conference
proceedings

2010 USENIX Conference on Web Application Development (WebApps '10)

Boston, MA, USA
June 23–24, 2010

Sponsored by
USENIX

Proceedings of the 2010 USENIX Conference on Web Application Development

Boston, MA, USA June 23–24, 2010

USENIX

ISBN 978-931971-76-8

USENIX Association

Proceedings of the 2010 USENIX Conference on Web Application Development

June 23–24, 2010
Boston, MA, USA

Conference Organizers

Program Chair

John Ousterhout, *Stanford University*

Program Committee

Dan Boneh, *Stanford University*
Mike Cafarella, *University of Michigan*
Stephen Chong, *Harvard University*
Thorsten von Eicken, *RightScale*
Armando Fox, *University of California, Berkeley*
Jeff Hammerbacher, *Cloudera*
Jon Howell, *Microsoft Research*
Wilson Hsieh, *Google*
Christopher Olston, *Yahoo! Research*
Marvin Theimer, *Amazon*
Helen Wang, *Microsoft Research*

The USENIX Association Staff

External Reviewers

Khaled Elmeleegy
Avi Shinnar

2010 USENIX Conference on Web Application Development
June 23–24, 2010
Boston, MA, USA

Wednesday, June 23

10:30–Noon

1:30–3:00

Thursday, June 24

10:30–Noon

1:30–3:00

3:30–4:30

Message from the Program Chair

Welcome to the 2010 USENIX Conference on Web Application Development! WebApps is a new conference this year, and we hope it will grow into a major annual event for discussing the latest ideas related to the development and deployment of Web applications. Web applications are revolutionizing software; over the next 10 years they are likely to change every aspect of the software development food chain, from languages and storage systems up to frameworks and management systems. However, until now there have been few opportunities for interaction and synergy across the diverse technologies related to Web applications. The goal of this conference is to bring together experts who span the entire ecosystem of Web applications.

The program committee accepted 14 excellent papers out of 26 submissions. The papers were reviewed in two rounds. In the first round each paper was read by four members of the program committee; most of the papers advanced to a second round, where they received another two reviews. The program committee met in person at Stanford University on March 9, 2010, to make the final selections for the conference; since most papers had been read by half of the program committee, the discussions were quite lively. Each accepted paper was assigned a program committee shepherd to guide the revision process for the paper.

I'd like to thank the many people who contributed time and effort to make this conference a success. First and foremost, thanks to all of the authors. Preparing a manuscript is a major undertaking; without you there would be no conference. Next I'd like to thank the program committee for their hard work in reviewing the submissions and shepherding the accepted papers. Finally, the USENIX organization did its usual stellar job in organizing and supporting the conference; thanks go to Ellie Young, Anne Dickison, Casey Henderson, Jane-Ellen Long, and the rest of the USENIX staff.

I hope you enjoy the conference and that it will inspire you to develop exciting ideas to submit to future WebApps conferences.

John Ousterhout, ***Stanford University***

Separating Web Applications from User Data Storage with BSTORE

Ramesh Chandra, Priya Gupta, and Nickolai Zeldovich
MIT CSAIL

ABSTRACT

This paper presents BSTORE, a framework that allows developers to separate their web application code from user data storage. With BSTORE, *storage providers* implement a standard file system API, and *applications* access user data through that same API without having to worry about where the data might be stored. A *file system manager* allows the user and applications to combine multiple file systems into a single namespace, and to control what data each application can access. One key idea in BSTORE's design is the use of *tags* on files, which allows applications both to organize data in different ways, and to delegate fine-grained access to other applications. We have implemented a prototype of BSTORE in Javascript that runs in unmodified Firefox and Chrome browsers. We also implemented three file systems and ported three different applications to BSTORE. Our prototype incurs an acceptable performance overhead of less than 5% on a 10Mbps network connection, and porting existing client-side applications to BSTORE required small amounts of source code changes.

1 INTRODUCTION

Today's web applications have *application-centric* data storage: each application stores all of its data on servers provided by that application's developer. For example, Google Spreadsheets [14] stores all of its documents on Google's servers, and Flickr [37] stores all of its photos at `flickr.com`. Coupling applications with their storage provides a number of benefits. First, users need not setup or manage storage for each application, which makes it easy to start using new apps. Second, each application's data is isolated from others, which prevents a malicious application from accessing the data of other applications. Finally, users can access their data from any computer, and collaborate with others on the same document, such as with Google Spreadsheets. Indeed, application-centric data storage is a natural fit for applications that perform server-side data processing, so that both client-side and server-side code can easily access the same data.

Although the application-centric data model works well in many cases, it also has a number of drawbacks. First, users are locked into using a single application for accessing any given piece of data—it is difficult to access the same data from multiple applications, or to migrate data from one application to another. By contrast, in the desktop environment, `vi` and `grep` have no problems accessing the same files. Second, application developers are forced to provide storage or server resources even if they just want to publish code for a new application, and even if users' data is already stored in other applications. By contrast, in the desktop environment, `http://gnu.org/grep` might only distribute the code for `grep`, and would not maintain servers to service users' `grep` requests. The makes it difficult for application developers to build client-side web applications.

To address this problem, we present BSTORE, a system that allows web application developers to decouple data storage from application code. In BSTORE, data can be stored in *file systems*, which provide a common interface to store and retrieve user data. File systems can be implemented by online services like Amazon S3 [4], so that the data can be accessed from any browser, or by local storage in the browser [12, 36], if stronger privacy and performance are desired. Multiple file systems are combined into a single namespace by the *file system manager*, much like the way different file systems in Unix are mounted into a single namespace. Finally, applications access data in BSTORE through the file system manager, without worrying about how or where the data is stored.

One challenge facing BSTORE is in providing security in the face of potentially malicious applications or file systems. While the application-centric model made it impossible for one application to access another application's data by design, BSTORE must control how each application accesses the user's shared data. Moreover, different users may place varying amounts of trust in file systems: while one user may be happy to store all of their data in Amazon S3, another user may want to encrypt any financial data stored with Amazon, and yet another user may want his grade spreadsheets to be stored only on the university's servers.

A second challenge lies in designing a single BSTORE interface that can be used by all applications and file systems. To start with, BSTORE's data storage interface must be flexible enough to support a wide range of application data access patterns. Equally important, however, is that any *management* interfaces provided by BSTORE be accessible to all applications. For example, any application should be able to specify its own access rights delegation or to mount its own file systems. If our design were to allow only the user to specify rights delegation, applications might be tempted to use their own file system manager when they find the need to specify finer-grained

access control policies or mount application-specific file systems. This would fracture the user's file namespace and preclude data sharing between applications.

A final challenge for BSTORE is to support existing web browsers without requiring users to install new plug-ins or browser extensions. While a browser plug-in could provide arbitrary new security models, we want to allow users and application developers to start using BSTORE incrementally, without requiring that all users switch to a new browser or plug-in simultaneously.

BSTORE's design addresses these challenges using three main ideas. First, BSTORE presents a unified file system namespace to applications. Applications can mount a new file system by simply supplying the file system's URL. A file system can either implement its own backend storage server or can use another BSTORE file system for its storage. Second, BSTORE allows applications to associate free-form *tags* with any files, even ones they might not have write access to. Using this single underlying mechanism, BSTORE enables an application to organize files as it chooses to, and to delegate access rights to other applications. Finally, BSTORE uses *URL origins* as principals, which are then used to partition the tag namespace, and specify rights delegation for files with different tags.

To illustrate how BSTORE would be used in practice, we ported a number of Javascript applications to BSTORE, including a photo editor, a `vi` clone, and a spreadsheet application. All of the applications required minimal amount of code changes to store and access data through BSTORE. We also implemented several BSTORE file systems, including an encrypting file system and a checkpointing file system. Our prototype of BSTORE incurs some in-browser processing overheads, but achieves overall performance comparable to using `XMLHttpRequest` directly for a typical home network connection.

The rest of this paper is organized as follows. Section 2 provides motivation and use cases for BSTORE. Section 3 details BSTORE's design, and Section 4 discusses our prototype implementation. Section 5 describes our experience using BSTORE in porting existing Javascript applications and in implementing file systems. Section 6 evaluates BSTORE's performance overheads, and Section 7 discusses some of BSTORE's limitations. Related work is discussed in Section 8 and finally Section 9 concludes.

2 MOTIVATING EXAMPLES

BSTORE can benefit the users and developers of a wide range of web applications, by giving users control over their data, by making it easier for applications to share data, and by removing the need for application developers to provide their own storage servers. An existing web application that has its own storage servers can also use BSTORE to either export its data to other applications, or to access additional data that the user might have stored elsewhere. The rest of this section describes a few use cases of BSTORE in more detail.

User wants control over data storage. In the current web application model, application developers are in full control of how user data is stored. Unfortunately, even well-known companies like Google and Amazon have had outages lasting several days [2, 23, 30], and smaller-scale sites like the Ma.gnolia social bookmarking site have lost user data altogether [21]. To make matters worse, the current model does not allow users to prevent such problems from re-occuring in the future. Some sites provide specialized backup interfaces, such as Google's GData [13], but using them requires a separate backup tool for each site, and even then, the user would still be unable to access their backed-up data through the application while the application's storage servers were down.

With BSTORE, users have a choice of storage providers, so that they can store their data with a reliable provider like Amazon or Google, even if they are using a small-scale application like Ma.gnolia. If the user is concerned that Amazon or Google might be unavailable, they can set up a *mirroring* file system that keeps a copy of their data on the local machine, or on another storage service, so that even if one service goes down, the data would still be available.

Finally, users might be concerned about how a financial application like Mint.com stores their financial data on its servers, and what would happen if those servers were compromised. Using BSTORE, users can ensure that their financial data is encrypted by mounting an encrypting file system, so that all data is encrypted before being stored on a server, be it Mint.com or some other storage provider. In this model the user's data would still be vulnerable if the Mint.com application were itself malicious, but the user would be protected from the storage servers being compromised by an attacker.

User accesses photos from multiple applications. There is a wide range of web applications that provide photo editing [9], manipulation [26], sharing [15], viewing [32], and printing [31]. Unfortunately, because of the application-centric storage model, users must usually maintain separate copies of photos with each of these applications, manually propagate updates from one application to another, and re-upload their entire photo collection when they want to try out a new application. While some cross-application communication support is available today through OAuth [25], it requires both applications to be aware of each other, thus greatly limiting the choice of applications for the user.

With BSTORE, all photo applications can easily access each other's files in the user's file system namespace. The user can also specify fine-grained delegation rules, such as granting Picasa access to lower-resolution photos for publishing on the web, without granting access to full-resolution originals. Web sharing applications such as Picasa could still store separate copies of photos on their servers (for example, for sharing on the web); this could be done either through an application-specific interface, as today, or by mounting that user's Picasa file system in BSTORE's file system manager.

Small-scale developer builds a popular application. In addition to common web applications like GMail and Picasa, there are a large number of niche web applications written by developers that might not have all of Google's resources to host every user's data. For example, MIT's Haystack group has written a number of client-side web applications that are popular with specific groups of users. One such application is NB [22], which allows users to write notes about PDF documents they are viewing. Currently, NB must store everyone's annotations on MIT's servers, which is a poor design both from a scalability and security perspective. BSTORE would allow individual users to provide storage for their own annotations. Many other small-scale web application developers are facing similar problems in having to provision significant server-side storage resources for hosting essentially client-side web applications, and BSTORE could help them as well.

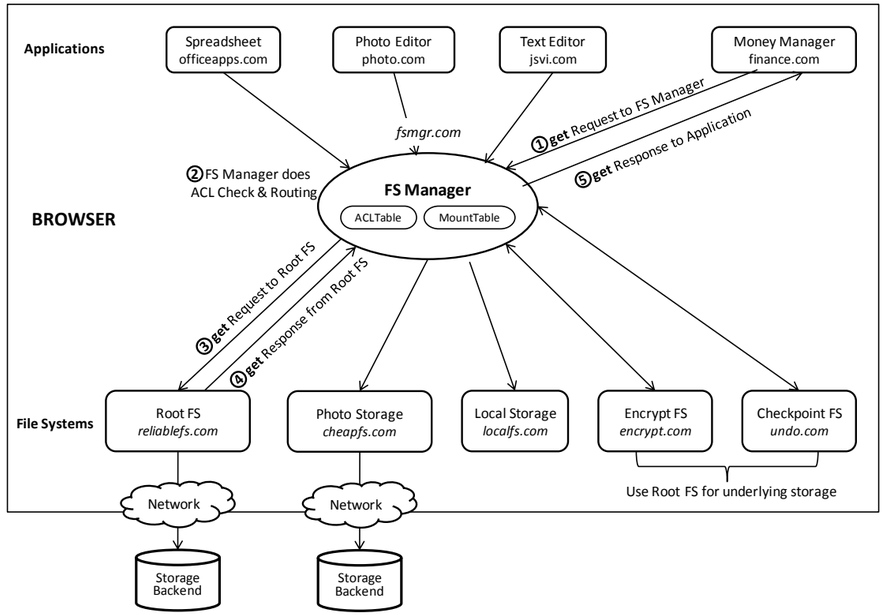


Figure 1: Overview of the BSTORE architecture. Each component in the browser corresponds to a separate window whose web page is running Javascript code. All requests to the BSTORE file system are mediated by the FS manager. Arrows (except in the **get** response flow) indicate possible direction of requests in the system.

3 DESIGN

The BSTORE design is based on the following goals:

Independence between applications and storage providers. In an ideal design, applications should be able to use any storage providers that the user might have access to, and the user should be able to manage their data storage providers independent of the applications. In particular, this would enable a pure Javascript application to store data without requiring that application's developer to provide server-side storage resources.

Egalitarian design. BSTORE should allow any application to make full use of the BSTORE API, and avoid reserving any special functionality for the user. One example is access control mechanisms: we would like to allow applications to be able to subdivide their privileges and delegate rights to other applications. Another example is mounting new file systems into the user's namespace: any application should be able to mount its own server-side resources into BSTORE, or to encrypt its data by mounting an encrypting file system.

No browser modifications. BSTORE should not require browser modifications and should work with existing browser protection mechanisms (same origin policy or SOP).

3.1 Overview

Figure 1 illustrates BSTORE's architecture. A user's BSTORE is managed by a trusted file system manager (FS manager). It mediates all file system requests in the system. Applications (shown at the top of the figure), send their file system requests to the FS manager, which routes them to the appropriate file system (shown at the bottom of the figure). The FS manager also performs access control during this request routing process.

The Javascript code for the FS manager, applications, and file systems run in separate protection domains (browser windows) and are isolated from each other by the browser's same origin policy. All communication happens via `postMessage`, which is the browser's cross-domain communication mechanism.

The FS manager allows users and applications to mount different file systems and stitch them together into a unified namespace. A BSTORE file system exports a flat namespace of files, and does not have directories. Files do not have a single user-defined name; instead each file has a set of tags. An application can categorize a file as it sees fit by setting the appropriate tags on that file. At a later time, it can search the file system and recall files that match a specific tag expression. Using tags in this manner for file system organization allows each application the flexibility to organize the file system as it chooses.

File tags are also the basis for access control in BSTORE. An application uses tags to delegate a subset of its access rights to another application. The FS manager keeps track of these delegations in the ACL table and enforces access control on every file system request.

3.2 BSTORE API

Table 1 shows the API calls exported by the FS manager to each application, and Table 2 shows the API calls exported by each BSTORE file system to the FS manager.

The API calls in Table 1 are divided into two categories, shown separated by a line in the table. The calls in the top category are file system calls that correspond directly to API calls in Table 2. The calls in the lower category are FS manager-only calls related to access rights delegation and creating mountpoints. Among the file system calls, **create** and **search** operate on a file system handle (*fs_h*), and the rest of the calls operate on a file handle (*fh*). The **get** and **set** calls operate on entire objects; there are no partial reads and writes. This design choice is consistent with the behavior of majority of web applications, which read and write entire objects, and with other web storage APIs, such as Amazon S3 and HTML5 local storage.

FS manager API call	Return on success	Rights needed
create(fs_h, init_tags)	fh, ver	write to creator tag
search(fs_h, tag_expr)	fh_list	read matching files
set(fh, data, [match_ver])	ver	write on file
get(fh, [match_ver])	ver	read on file
stat(fh)	ver, size	read on file
delete(fh, [match_ver])	—	write on file
settag(fh, tag)	ver	read on file
gettag(fh)	ver, tag_list	read on file
rmtag(fh, tag)	ver	read on file
setacl(target_principal, tags, perms)	—	—
getacl([target_principal])	—	—
rmacl(target_principal, tags, perms)	—	—
encrypt(plaintext)	ciphertext	—

Table 1: API exported by the FS manager to each BSTORE application. The rights column shows the rights needed by an application to perform each API call.

FS API call	Return on success
create(init_tags, acltagset)	fh, ver
search(tag_expr, acltagset)	fh_list
set(fh, data, acltagset, [match_ver])	ver
get(fh, acltagset, [match_ver])	ver
stat(fh, acltagset)	ver, size
delete(fh, acltagset, [match_ver])	—
settag(fh, tag, acltagset)	ver
gettag(fh, acltagset)	ver, tag_list
rmtag(fh, tag, acltagset)	ver

Table 2: API exported by each BSTORE file system to the FS manager. The FS manager fills in the *acltagset* based on the requesting application's access rights.

The rights column in Table 1 shows the access rights needed by an application to perform each API call. When an application makes a file system call, the FS manager uses the access rights delegated to the application to fill in the *acltagset* shown in API calls in Table 2. The file system uses the *acltagset* to perform access control, as will be described in more detail in Section 3.5.

Current browsers use the URL domain of a web application's origin as the principal while applying same origin policy to enforce protection between browser windows. Since one of our goals is to not modify browsers, we also choose URL domains of an application's or file server's origin as principals in BSTORE.

3.3 Tags

A tag is an arbitrary label that a principal can attach to a file (e.g., *low-res* on a photo file). A file can have multiple tags set by different principals. Tagging is the underlying mechanism using which an application (or user) can both organize the BSTORE namespace in ways meaningful to it, as well as to delegate access rights to other applications.

Applications use **settag** to tag a file. An application can tag other applications' files as long as it can read those files. This allows an application to categorize files in BSTORE in a manner that best fits its purpose. For example, a photo editor application can tag a photo file with the date of the photo, location at which it was taken, or the names of people in it. To avoid applications clobbering each others tags, each principal gets its own tag namespace, and can only set tags within that namespace. So, a *low-res* tag set by Flickr is actually *flickr.com#low-res*, and is different from *low-res* tag set by Google Picasa (which is *picasa.com#low-res*).

In BSTORE, tag-based search is the only mechanism by which applications can lookup files and get handles on them. **search** takes a file system handle and a tag query expression and returns files in that file system whose list of tags satisfy the tag query, and which are readable by the requesting application. The tag query expression supports simple wildcard matches. **settag** and **search** together allow an application to organize the BSTORE namespace, and recall data in ways meaningful to it.

Applications use **gettag** to retrieve a list of all tags for a file, including those set by other applications. **rmtag** allows an application to delete only the tags that were set by the same principal as the application.

3.4 File systems

Every mounted BSTORE file system has a Javascript component running in a browser window. This Javascript component exports the BSTORE file system API to the FS manager, and services requests from it. Some file systems store their data on network storage or on local storage (e.g., *reliablefs.com* and *localfs.com* in Figure 1). Others, called *layered file systems*, store their data on existing BSTORE file systems and layer additional functionality on them (e.g. *encrypt.com* encrypting data and storing it on *reliablefs.com*).

All file systems (except the root file system) are specified in mountpoint files. A mountpoint file for a file system contains information to mount the file system, in a well known format. This information includes the URL for the file system Javascript code, file system configuration parameters, and mountpoint tags. Just as it creates other files, an application uses the **create** call to create a mountpoint file with the right configuration information in the right format. Other applications can use the mountpoint file to access the file system referenced by it, as long as the access control policy allows it. Since mountpoint information can contain sensitive data (such as credentials and mountpoint tags), the application uses the FS manager's **encrypt** API call to encrypt the information and stores the encrypted information in the mountpoint file. The FS manager encrypts data using a key that is initialized when it is first set up, as described in Section 3.8.

To access a file system, an application passes a handle to the file system's mountpoint file to **create** and **search** API calls. If the file system is not already mounted, the FS manager uses this handle to read the mountpoint file, decrypt the mountpoint information, and launch the file system in a separate browser window using this information. Once the file system is initialized, the FS manager adds the file system to its mount table and routes requests to it.

Storing the mountpoint information encrypted allows an application to safely give other applications access to a file system, without giving away credentials to the file system. An application can also safely give other applications a copy of the mountpoint files it created, which can be used for mounting without leaking any sensitive information.

A user or application that created a mountpoint file may want to tag all files in that file system with a particular tag to enforce access control on all files in that file system. It is cumbersome to tag each file individually with that tag at mount time. Instead, BSTORE allows mounting file systems with *mountpoint tags* that are logically applied to every file on that file system.

3.5 Access control

A BSTORE principal obtains all its access rights through delegation from other principals. As the base case, the file system's principal is assumed to have full access to all files on that file system. A principal A can specify a *delegation rule* to delegate a subset of its access rights to another principal B. Delegation is transitive and principal B can in turn delegate a subset of rights it obtained from A to another principal C. Given all the delegation rules in the system, BSTORE computes the access rights that are transitively granted by any principal A to another principal B, by computing the transitive closure of the delegation rules. The access rights for an application with principal A to a file on principal F's file system are the rights that are transitively delegated by F to A.

Delegation rules use file tags to specify the files to which they apply. Since tags are application-defined properties on files, this allows an application (say, Picasa) to easily express intentions such as "Flickr can read all my low resolution photos on any file system," without having to search through the entire file system for low resolution photos and adding Flickr to each file's access control list.

The delegation rules described above are decentralized, and each file system can independently delegate access, maintain delegation rules from other principals, and compute transitive delegations itself. However, in practice, it is convenient for users to specify their access control policy in a single place. To achieve this, all delegation rules are centralized in the FS manager. File systems also follow a convention that, when mounted, they delegate all access to the FS manager. This allows the FS manager to make access control decisions for that file system (by using delegation from the FS manager's principal to specific applications). In case a file system does not want to use the FS manager's centralized ACL manager (e.g. in the case of an application mounting its own

application-specific file system), it can choose not to follow this convention.

The FS manager maintains an ACL table of all delegation rules. A delegation rule is of the form $\langle A, B, T, R \rangle$, and signifies that the rights R (read or read_write) are delegated from principal A to principal B for every file that is tagged with all the tags in tag set T. Going back to the example above, Picasa's intentions translate to a delegation rule $\langle$*picasa.com*, *flickr.com*, *picasa.com#low-res*, *read*$\rangle$. Any principal (application or file system) can invoke **setacl** and **rmacl** API calls on the FS manager to add and remove delegation rules. Both these calls take B, T, and R as arguments. A principal can use the **getacl** call to retrieve its delegations to a particular target principal. It can obtain all its delegations to any principal by omitting the target principal parameter to **getacl**.

The FS manager computes and maintains the transitive closure of the delegation rules added by all the principals. On a request from application A to file system F, the FS manager looks up the transitive closure and includes all the rights transitively delegated by F to A in the request before sending it to F (these rights are indicated by argument *acltagset* in Table 2). The FS manager does not fetch tags from F and do the rights check itself to avoid an extra round trip. The file system F checks the rights before performing the operation.

BSTORE's access control rules allow one principal to delegate access based on tags from another principal. In our example, if the user's low-resolution photos are already tagged *kodak.com#low-res*, Picasa can delegate access to *flickr.com* based on that tag, instead of tagging files on its own. However, this opens up the possibility of *kodak.com* colluding with flickr: if kodak tags all files with this tag, flickr will be able to access any file accessible to Picasa. The FS manager makes this trust relation explicit by verifying, on each ACL check, that the principals in a delegation rule's tags are also allowed to perform the operation in question. Thus, in our example, Picasa's delegation rule would not apply to files tagged *kodak.com#low-res* that *kodak.com* was not able to access itself.

3.6 File creation

An application (say *app.com*) creates a file using the **create** call, by specifying a handle to the target file system (say on *fs.com*). The application also specifies a list of initial tags to set on the new file. In addition, the FS manager sets a special creator tag (*fsmgr.com#app.com_creator*) on the file.

For create to succeed, *app.com* should have write access to the tag set consisting of the mountpoint tags of the file system *fs.com* and the creator tag. The creator tag is also used to delegate all access rights on that file to the creator application. This is done by the FS manager adding a delegation rule $\langle$*fsmgr.com, app.com, fsmgr.com#app.com_creator, read_write*$\rangle$.

3.7 File versions

Since BSTORE is shared storage, multiple applications could be concurrently accessing a file. To help applications detect concurrent modifications, the BSTORE API provides compare-and-swap like functionality. All files are versioned. Mutating file system calls take an optional *match_version* argument and fail with a `EMODIFIED` error if the current file version does not equal *match_version*. A file's version number is automatically incremented on every **set** operation, and most API calls also return the current file version. Versioning proves useful in other scenarios as well: it can be used to keep track of change history, as in the case of the checkpointing file system described in Section 5.2.2, and to support transparent local caching of network storage.

3.8 Bootstrapping

We now give a brief overview of how a user interacts with BSTORE, including initial setup of the different components and subsequent use.

FS manager. We imagine that users will be able to choose from multiple implementations of FS manager from different vendors, depending on whom they trust and whose UI they prefer. Once a user chooses a FS manager, she sets up her browser by browsing to the FS manager and configuring it with the information to mount the root file system. The FS manager stores this mount information in the browser's local storage and uses it for subsequent sessions. It also mounts the root file system, and sets up BSTORE by initializing its encryption key and delegation rules table. The user needs to repeat the browser setup step for every browser she uses. The BSTORE setup only happens once. During normal use, the FS manager also provides the user with a UI to create and manage delegation rules.

Applications. For each application, the user adds delegation rules to the FS manager to grant the application access to specific files in BSTORE. The user initializes the application with the FS manager URL, which it uses to launch the FS manager if it is not already launched. The application also stores the FS manager URL in its network storage or local storage for subsequent sessions.

During normal use, when the user needs to choose a file, applications present a file chooser interface to the user. The file chooser allows her to search specific file systems for files with specific tags, and to choose a file from the results.

File systems. The user obtains an account for file systems that store data on the network. File systems provide a friendly user interface using which the user can create a mountpoint file containing the parameters for that particular file system.

For example, when setting up the *Photo Storage* file system from *cheapfs.com* (shown in Figure 1), the file system UI prompts the user for her FS manager URL, file system on which to create the mountpoint file, the initial tag for the mountpoint file and mountpoint tags, and creates an encrypted mountpoint file that includes the user's credentials to access the file system.

When the file system is launched during normal use, it does not need any input from the user—the FS manager sends it the mountpoint information stored in the mountpoint file as part of the file system initialization, which includes the credentials required to access the file system.

3.9 Putting it all together

We now describe an example that illustrates how the different BSTORE components work together.

Say a user uses the *Money Manager* application shown in Figure 1 to compute her taxes. Assume that the tax files are tagged with *finance.com#tax*, where the principal of *Money Manager* is *finance.com*, and that the tax files are stored on *reliablefs.com*.

When the user launches *Money Manager*, it launches the user's FS manager in a separate browser window and initializes communication with it. It then sends the FS manager a **search** request to search for files with tag *finance.com#tax* on the *reliablefs.com* file system. The FS manager mounts the file system (if necessary) by launching it in a separate browser window. Using its delegation rules table, the FS manager computes the *acltagset* for *finance.com* on *reliablefs.com* and forwards the request to *reliablefs.com* file system with the *acltagset*. The file system retrieves all files that are tagged with *finance.com#tax* and returns only the handles of those files which are readable by *finance.com*, as specified by the *acltagset*. This result is routed back to *Money Manager* by the FS manager.

Money Manager uses these file handles to fetch the file data by issuing **get** requests; these requests follow a similar path as the **search** requests. Figure 1 illustrates the request flow for a **get** request from *Money Manager* to the *reliablefs.com* file system.

4 IMPLEMENTATION

We built a prototype of BSTORE, including the FS manager and a storage file system, which together comprise 2199 lines of Javascript code, 712 lines of PHP, and 136 lines of HTML.

All BSTORE client-side components are written in Javascript and work with the Firefox and Google Chrome browsers. Each component runs in a separate browser window, and is loaded from a separate domain. Inter-domain communication within the browser is via `postMessage`. The origin domain of the caller is included by the browser platform as part of the `postMessage` call, and is used as the caller principal.

4.1 Storage file system

The Javascript component of the prototype storage file system implements the BSTORE storage API. The network storage component is written in PHP, and currently runs on a Linux/Apache stack. Communication between the file system Javascript and the PHP components is via `XMLHttpRequest` POST. All information in the request and response, except for file data, is JSON encoded. File data in a **set** request is sent as a binary byte stream. File data in the response of a **get** request is Base64 encoded, since `XMLHttpRequest` does not support byte streams in the response. The response is also gzip encoded to offset the increase in size due to Base64 encoding.

User authentication to the file system is via username and password. The network storage uses a ext3 file system for storage and BSTORE files are stored as files in the file system. A file's version number is stored as an extended attribute of the ext3 file and file tags are stored in a database. The tags database is indexed for efficient search.

4.2 FS manager

To keep track of delegation rules, BSTORE represents each principal with a file in the root file system, containing the delegation rules from that principal to other principals. To avoid file system access on every access check, the FS manager caches these rules in memory. A cached entry for a principal is invalidated if the delegation rules were changed for that principal in that FS manager. To account for other FS manager instances, the cache entries also have an expiry time (currently 5 minutes) and are refreshed in the background. A better approach would be to have the FS manager be able to register for notifications from the file system when the files storing the delegation rules are modified. Due to this and other similar examples, we are considering adding version change notifications to BSTORE.

5 APPLICATIONS AND FILE SYSTEMS

To illustrate how BSTORE would be used in practice, we implemented one Javascript application that uses BSTORE, and ported three existing Javascript web applications to use BSTORE. We also implemented encryption and checkpointing file systems, to demonstrate the use of layered file systems. Table 3 summarizes the amount of code involved in these applications and file systems, and the rest of this section describes them in more detail.

5.1 Applications

The following applications work with our current BSTORE prototype:

Application/FS	Original LOC	Modified LOC
Shell	337	NA
Pixastic	4,245	81
jsvi	3,471	74
TrimSpreadsheet	1,293	66
EncryptFS	239	NA
CheckpointFS	595	NA

Table 3: Lines of Javascript code for BSTORE applications and file systems.

Shell provides basic commands that the user can use to interact with the storage system, including search, set, cat, stat, and unlink. We implemented this application from scratch.

Pixastic [28] is an online Javascript photo editor that performs basic transformations, such as rotate, crop, and sharpen, on a user's images. The version on the author's website allows a user to upload an image from their computer, perform transformations on it, and download it back to their computer. We modified Pixastic to work with BSTORE, so that a user can load, manipulate, and save images to BSTORE.

jsvi [18] is a vi-clone written purely in Javascript. The original version supports most vi commands and is a fully functional editor, but does not allow a user to load or save files (even to her local desktop). It only temporarily saves the file contents to a HTML text area, but they are lost once the web page is refreshed. Our modified jsvi works with BSTORE, and loads and saves files like a typical vi editor.

TrimSpreadsheet [34] is an open-source Javascript spreadsheet engine, and lets a user work with spreadsheet data in the browser. In the original application, the spreadsheet data was stored as HTML markup, which meant spreadsheets could be edited in a browser but the changes did not persist across browser sessions. We modified TrimSpreadsheet so that it can save and load spreadsheet data from BSTORE.

Modifying Pixastic, jsvi, and TrimSpreadsheet to work with BSTORE was straightforward and involved less than a day's worth of work to understand each application and add code to interact with BSTORE. Table 3 gives a tally of the lines of code for each application; for the applications we modified, the table also gives the number of lines changed to port the application to BSTORE. As can be seen from the table, porting applications required relatively few modifications to the application code.

5.2 Layered file systems

This section describes the motivation, design details, and usage of the encryption and checkpointing file systems. Each file system is a few hundred lines of code and involved a few days of effort to implement (as opposed to a couple of months of work to implement the base system). Given this experience, we feel that layering additional functionality on an existing BSTORE system is relatively easy.

5.2.1 *Encrypting file system*

Consider the scenario described in Section 2, where a user wants to encrypt her financial data before storing it. The user does not trust the underlying storage provider enough to store sensitive data in the clear, but trusts encryption functionality provided by, say, `encrypt.com`. In BSTORE, she can configure her applications to talk to the underlying storage via `encrypt.com`'s encrypting file system. We built a sample encrypting file system, *EncryptFS*, for this scenario, which works as described below.

EncryptFS provides a setup UI that the user can use to create a mountpoint. The setup process is similar to that of the Photo Storage file system, described in Section 3.8. The resulting mountpoint file stores the encryption password and the backing file system that EncryptFS uses to store its data. Once the mountpoint is created, the user adds a delegation rule to the FS manager allowing EncryptFS access to its backing file system. The user then proceeds to configure her financial application to use EncryptFS as discussed in Section 3.8.

We use the jsCrypto library from Stanford to perform cryptographic operations in Javascript [33]. We use the OCB mode of encryption with block size, key, IV, and MAC being 16 bytes. On a **set** request, the object contents are encrypted using the encryption key. The MAC and IV are attached to the beginning of the encrypted content, and the request is then written to the storage. On a **get** request, the encrypted content is fetched from storage, MAC and IV are extracted, the rest of the content is decrypted, the MAC on relevant tags is verified and the decrypted content returned.

Tags are critical in making access control decisions and so an untrusted FS cannot be trusted to return the correct tags. To get around this problem, EncryptFS MACs tags during **settag** and on a **gettag** will only return tags that have a valid tag. EncryptFS is relatively simple and does not prevent replay or rollback attacks. We can use well known techniques, such as those in the SUNDR file system [19], to get around such attacks.

5.2.2 *Checkpointing file system*

Imagine a user who wants to try a new application, say a photo retouching software that runs through her photo collection and enhances images that are below a certain quality. She does not trust the reliability of the software and does not know whether it would leave her photo collection in a damaged state if she lets it run on it. The simple checkpointing file system, *CheckpointFS*, that we describe here, helps with this situation by keeping an undo log of all the changes that the application made to the storage system, and at the end of the application run, giving the user an option either to commit the changes or to revert them.

The set up for CheckpointFS is similar to that of EncryptFS, except that the mountpoint configuration in this case only consists of the backing file system where CheckpointFS stores the data and its undo log.

CheckpointFS records undo information in its log for every mutating operation. The undo information consists of the operation performed, timestamp, and version information for the file on which the operation was performed. In addition, for **settag** and **set**, a copy of the file tags and file contents respectively, is stored in the undo log. CheckpointFS stores these undo log records in memory and dumps them every minute to log files (numbered sequentially). These records could also be stored in browser local storage instead of memory if crash safety is an issue.

CheckpointFS keeps logging requests until the user indicates in its UI that she is done with her application session. At this point, all in memory logs are dumped, and CheckpointFS temporarily stops accepting further requests. The user is then given the choice to either to rollback to previous checkpoint or to commit the changes thereby wiping out the old checkpoint and creating a new one. If the user chooses to rollback, all the logs are read and the version information is checked to make sure that no other application performed an intervening mutating operation on the backing file system that will be clobbered by the rollback.

Though the current UI is simple and provides only one checkpoint, the information CheckpointFS logs could be used to provide more finer grained rollback capabilities. For example, CheckpointFS could store multiple checkpoints and allow the user to rollback to any previous checkpoints, it could automatically take a checkpoint at regular intervals, or it could provide a finer-grained undo of a subset of files.

6 BSTORE PERFORMANCE

For BSTORE to be practical, it should have acceptable overhead and its performance should be competitive with alternate data access mechanisms for web applications. Web applications today typically use `XMLHttpRequest` (XHR) to asynchronously GET data from, and POST data to web servers. We ran a set of experiments to measure the performance of BSTORE under different configurations, and compared it with XHR GET and XHR POST. We also measured the overhead of BSTORE's layering file systems, and compared BSTORE's performance on two different browsers.

For all experiments, the BSTORE file system server was an Intel Xeon with 2GB of RAM running Debian Squeeze, with Apache 2.2 and PHP 5.3.2. The client machine was an Intel Core i7 950 with 12GB of RAM running Ubuntu 9.10. The web browsers we used were Firefox 3.5.9 and Google Chrome beta for Linux. The local network between the client and the server is 100Mbps ethernet.

Size	BSTORE-Get	XHR-Get	BSTORE-Set	XHR-Post
1 KB	17.6 ms	5.0 ms	18.9 ms	5.3 ms
5 KB	18.6 ms	6.0 ms	19.0 ms	5.9 ms
10 KB	19.7 ms	6.6 ms	19.4 ms	6.5 ms
100 KB	40.2 ms	18.8 ms	34.0 ms	15.7 ms
500 KB	117.5 ms	66.6 ms	102.3 ms	59.4 ms
1 MB	225.9 ms	141.2 ms	174.8 ms	116.8 ms

Table 4: Comparison of get and set operation performance on a BSTORE file system to XHR-get and XHR-post.

6.1 BSTORE file system performance

In our first experiment, we compare the performance of BSTORE **get** and **set** with XHR GET and XHR POST on a local network. The experiment consists of fetching and writing image files of various sizes ranging from 1 KB to 1 MB. The server side for XHR GET and XHR POST is a simple PHP script that sends back or accepts the required binary data. The BSTORE **get** and **set** requests are to the root file system, and mirror the request flow illustrated in Figure 1, with delegation rules set to allow read and write access to the required files for the application running the experiment. Since this experiment is on a local network, it highlights the overhead of BSTORE mechanisms, as opposed to the network transfer cost. The web browser used in this experiment is Firefox.

The results of the experiment are shown in Table 4. To remove variability, we ran 24 runs of the experiment and removed the runs with the two highest and two lowest timings. The numbers shown in the table are the average times of the remaining 20 runs. From the table, we see that XHR operations are about three times faster than BSTORE operations for small files. For a large file of 1 MB size, BSTORE-Get is about 60% slower than XHR-Get and BSTORE-Set is about 50% slower than XHR-Post. Most of this overhead is due to processing within the browser. For example, for BSTORE-Set on a 1 MB file, 19.9% of the time is spent in encoding/decoding data in the browser, 5.7% in communication within the browser using `postMessage`, and 73.6% in communication between the storage file system Javascript and backend server using XHR POST.

BSTORE-Get is slower than BSTORE-Set due to gzip overhead. Recall from Section 4 that the requests from BSTORE storage file system Javascript to the backend are in binary; the responses, however, are Base64 encoded and gzipped, as XHR does not support byte streams in the response. This means that BSTORE-Get involves compression of data on the backend and decompression in the browser, which is not present in BSTORE-Set. The overhead for these operations dominates the total time as the local network is fast.

Size	BSTORE-Get	XHR-Get	BSTORE-Set	XHR-Post
1 KB	117.7 ms	105.0 ms	117.6 ms	105.5 ms
5 KB	217.5 ms	205.0 ms	268.4 ms	255.3 ms
10 KB	218.9 ms	205.5 ms	268.9 ms	255.3 ms
100 KB	894.0 ms	927.7 ms	981.0 ms	1059.7 ms
500 KB	4223.2 ms	4172.9 ms	4451.5 ms	4409.4 ms
1 MB	8622.3 ms	8500.9 ms	8978.7 ms	8916.6 ms

Table 5: BSTORE FS performance on a 1Mbps, 100ms latency network.

Size	BSTORE-Get	XHR-Get	BSTORE-Set	XHR-Post
1 KB	27.5 ms	15.2 ms	29.1 ms	15.6 ms
5 KB	37.4 ms	16.6 ms	48.7 ms	36.1 ms
10 KB	38.5 ms	20.6 ms	49.5 ms	36.9 ms
100 KB	116.2 ms	94.8 ms	138.3 ms	119.8 ms
500 KB	451.4 ms	423.9 ms	498.9 ms	472.9 ms
1 MB	901.1 ms	863.2 ms	982.1 ms	935.1 ms

Table 6: BSTORE FS performance on a 10Mbps, 10ms latency network.

Though overhead of BSTORE requests seem high, they represent performance on an unrealistically fast network. On a more realistic network with lower bandwidth, the network cost dwarfs BSTORE overhead as the next experiment illustrates. Furthermore, these overheads are primarily due to lack of support for binary data in XHR responses and in `postMessage`, and can be significantly reduced by adding this support. The `responseBody` attribute of XHR, being considered by W3C for a future version of the XHR specification, supports binary byte streams, and is a step in this direction.

6.2 Wide-area network performance

In order to evaluate BSTORE overhead in real-world networks, we ran the same experiment as above on simulated networks with realistic bandwidths and latencies. We chose two network configurations: a 1Mbps network with 100ms round-trip latency, and a 10Mbps network with 10ms round-trip latency. The slow networks are emulated using the Linux traffic control tool (tc).

The results are shown in Tables 5 and 6. We see from the tables that the BSTORE overhead, as compared to plain XHR, drops considerably. Overhead of BSTORE-Get over XHR-Get for a 1 MB file drops from 60% in local network to 4% on 10Mbps network, and 1.4% on 1Mbps network. Similarly, overhead of BSTORE-Set over XHR-Post for a 1 MB file drops from 50% in local network to 5% on 10Mbps network, and 0.7% on 1Mbps network. It is clear from these results that the browser overheads in BSTORE become insignificant compared to network cost for realistic networks.

Another point illustrated in these tables is the effect of gzip in slower networks. For a 1 MB file on slower networks, BSTORE-Get is faster than BSTORE-Set, whereas the opposite held true on the local network. This is because gzip reduces the number of bytes transferred over the network; on slow networks the resulting time saved more than offsets the time taken to compress and decompress the data.

6.3 Performance of layered file systems

The previous experiments focused on the scenario of an application accessing data on the BSTORE root file system. BSTORE also supports layered file systems. In this experiment, we measure the overhead of layered file systems on the same workload as the previous experiments. We use three layered file systems: a null layered file system which passes data back and forth without any modification, and *EncryptFS* and *CheckpointFS* described in Section 5.2. The measurements are performed on the local network.

Table 7 shows the results. From the table we see that the overhead of a null layered file system is small—about 6% for BSTORE-Get and 7% for BSTORE-Set, for the 1 MB file. This overhead is due to the extra `postMessage` calls and encoding/decoding as requests pass through the layered file system and FS manager. We believe this overhead is reasonable; also, on a slower network, this overhead becomes a smaller fraction of the overall time, further reducing its impact.

For *EncryptFS*, the bulk of the time is spent in cryptographic operations. For a 1 MB file, decryption takes 3095ms (85.1% of Get time) and encryption takes 2534ms (90.3% of Set time). We also observed that a `postMessage` that follows a crypto operation takes more than an order of magnitude longer than other `postMessage` calls. We believe that this variability is a characteristic of the Firefox Javascript engine. We confirmed our suspicion by testing `postMessage` times after a CPU intensive tight loop, and observing that it does indeed take an order of magnitude longer than normal.

For *CheckpointFS*, Get performance is close to that of null layered file system, as it does not do anything on a Get. The overhead in its Set operation is due to logging—this involves an extra RPC to fetch the old file contents and store them in the log. For this experiment, the old file was always 1 KB in size.

6.4 Browser compatibility

Today people use many different browsers, and an important consideration for a good web application framework is cross-browser support. We primarily implemented BSTORE for Firefox, but were able to run it on Google Chrome with a small modification. XHR POST on Chrome does not support sending binary data, and so we had to change our implementation to send data in Base64 encoded form. We predict that porting BSTORE to other browsers such as IE and Safari will require some changes, mainly because of the difference in ways these browsers handle events, `postMessage`, and some other differences in Javascript engine.

To see how BSTORE performs on Chrome, we ran the first experiment using Chrome. Table 8 shows the results as compared to performance on Firefox. Chrome is slower

Size	No Layering		Null Layered FS		EncryptFS		CheckpointFS	
	Get	Set	Get	Set	Get	Set	Get	Set
1 KB	17.6 ms	18.9 ms	19.2 ms	19.9 ms	24.9 ms	28.2 ms	19.2 ms	37.8 ms
5 KB	18.6 ms	19.0 ms	20.3 ms	20.4 ms	45.0 ms	47.2 ms	20.4 ms	37.3 ms
10 KB	19.7 ms	19.4 ms	21.4 ms	21.1 ms	63.4 ms	65.2 ms	21.8 ms	38.8 ms
100 KB	40.2 ms	34.0 ms	43.0 ms	36.8 ms	300.5 ms	292.3 ms	45.0 ms	53.7 ms
500 KB	117.5 ms	102.3 ms	132.9 ms	110.0 ms	1494.5 ms	1359.8 ms	136.0 ms	123.2 ms
1 MB	225.9 ms	174.8 ms	238.5 ms	187.8 ms	3636.0 ms	2806.8 ms	247.4 ms	208.6 ms

Table 7: Performance of various layered file systems under BSTORE performing get and set operations.

Size	Firefox		Chrome	
	Get	Set	Get	Set
1 KB	17.6 ms	18.9 ms	14.8 ms	15.6 ms
5 KB	18.6 ms	19.0 ms	15.8 ms	16.1 ms
10 KB	19.7 ms	19.4 ms	17.4 ms	18.6 ms
100 KB	40.2 ms	34.0 ms	47.6 ms	44.6 ms
500 KB	117.5 ms	102.3 ms	141.8 ms	143.4 ms
1 MB	225.9 ms	174.8 ms	258.4 ms	256.4 ms

Table 8: Performance of BSTORE on Firefox and Chrome.

than Firefox on larger files, and the difference is primarily due to slower `postMessage` and slower XHR POST. For example, for a 1 MB file get, `postMessage` was 30.2ms in Chrome as compared to 7.2ms in Firefox, and XHR POST was 226.1ms in Chrome as compared to 202.9ms in Firefox. From these results, overall we can conclude that BSTORE works reasonably well in Chrome.

7 DISCUSSION

In its current form, BSTORE does not support all applications. Collaborative web applications (such as email and chat) need a server for reasons other than storage, and BSTORE does not eliminate the need for such servers. Social applications, such as Facebook, require sharing data between users. BSTORE currently does not support cross-user sharing; for one, the principals do not include a notion of a user. We plan to explore extending BSTORE to support cross-user sharing, perhaps by building on top of OpenID. In the meanwhile, social applications can still use BSTORE to store individual users' files, and implement cross-user sharing themselves.

The principal of an application in BSTORE is the URL origin from where the application is loaded. This makes it difficult to support serverless applications, where the Javascript code for the application can be hosted anywhere, or even passed around by email. BSTORE could be extended to add support for principals that are a hash or a public key corresponding to application code.

We have chosen to support an object get and set API in BSTORE, which works well for many applications, including the ones we used in our evaluation. Likewise, our tagging model fits well with the data model of existing applications like GMail and Google Docs [14], and can be also used to express traditional file-system-like hierarchies. However, some applications may require a richer interface for querying their data, such as SQL, and tags cannot express the full range of such queries. Storing an entire SQL table as a file in BSTORE may be acceptable for small data sets, but accessing large data sets efficiently would require adding a database-like interface along the lines of Sync Kit [5].

With BSTORE, users potentially have to worry about providing storage, whereas in the current model all storage is managed by the application provider. The simplicity of today's model could also be supported with BSTORE, where each application mounts its own storage in the user's file system manager, with the added benefit that applications can now easily share data with each other. At the same time, the user could have the option of procuring separate storage from a well-known online storage provider, such as Amazon S3 or Google, which would then store data for other applications the user runs, and backup data from existing application stores.

Due to the level of indirection between tags and the associated access rights, a user may inadvertently leak rights by tagging a file incorrectly without realizing it. To avoid this risk, the file tagging UI in applications and FS manager can resolve and display the new principals being granted access due to the addition of the tag.

Currently, all BSTORE components run in separate browser windows; this can present the user with too many windows. This can be mitigated by running the FS manager in a window and all the file systems as iframes within the FS manager window. If extending the browser is feasible, a browser extension to support "background windows" would provide a better user experience.

8 RELATED WORK

The idea of a unified file system namespace has been explored in earlier systems like Unix and Plan 9 [27]. Moreover, various distributed file systems [1, 6, 16, 29] provide a uniform storage platform, along with the security mechanisms needed for authentication and access control. BSTORE addresses new challenges posed by web applications, including the need for different storage models (using tags and tag search), the need for applications to mount their own file systems, and the need for flexible delegation of access control, without requiring any changes to client-side browsers.

Similar to tags in BSTORE, semantic file systems [11] and the Presto document system [8] provide alternate file

system organization using file attributes, in addition to the hierarchical file system namespace.

SUNDR [19] provides a network file system designed to store data securely on untrusted servers, and ideas from it would be directly applicable to designing a better encrypting file system for BSTORE.

Google gears [12], HTML5 [36] local storage, and Sync Kit [5] aim to improve web application performance and enable offline use with client-side persistent storage. However, these mechanisms still provide isolated storage for each application, and do not address sharing of data between applications.

There are also a number of browser plug-ins that provide alternative environments for deploying client-side applications [3, 7, 20, 38]. While some of them provide machine-local storage, none of them provide *user*-local storage that is available from any machine that the user might access. Accessing data stored in BSTORE from one of these environments currently requires going through Javascript; in the future, BSTORE could support native bindings for other execution environments.

Some websites provide mechanisms, such as REST APIs or OAuth, for users to access their data from external services. OAuth is an open protocol that allows a user to share her web application data with another web application from a different origin. However, unlike BSTORE, both applications should know of each other before hand, limiting the number of applications that can use this. Also, OAuth requires involvement of the web application servers and cannot support Javascript-only applications with no backend. Google provides external access to user data through APIs utilizing the Google Data Protocol [13] in a similar manner. BSTORE simplifies data sharing by avoiding the need for all applications to know about each other ahead of time, and does not require server involvement for data sharing.

Menagerie [10] allows sharing of a user's personal data between multiple web applications and provides standardized hierarchical naming and capability-based protection. However, like OAuth, it requires backend servers to communicate with each other. Also, unlike BSTORE's tag-based mechanisms, Menagerie's file systems don't support per-application file system organization and delegation of access rights based on file properties.

Cloud computing and storage services such as Amazon S3 [4], and Nirvanix [24] provide web application developers with the option of renting storage and on-demand computing. However, developers still need to bear the costs of renting the server capacity, and make data management decisions on behalf of users. BSTORE allows users to control their own data, such as by encrypting, mirroring, or backing it up.

Cross-origin resource sharing [35] provides a mechanism for client-side cross-origin requests. This allows for pure client-side apps to access data from other websites which will in turn implement cross domain authentication and access control. However, using this mechanism alone does not provide a single namespace for all user data, and does not provide an access control and delegation mechanism such as that provided by BSTORE.

Hsu and Chen [17] describe the design of a secure file storage service for Web 2.0 applications. While the motivation for their work is similar to ours, there are a number of limitations of their work that BSTORE's design addresses. First, their file system doesn't support a unified namespace and there are no mountpoints. It cannot support delegation, encryption, or checkpointing, and doesn't support versioning, which means that applications sharing data can run into problems. Finally, BSTORE's tags allow applications to annotate each others' files, and to delegate specific access, without requiring write privileges, something that is not possible in Hsu and Chen's system.

9 CONCLUSION

This paper presented BSTORE, a framework for separating application code from data storage in client-side web applications. BSTORE's architecture consists of three components: *file systems*, which export a storage API for accessing user data, a *file system manager*, which implements the user's namespace from a collection of file systems, and enforces access control, and *applications*, which access user data through the file system manager. A key idea in BSTORE is the use of tags on files. Tags allow applications to organize their data in different ways. An application also uses tags to designate the precise files it wants to delegate rights for to other applications, even if it cannot write or otherwise modify those files itself. The BSTORE file system manager interface is egalitarian, allowing any application to specify delegation rules or mount new file systems, in hopes of avoiding the need for applications to supply their own file system manager.

A prototype of BSTORE is implemented in pure Javascript, which runs on both the Firefox and Chrome browsers. We ported three existing applications to run on BSTORE, which required minimal source code modifications, and wrote one new application. We also implemented three file systems, including ones that transparently provide encryption or checkpointing capability using an existing file system for storage. Finally, our prototype achieves reasonable performance when accessing data over a typical home network connection.

ACKNOWLEDGMENTS

We thank our shepherd, John Ousterhout, as well as Jon Howell, Adam Marcus, Neha Narula, and the anonymous reviewers for providing feedback that helped improve this paper. This work was supported by Quanta Computer and by Google.

REFERENCES

[1] A. M. Vahdat, P. C. Eastham, and T. E. Anderson. WebFS: A global cache coherent file system. Technical report, UC Berkeley, 1996.

[2] S. Aaronson. Off the grid. http://scottaaronson.com/blog/?p=428.

[3] Adobe. Adobe Flash. http://www.adobe.com/flashplatform.

[4] Amazon. Amazon simple storage service. http://aws.amazon.com/s3/.

[5] E. Benson, A. Marcus, D. Karger, and S. Madden. Sync Kit: A persistent client-side database caching toolkit for data intensive websites. In *Proceedings of the World Wide Web Conference*, 2010.

[6] B. Callaghan. WebNFS Client Specification. RFC 2054 (Informational), 1996.

[7] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, San Diego, CA, December 2008.

[8] P. Dourish, W. K. Edwards, A. LaMarca, and M. Salisbury. Using properties for uniform interaction in the presto document system. In *Proceedings of the ACM Symposium on User Interface Software and Technology (USIT)*. ACM, 1999.

[9] FotoFlexer. Fotoflexer: The world's most advanced online image editor. http://www.fotoflexer.com.

[10] R. Geambasu, C. Cheung, A. Moshchuk, S. D. Gribble, and H. M. Levy. The organization and sharing of web-service objects with menagerie. In *Proceedings of the World Wide Web Conference (WWW)*, 2008.

[11] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. toole. Semantic file systems. In *13thACM Symposium on Operating Systems Principles*, pages 16–25. ACM, 1991.

[12] Google. Gears: Improving your browser. http://gears.google.com/.

[13] Google data protocol. http://code.google.com/apis/gdata/.

[14] Google docs. http://docs.google.com/.

[15] Google. Picasa web albums. http://picasaweb.google.com.

[16] J. H. Howar, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[17] F. Hsu and H. Chen. Secure file system services for web 2.0 applications. In *Proceedings of the ACM Cloud Computing Security Workshop*, Chicago, IL, November 2009.

[18] jsvi – javascript vi. http://gpl.internetconnection.net/vi/.

[19] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 91–106, San Francisco, CA, December 2004.

[20] Microsoft. Silverlight. http://silverlight.net/.

[21] R. Miller. Ma.gnolia data is gone for good. http://www.datacenterknowledge.com/archives/2009/02/19/magnolia-data-is-gone-for-good/.

[22] MIT Haystack Group. NB 2.0. http://nb.csail.mit.edu/.

[23] A. Modine. Web startups crumble under amazon s3 outage. http://www.theregister.co.uk/2008/02/15/amazon_s3_outage_feb_2008/.

[24] Nirvanix. http://www.nirvanix.com/.

[25] Oauth. http://oauth.net.

[26] Photofunia. http://www.photofunia.com.

[27] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in plan 9. *ACM SIGOPS Operating System Review*, 27(2):72–76, 1993.

[28] Pixastic – online javascript photo editor. http://www.pixastic.com.

[29] R. Sandberg, D. Goldberg, S. Kleinman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the Summer 1986 USENIX Conference*, 1985.

[30] S. Shankland and T. Krazit. Widespread google outages rattle users. http://news.cnet.com/widespread-google-outages-rattle-users/.

[31] Shutterfly. http://www.shutterfly.com.

[32] Slideroll. Slideroll online slideshows. http://www.slideroll.com.

[33] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in javascript. In *Proceedings of the Annual Computer Security Applications Conference*, 2009.

[34] TrimSpreadsheet. http://code.google.com/p/trimpath/wiki/TrimSpreadsheet.

[35] W3C. Cross-origin resource sharing: Editor draft. http://dev.w3.org/2006/waf/access-control/, December 2009.

[36] W3C. HTML 5 editor's draft. http://dev.w3.org/html5/spec/, January 2010.

[37] Yahoo. flickr. http://flickr.com.

[38] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

AjaxTracker: Active Measurement System for High-Fidelity Characterization of AJAX Applications

Myungjin Lee, Ramana Rao Kompella, Sumeet Singh[†]
Purdue University, [†]*Cisco Systems*

Abstract

Cloud-based Web applications powered by new technologies such as Asynchronous Javascript and XML (Ajax) place a significant burden on network operators and enterprises to effectively manage traffic. Despite increase of their popularity, we have little understanding of characteristics of these cloud applications. Part of the problem is that there exists no systematic way to generate their workloads, observe their network behavior today and keep track of the changing trends of these applications. This paper focuses on addressing these issues by developing a tool, called AJAXTRACKER, that automatically mimics a human interaction with a cloud application and collects associated network traces. These traces can further be post-processed to understand various characteristics of these applications and those characteristics can be fed into a classifier to identify new traffic for a particular application in a passive trace. The tool also can be used by service providers to automatically generate relevant workloads to monitor and test specific applications.

1 Introduction

The promise of cloud computing is fueling the migration of several traditional enterprise desktop applications such as email and office applications (*e.g.*, spreadsheets, presentations, and word processors) to the cloud. The key technology that is powering this transition of the browser into a full-fledged cloud computing platform is Asynchronous Javascript and XML (Ajax) [11]. Ajax allows application developers to provide users with very similar look-and-feel as their desktop counterparts, making the transition to the cloud significantly easier.

The modern cloud applications based on Ajax behave differently from the traditional Web applications that involve users clicking on a particular URL to pull objects from the Web server. Ajax-based cloud applications, however, may involve each mouse movement leading to a transaction between the client and the server. Further, these transactions may potentially involve an exchange of one or many messages *asynchronously* and sometimes, even *autonomously* without user involvement (*e.g.*, auto-save feature in email).

While there are a large number of studies that characterize (*e.g.*, [7]) and model (*e.g.*, [6]) classical Web traffic, we have very limited understanding of the network-level behavior of these Ajax-based applications. A comprehensive study of these applications is critical due to two reasons. First, enterprises are increasingly relying on cloud applications with Ajax as a core technology. As these services can potentially affect the employee productivity, it becomes crucial for operators (both enterprise as well as ISP) to constantly monitor the performance of these applications. Second, network operators need to project how application popularity changes may potentially affect network traffic growth, perform 'what-if' analyses, monitor for new threats and security vulnerabilities that may affect their network.

A standard approach (*e.g.*, [21]) for characterizing these applications is to collect a trace in the middle of the network and observe the network characteristics of these applications in the wild. Due to the reliance on passive network traces, however, this approach has two main limitations. The first limitation is that there is no easy way to isolate the network-traffic produced by individual operations (such as Zoom-in operation in Maps application, or drag-and-drop on Mail application), which may be important to understand which actions are most expensive or how network traffic may change if relative usage of different operations change in future. Second, there is no easy way to understand how network conditions affect the characteristics of these applications. This is since, at the middle of the network, the router only observes aggregate traffic comprising of clients from heterogeneous network environments. For some uses, aggregate view may actually be sufficient, but for certain management tasks such as, say, conducting what-if analyses, this aggregate view is *not* sufficient.

To address these challenges, in this paper, we propose an active measurement system for high-fidelity characterization of modern cloud applications, particularly those that are based on Ajax. Our approach comprises of two key ideas: First, we observe that running an application on an end-host with no other application can allow capturing *all* the packets associated with that application session with zero false positives or false negatives. Second, by controlling the network conditions and what operations we inject in isolation, we can get a deeper understanding of these applications in addition to predicting their impact on the network.

Our system called AJAXTRACKER, works by modeling high-level interaction operations (*e.g.*, drag-and-drop) on a particular Ajax-based cloud application and by injecting these operations through a browser to generate (and subsequently capture) relevant network activity between the client and the server. In addition, it incorporates mechanisms to generate representative client appli-

cation sessions by specifying either an explicit or model-driven sequence of atomic operations. The model that governs the sequence of operations may, for instance, control the distribution of time between two atomic operations. It also utilizes a traffic shaper to control network latencies and bandwidth to study the effects of end-host network conditions on the application performance. We have designed and implemented a prototype of this tool that is available for download[1].

Thus, our paper makes the following contributions: 1) Our first contribution in this paper is the design of AJAXTRACKER that provides a mechanism to automatically interact with Ajax-powered cloud services. We discuss the details of the tool in Section 3. 2) We present a characterization study of popular Ajax-based applications under different bandwidth conditions and different round-trip times. Section 4.2 discusses these results in more detail. 3) Our final contribution is a characterization of network activity generated by popular applications on a per-operation basis. To the best of our knowledge, our study is the first to consider the network activity of individual atomic operations in Ajax applications. We discuss these details in Section 4.3.

While the primary purpose of the tool is to characterize Ajax-based cloud applications, we believe that AJAXTRACKER will prove useful in many other scenarios. For instance, it could provide interference-free access to the ground-truth required to train classifiers in several statistical traffic classification systems [18, 15, 20]. Its fine-grained analysis capabilities will allow network operators to model, predict traffic characteristics and growth, conduct 'what-if' analyses and so on.

2 Background and motivation

Today, many cloud application providers are increasingly focusing on enriching the user interface to make these services resemble desktop look-and-feel as much as possible. Perhaps, the most prominent ones among these are Mail, Documents, and Maps[2] applications, which are now offered by companies such as Google, Microsoft and Yahoo among others.

In the traditional Web, the navigation model of a Web session is quite straightforward: A user first clicks on a URL, then, after the page is rendered, he thinks for some time and requests another object. This process continues until the user is done. On the other hand, the navigation model of modern Ajax web sessions is quite different: A user can click on a URL, drag-and-drop on the screen, zoom in or zoom out (if it is a maps application) using the mouse scroll button among several other such features. In addition, the Javascript engine on the client side can request objects asynchronously and autonomously without the user ever requesting one. For example, when using Ajax-based email services, the browser automatically tries to save data when a user is composing email.

[1] http://www.cs.purdue.edu/synlab/ajaxtracker

[2] While Maps application is not strictly an enterprise cloud application, it exports a rich set of Ajax features making it an interesting Ajax application to characterize.

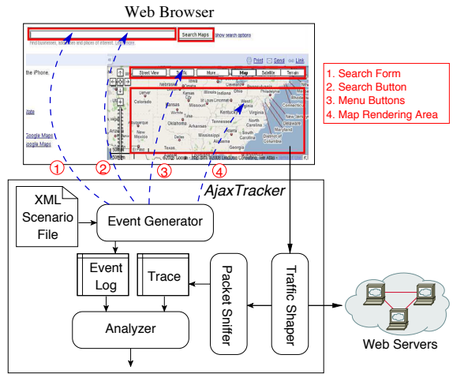


Figure 1: Structure of AJAXTRACKER.

Given their importance in the years to come, as increasingly more applications migrate to the cloud, it is important to characterize these applications and understand their impact on the network. Due to the forementioned shortcomings of passive approaches, we take an *active measurement* approach for characterizing these applications in this paper. The basic idea of our approach is to perform application measurement and characterization at the end-host. By ensuring that there exists only one application session at any given time, we can collect packet traces that are unique for that particular session, even if the session itself consists of connections to several servers or multiple connections to the same server. We, however, need a way to generate user application sessions in an *automated* fashion that can help repeatedly and automatically generate these sessions under different network conditions.

Unfortunately, there exist few tools that can interact with applications in an *automated* manner. Web crawlers lack the sophistication required on the client-side to generate Ajax-behavior. Traditional crawlers have no built-in mechanisms to interact with the interactive controls (like drag-and-drop), which require mouse or keyboard input, and are fairly common in these new applications. In the next section, we describe the design of AJAXTRACKER to overcome these limitations.

3 Design of AJAXTRACKER

The main components of AJAXTRACKER include an event generator, a Web browser, a packet sniffer, and a traffic shaper as shown in Figure 1. The event generator forms the bulk of the tool. It produces a sequence of mouse and/or keyboard events that simulate a human navigating a cloud-based Web site based on a configured scenario file written in XML. These events are then input to an off-the-shelf Web browser (*e.g.*, Mozilla Firefox) that then executes them in the order it receives individual operations. Note that AJAXTRACKER itself is agnostic to the choice of the Web browser and can work with any browser. Given the goal is to collect representative traces of a client session, AJAXTRACKER employs a packet sniffer (*e.g.*, tcpdump [2]) that captures the packets on the client machine. These packets can then be examined to obtain specific characteristics of the simulated client session.

In addition to the basic components described above, AJAXTRACKER also makes use of a traffic shaper that can be configured to simulate specific bandwidth and delay conditions under which the corresponding cloud application sessions are simulated. This feature enables the tool to obtain many client sessions, each possibly under different network conditions to simulate the real-world settings where each user is exposed to different set of network conditions. Finally, the tool has the ability to perform causality analysis between operations (obtained from the browser's event log) on a cloud Web site and the corresponding network activity captured from the packet sniffer's trace.

AJAXTRACKER works by first configuring the traffic shaper with delay and bandwidth parameters. Next, it runs tcpdump, then launches the Web browser with the corresponding url that is indicative of the cloud application that we wish to simulate. The event generator is then executed until all specified events have been processed. We describe individual components in detail next.

3.1 Scenario file

A scenario file is intended to model a sequence of user operations that need to be executed to emulate a user session. For example, a user session could consist of entering a location, say New York, in the search tool bar in the Maps application and clicking on the submit button. The next action, once the page is rendered, could be to zoom in at a particular location within the map page. After a certain duration, the session could consist of dragging the screen to move to a different location. The scenario file is intended to allow specifying these sequence of operations. In addition to static scenarios, the scenario file will enable the tool to explore random navigation scenarios that for example, execute events in a random order or navigate to random locations within the browser. Random scenarios help the tool users to explore a richer set of navigational patterns that are too tedious to specify using the static mechanisms one-by-one.

The scenario file is composed of mainly three categories—events, objects and actions. There are three broad categories of events: pre-processing events (specified with the tag `PRE_EVENTS`), main events (`MAIN_EVENTS` tag), and post-processing events (`POST_EVENTS` tag). Events in pre- and post-processing category are sequentially executed exactly once—before and after the main events are simulated. Main events can be specified to be executed in either static or random order using the values 'static' and 'random' within the `TYPE` attribute in `MAIN_EVENTS` tag.

Model	PDF	Attributes
static	-	PERIOD
uniform	$1/d$	PERIOD
exponential	$\mu e^{-\mu x}$	EXP_MU
weibull	$\frac{b}{a}(\frac{x}{a})b-1e^{-(x/a)^b}$	W_A, W_B
pareto	$\alpha k^{\alpha} x^{-(\alpha+1)}$	P_A, P_K

Table 1: Inter-operation time distribution models.

Each event (within the `EVENT` tag) enumerates a list of objects, with each object described by an identifier, action and a pause period (`PAUSE_TYPE` attribute) that specifies the amount of time the event generator should wait after executing the specified action on the object. The time specification can be either a constant or could be drawn from a distribution (such as Pareto or exponential). The pause period specification helps model human think time in a sequence of operations. Table 1 shows the five different distributions for pause between operations that are currently supported by our tool. In the `EVENT` element, if `LOG` attribute is specified, AJAXTRACKER records event logs into a file that can be later used for correlating with packet logs for causality analysis.

Objects are specified within the broader `OBJECTS` tag and individually enclosed within the `OBJECT` tag. Actions are associated with individual objects. Each object defines its position or area and possible actions. Depending on the type of the object, specific tags such as `TEXT` or `READ_FILE` are specified that are appropriate for those objects. For example, to fill submission form from a set of predefined texts, the input that needs to be supplied is specified using the `READ_FILE` object to indicate a file which contains a set of predefined texts. An input text from the file is fetched on a line-by-line basis by the object. The value encapsulated by `ACTIONS` tag defines supported actions (a list of support actions is depicted in Table 2) and the position where the action needs to be performed. If position values (*e.g.*, `X`, `S_X`) in `ACTION` tag are not defined, the values of `POSITION` or `AREA` tags are used.

Example. A small snippet of a sample scenario file for Google Maps is shown in Figure 2. Note that this example is not meant to exhaustively capture all the capabilities of the tool. The scenario forces AJAXTRACKER

Action	Meaning
left_click	click with left mouse button
right_click	click with right mouse button
select	pushing Ctrl+A
delete	pushing Backspace
copy	pushing Ctrl+C
cut	pushing Ctrl+X
paste	pushing Ctrl+V
drag	move object holding mouse left button
wheel_up	scroll up an object
wheel_down	scroll down an object

Table 2: Supported actions.

to work as follows: Events are executed in the order of 'navigate_map' and 'close_window'. If there exist events in PRE_EVENTS, these events are executed first. Then, the tool statically executes 'navigate_map' event twice (as specified in the EXEC_COUNT attribute). If there are more than one event listed, the tool will sequentially execute each event twice. For each instance of 'navigate_map' event, operations specified between lines 13 to 22 of the scenario file are executed. We only allow the tool to execute the operations serially because we defined an event as a series of operations to accomplish a task. Thus, in this event, a query string retrieved from list.site file in line 31 is put into 'search_form' object, and the tool takes inter-operation time of 1 second. Then, 'search_button' object is clicked and the tool lets another 1 second elapsed. After that, the tool generates 'drag mouse' window event which is followed by inter-operation time probabilistically selected by Pareto distribution. In addition, the tool records a log in the file "drag map" specified as part of the OBJ_REF description. For the tool to identify the coordinate of object or actions to be taken, the tool searches objects which are defined from lines 24 to 42 by using object ID whenever it executes an object.

Note that the scenario file can describe events at both semantic level or in the form of coordinates; our system allows both types of input. The choice of one over the other depends on the particular event that needs to be described. Drag-and-drop actions on maps applications, for instance, are better represented using coordinates, while actions that involve submitting forms (*e.g.*, save file) are better represented at the semantic level.

A scenario is presented in a hierarchical fashion. One can first list events to generate and flexibly compose events with one or more objects and actions against the objects. Multiple actions can be defined within an object which can be reused in several events. While the specification allows users to build various scenarios covering many user interactions, it is hard to cover all user actions due to complexity of user actions and coordinate-based specification of an object. Note that although the specification appears complicated, we have found in our experience that coding the scenario file does not take too long. In our experience with 14 scenario files, the longest was 454 lines that took us less than an hour to specify. Once the scenario file is specified, the tool itself performs completely automatically and can work repeatedly and continuously. Thus, the cost of specifying the scenario file is amortized over the duration over which the scenario is monitored. As part of our future work, we are working on automating the generation of the scenario file by recording and replaying user/client activities passively using a browser plugin.

```
 1: <SCENARIO>
 2:  <NAME> Google Maps </NAME>
 3:
 4:  <PRE_EVENTS>
 5:  </PRE_EVENTS>
 6:  <MAIN_EVENTS TYPE="static" EXEC_COUNT="2">
 7:   <EVT_REF IDREF="navigate_map" />
 8:  </MAIN_EVENTS>
 9:  <POST_EVENTS>
10:   <EVT_REF IDREF="close_window" />
11:  </POST_EVENTS>
12:
13:  <EVENT ID="navigate_map">
14:   <OBJ_REF IDREF="search_form" ACTION="paste"
15:    PAUSE_TYPE="static" PERIOD="1" />
16:   <OBJ_REF IDREF="search_button" ACTION="click"
17:    PAUSE_TYPE="static" PERIOD="1" />
18:   <OBJ_REF IDREF="map_area" ACTION="drag"
19:    LOG="drag map"
20:    PAUSE_TYPE="pareto"
21:    PARETO_K="1" PARETO_A="1.5" />
22:  </EVENT>
23:
24:  <OBJECTS>
25:   <OBJECT ID="search_form">
26:    <POSITION X="359" Y="225" />
27:    <ACTIONS>
28:     <ACTION ID="paste">paste</ACTION>
29:     <ACTION ID="click">left_click</ACTION>
30:    </ACTIONS>
31:    <READ_FILE>/ajax/env/list.site</READ_FILE>
32:   </OBJECT>
33:
34:   <OBJECT ID="map_area">
35:    <AREA LEFT="500" TOP="333"
36:          RIGHT="1241" BOTTOM="941" />
37:    <ACTIONS>
38:     <ACTION ID="drag" S_X="600" S_Y="400"
39:      E_X="900" E_Y="900" COUNT="1">drag</ACTION>
40:    </ACTIONS>
41:   </OBJECT>
42:  </OBJECTS>
43: </SCENARIO>
```

Figure 2: Example scenario file for Google maps.

3.2 Event generator

Given a scenario file, the event generator first parses it and builds data structures for elements listed in the scenario file. Then, the event generator registers a callback function called run_scenario() for a timer. The callback function plays a core role in generating mouse and keyboard events. Every time the function is called, it checks if the whole events in the scenario file were executed. If there is any event left, the function generates mouse or keyboard events accordingly. As discussed before, the event generator supports two basic navigation modes—static and random. In the static navigation mode, AJAXTRACKER generates the sequence of events exactly in the order specified in the scenario file. In the random mode, it provides different levels of randomness. First, the event generator can randomly select the order of events in main event class by assigning TYPE attribute as 'random' which implies uniform distribution. Second, in an event, it can adjust inter-operation time with four different probabilistic distributions if PAUSE_TYPE is defined as one of values (except 'static') listed in Table 1. Other distributions can optionally be added. Third, action can be executed randomly. For instance, if TYPE attribute in ACTION element is set 'random', the tool ignores position values (*e.g.*, X, S_X), and executes the action by uniformly selecting position or direction (in case of drag) within values of AREA element and the number of clicking objects within the value of COUNT attribute.

In case of action name called 'paste', the tool can randomly select one from text list which it manages and pastes it in the input form of a Web site. Moreover, by simply changing the number of main events and reorganizing the execution procedure of an event, we can let the event generator work completely differently. Thus, through this way of providing randomness, the event generator strives to generate random but guided navigation scenarios to simulate a larger set of client sessions.

We implemented the event generator as a command-line program with 3500+ lines developed using C++, GTK+, the X library and Xerces-C++ parser [4].

3.3 Traffic shaper

Often, it is important to study the characteristics of these applications under different network conditions. Given that the tool works on an isolated end-host, the range of network conditions it can support is quite dependent on the capacity of the bottleneck link at the end host. For example, if the tool is being used by an application service provider (ASP), typically, the ASP is going to use it in a local area network (close to the Web server) where the network conditions are not as constrained as clients connected via DSL or Cable Modem or Dial-up or some such ways to access the Web service. The traffic shaper, in such cases, provides a way to study the application performance by artificially constraining the bandwidth as well as increasing the round-trip times of the network.

The traffic shaper mainly implements bandwidth throttling and delay increases, and does not factor in packet drop rates. Packet losses are not directly considered in our tool at the moment since available bottleneck bandwidth, to some extent, forces packets to be lost as soon as the bottleneck capacity is reached. We can, however, augment the tool with arbitrary loss fairly easily. We used an open-source software router called Click [16] in our tool for implementing the traffic shaping functionality. We ran Click as a kernel module in our system. Note that any software that provides the required traffic shaping functionality would work equally well.

Time	Event	#click	S_X	S_Y	E_X	E_Y
1224694605.059651	click search button	1	620	220	620	220
1224694975.651213	zoom out	2	940	832	940	832
1224695045.303020	drag map	1	1128	537	1021	470
1224695062.083703	zoom in	10	824	554	824	554
1224695175.203356	drag map	1	858	693	867	411

Figure 3: Example event log snapshot generated.

3.4 Packet capture

To characterize the network-level performance of an application session, it is important to capture the packet-level traces that correspond to the session. One can potentially instrument the browser to obtain higher-level characteristics, such as URLs accessed and so on. However, our goal is to characterize network activity; thus, we employ off-the-shelf packet sniffer such as tcpdump to capture packet traces during the entire session. Since AJAXTRACKER aims to characterize mainly cloud Web applications, it filters out non-TCP non-Web traffic (i.e., packets that do have port 80 in either the source or destination port fields). AJAXTRACKER considers all captured 5-tuple flows of *<src, dst, src_port, dst_port, protocol>* to form the entire network-activity corresponding to a given session. We do not need to perform TCP flow-reassembly as we are mainly interested in the packet-level dynamics of these sessions.

Advertisement data, however, which are parts of a Web site but are not Ajax-based cloud application related, can also be included in the trace file. If we do not wish to analyze the non-Ajax content, depending on the application, we apply a simple keyword filter which is similar to ones used by Schneider *et al.* [21] to isolate Ajax-based flows alone. We find related flows whose HTTP request message contains keywords of interest and retrieve bidirectional flow data.

3.5 Causality analysis

Our tool generates traces as well as logs about which operations were performed with their timing information as shown in Figure 3. While information in the first two columns is mainly used for causality correlation, other five columns provide auxiliary information. The third column denotes the number of mouse clicks. The fourth and fifth columns denote the screen coordinates where an event begins to occur and the last two columns represent the screen coordinates where it ends. We do not at the moment use this auxiliary information however, and focus mainly on the first two columns. Based on these two pieces of information, i.e., timestamp and event name,

we can reconstruct the causality relationship between operations in the Web browser and the corresponding network activity by correlating the network activity using the timestamp.

Using this causality, we can isolate the network effects of individual operations, such as finding what Web servers are contacted for a given operation, the number of connections that are open or the number of requests generated by each operation. Such causality analysis helps when anomalies are found in different applications as one can isolate the effects of individual operations that are responsible for the anomalies. In addition, the causality analysis helps predict how the application traffic is going to look like, when we change the popularity distributions of different operations in a given sequence. For example, if users use the zoom features much more than drag-and-drops, we can study the underlying characteristics of such situations.

Note that while this timing-based causality works well for simple applications we considered in this paper such as Gmail and maps, it may not work easily for all events and applications. For instance, if we investigate 'autosave' event of Google Docs, we need to know when the event is triggered while a user composes, which may not be simple to know unless the browser is instrumented appropriately. Modifying the browser, however, introduces an additional degree of complexity that we tried to avoid in our system design.

3.6 Limitations

As with perhaps any other tool, AJAXTRACKER also has some limitations. First, since it works depending on the layout of user interface, significant changes to the user interface by the application provider may cause the tool to not operate as intended. Though this limitation may seem serious, observations made by the tool over a period of weeks shows considerable consistency in the results of the static navigation mode, barring a few user interface changes that were easy to modify in the scenario file. Second, in our current setup, we specify the mouse clicks in the form of coordinates, which assumes that we have access to the screen resolution. If the setup needs to run a separate platform, the scenario files need to be readjusted. One way to address this issue is to specify them relative to the screen size; we did not implement this feature yet and is part of our future work. Third, because the operation of our tool depends on a specified scenario file, the generated workloads cannot cover all possible user space. Instead, we try to configure scenario files with functions which are most likely to be used by users in each explored application. Currently, while we program these scenarios ourselves, we are also investigating representative client session models and deploying them into our tool. Note that these models are orthogonal to our tool design itself. Third, given the nature of the traffic shaper, we cannot emulate all types of network conditions; we can either reduce the bandwidth or increase the RTT in comparison with the actual network conditions at the location where the tool is deployed.

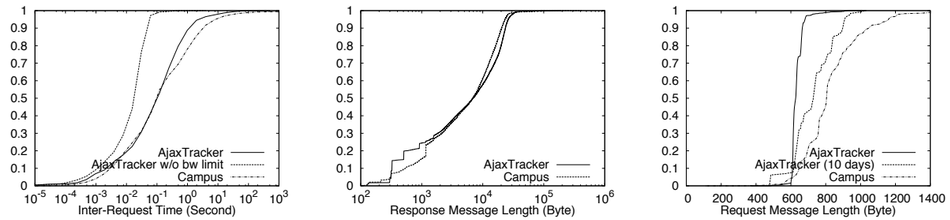
4 Evaluation

In this section, we present our measurement results obtained using the tool on real Ajax applications. We categorize our results into three main parts. First, we demonstrate that our tool produces representative traces by comparing our results with a passive campus trace. Second, we perform macroscopic characterization of full application sessions generated using our tool. We also show how Ajax application traffic characteristics change with different network conditions. Third, we show the characterization of individual operations such as 'click' and 'drag-drop' in two canonical Ajax applications—Google Maps and Mail—with the help of the causality analysis component of our tool.

4.1 Comparison with a real trace

The representativeness of our tool is completely dependent on the expressiveness of our tool and the scenario files specified. In order to demonstrate that the scenarios we have specified in our tool are representative, we show comparisons with a real passive trace. For the purposes of this experiment, we have obtained a real trace of Google Maps traffic from a campus switch of Purdue university. There are approximately 300 machines connected to the switch and users are mainly campus students. The Google Maps trace we collected represents 24 hours worth of client activity over which we observed about 182 unique clients totaling about 13,200 connections. While our campus trace is not representative of all settings, our trace is representative of network infrastructure environment that corresponds to typical enterprise networks, and hence, the use of Google Maps in this environment is arguably similar to that of any other organization's use of Google Maps.

Figure 4 shows the comparison results in terms of inter-request time (IRT), response and request message length (QML). IRT is an important metric because it can show or measure how proactive the application is. If IRTs are much smaller than RTT (if we measure RTT), it implies that the application is more proactive and relies on parallel connections for fast retrieval of traffic. For IRT, we calculated intervals between request messages sent to the same server through multiple flows, instead of calculating intervals between every request messages regardless of the destination. We believe that this is a reasonable approach to calculate IRT because ignoring the destination may lead to a much higher request frequency for Ajax application's traffic, but it is misleading



(a) CDF of inter-request time (b) CDF of response message length (c) CDF of request message length

Figure 4: Comparison between AJAXTRACKER's trace and Campus trace of Google Maps.

as some of the traffic will be destined to multiple destinations (typically within the same cluster).

First, we plot the comparison between AJAXTRACKER (the curve that says AJAXTRACKER without bandwidth limit) and campus trace in terms of their IRT distributions in Figure 4(a). When we compare the two, we can observe clearly that they are not similar. This is because, IRT distributions are easily affected by the available network bandwidth. Since the clients in the trace may potentially have a different throughput constraints from the machine we use AJAXTRACKER from, we need some calibration to match the trace. We first analyzed the average instantaneous throughput of Maps with a bin of size 1 second for every host in the trace. We excluded the case where there is no traffic in considering instantaneous throughput. The average instantaneous throughput was about 580Kbps. Specifically, in cumulative distribution, 82% of instantaneous throughput were less than 1Mbps, 16% were between 1-5Mbps, and 2% were between 5-20Mbps.

Based on the above observation, we ran our tool with different network bandwidth constraints to adjust available network bandwidth artificially. To simulate the distribution of instantaneous throughput, we differentiated the proportion of each trace generated by AJAXTRACKER under different network bandwidth conditions based on the distribution we have observed in the campus trace. Specifically, traces generated by our tool has fixed data rate configured by Click: 500Kbps, 1Mbps, 5Mbps, and 10 Mbps. On the other hand, campus trace has continuous distribution from around 500Kbps to 20Mbps. We envision that there are only a few different quantized access bandwidths for different clients within a network. By empirically finding these numbers, one can run the tool and mix different traces with different weights. Thus, we empirically gave 10% weight to a trace by 500Kbps constraint, 72% weight to a trace by 1Mbps constraint, 16% weight to a trace by 5 Mbps, and 2% weight to a trace by 10Mbps constraint, and conducted weighted IRT simulation.

We found that IRT distribution generated by AJAXTRACKER is quite close to the IRT distribution of the campus trace as shown in Figure 4(a). There are still a few minor discrepancies; at around 0.6-2 seconds, there is 10% discrepancy between two curves. IRTs larger than one second are typically because of human think time, as has been described by Schneider *et al.* in [21]. Thus, we believe this area of discrepancy that represents human think time exists because of the discrepancies between our scenario models that model the inter-operation duration and real user's behavior. If needed, therefore, we can carefully tune the scenarios to easily match the campus trace. Such tuning may or may not be necessary depending on the particular use of the tool; the more important aspect is that the tool allows such calibration.

The distribution of response messages (shown in Figure 4(b)) are quite similar between AJAXTRACKER and the campus trace for the most part. The big difference ranging from about 300 to 1,000 bytes is related to whether basic components (*i.e.*, icons, thumbnail images, etc.) that constitute Maps application are already cached or not. Because we ran AJAXTRACKER ensuring that the browser has no cached data, the fraction of that area in AJAXTRACKER's distribution is larger than that of campus.

While we have not conducted extensive experiments to study the impact of cached data in this paper, we note that storage size generally for cached data is limited and stale data is typically removed from cache storage. Thus, in general, the fact that Web browser caches data does not mean that it cached data for a particular application of interest. In addition, Ajax applications often consist of two portions: Application source (e.g., javascript code) that is static and needs to be downloaded only once when a user accesses the application for the first time and, application data that is typically more dynamic and diverse than application source. Thus, caching typically impacts only the application source download and not so much the data portion of the session.

Finally, in Figure 4(c), we can observe that QML dis-

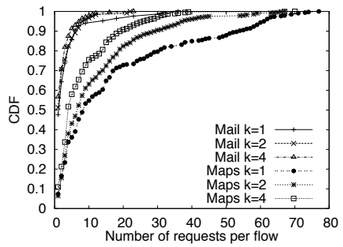


Figure 5: CDF of number of requests per flow. k is the scale parameter. The shape parameter α is fixed at 1.5.

tribution is significantly different from what is observed using AJAXTRACKER and the campus trace. This discrepancy is because QML can vary quite a bit because of the variation of cookie length. We collected 10 days of traces to find if there are changes in the length of request messages, and their QMLs showed diversity due to different cookie length. When we joined together those traces, we found that the distribution comes close to the curve of campus in Figure 4(c). There are also browser differences in campus trace can cause different size of requests for the same objects. When we adjust for these differences, the trends of the distribution by AJAXTRACKER and campus will be similar. The bigger takeaway from these experiments is that the results obtained using our tool can be made to be easily consistent with those observed in real life. These experiments also suggest an interesting side-application of our tool in reverse engineering the traffic shaping policy at the monitoring network. For the rest of the experiments, we do not put any restriction on the bandwidth, since there is nothing particularly fundamental about any given aggregate bandwidth and can vary depending on the situation.

4.2 Characterization of full sessions

In this section, we characterize overall application sessions. We first describe some flow-level characteristics of Ajax applications. Next, we discuss our results by simulating different network conditions at the end-host.

Number of requests per flow. Since persistent connections are supported by modern Web browsers, the number of requests per flow depends on the inter-operation duration. If there is no network activity, the flows are usually terminated. To understand this dependency, we varied the inter-operation duration as a Pareto distribution, which has been used before to model human think time for traditional Web traffic [6]. Given that we do not yet have representative parameters for Ajax applications, we chose to experiment with those reported in [6] for regular Web traffic and some new ones. For both Google Maps and Mail, we chose the Pareto scale parameter k to be 1, 2 and 4 and fixed the shape parameter α as 1.5 (as suggested in [6]).

Regardless of values of k, we can observe clear difference between Maps and Mail in Figure 5. While Mail has at most 31 requests per flow at $k = 1$, Maps generates 67 requests per flow at $k = 4$. In the head of distribution, while about 50% of all Mail's flows have only one request, only 10% of Maps' flows have one request. Interestingly, the top 93% exhibit similar trends for Mail, after which the $k = 1$ curve exhibits higher number of requests per flow. This phenomenon is expected since, smaller values of k imply better re-use of connections, which in turn leads to larger number of requests per flow.

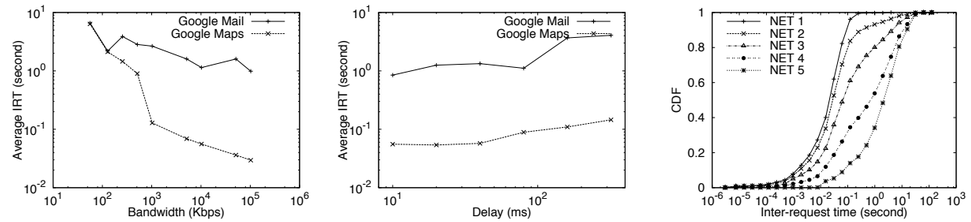
In Maps, on the other hand, the number of requests per flow exhibits big difference depending on values of k. The reason for the difference could be because of the lack of a mechanism in Maps similar to asynchronous updates in Mail. While Maps prefetches map tile images, the connections are closed faster than Mail's connections.

Our analysis on number of requests per flow uses flow-level granularity while Schneider *et al*. report session-level results in [21]. Despite this difference, our results roughly match their observations, in that Maps generates more requests than Mail.

Effects of different network conditions. To understand how these applications behave under different network conditions, we let AJAXTRACKER run on emulated network environments using its traffic shaping functionality. We conducted two experiments on Maps and Mail: The first is a bandwidth test where the traffic shaper throttles link capacity. The second is a delay test where it injects additional delays in the link to artificially increase the RTT. In our configuration, the set of bandwidths we have chosen include {56Kbps, 128Kbps, 256Kbps, 512Kbps, 1Mbps, 5Mbps, 10Mbps, 50Mbps, 100Mbps}. For delay, we chose values from 10ms to 320ms in multiples of 2 (*i.e.*, 10ms, 20ms, 40ms, etc.).

While our framework allows adding delay to both outbound as well as inbound packets, we added the simulation delay only to the inbound packets. This is because, it is the RTT that typically dictates the connection characteristics and hence it suffices to adding it in either of the directions. The inter-operation times were statically configured in the scenario files.

Figure 6 shows how IRT is affected according to the change of bandwidth and delay, respectively. We used the causality between operation and network activity from log information and traces in order to remove large IRTs which come from the the interval between the time when last request message of previous operation was seen and the time when the first request message of current operation is seen because these large IRTs affected by inter-operation time (which is decided by a user) restrain our understanding about applications' behavior.



(a) Bandwidth with no added delay (b) Delay with 50 Mbps (c) Simulated IRT

Figure 6: Average IRT variation for different network parameters.

From Figure 6(a), we can observe that, as network bandwidth increases, the average IRT of Maps decreases fast, but the extent to which Mail's IRT distribution decreases is small. The graph shows that Maps fetches contents more aggressively than Mail does. On the contrary, Figure 6(b) shows that IRTs of both applications are relatively less affected by the increase of delay. The figure indicates that IRT feature is more sensitive to bandwidth rather than delay. Because Ajax applications can actively fetch data before the data are actually required by users, network delay may have little sensitivity. On the other hand, network bandwidth directly impacts the performance of Ajax applications since it takes more time to pre-fetch the content. We believe that our tool helps answer whether a given Web site is more sensitive to either bandwidth or latency by allowing different settings for the Web site.

	10 Mbps	1 Mbps	56 Kbps
NET1	100%	0%	0%
NET2	70%	20%	10%
NET3	20%	50%	30%
NET4	10%	20%	70%
NET5	0%	0%	100%

Table 3: Configuration for weighted IRT simulation.

Since these results indicate a direct dependency of IRT on the network conditions, we consider how IRT distributions change when different clients with different network conditions are mixed together, as a router in the middle of the network would probably observe. Thus, we conduct a simple simulation by mixing together different proportion of clients with different network conditions (particularly 10Mbps, 1Mbps and 56Kbps clients) in Figure 6(a). The parameters for the simulation are summarized in Table 3. Figure 6(c) shows these different IRT distributions for these different combinations.

We believe our mechanism provides interesting projections into how the IRT distribution varies according to the traffic mixes. For example, as we move from NET1 which consists of users with extremely high bandwidth (100% users have 10 Mbps) to the NET5 (100% users have 56Kbps), we see the progressive shift in the curves to the right, indicating a stark increase in the IRT distributions.

We also investigate how the number of requests per flow is affected by changes of bandwidth and network delay constraints. (Due to space constraints, we omit showing the graphs.) We observed that as more bandwidth becomes available, both Maps and Mail services increase the number of requests per flow. As we increase the delay, we found that Mail application showed a slight dip in the number of requests (4 reqs/flow at 10ms down to about 2 reqs/flow at 160ms) while Maps shows variations of 9-15 reqs/flow at 10-80ms but decreases 7 reqs/flow from 160ms because the number of flows itself increases (about 200 flows to 80ms and about 340-400 flows from 160ms).

4.3 Characterizing individual operations

We begin our discussion with explaining methodology for characterizing individual operations. In Table 4, we show the candidate operations within Google Maps and mail applications we selected for our analysis. These operations are by no means exhaustive; we hand-selected only basic operations that seem to represent the main objective of the application. We can easily instrument the tool for characterizing other operations as well. For this analysis, we collected about 250 MB of data representing about 10,000 flows across the two candidate applications for the operations included within Table 4. Each operation has been repeated several times for statistical significance, with exact counts for each operation outlined in Table 4. The counts are different for different operations because we wrote different scenario files, each of which represent different sequences of operations and extracted out the individual operations from the sessions.

We configured the scenario files for each application with a 60 second interval between two adjacent operations (120 seconds for 'idle state' of Mail) to eliminate any interference between them. We then matched the

App.	Operation	Meaning	Count
Google Maps	drag map	dragging a mouse on the map area	140
	zoom in	zooming in map area by moving wheel button on a mouse	69
	zoom out	zooming out map area by moving wheel button on a mouse	62
	click search button	clicking the search button for finding a location	102
Google Mail	delete mail	clicking mail deletion button after selecting a mail	52
	attach file	inputting a file path, and then waiting to be uploaded	37
	idle state	letting Google Mail idle without operations for 120 seconds	56
	click inbox	clicking inbox and listing mails	125
	read mail	reading a mail by clicking mail	70
	send mail	clicking send button to send a mail	96

Table 4: Candidate set of operations selected within Google Maps and Mail applications for characterization. These are some of the most commonly used operations within these applications.

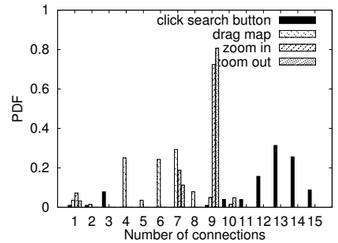
mouse event timing recorded in the logs with the trace files to extract out the relevant network activity. To determine the causal relationship, we collected all flows generated within 5 seconds of the mouse events and analyze their network-level behavior. In the means of correlating operation and traffic, the time value is selected empirically. While in different applications we may have to use the time value different from one used in this paper or require different approaches, we find that the approach works fine for the given two applications in our setting. Thus, only flows that are a direct result of the event are considered for the analysis; some asynchronous flows reflective of the polling mechanisms, such as those employed in Mail, are considered separately through an 'idle state'. To ensure minimal interference with active user activity, the 'idle state' is not mixed with any other user action.

We had to make a few other minor adjustments. The first involves the 'file attach' operation in Mail. File attachments in Mail are handled by specifying a file directory path information in the form, which then attaches and uploads the file automatically after a fixed 10 seconds because there is no explicit upload button; thus, we configured the tool to collect all network activity within 11 seconds (which ensures that any flows that occur as a result of this activity are started within 11 seconds). The other adjustment we required was for the 'idle state' of Mail, which unlike Maps, continuously exchanges information with the server even in the absence of any particular mouse event. To accommodate this, we chose 60 seconds as the window interval to include flows generated during 'idle state' to obtain a finite snapshot of the polling mechanisms in use. Note that once we identify the flows belonging to a specific operation, we analyzed every packets until the flows are closed. The time thresholds are only to identify which flows to keep track of.

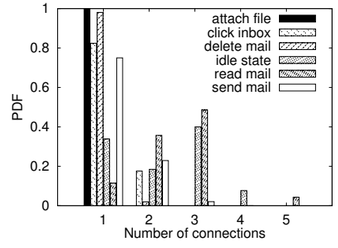
To obtain flow-concurrency, we counted the number of concurrent flows per distinct host (*i.e.*, on a per-IP address basis) and the number of distinct hosts contacted within a window of α seconds of each other. In our analysis, we set α to 0.1 seconds, which is an adjustable parameter in our analyzer. This ensures that flows that are generated automatically as a result of a particular action are obtained and not necessarily those that are related to human actions.

We considered two types of network-level metrics: flow- and message-oriented. Along flow-based metrics, we mainly considered concurrency in terms of number of simultaneous flows generated. For message-oriented metrics, we selected number of request messages per flow. We have analyzed a few more, such as flow inter-arrival time, inter-request time, bytes transferred, request and response message length, and so on, but in the interest of space, we do not show them in this paper. We can potentially also consider packet-level metrics, such as packet size distribution and their timing, but we chose to model mainly Ajax application's characteristics; packet size distributions are dependent on how the TCP layer segments individual packets, and thus is less intrinsic to the Ajax application itself.

Connection Distribution. We analyze connection distribution along three dimensions—total number of connections, number of connections per server, and the number of servers contacted per operation. However, due to space limitation, we only provide a graph about total number of connections in this paper, but briefly explain the other two results. From Figures 7(a) and 7(b), we observe that Maps opens the most number of total connections (up to 15) as well as the most number of connections per server (up to 8). This phenomenon is because Maps requires the fastest responsiveness as every user activity leads to a large number of requests for fairly large map image tiles. While the Mail application also requires fast responsiveness, the responses are usually small, and thus fewer connections are sufficient to fetch the required information.

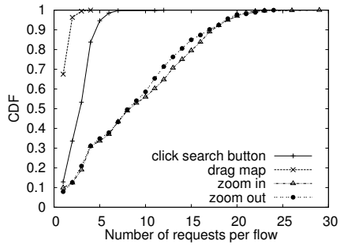


(a) Google Maps

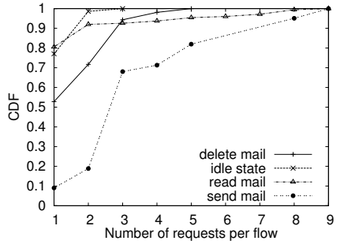


(b) Google Mail

Figure 7: Total connection distributions.



(a) Google Maps



(b) Google Mail

Figure 8: Number of requests per flow.

Among operations within Maps, 'click search button' starts the most number of connections with up to 8 connections per server and up to 5 different servers. This is because clicking on the search button typically leads to the whole refresh of a page thus involving most amount of response data. Other operations on Maps involve more connections (around 4–9) as well, but typically to a much lesser number of servers (around 1 or 2). We believe this might be because operations such as 'drag and drop' typically fetch a few tiles that are adjacent to the current location on the displayed map, thus resulting in smaller amount of data compared to the total refresh when the search button is clicked.

For Mail, we observe that most connections are found in the case of 'read mail' operation. To identify the reason, we inspected the HTTP headers and observed that along with the mail content, several advertisements were fetched from different servers causing more number of connections. The 'idle state' operation came next in terms of the number of connections and servers involved. This is because we use a window interval of 60 seconds, which in turn results in a lot of client requests generated to synchronize with the server for any further updates.

One concern is that the characteristics of Gmail sessions may be dependent on the inbox content. Our approach cannot effectively model the application if there is a lot of variation based on different inboxes, unless appropriate state is created in the form of different logins and different numbers of emails at the server side. In our future work, we will study how sensitive the characteristics of these applications are to the size of the inbox contents or the amount of content at the server side.

Number of request messages per flow. The middle graphs of Figures 8(a) and 8(b) show the CDF of the number of request messages per flow. In the case of Maps, most operations tend to have multiple requests in a flow. Specifically, zooming in and out results in changes of the map scale, and, as a result, the client needs to fetch whole new tile images repeatedly. On the other hand, in the case of 'drag map' operation, the probability of repeated fetching is lower. The 'click search button' operation also requires to fetch several tile images, but it achieves its goal through multiple connections to servers (shown later in Figure 7(a)). Unlike Maps, most operations in Mail have less than 9 requests in a flow.

5 Related Work

Given the recent emergence of Cloud-based applications into the mainstream Web, there has been limited research work in characterization of these applications. A recent paper by Schneider *et al.* [21] made one of the first attempts to characterize these application protocols by examining the contents of HTTP headers over ISP traces. In this work, they study the distributions of several fea-

tures such as bytes transferred, inter-request times, etc., associated with popular cloud-based application traffic. Our work is significantly different from (and in many respects complimentary to) theirs, as we focus on generating user interactions with a Web service in a controlled fashion to generate and obtain our traces.

Web traffic has been extensively studied in the past. While some studies focus on the user-level behavior in terms of the number of request and response messages and application-specific properties such as reference counts and page complexity (*e.g.*, [5, 9]), network-level characteristics were examined in others such as [12, 6, 7]. Several traffic generators based on statistical models have also contributed to understanding the impact of Web traffic (see [3] for a list of these) on the network. These models and tools, however, do not factor the exact cloud application traffic characteristics.

There also exist a lot of tools developed for the classical Web. Web automation tools such as Chickenfoot [8], CARENA [19], and SWAT [1] enable one to navigate a Web site automatically and repeatedly. However, these tools are mainly tailored for static navigation of Websites using automatic filling of forms and lack the functionality to interact with cloud applications.

Traditionally, characterizing and modeling network-level behavior of Web applications is largely trace-driven. For example, prior measurement efforts [21, 7, 10, 14] used Gigabytes to Terabytes of traces collected from core routers. On the other hand, we use end-host based active profiling to study the network-level behavior of individual operations within application.

The idea of end-host based active profiling is not new by itself. There have been several contexts where the idea has been applied. For example, Cunha *et al*. characterized Web traffic by collecting measurements from a modified Web browser [13]. In [17], Krishnamurthy *et al*. use Firefox add-on features to demonstrate how to collect statistics about page downloads. Our tool, shares some similarity, but is more tuned towards cloud applications and is designed to be browser agnostic as opposed to the other approaches.

6 Conclusion

As the popularity of cloud Web services increases, it becomes critical to study and understand their network-level behavior. A lot of tools designed for classical Web characterization are difficult to adapt to cloud applications due to the rich interface used by them. While trace-based approaches are standard for characterization, they do not allow characterizing individual operations within an application or provide any understanding on how these applications behave under different network conditions. We described the design and implementation details of AJAXTRACKER that is designed to address these limitations. It successfully captures realistic network-level flow measurements by imitating the set of mouse/keyboard events specified in the scenario file. The traffic shaper functionality allows it to simulate arbitrary network conditions.As part of ongoing work, based on our tool's capability, we hope to classify and detect cloud applications in the middle of the network. This is particularly useful given that cloud applications cannot easily be detected using port number approaches alone.

Acknowledgments

The authors are indebted to the anonymous reviewers and Marvin Theimer, our shepherd, for comments on previous versions of this manuscript. We also thank the IT staff at Purdue, Addam Schroll, William Harshbarger, Greg Hedrick for their immense help in obtaining the network traces. This work was supported in part by NSF Award CNS 0831647 and a grant from Cisco Systems.

References

[1] Simple Web Automation Tool. http://swat.sourceforge.net/.

[2] tcpdump. http://www.tcpdump.org/.

[3] Traffic Generators for Internet Traffic. http://www.icir.org/models/trafficgenerators.html.

[4] Xerces C++ Parser. http://xerces.apache.org/xerces-c/.

[5] M. Arlitt and C. Williamson. Internet Web Servers: Workload Characterizations and Implications. *IEEE/ACM Transactions on Networking*, 5(5), Oct. 1997.

[6] P. Barford and M. E. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of Performance '98/SIGMETRICS '98*, 1998.

[7] N. Basher, A. Mahanti, A. Mahanti, C. Williamson, and M. Arlitt. A comparative analysis of web and peer-to-peer traffic. In *WWW*, 2008.

[8] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. In *Proc. UIST '05. ACM Press*, pages 163–172, 2005.

[9] F. Campos, K. Jeffay, and F. Smith. Tracking the evolution of web traffic: 1995 - 2003. In *MASCOTS*, 2003.

[10] J. Cao, W. S. Clevel, Y. Gao, K. Jeffay, F. D. Smith, and M. Weigle. Stochastic models for generating synthetic http source traffic. In *INFOCOM*, pages 1546–1557, Mar. 2004.

[11] D. Crane, E. Pascarello, and D. James. *Ajax in Action*. Manning, 2006.

[12] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, Dec. 1997.

[13] C. R. Cunha, A. Bestavros, and M. E. Crovella. Characteristics of WWW Client-based Traces. Technical Report BUCS-TR-1995-010, Boston University, 1995.

[14] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. YouTube Traffic Characterization: A View From the Edge. In *ACM IMC*, 2007.

[15] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: Multilevel traffic classification in the dark. In *ACM SIGCOMM*, 2005.

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[17] B. Krishnamurthy and C. E. Wills. Generating a privacy footprint on the internet. In *ACM IMC*, 2006.

[18] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected Means of Protocol Inference. In *ACM IMC*, 2006.

[19] I. J. Nino, B. de la Ossa, J. A. Gil, J. Sahuquillo, and A. Pont. Carena: a tool to capture and replay web navigation sessions. In *E2EMON '05: Proceedings of the End-to-End Monitoring Techniques and Services on 2005 Workshop*, pages 127–141, 2005.

[20] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-Service Mapping for QoS: A Statistical Signature-based Approach to IP Traffic Classification. In *ACM IMC*, 2004.

[21] F. Schneider, S. Agarwal, T. Alpcan, and A. Feldmann. The New Web: Characterizing AJAX Traffic. In *International Conference on Passive and Active Network Measurement*, 2008.

JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications

Paruj Ratanaworabhan
Kasetsart University
paruj.r@ku.ac.th

Benjamin Livshits and Benjamin G. Zorn
Microsoft Research
{livshits,zorn}@microsoft.com

Abstract

JavaScript is widely used in web-based applications and is increasingly popular with developers. So-called browser wars in recent years have focused on JavaScript performance, specifically claiming comparative results based on benchmark suites such as SunSpider and V8. In this paper we evaluate the behavior of JavaScript web applications from commercial web sites and compare this behavior with the benchmarks.

We measure two specific areas of JavaScript runtime behavior: 1) functions and code and 2) events and handlers. We find that the benchmarks are not representative of many real web sites and that conclusions reached from measuring the benchmarks may be misleading. Specific common behaviors of real web sites that are underemphasized in the benchmarks include event-driven execution, instruction mix similarity, cold-code dominance, and the prevalence of short functions. We hope our results will convince the JavaScript community to develop and adopt benchmarks that are more representative of real web applications.

1 Introduction

JavaScript is a widely used programming language that is enabling a new generation of computer applications. Used by large fraction of all web sites, including Google, Facebook, and Yahoo, JavaScript allows web applications to be more dynamic, interesting, and responsive. Because JavaScript is so widely used to enable Web 2.0, the performance of JavaScript is now a concern of vendors of every major browser, including Mozilla Firefox, Google Chrome, and Microsoft Internet Explorer. The competition between major vendors, also known as the 'browser wars" [24], has inspired aggressive new JavaScript implementations based on Just-In-Time (JIT) compilation strategies [8].

Because browser market share is extremely important to companies competing in the web services marketplace, an objective comparison of the performance of different browsers is valuable to both consumers and service providers. JavaScript benchmarks, including SunSpider [23] and V8 [10], are widely used to evaluate JavaScript performance (for example, see [13]). These benchmark results are used to market and promote browers, and the benchmarks influence the design of JavaScript runtime implementations. Performance of JavaScript on the SunSpider and V8 benchmarks has improved dramatically in recent years.

This paper examines the following question: How representative are the SunSpider and V8 benchmarks suites when compared with the behavior of real JavaScript-based web applications? More importantly, we examine how benchmark behavior that differs quite significantly from real web applications might mislead JavaScript runtime developers.

By instrumenting the Internet Explorer 8 JavaScript runtime, we measure the JavaScript behavior of 11 important web applications and pages, including Gmail, Facebook, Amazon, and Yahoo. For each application, we conduct a typical user interaction scenario that uses the web application for a productive purpose such as reading email, ordering a book, or finding travel directions. We measure a variety of different program characteristics, ranging from the mix of operations executed to the frequency and types of events generated and handled.

Our results show that real web applications behave very differently from the benchmarks and that there are definite ways in which the benchmark behavior might mislead a designer. Because of the space limitations, this paper presents a relatively brief summary of our findings. The interested reader is referred to a companion technical report [17] for a more comprehensive set of results.

The contributions of this paper include:

- We are among the first to publish a detailed characterization of JavaScript execution behavior in real web applications, the SunSpider, and the V8 bench-

marks. In this paper we focus on functions and code as well as events and handlers. Our technical report [17] considers heap-allocated objects and data.

- We conclude that the benchmarks are not representative of real applications in many ways. Focusing on benchmark performance may result in overspecialization for benchmark behavior that does not occur in practice, and in missing optimization opportunities that are present in the real applications but not present in the benchmarks.
- We find that real web applications have code that is one to two orders of magnitude larger than most of the benchmarks and that managing code (both allocating and translating) is an important activity in a real JavaScript engine. Our case study in Section 4.7 demonstrates this point.
- We find that while the benchmarks are compute-intensive and batch-oriented, real web applications are event-driven, handling thousands of events. To be responsive, most event handlers execute only tens to hundreds of bytecodes. As a result, functions are typically short-lived, and long-running loops are uncommon.
- While existing JavaScript benchmarks make minimal use of event handlers, we find that they are extensively used in real web applications. The importance of responsiveness in web application design is not captured adequately by any of the benchmarks available today.

2 Background

JavaScript is a garbage-collected, memory-safe programming language with a number of interesting properties [6]. Unlike class-based object-oriented languages like C# and Java, JavaScript is a prototype-based language, influenced heavily in its design by Self [22]. JavaScript became widely used because it is standardized, available in every browser implementation, and tightly coupled with the browser's Document Object Model [2].

Importance of JavaScript. JavaScript's popularity has grown with the success of the web. Scripts in web pages have become increasingly complex as AJAX (Asynchronous JavaScript and XML) programming has transformed static web pages into responsive applications [11]. Web sites such as Amazon, Gmail, and Facebook contain and execute significant amounts of JavaScript code, as we document in this paper. Web applications (or apps) are applications that are hosted entirely in a browser and delivered through the web. Web apps have the advantage that they require no additional installation, will run on any machine that has a browser, and provide access to information stored in the cloud. Sophisticated mobile phones, such as the iPhone, broaden the base of Internet users, further increasing the importance and reach of web apps.

In recent years, the complexity of web content has spurred browser developers to increase browser performance in a number of dimensions, including improving JavaScript performance. Many of the techniques for improving traditional object-oriented languages such as Java and C# can and have been applied to JavaScript [8, 9]. JIT compilation has also been effectively applied, increasing measured benchmark performance of JavaScript dramatically.

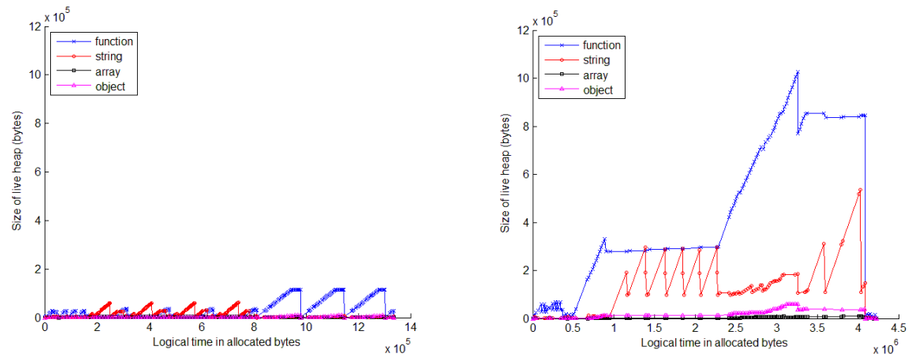
Value of benchmarks. Because browser performance can significantly affect a user's experience using a web application, there is commercial pressure for browser vendors to demonstrate that they have improved performance. As a result, JavaScript benchmark results are widely used in marketing and in evaluating new browser implementations. The two most widely used JavaScript benchmark suites are SunSpider, a collection of small benchmarks available from WebKit.org [23], and the V8 benchmarks, a collection of seven slightly larger benchmarks published by Google [10]. The benchmarks in both of these suites are relatively small programs; for example, the V8 benchmarks range from approximately 600 to 5,000 lines of code.

Illustrative example. Before we discuss how we collect JavaScript behavior data from real sites and benchmarks, we illustrate how this data is useful. Figure 1 shows live heap graphs for visits to the `google` and `bing` web sites[1]. These graphs show the number of live bytes of different types of data in the JavaScript heap as a function of time (measured by bytes of data allocated). In the figures, we show only the four most important data types: functions, strings, arrays, and objects. When the JavaScript heap is discarded, for example because the user navigates to a new page, the live bytes drops to zero, as we see in `google`.

These two search web sites shown offer very similar functionality, and we performed the same sequence of operations on them during our visit: we searched for "New York" in both cases and then proceeded to page through the results, first web page results and then the relevant news items.

We see from our measurements of the JavaScript heap, however, that the implementations of the two applications are very different, with `google` being implemented as a series of visits to different pages, and `bing` implemented as a single page visit. The benefit of the `bing` approach is highlighted in this case by looking at the right hand side of each subfigure. In the case of `google`, we see that the contents of the JavaScript heap, including all the functions, are discarded and recreated repeatedly during our visit, whereas in the `bing` heap the functions are allocated only once. The size of the `google` heap is significantly smaller than the `bing` heap (approximately an order of magnitude), so it could be argued that the `google` approach is better. On the other hand, the `bing` approach does not lead to the JavaScript heap being repeatedly recreated.

[1] Similar graphs for all the real web sites and benchmarks can be found in our tech report [17].



(a) Live heap for `google`.

(b) Live heap for `bing`.

Figure 1: Live heap contents as a function of time for two search applications.

In conclusion, we note that this kind of dynamic heap behavior is not captured by any of the V8 or SunSpider benchmarks, even though it is common among real web applications. Knowledge about such allocation behavior can be useful when, for example, designing and optimizing the garbage collection systems.

3 Experimental Design

In this section, we describe the benchmarks and applications we used and provide an overview of our measurements.

Figure 2 lists the 11 real web applications that we used for our study[2]. These sites were selected because of their popularity according to Alexa.com, and also because they represent a cross-section of diverse activities. Specifically, our applications represent search (`google`, `bing`), mapping (`googlemap`, `bingmap`), email (`hotmail`, `gmail`), e-commerce (`amazon`, `ebay`), news (`cnn`, `economist`), and social networking (`facebook`). Part of our goal was to understand both the differences between the real sites and the benchmarks as well as the differences among different classes of real web applications. For the remainder of this paper, we will refer to the different web sites using the names from Figure 2.

The workload for each site mimics the behavior of a user on a short, but complete and representative, visit to the site. This approach is dictated partly by expedience — it would be logistically complicated to measure long-term use of each web application — and partly because we believe that many applications are actually used in this way. For example, search and mapping applications are often used for targeted interactions.

[2] Throughout this discussion, we use the terms web application and web site interchangeably. When we refer to the site, we specifically mean the JavaScript executed when you visit the site.

3.1 Web Applications and Benchmarks

In measuring the JavaScript benchmarks, we chose to use the entire V8 benchmark suite, which comprises 7 programs, and selected programs from the SunSpider suite, which consists of 26 different programs. In order to reduce the amount of data collected and displayed, for SunSpider we chose the longest running benchmark in each of the 9 different benchmark categories — 3d: `raytrace`, access: `nbody`, bitops: `nseive` – `bits`, controlflow: `recursive`, crypto: `aes`, date: `xparb`, math: `cordic`, regexp: `dna`, and string: `tagcloud`.

3.2 Instrumenting Internet Explorer

Our approach to data collection is illustrated in Figure 3. The platform we chose for instrumentation is Internet Explorer (IE), version 8, running on a 32-bit Windows Vista operating system. While our results are in some ways specific to IE, the methods described here can be

Site	URL	Actions performed
amazon	amazon.com	Search for the book "Quantitative Computer Architecture," add to shopping cart, sign in, and sign out
bing	bing.com	Type in the search query "New York" and look at resulting images and news
bingmap	maps.bing.com	Search for directions from Austin to Houston, search for a location in Seattle, zoom-in, and use the bird's-eye view feature
cnn	cnn.com	Read the front-page news and three other news articles
ebay	ebay.com	Search for a notebook computer, sign in, bid, and sign out
economist	economist.com	Read the front-page news, read three other articles, view comments
facebook	facebook.com	Log in, visit a friend's page, browser through photos and comments
gmail	mail.google.com	Sign in, check inbox, delete a mail item, sign out
google	google.com	Type in the search query "New York" and look at resulting images and news
googlemap	maps.google.com	Search for directions from Austin to Houston, search for a location in Seattle, zoom-in, and use the street view feature
hotmail	hotmail.com	Sign in, check inbox, delete a mail item, sign out

Figure 2: Real web sites visited and actions taken.

applied to other browsers as well.

Our measurement approach works as follows: we have instrumented the C++ code that implements the IE 8 JavaScript runtime. For IE, the code that is responsible for executing JavaScript programs is not bundled in the main IE executable. Instead, it resides in a dynamic linked library, `jscript.dll`. After performing the instrumentation, we recompiled the engine source code to create a custom `jscript.dll`. (see Step 1 in Figure 3).

Next, we set up IE to use the instrumented `jscript.dll`. We then visit the web sites and run the benchmark programs described in the previous section with our special version of IE. A set of binary trace files is created in the process of visiting the web site or running a benchmark. These traces typically comprise megabytes of data, often up to 800 megabytes in the case of instruction traces. Finally, we use offline analyzers to process these custom trace files to obtain the results presented here.

3.3 Behavior Measurements

In studying the behavior of JavaScript programs, we focused on three broad areas: functions and code, objects and data (omitted here), and events and handlers. In each of these dimensions, we consider both static measurements (e.g., number of unique functions) and dynamic measurements (e.g., total number of function calls). We measure mostly the logical behavior of

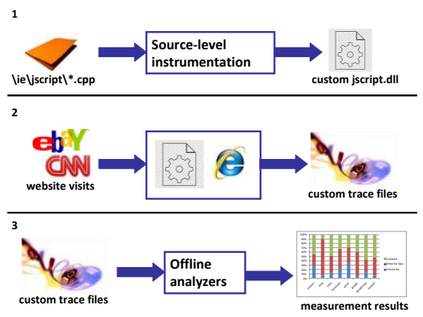


Figure 3: Instrumentation framework for measuring JavaScript execution using Internet Explorer.

JavaScript programs, avoiding characteristics that are browser-dependent. Thus, our measurements are largely machine-independent. However, we also look at specific characteristics of the IE's JavaScript engine (e.g., we count IE 8 bytecodes as a measure of execution) that pertain to interpreter-based engines. We leave measurements for characteristics relevant to JIT-based engines such as those found in Firefox and Chrome for future work.

3.3.1 Functions and Code

The JavaScript engine in IE 8 interprets JavaScript source after compiling it to an intermediate representation called bytecode. The interpreter has a loop that reads each bytecode instruction and implements its effect in a virtual machine. Because no actual machine instructions are generated in IE 8, we cannot measure the execution of JavaScript in terms of machine instructions. The bytecode instruction set implemented by the IE 8 interpreter is a well-optimized, traditional stack-oriented bytecode.

We count each bytecode execution as an "instruction" and use the term bytecode and instruction interchangeably throughout our evaluation. In our measurements, we look at the code behavior at two levels, the function and the bytecode level. Therefore, we instrument the engine at the points when it creates functions as well as in its main interpreter loop. Prior work measuring architecture characteristics of interpreters also measures behavior in terms of bytecode execution [19].

3.3.2 Events and Handlers

JavaScript has a single-threaded event-based programming model, with each event being processed by a non-preemptive handler. In other words, JavaScript code runs in response to specific user-initiated events such as a

Behavior	Real applications	Benchmarks	Implications
	CODE AND FUNCTIONS		
Code size	100s of kilobytes to a few megabytes	100s of bytes to 10s of kilobytes	Efficient in-memory function and bytecode representation
Number of functions	1000s of functions	10s to 100s of functions	Minimize per-function fixed costs
Number of hot functions	10s to 100s of functions	10 functions or less	Size hot function cache appropriately
Instruction mix	Similar to each other	Different across benchmarks and from real applications	Optimize for real application instruction mix
Cold code	Majority of code	Minority of code	Download, parse, and JIT code lazily
Function duration	Mostly short	Mostly short, some very long running	Loop optimizations less effective
	EVENTS AND EVENT HANDLERS		
Handler invocations	1000s of invocations	Less than 10 invocations	Optimize for frequent handler calls
Handler duration	10s to 100s of bytecodes	Very long	Make common short handler case fast
	MEMORY ALLOCATION AND OBJECT LIFETIMES		
Allocation rate	Significant, sustained	Only significant in a few	GC performance not a factor in benchmark results
Data types	Functions and strings dominate	Varies, JS objects dominate in some	Optimize allocation of functions, strings
Object lifetimes	Depends on type, some long-lived	Very long or very short	Approaches like generational collection hard to evaluate with benchmarks
Heap reuse	Web 1.0 has significant reuse between page loads	No heap reuse	Optimize code, heap for reuse case—cache functions, DOM, possibly heap contents

Figure 4: A summary of lessons learned from JSMeter.

mouse click, becomes idle, and waits for another event to process. Therefore, to completely understand behaviors of JavaScript that are relevant to its predominant usage, we must consider the event-driven programming model of JavaScript. Generally speaking, the faster handlers complete, the more responsive an application appears.

However, event handling is an aspect of program behavior that is largely unexplored in related work measuring C++ and Java execution (e.g., see [5] for a thorough analysis of Java execution). Most related work considers the behavior of benchmarks, such as SPECjvm98 [4] and SPECcpu2000 [1], that have no interactive component. For JavaScript, however, such batch processing is mostly irrelevant.

For our measurements, we insert instrumentation hooks before and after event handling routines to measure characteristics such as the number of events handled and the dynamic size of each event handler invocation as measured by the number of executed bytecode instructions.

4 Evaluation

We begin this section with an overview of our results. We then consider the behavior of the JavaScript functions and code, including the size of functions, opcodes executed, etc. Next, we investigate the use of events and event handlers in the applications. We conclude the section with a case study showing that introducing cold code, i.e., code that is never executed, into existing benchmarks has a substantial effect on performance results.

4.1 Overview

Before drilling down into our results, we summarize the main conclusions of our comparison in Figure 4. The first column of the table indicates the specific behavior we measured, and the next two columns compare and contrast results for the real web applications and benchmarks. The last column summarizes the implications of the observed differences, specifically providing insights for future JavaScript engine designers. Due to space constraints, a detailed comparison of all aspects of behavior is beyond the scope of this paper, and we refer the reader to our tech report for those details [17].

4.2 Functions and Code Behavior

We begin our discussion by looking at a summary of the functions and behavior of the real applications and benchmarks. Figure 5 summarizes our static and dynamic measurements of JavaScript functions.

The real web sites. In Figure 5a, we see that the real web applications comprise many functions, ranging from a low of around 1,000 in `google` to a high of 10,000 in `gmail`. The total amount of JavaScript

	Static				Dynamic				
	Unique Func.	Source (bytes)	Compiled (bytes)	Global Context	Unique Func.	Total Calls	Total Opcodes	Opcodes / Call	% Unique Exec. Func.
amazon	1,833	692,173	312,056	210	808	158,953	9,941,596	62.54	44.08%
bing	2,605	1,115,623	657,118	50	876	23,759	1,226,116	51.61	33.63%
bingmap	4,258	1,776,336	1,053,174	93	1,826	274,446	12,560,049	45.77	42.88%
cnn	1,246	551,257	252,214	124	526	99,731	5,030,647	50.44	42.22%
ebay	2,799	1,103,079	595,424	210	1,337	189,805	7,530,843	39.68	47.77%
economist	2,025	899,345	423,087	184	1,040	116,562	21,488,257	184.35	51.36%
facebook	3,553	1,884,554	645,559	130	1,296	210,315	20,855,870	99.16	36.48%
gmail	10,193	2,396,062	2,018,450	129	3,660	420,839	9,763,506	23.20	35.91%
google	987	235,996	178,186	42	341	10,166	427,848	42.09	34.55%
googlemap	5,747	2,024,655	1,218,119	144	2,749	1,121,777	29,336,582	26.15	47.83%
hotmail	3,747	1,233,520	725,690	146	1,174	15,474	585,605	37.84	31.33%

(a) Real web application summary.

	Static				Dynamic				
	Unique Func.	Source (bytes)	Compiled (bytes)	Global Context	Unique Func.	Total Calls	Total Opcodes	Opcodes / Call	% Unique Exec. Func.
richards	67	22,738	7,617	3	59	81,009	2,403,338	29.67	88.06%
deltablue	101	33,309	11,263	3	95	113,276	1,463,921	12.92	94.06%
crypto	163	55,339	31,304	3	91	103,451	90,395,272	873.80	55.83%
raytrace	90	37,278	15,014	3	72	214,983	5,745,822	26.73	80.00%
earley	416	203,933	65,693	3	112	813,683	25,285,901	31.08	26.92%
regexp	44	112,229	35,370	3	41	96	935,322	9742.94	93.18%
splay	47	17,167	5,874	3	45	678,417	25,597,696	37.73	95.74%

(b) V8 benchmark summary.

	Static				Dynamic				
	Unique Func.	Source (bytes)	Compiled (bytes)	Global Context	Unique Func.	Total Calls	Total Opcodes	Opcodes / Call	% Unique Exec. Func.
3d-raytrace	31	14,614	7,419	2	30	56,631	5,954,264	105.14	96.77%
access-nbody	14	4,437	2,363	2	14	4,563	8,177,321	1,792.09	100.00%
bitops-nsieve	6	939	564	2	5	5	13,737,420	2,747,484.00	83.33%
controlflow	6	790	564	2	6	245,492	3,423,090	13.94	100.00%
crypto-aes	22	17,332	6,215	2	17	10,071	5,961,096	591.91	77.27%
date-xparb	24	12,914	5,341	4	12	36,040	1,266,736	35.15	50.00%
math-cordic	8	2,942	862	2	6	75,016	12,650,198	168.63	75.00%
regexp-dna	3	108,181	630	2	3	3	594	198.00	100.00%
string-tagcloud	16	321,894	55,219	3	10	63,874	2,133,324	33.40	62.50%

(c) SunSpider benchmark summary.

Figure 5: Summary measurements of web applications and benchmarks.

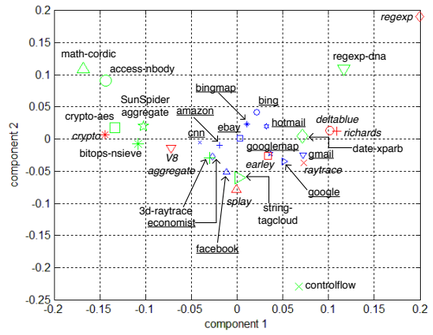


Figure 6: Opcode frequency distribution comparison.

source code associated with these web sites is significant, ranging from 200 kilobytes to more than two megabytes of source. Most of the JavaScript source code in these applications has been "minified", that is, had the whitespace removed and local variable names minimized using available tools such as JScrunch [7] or JSmin [3]. This source code is translated to the smaller bytecode representation, which from the figure we see is roughly 60% the size of the source.

In the last column, which captures the percentage of static unique functions executed, we see that as many as 50–70% are *not* executed during our use of the applications, suggesting that much of the code delivered applies to specific functionality that we did not exercise when we visited the sites. Code-splitting approaches such as Doloto [15] exploit this fact to reduce the wasted effort of downloading and compiling cold code.

The number of bytecodes executed during our visits ranged from around 400,000 to over 20 million. The most compute-intensive applications were `facebook`, `gmail`, and `economist`. As we show below, the large number of executed bytecodes in `economist` is an anomaly caused by a hot function with a tight loop. This anomaly is also clearly visible from the opcodes/call column. We see that `economist` averages over 180 bytecodes per call, while most of the other sites average between 25 and 65 bytecodes per call. This low number suggests that a majority of JavaScript function executions in these programs *do not* execute long-running loops. Our discussion of event handler behavior in Section 4.6 expands on this observation.

Because it is an outlier, the `economist` application deserves further comment. We looked at the hottest function in the application and found a single function which accounts for over 50% of the total bytecodes executed in our visit to the web site. This function loops over the elements of the DOM looking for elements with a specific node type and placing those elements into an array. Given that the DOM can be quite large, using an interpreted loop to gather specific kinds of elements can be quite expensive to compute. An alternative, more efficient implementation might use DOM APIs like `getElementById` to find the specific elements of interest directly.

On a final note, in column five of Figure 5 we show the number of instances of separate matching < `script` > tags that appeared in the web pages that implemented the applications. We see that in the real applications, there are many such instances, ranging to over 200 in `ebay`. This high number indicates that JavaScript code is coming from a number of sources in the applications, including different modules and/or feature teams from within the same site, and also coming from third party sites, for advertising, analytics, etc.

The benchmarks. In Figure 5, we also see the summary of the V8 and SunSpider benchmarks. We see immediately that the benchmarks are much smaller, in terms of both source code and compiled bytecode, than the real applications. Furthermore, the largest of the benchmarks, `string` – `tagcloud`, is large not because of the amount of code, but because it contains a large number of string constants. Of the benchmarks, `earley` has the most real code and is an outlier, with 400 functions compared to the average of the rest, which is well below 100 functions. These functions compile down to very compact bytecode, often more than 10 times smaller than the real applications. Looking at the fraction of these functions that are executed when the benchmarks are run, we see that in many cases the percentage is high, ranging from 55–100%. The benchmark `earley` is again an outlier, with only 27% of the code actually executed in the course of running the benchmark.

The opcodes per call measure also shows significant differences with the real applications. Some of the SunSpider benchmarks, in particular, have long-running loops, resulting in high average bytecodes executed per call. Other benchmarks, such as `controlflow`, have artificially low counts of opcodes per call. Finally, none of the benchmarks has a significant number of distinct contexts in which JavaScript code is introduced (global scope), emphasizing the homogeneous nature of the code in each benchmark.

4.3 Opcode Distribution

We examined the distribution of opcodes that each of the real applications and benchmarks executed. To do this, we counted how many times each of the 160 different opcodes was executed in each program and normalized

these values to fractions. We then compared the 160-dimensional vector generated by each real application and benchmark.

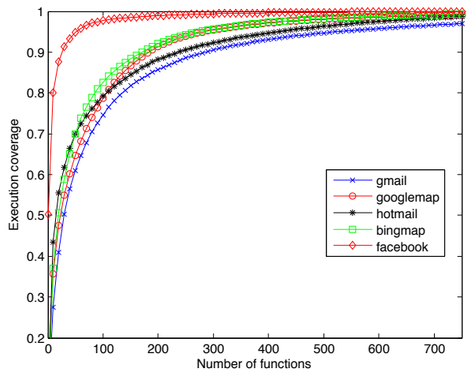
Our goal was to characterize the kinds of operations that these programs perform and determine how representative the benchmarks are of the opcode mix performed by the real applications. We were also interested in understanding how much variation exists between the individual real applications themselves, given their diverse functionality.

To compare the resulting vectors, we used Principal Component Analysis (PCA) [12] to reduce the 160-dimensional space to two principal dimensions. This dimension reduction is a way to avoid the curse of dimensionality problem. We found that components after the third are insignificant and chose to present only the two principal components for readability. Figure 6 shows the result of this analysis. In the figure, we see the three different program collections (real, V8, and SunSpider). The figure shows that the real sites cluster in the center of the graph, showing relatively small variation among themselves.

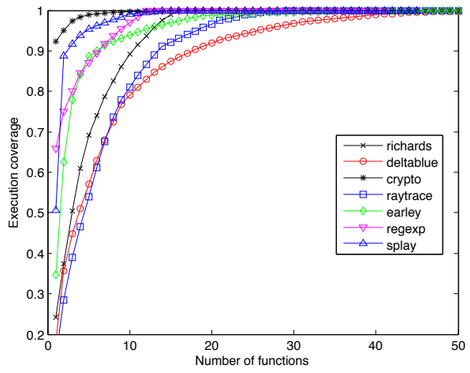
For example, `ebay` and `bingmap`, very different in their functionality, cluster quite closely. In contrast, both sets of benchmarks are more widely distributed, with several obvious outliers. For SunSpider, `controlflow` is clearly different from the other applications, while in V8, `regexp` sits by itself. Surprisingly, few of the benchmarks overlap the cluster of real applications, with `earley` being the closest in overall opcode mix to the real applications. While we expect some variation in the behavior of a collection of smaller programs, what is most surprising is that almost all the benchmarks have behaviors that are significantly different than the real applications. Furthermore, it is also surprising that the real web applications cluster as tightly as they do. This result suggests that while the external functionality provided may appear quite different from site to site, much of the work being done in JavaScript on these sites is quite similar.

4.4 Hot Function Distribution

We next consider the distribution of hot functions in the applications, which tells us what code needs to be highly optimized. Figure 7 shows the distribution of hot functions in a subset of the real applications and the V8 benchmarks (full results, including the SunSpider benchmarks are included in [17]). Each figure shows the cumulative contribution of each function, sorted by hottest functions first on the x-axis, to normalized total opcodes executed on the y-axis. We truncate the x-axis (not considering all functions) to get a better view of the left end of the curve. The figures show that all programs, both



(a) Real web application hot function distribution.



(b) V8 benchmarks hot function distribution.

Figure 7: Hot function distribution.

real applications and benchmarks, exhibit high code locality, with a small number of functions accounting for a large majority of total execution. In the real applications, 80% of total execution is covered by 50 to 150 functions, while in the benchmarks, at most 10 functions are required. `facebook` is an outlier among the real applications, with a small number of functions accounting for almost all the execution time.

4.5 Implications of Code Measurements

We have considered static and dynamic measures of JavaScript program execution, and discovered numerous important differences between the behaviors of the real applications and the benchmarks. Here we discuss how these differences might lead designers astray when building JavaScript engines that optimize benchmark performance.

First, we note a significant difference in the code size of the benchmarks and real applications. Real web applications have large code bases, containing thousands of functions from hundreds of individual < `script` > bodies. Much of this code is never or rarely executed, meaning that efforts to compile, optimize, or tune this code are unnecessary and can be expensive relative to what the benchmarks would indicate. We also observe that a substantial fraction of the downloaded code is not executed in a typical interaction with a real application. Attempts to avoid downloading this code, or minimizing the resources that it consumes once it is downloaded, will show much greater benefits in the real applications than in the benchmarks.

Second, we observe that based on the distribution of opcodes executed, benchmark programs represent a much broader and skewed spectrum of behavior than the real applications, which are quite closely clustered. Tuning a JavaScript engine to run `controlflow` or `regexp` may improve benchmark results, but tuning the engine to run any one of the real applications is also likely to significantly help the other real applications as well. Surprisingly, few of the benchmarks approximate the instruction stream mix of the real applications, suggesting that there are activities being performed in the real applications that are not well emulated by the benchmark code.

Third, we observe that each individual function execution in the real applications is relatively short. Because these applications are not compute-intensive, benchmarks with high loop counts, such as `bitops` – `nsieve`, distort the benefit that loop optimizations will provide in real applications. Because the benchmarks are batch-oriented to facilitate data collection, they fail to match a fundamental characteristic of all real web applications — the need for responsiveness. The very nature of an interactive application prevents developers from writing code that executes for long periods of time without interruption.

Finally, we observe that a tiny fraction of the code accounts for a large fraction of total execution in both the benchmarks and the real applications. The size of the hot code differs by one to two orders of magnitude between the benchmarks and applications, but even in the real applications the hot code is still quite compact.

4.6 Event Behavior

In this section, we consider the event-handling behavior of the JavaScript programs. We observe that handling events is commonplace in the real applications and almost never occurs in the benchmarks. Thus the focus of this section is on characterizing the handler behavior of the real applications.

	# of events	unique events	executed instructions handler	executed instructions total
richards	8	6	2,403,333	2,403,338
deltablue	8	6	1,463,916	1,463,921
crypto	11	6	86,854,336	86,854,341
raytrace	8	6	5,745,817	5,745,822
earley	11	6	25,285,896	25,285,901
regexp	8	6	935,317	935,322
splay	8	6	25,597,691	25,597,696

Figure 9: Event handler characteristics in the V8 benchmarks.

Before discussing the results, it is important to explain how handlers affect JavaScript execution. In some cases, handlers are attached to events that occur when a user interacts with a web page. Handlers can be attached to any element of the DOM, and interactions such as clicking on an element, moving the mouse over an element, etc., can cause handlers to be invoked. Handlers also are executed when a timer times out, when a page loads, or when an asynchronous `XMLHttpRequest` is completed. JavaScript code is also executed outside of a handler context, such as when a < `script` > block is processed as part of parsing the web page. Often code that initializes the JavaScript for the page executes outside of a handler.

Because JavaScript has a non-preemptive execution model, once a JavaScript handler is started, the rest of the browser thread for that particular web page is stalled until it completes. A handler that takes a significant amount of time to execute will make the web application appear sluggish and non-responsive.

Figures 8 and 9 present measures of the event handling behavior in the real applications and the V8 benchmarks[3]. In both tables, unique events are defined as follows. Events are nominally unique when they invoke the same sequences of handler instructions with the same inputs. Our measurements in the figures only approximate this definition. We associate each event with three attributes: name, the set of handler functions invoked, and the total number of instructions executed. If the two events have the same three attributes, we say that they are unique.

We see that the real applications typically handle thousands of events while the benchmarks all handle 11 or fewer. In all the benchmarks, one `onload` event (for loading and, subsequently, running the benchmark program) is responsible for almost 100% of all JavaScript execution. We will see shortly that this is in stark contrast to the behavior seen in the real applications. Even though real web sites typically process thousands of events, the unique events column in the figure indicates that there are only around one hundred unique events per application. This means that a given event is likely to be repeated and

[3] SunSpider results are similar to V8 results, so we omit them here.

	# of events	unique events	executed instructions handler	executed instructions total	% of handler instructions	handler size average	handler size median	handler size maximum
amazon	6,424	224	7,237,073	9,941,596	72.80%	1,127	8	1,041,744
bing	4,370	103	598,350	1,226,116	48.80%	137	24	68,780
bingmap	4,669	138	8,274,169	12,560,049	65.88%	1,772	314	281,887
cnn	1,614	133	4,939,776	5,030,647	98.19%	3,061	11	4,208,115
ebay	2,729	136	7,463,521	7,530,843	99.11%	2,735	80	879,798
economist	2,338	179	21,146,767	21,488,257	98.41%	9,045	30	270,616
facebook	5,440	143	17,527,035	20,855,870	84.04%	3,222	380	89,785
gmail	1,520	98	3,085,482	9,763,506	31.60%	2,030	506	594,437
google	569	64	143,039	427,848	33.43%	251	43	10,025
googlemap	3,658	74	26,848,187	29,336,582	91.52%	7,340	2,137	1,074,568
hotmail	552	194	474,693	585,605	81.06%	860	26	202,105

Figure 8: Event handler characteristics in real applications.

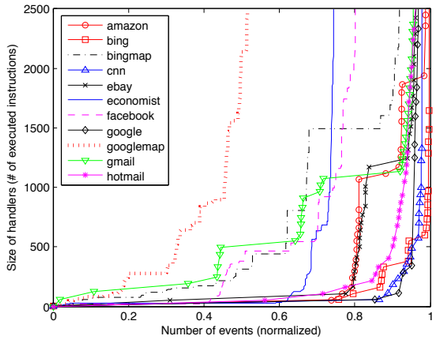


Figure 10: Distribution of handler durations.

handled many times throughout the course of a user visit to the site.

We see the diversity of the collection of handlers in the results comparing the mean, median, and maximum of handler durations for the real applications. Some handlers run for a long time, such as in `cnn`, where a single handler accounts for a significant fraction of the total JavaScript activity. Many handlers execute for a very short time, however. The median handler duration in `amazon`, for example, is only 8 bytecodes. `amazon` is also unusual in that it has the highest number of events. We hypothesize that such short-duration handlers probably are invoked, test a single value, and then return.

These results demonstrate that handlers are written so that they almost always complete in a short time. For example, in `bing` and `google`, both highly optimized for delivering search results quickly, we see low average and median handler times. It is also clear that `google`, `bing`, and `facebook` have taken care to reduce the duration of the longest handler, with the maximum of all three below 100,000 bytecodes.

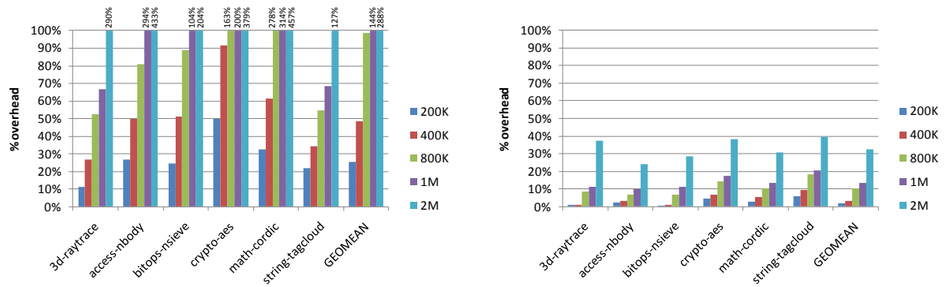
Figure 10 illustrates the distribution of handler durations for each of the applications. The x-axis depicts the instances of handler invocations, sorted by smallest first and normalized to one. The y-axis depicts the number of bytecodes executed by each handler invocation. For example, in the figure, approximate 40% of the handlers in `googlemap` executed for 1000 bytecodes or less.

Figure 10 confirms that most handler invocations are short. This figure provides additional context to understand the distribution. For example, we can determine the 95th percentile handler duration by drawing a vertical line at 0.95 and seeing where each line crosses it. The figure also illustrates that the durations in many of the applications reach plateaus, indicating that there are many instances of handlers that execute for the same number of instructions. For example, we see a significant number of `bingmap` instances that take 1,500 bytcodes to complete.

4.7 Cold Code Case Study

Our results show that real web applications have much more JavaScript code than the SunSpider and V8 benchmarks and that most of that code is cold. We were curious how much impact the presence of such cold code would have on benchmark performance results. Based on our understanding of the complexity and performance overhead of code translation, especially in a JIT-compiler, we hypothesized that simply increasing the amount of cold code in existing benchmarks would have a significant non-uniform impact on benchmark results. If this hypothesis is true, then a simple way to make results from current benchmarks more representative of actual web applications would be to add cold code to each of them.

To test this hypothesis, we selected six SunSpider benchmarks that are small and have mostly hot code. To each of these benchmarks, we added 200 kilobytes, 400 kilobytes, 800 kilobytes, 1 megabyte and 2 megabytes



(a) Impact of cold code in Chrome.

(b) Impact of cold code Internet Explorer 8.

Figure 11: Impact of cold code using a subset of the SunSpider benchmarks.

of cold code from the jQuery library. The added code is never called in the benchmark but the JavaScript runtime still processes it. We executed each benchmark with the added code and recorded its performance on both the Google Chrome and Internet Explorer browsers[4].

Figure 11 presents the results of the experiment. It shows the execution overhead observed in each browser as a function of the size of the additional cold code added in each benchmark. At a high level, we see immediately that the addition of cold code affects the benchmark performance on the two browsers differently. In the case of Chrome (Figure 11a), adding two megabytes of cold code can add up to 450% overhead to the benchmark performance. In Internet Explorer (Figure 11b), cold code has much less impact.

In IE, the addition of 200 to 400 kilobytes does not impact its performance significantly. On average, we observe the overhead due to cold code of 1.8% and 3.2%, respectively. With 1 megabyte of cold code, the overhead is around 13%, still relatively small given the large amount of code being processed. In Chrome, on the other hand, even at 200 kilobytes, we observe quite a significant overhead, 25% on average across the six benchmarks. Even between the benchmarks on the same browser, the addition of cold code has widely varying effects (consider the effect of 1 megabyte of cold code on the different benchmarks in Chrome).

There are several reasons for these observed differences. First, because Chrome executes the benchmarks faster than IE, the additional fixed time processing the cold code will have a greater effect on Chrome's overall runtime. Second, Chrome and IE process JavaScript source differently, and large amounts of additional source, even if it is cold code, will have different effects on runtime. The important takeaway here is not that one browser processes cold code any better than another, but that results of benchmarks containing 1 megabyte of cold code will look different than results without the cold code. Furthermore, results with cold code are likely to be more representative of browser performance on real web sites.

[4]We use Chrome version 3.0.195.38 and Internet Explorer version 8.0.6001.18865. We collected measurements on a machine with a 1.2 GHz Intel Core Duo processor with 1.5 gigabytes of RAM, running 32-bit Windows Vista operating system.

5 Related Work

There are surprisingly few papers measuring specific aspects of JavaScript behavior, despite how widely used it is in practice. A concurrently submitted paper by Richards et al. measures static and dynamic aspects of JavaScript programs, much as we do [18]. Like us, their goals are to understand the behavior of JavaScript applications in practice, and specifically they investigate the degree of dynamism present in these applications (such as uses of eval). They also consider the behavior of JavaScript benchmarks, although this is not a major focus of the research. Unlike us, they do not consider the use of events in applications, or consider the size and effect of cold code.

One closely related paper focuses on the behavior of interpreted languages. Romer et al. [19] consider the runtime behavior of several interpreted languages, including Tcl, Perl, and Java, and show that architectural characteristics, such as cache locality, are a function of the interpreter itself and not the program that it is interpreting. While the goals are similar, our methods, and the language we consider (JavaScript), are very different.

Dieckmann and Hölzle consider the memory allocation behavior of the SPECJVM Java benchmarks [4]. A number of papers have examined the memory reference characteristics of Java programs [4, 14, 16, 20, 21] specifically to understand how hardware tailored for Java ex-

ecution might improve performance. Our work differs from this previous work in that we measure JavaScript and not Java, we look at characteristics beyond memory allocation, and we consider differences between benchmarks and real applications.

Dufour et al. present a framework for categorizing the runtime behavior of programs using precise and concise metrics [5]. They classify behavior in terms of five general categories of measurement and report measurements of a number of Java applications and benchmarks, using their results to classify the programs into more precise categories. Our measurements correspond to some metrics mentioned by Dufour et al., but we consider some dimensions of execution that they do not, such as event handler metrics, and compare benchmark behavior with real application behavior.

6 Conclusions

We have presented detailed measurements of the behavior of JavaScript applications, including commercially important web applications such as Gmail and Facebook, as well as the SunSpider and V8 benchmark suites. We measure two specific areas of JavaScript runtime behavior: 1) functions and code and 2) events and handlers. We find that the benchmarks are not representative of many real web sites and that conclusions reached from measuring the benchmarks may be misleading.

Our results show that JavaScript web applications are large, complex, and highly interactive programs. While the functionality they implement varies significantly, we observe that the real applications have much in common with each other as well. In contrast, the JavaScript benchmarks are small, and behave in ways that are significantly different than the real applications. We have documented numerous differences in behavior, and we conclude from these measured differences that results based on the benchmarks may mislead JavaScript engine implementers.

Furthermore, we observe interesting behaviors in real JavaScript applications that the benchmarks fail to exhibit. Our measurements suggest a number of valuable follow-up efforts. These include working on building a more representative collection of benchmarks, modifying JavaScript engines to more effectively implement some of the real behaviors we observed, and building developer tools that expose the kind of measurement data we report.

Acknowledgments

We thank Corneliu Barsan, Trishul Chilimbi, David Detlefs, Leo Meyerovich, Karthik Pattabiraman, David Simmons, Herman Venter, and Allen Wirfs-Brock for their support and feedback during the course of this research. We thank the anonymous reviewers for their feedback, and specifically Wilson Hsieh, who made a number of concrete and helpful suggestions.

References

[1] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2:313–351, 1995.

[2] W. W. W. Consortium. Document object model (DOM). `http://www.w3.org/DOM/`.

[3] D. Crockford. JSMin: The JavaScript minifier. `http://www.crockford.com/javascript/jsmin.html`.

[4] S. Dieckmann and U. Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In *Proceedings of European Conference on Object Oriented Programming*, pages 92–115, July 1999.

[5] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *SIGPLAN Not.*, 38(11):149–168, 2003.

[6] ECMA International. ECMAScript language specification. Standard ECMA-262, Dec. 1999.

[7] C. Foster. JSCrunch: JavaScript cruncher. `http://www.cfoster.net/jscrunch/`.

[8] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 465–478, 2009.

[9] Google. V8 JavaScript engine. `http://code.google.com/apis/v8/design.html`.

[10] Google. V8 benchmark suite - version 5. `http://v8.googlecode.com/svn/data/benchmarks/v5/run.html`, 2009.

[11] A. T. Holdener, III. *Ajax: The Definitive Guide.* O'Reilly, 2008.

[12] I. T. Jolliffe. *Principal Component Analysis.* Series in Statistics. Springer Verlag, 2002.

[13] G. Keizer. Chrome buries Windows rivals in browser drag race. `http://www.computerworld.com/s/article/9138331/Chrome_buries_Windows_rivals_in_browser_drag_race`, 2009.

[14] J.-S. Kim and Y. Hsu. Memory system behavior of Java programs: methodology and analysis. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 264–274, 2000.

[15] B. Livshits and E. Kiciman. Doloto: code splitting for network-bound Web 2.0 applications. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 350–360, 2008.

[16] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Trans. Computers*, 50(2):131–146, 2001.

[17] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn. JSMeter: Characterizing real-world behavior of JavaScript programs. Technical Report MSR-TR-2009-173, Microsoft Research, Dec. 2009.

[18] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10)*, pages 1–12, 2010.

[19] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The structure and performance of interpreters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, Oct. 1996.

[20] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 194–205, 2001.

[21] T. Systä. Understanding the behavior of Java programs. In *Proceedings of the Working Conference on Reverse Engineering*, pages 214–223, 2000.

[22] D. Unger and R. B. Smith. Self: The power of simplicity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242, Dec. 1987.

[23] WebKit. Sunspider JavaScript benchmark, 2008. `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`, 2008.

[24] Wikipedia. Browser wars. `http://en.wikipedia.org/wiki/Browser_wars`, 2009.

JSZap: Compressing JavaScript Code

Martin Burtscher
University of Texas at Austin
burtscher@ices.utexas.edu

Benjamin Livshits and Benjamin G. Zorn
Microsoft Research
{livshits,zorn}@microsoft.com

Gaurav Sinha
IIT Kanpur
gsinha@iitk.ac.in

Abstract

JavaScript is widely used in web-based applications, and gigabytes of JavaScript code are transmitted over the Internet every day. Current efforts to compress JavaScript to reduce network delays and server bandwidth requirements rely on syntactic changes to the source code and content encoding using gzip. This paper considers reducing the JavaScript source to a compressed abstract syntax tree (AST) and transmitting it in this format. An AST-based representation has a number of benefits including reducing parsing time on the client, fast checking for well-formedness, and, as we show, compression.

With JSZAP, we transform the JavaScript source into three streams: AST production rules, identifiers, and literals, each of which is compressed independently. While previous work has compressed Java programs using ASTs for network transmission, no prior work has applied and evaluated these techniques for JavaScript source code, despite the fact that it is by far the most commonly transmitted program representation.

We show that in JavaScript the literals and identifiers constitute the majority of the total file size and we describe techniques that compress each stream effectively. On average, compared to gzip we reduce the production, identifier, and literal streams by 30%, 12%, and 4%, respectively. Overall, we reduce total file size by 10% compared to gzip while, at the same time, benefiting the client by moving some of the necessary processing to the server.

1 Introduction

Over the last decade, JavaScript has become the lingua franca of the web, meaning that increasingly large JavaScript applications are being delivered to users over the wire. The JavaScript code of large applications such as Gmail, Office Web Apps, and Facebook amounts to several megabytes of uncompressed and hundreds of kilobytes of compressed data.

This paper argues that the current Internet infrastructure, which transmits and treats executable JavaScript as files, is ill-suited for building the increasingly large and complex web applications of the future. Instead of using a flat, file-based format for JavaScript transmission, we advocate the use of a hierarchical, abstract syntax tree-based representation.

Prior research by Franz et al. [4, 22] has argued that switching to an AST-based format has a number of valuable benefits, including the ability to quickly check that the code is well-formed and has not been tampered with, the ability to introduce better caching mechanisms in the browser, etc. However, if not engineered properly, an AST-based representation can be quite heavy-weight, leading to a reduction in application responsiveness.

This paper presents JSZAP, a tool that generates and compresses an AST-based representation of JavaScript source code, and shows that JSZAP outperforms de facto compression techniques such as gzip [13] by 10% on average. The power of JSZAP comes from the fact that JavaScript code conforms to a well-defined grammar [15]. This allows us to represent the grammar productions separately from the identifiers and the literals found in the source code so that we can apply different compression techniques to each component.

With JSZAP, we transform the JavaScript source into three streams: AST production rules, identifiers, and literals, each of which is compressed independently. While previous work has considered compressing Java programs for network transmission [4, 21, 22, 38], no prior work has considered applying these techniques to JavaScript. We show that in JavaScript the literals and identifiers constitute the majority of the total file size and describe techniques that compress each stream effectively.

1.1 Contributions

This paper makes the following contributions.

- It demonstrates the benefits of an AST-based JavaScript program representation, which include the ability to parse code in parallel [25], the potential to remove blocking HTML parser operations, and the opportunity for better caching, leading to more responsive web applications.
- It introduces JSZAP, the first grammar-based compression algorithm for JavaScript. JSZAP represents productions, identifiers, and literals as independent streams and uses customized compression strategies for each of them.
- It evaluates JSZAP on nine JavaScript programs, covering various program sizes and application domains, and ranging in size between about 1,000 to 22,000 lines of code. We conclude that JSZAP is able to compress JavaScript code 10% better than gzip. JSZAP compression ratios appear to apply across a wide range of JavaScript inputs.

1.2 Paper Organization

The rest of this paper is organized as follows. Section 2 provides background on JavaScript and AST-based program representation. Section 3 gives an overview of AST compression and Section 4 goes into the technical details of our JSZAP implementation. Section 5 presents the evaluation methodology and our experimental results. Section 6 describes related work and Section 7 concludes.

2 Background

This section covers the fundamentals of how JavaScript-based web applications are constructed and advocates an AST representation as a transfer format for JavaScript.

2.1 Web Application Background

Over the last several years, we have witnessed the creation of a new generation of sophisticated distributed Web 2.0 applications as diverse as Gmail, Bing Maps, Redfin, MySpace, and Netflix. A key enabler for these applications is their use of client-side code—usually JavaScript executed within the web browser—to provide a smooth and highly responsive user experience while the rendered web page is dynamically updated in response to user actions and client-server interactions. As the sophistication and feature sets of these web applications grow, downloading their client-side code is increasingly becoming a bottleneck in both initial startup time and subsequent application reaction time. Given the importance of performance and instant gratification in the adoption of applications, a key challenge thus lies in maintaining and improving application responsiveness despite increased code size.

Indeed, for many of today's popular Web 2.0 applications, client-side components already approach or exceed one megabyte of (uncompressed) code. Clearly, having the user wait until the *entire* code base has been transferred to the client before execution can commence does not result in the most responsive user experience, especially on slower connections. For example, over a typical 802.11b wireless connection, the simple act of opening an email in a Hotmail inbox can take 24 seconds on the first visit. The second visit can still take 11 seconds—even after much of the static resources and code have been cached. Users on dial-up, cell phone, or other slow networks see much worse latencies, of course, and large applications become virtually unusable. Bing Maps, for instance, takes over 3 minutes to download on a second (cached) visit over a 56k modem. (According to a recent Pew research poll, 23% of people who use the Internet at home rely on dial-up connections [30].) In addition to increased application responsiveness, reducing the amount of code needed for applications to run has the benefit of reducing the overall download size, which is important in mobile and some international contexts, where network connectivity is often paid per byte instead of a flat rate.

From the technical standpoint, a key distinguishing characteristic of Web 2.0 applications is the fact that code executes both on the client, within the web browser, and on the server, whose capacity ranges from a standalone machine to a full-fledged data center. Simply put, today's Web 2.0 applications are effectively sophisticated distributed systems, with the client portion typically written in JavaScript running within the browser. Client-side execution leads to faster, more responsive client-side experience, which makes Web 2.0 sites shine compared to their Web 1.0 counterparts.

In traditional web applications, execution occurs *entirely* on the server so that every client-side update within the browser triggers a round-trip message to the server, followed by a refresh of the entire browser window. In contrast, Web 2.0 applications make requests to fetch only the data that are necessary and are able to repaint individual portions of the screen. For instance, a mapping application such as Google Maps or Bing Maps may only fetch map tiles around a particular point of interest such as a street address or a landmark. Once additional bandwidth becomes available, such an application may use speculative data prefetch; it could push additional map tiles for the surrounding regions of the map. This is beneficial because, if the user chooses to move the map around, surrounding tiles will already be available on the client side in the browser cache.

However, there is an even more basic bottleneck associated with today's sophisticated Web 2.0 applications: they contain a great deal of code. For large applications such as Bing Maps, downloading as much as one megabyte of JavaScript code on the first visit to the front page is not uncommon [27]. This number is for the initial application download; often even more code may be needed as the user continues to interact with the application. The opportunity to optimize this large amount of code motivates our interest in JSZAP.

2.2 Benefits of an AST-based Representation

Franz's Slim Binaries project was the first to propose transmitting mobile code in the form of an abstract syntax tree [22]. In that project, Oberon source programs were converted to ASTs and compressed with a variant of LZW [40] compression. In later work, Franz also investigated the use of ASTs for compressing and transmitting Java programs [4, 21, 38].

Since this original work, JavaScript has become the de facto standard for transmission of mobile code on the web. Surprisingly, however, no one has investigated applying Franz's techniques to JavaScript programs. Below we list the benefits of AST-based representation, both those proposed earlier as well as unique opportunities present only in the context of web browsers.

Well-formedness and security. By requiring that JavaScript be transferred in the form of an AST, the browser can easily and quickly enforce important code properties. For instance, it can ensure that the code will parse or that it belongs to a smaller, safer JavaScript subset such as ADSafe [10]. Furthermore, simple code signing techniques can be used to ensure that the code is not being tampered with, which is common according to a recent study [34].

Caching and incremental updates. It is typical for large Internet sites to go through many small code revisions. This kind of JavaScript code churn results in cache misses, followed by code retransmission: the browser queries the server to see if there are any changes to a particular JavaScript file, and if so, requests a new version of it. Instead of redelivering entire JavaScript files, however, an AST-based approach provides a more natural way to allow fine-grained updates to individual functions, modules, etc. While unparsed source text can also be incrementally updated by specifying source ranges, AST updates can be guaranteed to preserve well-formed code with only local tree updates, while source-level updates cannot. Source-level updates may require the entire source to be parsed again once an update has been received.

Unblocking the HTML parser. The HTML parser has to parse and execute JavaScript code synchronously, because JavaScript execution can, for instance, inject additional HTML markup into the existing page. This is why many pages place JavaScript towards the end so that it can run once the rest of the page has been rendered. An AST-based representation can explicitly represent whether the code contains code execution or just code declaration, as is the case for most JavaScript libraries. Based on this information, the browser should be able to unblock the HTML parser, effectively removing a major bubble in the HTML parser pipeline.

Compression. This paper shows that an AST-based representation can be used to achieve better compression for JavaScript, reducing the amount of data that needs to be transferred across the network and shortening the processing time required by the browser to parse the code.

While some of the benefits mentioned can also be obtained by extending the existing source-based transmission method, we argue the if changes are required, then an AST-based approach is both more natural and more efficient to use than adding ad hoc mechanisms onto the existing techniques.

2.3 JavaScript Compression: State of the Art

Currently, the most commonly used approach to JavaScript compression is to "minify" the source code by removing superfluous whitespace. JSCrunch [20] and JSMin [11] are some of the more commonly used tools for this task. Some of the more advanced minifiers attempt to also rename variables to use shorter identifiers for temporaries, etc. In general, such renaming is difficult to perform soundly, as the prevalence of dynamic constructs like `eval` makes the safety of such renaming difficult to guarantee. When considered in isolation, however, minification generally does not produce very high compression ratios.

After minification, the code is usually compressed with gzip, an industry-standard compression utility that works well on source code and can eliminate much of the redundancy present in JavaScript programs. Needless to say, gzip is not aware of the JavaScript program structure and treats it as it would any other text file. On the client machine, the browser proceeds to decompress the code, parse it, and execute it.

To better understand the benefit of minification over straight gzip compression of the original source, we did the following experiment: for each of the benchmarks listed in Table 1, we either obtained the original unminified source if it was available, or we created a pretty-printed version of the source from the original minified source. These files approximate what the original source contained prior to minification (not including the comments). We then compressed the pretty-printed source and the minified source (created using the tool JSCrunch)

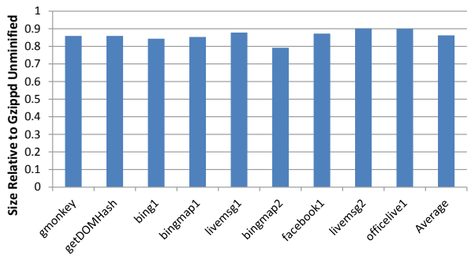


Figure 1: Relative benefit of using minification before gzip compression.

with gzip and compared the resulting file sizes. The results of this experiment are presented in Figure 1. The figure shows that the overall benefit that a programmer gets from using JSCruch prior to gzip is between 10 and 20%. Because minification is widely used in web applications, we conclude that JavaScript file size reductions on the order of 10-20% would be of interest to many web developers.

3 Compression Overview

This section gives an overview of the JSZAP approach, with Section 4 focusing on the details.

3.1 JavaScript Compression

JavaScript code, like the source code of most other high-level programming languages, is expressed as a sequence of characters that has to follow a specific structure to represent a valid program. This sequence can be broken down into tokens, which consist of keywords, predefined symbols, whitespace, user-provided constants, and user-provided names. Keywords include strings such as `while` and `if`. Symbols are operators such as − and ++ as well as semicolons, parentheses, etc. Whitespace typically includes all non-printable characters but most commonly refers to one or more space (blank) or tab characters. User-provided constants include hardcoded string, integer, and floating-point values. User-provided identifiers are variable names, function names, and so on.

The order in which these tokens are allowed to appear is defined by the syntax rules of the JavaScript grammar [15]. For instance, one such rule is that the keyword `while` must be followed by an opening parenthesis that is optionally preceded by whitespace. These syntax rules force legal programs to conform to a strict structure, which makes JavaScript code compressible. For example, the whitespace and the opening parenthesis after the keyword `while` are only there to make the code look more appealing to the programmer. They can safely be omitted in a compressed version of the source code because the uncompressed source code can easily be regenerated from the compressed form by inserting an opening parenthesis after every occurrence of the word `while` (outside of string constants).

Because the compressed code is not directly executable but must first be decompressed, crunching tools like JSCrunch [20] and JSMin [11] do not go this far. They primarily focus on minimizing whitespace, shortening local variable names, and removing comments. As in the `while` example above, whitespace is often optional and can be removed. Comments can always be removed. Local variables can be arbitrarily renamed without changing the meaning of the program as long as they remain unique and do not collide with a reserved word or global name that needs to be visible. Crunching tools exploit this fact and rename local variables to the shortest possible variable names such as `a`, `b`, `c`, etc. The resulting code is compressed because it is void of comments and unnecessary whitespace such as indentation and uses short but meaningless variable names, making it hard to read for humans.

If we are willing to forego direct execution, i.e., to introduce a decompression step, we can achieve much higher compression ratios than crunching tools are capable of achieving. For example, general-purpose compressors such as gzip are often able to further compress crunched JavaScript programs by a large amount. In the case of gzip, recurring character sequences are compressed by replacing later occurrences with a reference to an earlier occurrence. These references, which specify the position and length of the earlier occurrence, and the remaining symbols are then encoded using an adaptive Huffman scheme [19, 23] to minimize the number of bits required to express them. This way, keywords and longer recurring sequences such as `while(a < b)` can be compressed down to just a few bits. As mentioned, gzip compression of JavaScript and other files is so successful that many web servers and browsers provide support for it, i.e., files are transparently compressed by the server and decompressed by the client. Nevertheless, gzip was not designed for JavaScript. Rather, it was designed as a general-purpose text compressor, making it possible to compress JavaScript even better with a special-purpose compressor like JSZAP that takes advantage of specific properties such as the structure imposed by the grammar.

3.2 AST-based Compression

One way to expose the structure in JavaScript programs is to use a parser, which breaks down the source code into an abstract syntax tree (AST) whose nodes contain the tokens mentioned above. The AST specifies the order in which the grammar rules have to be applied to obtain the

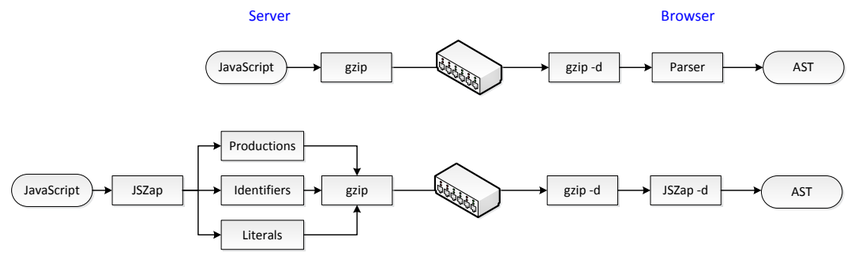


Figure 2: Architecture of JSZAP (bottom) compared to the current practice (top).

program at hand. In compiler terminology, these rules are called productions, the constants are called literals, and the variable and function names are called identifiers. Thus, the use of a parser in JSZAP makes it possible to extract and separate the productions, identifiers, and literals that represent a JavaScript program. Figure 2 illustrates this process, including further compression with gzip. The top portion of the figure shows the typical current process; the bottom portion illustrates the JSZAP approach of breaking down the code into multiple streams and compressing them separately.

Figure 3 provides an example of how a small piece of JavaScript code is converted into these three data streams. The productions are shown in linearized format. The figure illustrates that the three categories exhibit very different properties, making it unlikely that a single compression algorithm will be able to compress all of them well. Instead, JSZAP applies different compression techniques to each category to maximize the compression ratio. Each compressor is designed to maximally exploit the characteristics of the corresponding category, as explained in the next section. Figure 4 shows that each category represents a significant fraction of the total amount of data, meaning that all three categories must be compressed well to obtain good overall compression. The figure shows results for our nine benchmarks, ordered in increasing size, ranging from 17 kilobytes to 668 kilobytes (see also Table 1). The fraction of each kind of data is consistent across the programs, with a slight trend towards larger files having a larger fraction of identifiers and a smaller fraction of literals.

4 JSZap Design and Implementation

Because data sent over the Internet are typically compressed with a general compression algorithm like gzip, we not only want to determine how to best compress ASTs but also how to do it in a way that complements this preexisting compression stage well.

Such a two-stage compression scheme has interesting

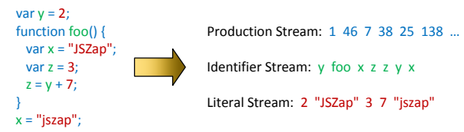


Figure 3: A simple JavaScript example.

implications. For example, to optimize the overall compression ratio, it is not generally best to compress the data as much as possible in the first stage because doing so obfuscates patterns and makes it difficult for the second stage to be effective. In fact, the best approach may be to *expand* the amount of data in the first stage to better expose patterns and thereby making the second stage as useful as possible [6]. We now describe how JSZAP exploits the different properties of the productions, identifiers, and literals to compress them well in the presence of a gzip-based second compression stage.

4.1 Compressing Productions

A JavaScript abstract syntax tree consists of non-terminal and terminal nodes from a JavaScript grammar. The JavaScript grammar we used for JSZAP is a top-down LALR(k) grammar implemented using Visual Parse++ [36]. The grammar has 236 rules, meaning that each terminal and non-terminal in the grammar can be encoded as a single byte. Because we are starting with a tree, there are two approaches to compressing data in this form: either convert the tree to a linear form (such as doing a pre-order traversal) and compress the sequence, or compress the tree directly. Our first approach was to reduce the tree to a linear form, attempt optimizations on the sequence, and then use gzip as a final compression stage. We attempted to compress the linear sequence using production renaming, differential encoding, and by removing chain productions.

Production renaming. Production renaming attempts to

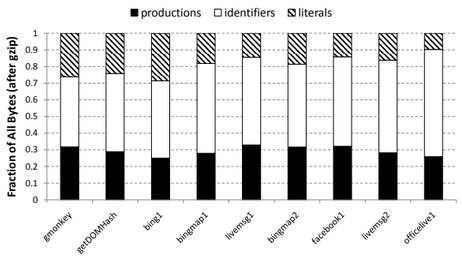


Figure 4: Breakdown of component types in our benchmarks (after gzip).

change the assignments of productions to integers (e.g., the production `Program => SourceElements` might be represented by the integer 225). We might choose to rename this production to integer 1 instead, if it was a common production in our streams. The idea behind renaming is to maximize the frequency of small production IDs. However, gzip is insensitive to absolute values as it only exploits repeating patterns, which is why this transformation does not help.

Differential encoding. Differential encoding works based on the observation that only a few productions can follow a given production. Hence, we renamed the possible following n productions to 0 through $n-1$ for each production. For example, if production x can only be followed by the two productions with IDs 56 and 77, we would rename production 56 to 0 and production 77 to 1. Differential encoding can potentially help gzip because it reduces the number of unique bytes and increases their frequency, but it is unclear whether this results in longer or more frequent repeating patterns that gzip can exploit.

Chain rule. Some productions always follow one specific production. For such chains of productions, it suffices to record only the first production. While this approach substantially reduces the amount of data emitted, it does not help gzip because it only makes the repeating patterns shorter. Because gzip uses an adaptive Huffman coder to encode the lengths of the patterns, not much if anything is gained by this transformation. Moreover, differential encoding and chain production removal are antagonistic. By removing the chains, the number of symbols that can follow a specific symbol often increases.

Overall, the techniques we investigated to compress linearized production streams are not effective. Nevertheless, chain production removal is quite useful when compressing the productions in tree form, as the following subsection explains.

4.1.1 Compressing Productions in AST Format

We found that productions are more compressible in tree format. We believe the reason for this to be the following. Assume a production with two symbols on the right-hand-side, e.g., an `if` statement with a `then` and an `else` block. Such a production always corresponds to a node and its two children in the AST, no matter what context the production occurs in. In linearized form, e.g., in a pre-order traversal of the tree, the first child appears right after the parent, but the second child appears at an arbitrary distance from the parent where the distance depends on the size of the subtree rooted in the first child (the size of the `then` block in our example). This irregularity makes it difficult for any linear data model such as gzip's to anticipate the second symbol and therefore to achieve good compression.

Compressing the productions in tree form eliminates this problem. The children of a node can always be encoded in the context of the parent, making it easier to predict and compress the productions. The only additional piece of information needed is the position of the child since each child of a node has the same parent, grandparent, etc. In other words, we need to use the path from the root to a node as context for compressing that node plus information about which child it is. Without the position information, all children of a node would have the same context.

One powerful context-based data compression technique is prediction by partial match (PPM) [8]. PPM works by recording, for each encountered context, what symbol follows so that the next time the same context is seen, a lookup can be performed to provide the likely next symbols together with their probability of occurrence. The maximum allowed context length determines the size of the lookup table. We experimentally found a context length of one, i.e., just using the parent and the empty context, to yield the best performance after chain-production removal. Aside from maximizing the compression ratio, using short contexts also reduces the amount of memory needed for table space and makes decompression very fast, both of which are important when running on a constrained client such a cell phone.

Since the algorithm may produce a different prediction for the empty context (a single table) and the order-1 context (one table per possible parent ID), we need to specify what to do if this happens. We use a PPM scheme that incorporates ideas from PPMA and PPMC [29], which have been shown to work well in practice. JSZAP's scheme always picks the longest context that has occurred at least once before, defaulting to the empty context if necessary. Because our tree nodes can have up to four children, JSZAP uses four distinct PPM tables, one for each child position. For each context, the tables record how often each symbol follows. PPM then predicts the next symbol with a probability that is proportional to its frequency and uses an arithmetic coder [35] to compactly encode which symbol it actually is. This approach is so powerful that further compression with gzip is useless.

To ensure that each context can always make a prediction, the first-order contexts include an escape symbol, which is used to indicate that the current production has not been seen before and that the empty context needs to be queried. The frequency of the escape symbol is fixed at 1 (like in the PPMA method), which we found to work best. JSZAP primes the empty context with each possible production, which is to say that they are all initialized with a frequency of one. This way, no escape symbol is necessary. Unlike in conventional PPM implementations, where an order -1 context is used for this purpose, we opted to use the empty (i.e., order 0) context because it tends to encounter most productions relatively quickly in any case.

To add aging, which gives more weight to recently seen productions, JSZAP scales down all frequency counts by a factor of two whenever one of the counts reaches a predefined maximum (as is done in the PPMC method). We found a maximum of 127 to work best. JSZAP further employs update exclusion, that is, the empty context is not updated if the first-order context was able to predict the current production. Finally, and unlike most other PPM implementations, JSZAP does not need to encode an end-of-file symbol or record the length of the file because decompression automatically terminates when the complete tree has been recreated.

4.2 Compressing Identifiers

The identifiers are emitted in the order in which the parser encounters them. We considered several transformations to reduce the size of this identifier stream. First, the same identifiers are often used repeatedly. Second, some identifiers occur more often than others. Third, many identifier names are irrelevant.

Symbol tables. To exploit the fact that many identifiers appear frequently, JSZAP records each unique identifier in a symbol table and replaces the stream of identifiers by indices into this table. Per se, this transformation does not shrink the amount of data, but it enables the following optimizations.

At any one time, only a few identifiers are usually in scope. Hence, it is advantageous to split the symbol table into a global scope table and several local scope tables. Only one local scope table is active at a time, and function boundary information, which can be derived from the productions, is used to determine when to switch local scope tables. The benefit of this approach is that only a small number of indices are needed to specify the identifiers. Moreover, this approach enables several important additional optimizations. For instance, we can sort the global table by frequency to make small offsets more frequent.

Symbol table sorting. Because not all identifiers appear equally often, it pays to sort the symbol table from most to least frequently used identifier. As a result, the index stream contains mostly small values, which makes it more compressible when using variable-length encodings, which JSZAP does.

Local renaming. The actual names of local variables are irrelevant because JSZAP does not need to reproduce the variable names at the receiving end. One can rename local variables arbitrarily as long as uniqueness is guaranteed and there are no clashes with keywords or global identifiers. As mentioned, one can assign very short names to local variables, such as `a`, `b`, `c`, etc., which is what many of the publicly available minifiers and JavaScript-generating compilers do.

Renaming allows JSZAP to use a built-in table of common variable names to eliminate the need to store the names explicitly. Consequently, most local scopes become empty and the index stream alone suffices to specify which identifier is used. (Essentially, the index is the variable name.) Note that JSZAP does not rename global identifiers such as function names because external code may call these functions by name.

Variable-length encoding. Ideally, we would like to encode a symbol table index as a single byte. Unfortunately, because we can only address 256 values with a single byte, a table that includes all the global identifiers used in a typical JavaScript program would be too large. To overcome this drawback, we allow a variable-length encoding of table index sizes (one and two bytes), and encode the most common identifiers in a single byte. We subdivide the 256-values addressable with a byte into distinct categories: local built-in symbols (mentioned above), common local symbols, common global symbols, and an escape value. The escape value is used to encode the remaining categories of symbols (uncommon local symbols, uncommon global symbols, and symbols found in the enclosing local symbol table) into two bytes.

4.3 Compressing Literals

The literals are also generated in the order in which the parser encounters them. The stream of literals contains three types of redundancy that we have tried to exploit. First, the same literal may occur multiple times. Second, there are different categories of literals such as strings, integers, and floating-point values. Third, some categories include known pre- and postfixes such as quotes

around strings.

Symbol tables. We have attempted to take advantage of multiple occurrences of the same literal by storing all unique literals in a table and replacing the literal stream with a stream of indices into this table. Unfortunately, most literals occur only once. As a consequence, the index stream adds more overhead than the table of unique values saves, both with and without gzip compression. Thus, this approach expands instead of shrinks the amount of data.

Grouping literals by type. Exploiting the different categories of literals proved more fruitful, especially because the category can be determined from the productions, so no additional information needs to be recorded. JSZAP separates the string and numeric literals, which makes gzip more effective. For example, it is usually better to compress all strings and then all integer constants as opposed to compressing an interleaving of strings and integers.

Prefixes and postfixes. Eliminating known pre- and postfixes also aids the second compressor stage by not burdening it with having to compress unnecessary information and by transferring less data to it, which can make it faster and more effective because a larger fraction of the data fits into the search window [41]. The two optimizations JSZAP performs in this regard are removing the quotes around strings and using a single-character separator to delineate the literals instead of a newline, carriage-return pair. In practice, this optimization does not help much because gzip is largely insensitive to the length of repeating patterns.

5 Evaluation

In this section, we evaluate the performance of JSZAP using a variety of JavaScript source code taken from commercial web sites.

5.1 Experimental Setup

Table 1 provides a summary of information about the benchmarks we have chosen to test JSZAP on. Each of the nine benchmarks is a JavaScript file, with its size indicated in the table both in terms of the number of bytes and lines of code, after pretty-printing (columns 2 and 3).

Many of the inputs come from online sources, including Bing, Bing Maps, Microsoft Live Messenger, and Microsoft Office Live. Two of the smaller scripts (`gmonkey`, `getDOMHash`) are hand-coded JavaScript applications used in browser plug-ins. The source files vary in size from 17 kilobytes to 668 kilobytes—results show that 100 kilobytes is not an uncommon size for JavaScript source in a high-function web application like Bing or Bing Maps [33].

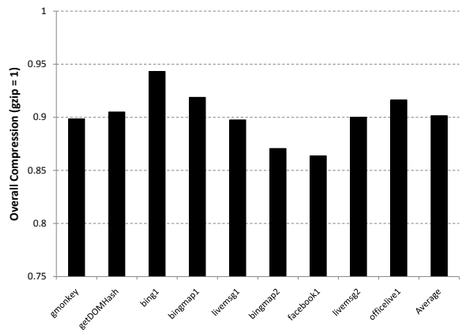


Figure 5: JSZAP overall compression relative to gzip. Note that the y-axis does not start at 0.

We processed each input by running JSCrunch on the source code before compressing with gzip and JSZAP (although in most cases this crunching had no effect because the code had already been crunched). We did not perform automatic local variable renaming with JSCrunch during our processing, because all but one of these files (`getDOMHash`) was received in a form that had been previous crunched with a tool such as JSCrunch or JSMin, meaning that the local variables had largely been renamed in the original source. Pretty-printing the sources results in files that range from 900 to 22,000 lines of code.

The size of the AST representation, composed of independent production, identifier, and literal streams, is shown in columns 4 and 5. Note that the AST is in fact already smaller than the corresponding JavaScript sources, by about 25% on average. This reduction results from elimination of source syntax such as keywords, delimiters, etc. The last two columns show the size of the gzipped representation and the gzip-to-source code ratio. In most cases, gzip compresses the source by a factor of three to five.

5.2 Total Compression

Figure 5 shows overall compression results of applying JSZAP to our benchmarks. We show the compression ratio of JSZAP compared to gzip. We see that in the majority of cases, the reduction in the overall size is 10% or more. It should be noted that JSZAP's AST-based representation already includes several processing steps such as parsing and semantic checking, thus reducing the amount of processing the client will have to do. Despite this fact, we are able to achieve compression ratios better than gzip.

Figure 5 demonstrates that the benefits of JSZAP compression are largely independent of the input size. There is no clear correlation of compression ratios and whether the source has been produced by a tool or framework. This leads us to believe that similar compression benefits can be obtained with JSZAP for a wide range of JavaScript sources. The input with the greatest compression, `facebook1`, is also the input with the most effective compression of the productions relative to gzip (see next section), suggesting that the PPM compression of productions is a central part of an effective overall compression strategy.

Benchmark name	Source bytes	Source lines	Uncompressed AST (bytes)	Uncompressed AST/src ratio	gzip bytes	gzip/source ratio
`gmonkey`	17,382	922	13,108	0.75	5,340	0.30
`getDOMHash`	25,467	1,136	17,462	0.68	6,908	0.27
`bing1`	77,891	3,758	65,301	0.83	23,454	0.30
`bingmap1`	80,066	3,473	56,045	0.69	19,537	0.24
`livemsg1`	93,982	5,307	70,628	0.75	22,257	0.23
`bingmap2`	113,393	9,726	108,082	0.95	41,844	0.36
`facebook1`	141,469	5,886	94,914	0.67	36,611	0.25
`livemsg2`	156,282	7,139	104,101	0.66	32,058	0.20
`officelive1`	668,051	22,016	447,122	0.66	132,289	0.19
Average				0.7432		0.2657

Table 1: Summary of information about our benchmarks.

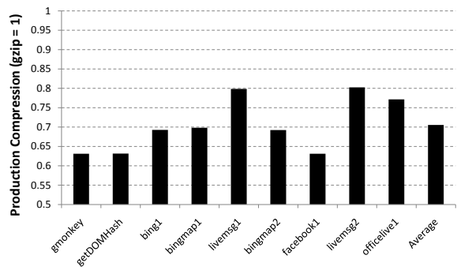


Figure 6: JSZAP production compression relative to gzip. Note that the y-axis does not start at 0.

5.3 Productions

Figure 6 shows the benefits of using PPM to compress the production stream. As we have discussed, the structured nature of the productions allows PPM compression to be very effective, producing a significant advantage over gzip. Just as before, we normalize the size produced using PPM compression relative to compressing the productions with gzip. We see that JSZAP compresses the productions 20% to over 35% better than gzip, with an average improvement of 30%. Again, JSZAP's compression benefits appear to be independent of the benchmark size.

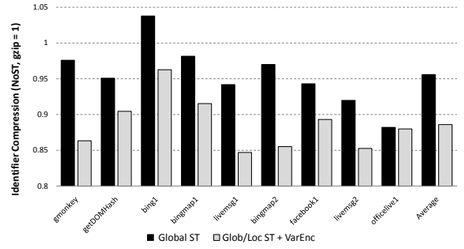


Figure 7: JSZAP identifier compression relative to gzip. Note that the y-axis does not start at 0.

We note that PPM compression can easily be changed with a number of control parameters. The results reported here are based on a context length of one and a maximum symbol frequency of 127. Varying these as well as other parameters slightly resulted in minor differences in overall compression, with individual file sizes changing by a few percent.

5.4 Identifiers

Figure 7 presents the results of applying JSZAP to compress the identifier stream. The figure shows results normalized to using gzip to compress the identifier stream without any symbol table being used. The figure includes two different symbol table encodings: a single global symbol table with a fixed-length 2-byte encoding (Global ST) as well as using both global and local symbol tables with variable-length encoding (Glob/Loc ST + VarEnc),

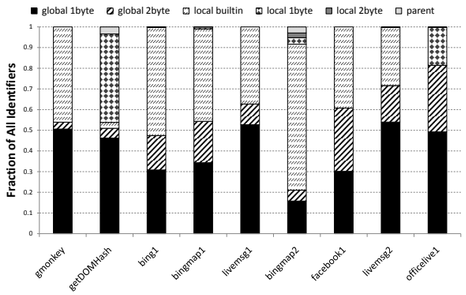


Figure 8: Breakdown of categories in variable-length identifier encoding.

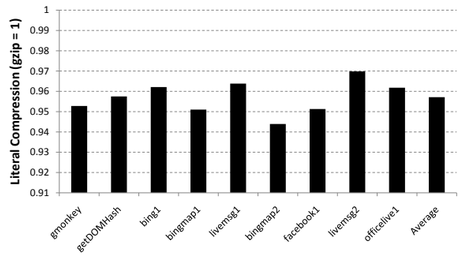


Figure 9: JSZAP literal compression relative to gzip. Note that the y-axis does not start at 0.

as described in Section 4.2. We observe that the 2-byte encoding generally produces only small benefits and in one case hurts overall compression. Using the variable encoding results in a 12% improvement overall. Interestingly, we see that our variable encoding provides no benefit over the global symbol table in our largest application, `officelive1`.

To further illustrate the effectiveness of our variable-length encoding, Figure 8 shows a breakdown of different encoding types for our benchmarks. From the figure, we see that our strategy to represent as many identifiers as possible with 1 byte succeeded, with the only major category of 2-byte identifiers being globals. We see that global 1- and 2-byte identifiers account for more than half the total identifiers in most applications. Local built-ins are also very common in the applications, especially for `bingmap2`, where they account for over 75% of all identifiers. `bingmap2` is also one of the applications where we get the greatest benefit relative to gzip in Figure 7. Figure 8 explains why `officelive1` does not benefit from our variable-length encoding in the previous figure. Because of the framework that was used to generate `officelive1`, we see that more than 80% of all identifiers are globals, and there are no local built-ins (`a`, `b`, `c`, etc.) We anticipate that if we tailored variable renaming appropriately by preprocessing `officelive1`, we could replace many of the local 1-byte identifiers with built-ins. `getDOMHash`, which was written by hand and did not have automatic variable renaming performed during crunching, also has many 1-byte local variables.

To conclude, we see that we obtain a relatively modest compression (12%) of the identifiers over gzip, but because gzip is designed explicitly to compress characters strings, this result is not surprising. We do find tailoring identifier compression using source-level information about local and global symbols to be beneficial.

5.5 Literals

Figure 9 shows the results of using JSZAP to process the literals before compressing them with gzip. As with the previous figures, we compare against compressing an unoptimized literal stream with gzip. Because literals are mostly single-use (except common ones such as 0, 1, etc.), using a literal table increases space usage over gzip and is not shown. Our other optimizations, including grouping literals by type and eliminating prefixes and postfixes have modest benefits, averaging around 4–5%.

6 Related Work

This section discusses compression in general and the specific work related to compressing JavaScript.

6.1 Mobile Code Compression

The work most closely related to JSZAP is the Slim Binaries and TransPROse projects by Franz et al. [4, 22]. Slim Binaries is the first project to promote the use of transmitting mobile code using abstract syntax trees for benefits including compression, code generation, and security. The original Slim Binaries project compressed Oberon ASTs using a variation of LZW compression [40] extended with abstract grammars. Later, in the TransPROse project [4, 21, 38], Franz et al. describe a new compression technique for ASTs based on abstract grammars, arithmetic coding, and prediction by partial match. They apply their technique to Java class files and compare the results against Pugh's highly-effective jar file compressor [31], as well as general compressors like gzip and bzip2. With PPM compression, they achieve a compression ratio of approximately 15% over the uncompressed source.

Our work is motivated by the goals of Franz's earlier work. We also share some of the techniques with that work, including PPM compression of the production rules. Our work differs in that we adapt and optimize these ideas to compressing JavaScript. We show that for real JavaScript source code, we achieve significant benefits over the current state-of-the-art. In addition, we show that identifiers and literals constitute a significant fraction of the data that requires compression, and describe JavaScript-specific techniques to compress these streams.

In a one-page abstract, Evans describes the compression of Java and Pascal programs based on *guided parsing*, which also uses the language grammar to make compression more efficient [17]. In another one-page abstract, Eck et al. propose Syntax-oriented Coding [14]. Guided parsing and SoC have many elements in common with Slim Binaries and TransPROse, but due to their shortness, both papers lack detail.

Other approaches to compressing mobile code have been proposed. Many of them focus on compressing a program representation that is close to the target language, specifically native machine code [16] or some form of bytecode [18, 28]. Some proposals consider dynamically compressing unused portions of code to reduce the in-memory footprint of the executing program [12]. The main difference between this work and ours is our focus on using the augmented AST as the medium of transfer between the server and client as well as our focus on compressing the tree itself instead of a linearized format, such as an instruction stream. While bytecode-based approaches have advantages, they also require agreement about what the best translation of the source to bytecode would be. Our approach follows the current JavaScript transfer model, and maintains the content of the source without assuming a translation to a lower-level representation.

Pugh considers ways to compress Java class files. He splits data into multiple streams using redundancies in the class file information and finds a number of format specific opportunities to achieve good compression [31]. Like our work, he examines opportunities to improve second-stage gzip compression, although he does not consider using the grammar to compress the program text. Jazz [5] and Clazz [24] also improve the representation of the entire Java archive but do not consider source compression.

6.2 Syntax-based Compression

The problem of compressing source code has been considered since the 1980s. The idea of using the program parse as a program representation and the grammar as a means of compressing the parse was proposed by Contla [9]. He applied the approach to Pascal source code and demonstrated compression on three small programs. Katajainen et al. describe a source program compressor for Pascal that encodes the parse tree and symbol tables [26]. They show that their Prolog implementation of the compressor results in space gains of 50–60%. Stone describes *analytic encoding*, which combines parsing with compression [37]. Stone considers how the parser used (LL versus LR) affects the resulting compressibility and reports that LR parsers are more appropriate, which is what JSZAP uses.

Cameron describes source compression using the language syntax as the data model [7]. He suggests using arithmetic coding to compress the production rules and separating the local and global symbols to improve the compression of symbols. In applying the technique to Pascal programs, he shows a result that is approximately 15% of the size of the original source. Tarhio reports the use of PPM compression on parse trees [39]. Applying the approach to four Pascal programs, he shows that the the number of bits per production can be reduced below more general purpose techniques such as gzip and bzip2.

Rai and Shankar consider compressing program intermediate code using tree compression [32]. They consider using tree grammars to encode the intermediate form (unlike work based on the source syntax) and show that they outperform gzip and bzip2 on lcc-generated intermediate files. They speculate that their technique could be applied to compression of XML-structured documents.

6.3 XML / Semistructured Text Compression

Adiego et al. describe LZCS [1, 2], a Lempel-Ziv-based algorithm to compress structured data such as XML files. Their approach takes advantage of repeated substructures by replacing them with a backward reference to an earlier occurrence. JSZAP employs the same general approach; it also transforms the original data and uses a second compression stage to maximize the overall compression ratio.

The same authors further describe the Structural Contexts Model (SCM) [3], which exploits structural information such as XML tags to combine data belonging to the same structure. The combined data are more compressible than the original data because combining brings data with similar properties together. JSZAP adopts the idea of separately compressing data with similar properties, i.e., identifiers, literals, and productions, to boost the compression ratio.

7 Conclusions

This paper advocates an AST-based representation for delivering JavaScript code over the Internet and presents

JSZAP, a tool that employs such an AST-based representation for compression. JSZAP compresses JavaScript code 10% better than gzip, which is the current standard. In the context of a high-traffic host serving gigabytes of JavaScript to thousands of users, the savings demonstrated by JSZAP may amount to hundreds of megabytes less to be transmitted. It is our hope that our work will inspire developers of next-generation browsers to reexamine their approach to representing, transmitting, and executing JavaScript code.

References

[1] J. Adiego, G. Navarro, and P. de la Fuente. Lempel-Ziv compression of structured text. pages 112–121, March 2004.

[2] J. Adiego, G. Navarro, and P. de la Fuente. Lempel-Ziv compression of highly structured documents: Research articles. *J. Am. Soc. Inf. Sci. Technol.*, 58(4):461–478, 2007.

[3] J. Adiego, G. Navarro, and P. de la Fuente. Using structural contexts to compress semistructured text collections. *Information Processing & Management*, 43(3):769–790, 2007. Special Issue on Heterogeneous and Distributed IR.

[4] W. Amme, P. S. Housel, N. Dalton, J. V. Ronne, P. H. Frohlich, C. H. Stork, V. Haldar, S. Zhenochin, and M. Franz. Project TRANSPROSE: Reconciling mobile-code security with execution efficiency. 2001.

[5] Q. Bradley, R. N. Horspool, and J. Vitek. JAZZ: an efficient compressed format for Java archive files. In S. A. MacKay and J. H. Johnson, editors, *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*, page 7. IBM, 1998.

[6] M. Burtscher, I. Ganusov, S. Jackson, J. Ke, P. Ratanaworabhan, and N. Sam. The VPC trace-compression algorithms. *IEEE Transactions on Computers*, 54(11):1329–1344, Nov. 2005.

[7] R. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, Jul 1988.

[8] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[9] J. F. Contla. Compact coding of syntactically correct source programs. *Softw. Pract. Exper.*, 15(7):625–636, 1985.

[10] D. Crockford. ADsafe. adsafe.org.

[11] D. Crockford. JSMin: The JavaScript minifier. `http://www.crockford.com/javascript/jsmin.html`.

[12] S. Debray and W. Evans. Profile-guided code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, June 2002.

[13] L. P. Deutsch. GZIP file format specification version 4.3. Internet RFC 1952, May 1996.

[14] P. Eck, X. Changsong, and R. Matzner. A new compression scheme for syntactically structured messages (programs) and its application to Java and the Internet. pages 542–, 1998.

[15] ECMA. ECMAScript language specification.

[16] J. Ernst, W. S. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In *PLDI*, pages 358–365, 1997.

[17] W. S. Evans. Compression via guided parsing. In *Proceedings of the Conference on Data Compression*, page 544, Washington, DC, USA, 1998. IEEE Computer Society.

[18] W. S. Evans and C. W. Fraser. Bytecode compression via profiled grammar rewriting. In *PLDI*, pages 148–155, 2001.

[19] N. Faller. An adaptive system for data compression. In *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, pages 593–597, 1973.

[20] C. Foster. JSCrunch: JavaScript cruncher. `http://www.cfoster.net/jscrunch/`.

[21] M. Franz, W. Amme, M. Beers, N. Dalton, P. H. Frohlich, V. Haldar, A. Hartmann, P. S. Housel, F. Reig, J. Ronne, C. H. Stork, and S. Zhenochin. Making mobile code both safe and efficient. In *Foundations of Intrusion Tolerant Systems*, Dec. 2002.

[22] M. Franz and T. Kistler. Slim binaries. *Commun. ACM*, 40(12):87–94, 1997.

[23] R. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978.

[24] R. N. Horspool and J. Corless. Tailored compression of Java class files. *Softw, Pract. Exper*, 28(12):1253–1268, 1998.

[25] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodk. Parallelizing the Web Browser. In *Proceedings of the Workshop on Hot Topics in Parallelism*. USENIX, March 2009.

[26] J. Katajainen, M. Penttonen, and J. Teuhola. Syntax-directed compression of program files. *Softw. Pract. Exper.*, 16(3):269–276, 1986.

[27] E. Kıcıman and B. Livshits. AjaxScope: a Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications. In *Proceedings of Symposium on Operating Systems Principles*, Oct. 2007.

[28] S. Lucco. Split-stream dictionary program compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 27–34, 2000.

[29] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, 1990.

[30] Pew Internet and American Project. Home broadband adoption 2007. `http://www.pewinternet.org/pdfs/PIP_Broadband\%202007.pdf`, 2007.

[31] W. Pugh. Compressing Java class files. In *PLDI*, pages 247–258, 1999.

[32] S. Rai and P. Shankar. Efficient statistical modeling for the compression of tree structured intermediate code. *Comput. J*, 46(5):476–486, 2003.

[33] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn. JSMeter: Characterizing real-world behavior of JavaScript programs. Technical Report MSR-TR-2002-138, Microsoft Research, Microsoft Corporation, 2009.

[34] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 31–44, 2008.

[35] J. J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, 1979.

[36] Sandstone Technologies Inc. Parsing with Sandstone's Visual Parse++. `http://visualparse.s3.amazonaws.com/pvpp-fix.pdf`, 2001.

[37] R. G. Stone. On the choice of grammar and parser for the compact analytical encoding of programs. *The Computer Journal*, 29(5):307–314, 1986.

[38] C. H. Stork, V. Haldar, and M. Franz. Generic adaptive syntax-directed compression for mobile code. Technical Report 00-42, Department of Infomation and Computer Science, University of California, Irvine, 2000.

[39] J. Tarhio. On compression of parse trees. pages 205–211, Nov. 2001.

[40] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[41] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

Leveraging Cognitive Factors in Securing WWW with CAPTCHA

Amalia Rusu and Rebecca Docimo

Fairfield University, Fairfield, CT, USA, arusu@fairfield.edu

Adrian Rusu

Rowan University, Glassboro, NJ, USA, rusu@rowan.edu

Abstract

Human Interactive Proofs systems using CAPTCHA help protect services on the World Wide Web (WWW) from widespread abuse by verifying that a human, not an automated program, is making a request. To authenticate a user as human, a test must be passable by virtually all humans, but not by computer programs. For a CAPTCHA to be useful online, it must be easy to interpret by humans. In this paper, we present a new method to combine handwritten CAPTCHAs with a random tree structure and random test questions to create a novel and more robust implementation that leverages unique features of human cognition, including the superior ability over machines in recognizing graphics and reading unconstrained handwriting text that has been transformed in precise ways. This combined CAPTCHA protects against advances in recognition systems to ensure it remains viable in the future without causing additional difficulties for humans.

We present motivation for our approach, algorithm development, and experimental results that support our CAPTCHA in protecting web services while providing important insights into human cognitive factors at play during human-computer interaction.

1. Introduction

Most users of the WWW today are familiar with CAPTCHAs, which are presented to them as machine-printed text or sound samples to be interpreted. CAPTCHAs are typically used to prevent automated programs from gaining access to various Web resources for the purpose of spamming or other illegitimate use. CAPTCHA is needed because of the sheer volume of spam crossing the Internet and the agility and tenacity of spammers [12].

Artificial Intelligence (AI) experts consider CAPTCHA a win-win situation and point out that CAPTCHAs are useful even when broken for the insights provided to the field of AI [29, 30]. While breaking CAPTCHAs can be useful for advancing the field of AI as well as Image Processing, Pattern Recognition, etc., the current usefulness of CAPTCHAs in protecting Web resources from widespread illegitimate use by automated programs must not be overlooked.

In this paper, we present the development of a new CAPTCHA-based Human Interactive Proofs (HIP) authentication system to protect services on the WWW. In our system, users are authenticated as humans to gain access to Web services by correctly interpreting a tree structure with handwriting samples transformed according to specific principles of cognitive psychology, explained in greater detail in the next sections (Figure 1). To correctly solve the challenge, the tree structure and handwriting samples must be segmented out and interpreted, a task that presents much difficulty for machines, while being trivial for humans. With this CAPTCHA, we further the work begun on handwritten CAPTCHAs [22, 23, 24]. As CAPTCHAs are currently a readily available, relatively low cost and easy to administer solution to protect Web resources, our goal is to provide a useful CAPTCHA to overcome the security and usability difficulties present in other CAPTCHAs [17, 21, 33, 34]. Such a CAPTCHA can also offer extremely valuable insights not only related to the parsing of handwriting. By using a tree structure with handwritten images, both of which must be parsed to pass the CAPTCHA test, we can also offer important insights for the fields of AI, Image Analysis, Graphics Recognition and others.

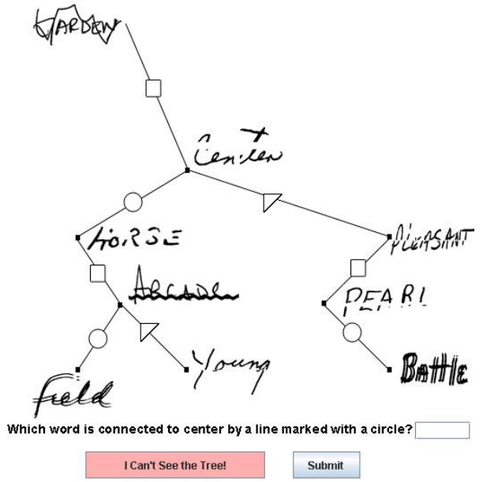


Figure 1. A tree-based handwritten CAPTCHA.

Tree drawings and handwriting are used in our CAPTCHA rather than the typical machine-printed text, both for the important advances to be gained in graphics and handwriting recognition fields if our CAPTCHA is broken, as well as for the security provided due to the extra challenges posed to machines. Human skill at interpreting basic drawings and handwriting, no matter the condition (i.e., rotated, occluded, or deformed) [20], is gained from an early age, while consistent machine recognition of graphics in general, and handwriting in particular, continues to be problematic mostly due to unconstrained writing style and segmentation, especially in the absence of a context [4, 7, 18, 26]. Moreover, when applying certain transformations to the handwriting and rendering it on a tree structure, the recognition drastically decreases. While humans are able to make use of certain aspects of perception and cognition to interpret transformed samples, this remains a difficult open problem for machines [9].

We begin by reviewing the concepts of both HIPs and CAPTCHAs. We then discuss advances in handwriting recognition and present human cognitive factors that relate to handwriting interpretation. In this context we introduce and discuss the Gestalt laws of perception and Geon theory related to human perception and reading skills to motivate the transformations we have applied to the images. The technical approach and methodology is then presented, as well as the findings of user studies and machine testing of our CAPTCHA to validate the usefulness of our system in protecting Web services. We conclude with important insights gained from our work and discuss possible future enhancements.

1.1 Overview of HIPs and CAPTCHA

The purpose of HIPs is to distinguish one class of users from another, most commonly humans from computer programs (also referred to as "machines") [1]. CAPTCHA is the test used by HIPs to distinguish a human user from a machine by presenting a challenge that can only be passed by the human. CAPTCHAs leverage AI factors and similarly to the tests of Alan Turing [28], they determine whether a user is human or machine. CAPTCHAs differ from Turing Tests, however, by making the computer the judge and administrator, though the computer itself should not be able to pass the test. In a CAPTCHA, if the individual completing the challenge presented passes correctly, they gain access to the service they are requesting. Otherwise, they are deemed to be an illegitimate program and are not allowed access. For a CAPTCHA to be useful, it must be easily passable by virtually all human users but not by machines [30]. If a CAPTCHA presents difficulty to machines, but also to humans, it has failed in its function [6, 21].

Primitive use of a commercial text-based riddle dates back to 1998 on the AltaVista search engine Web site (at altavista.com). Approximately two years later CAPTCHA was defined, along with the first commercial implementation by Carnegie Mellon University researchers. They set forth the basic properties of CAPTCHA: it must be automated so it can be administered by a computer program without human intervention, it must be public in that the test should not be unsolvable by machines simply because it is novel or the method or code is hidden, and it must be passable by virtually all humans but not by computer programs [30]. Many text-based visual CAPTCHAs have been created from Gimpy [10] to Baffletext [8] to reCAPTCHA [19]. An example text-based visual CAPTCHA is shown in Figure 2. Non-text visual CAPTCHAs have also been created, including those that leverage human cognitive abilities beyond word recognition, such as ARTiFACIAL [21] where users are asked to identify facial features.

Figure 2. Example of commercial CAPTCHA at msn.com.

1.2 Motivation for a Tree-based Handwritten CAPTCHA

While many CAPTCHAs have been created, more secure CAPTCHAs are needed to help secure the WWW against widespread abuse by programs posing as humans on the WWW. While a typical response to foil machine recognition to maintain the usefulness of CAPTCHAs is to make them harder for machines, care must be taken to ensure that human ease of use does not suffer. Accordingly, many current CAPTCHAs, both text and image based suffer from usability issues [6, 34]. Intrinsic security flaws of various CAPTCHAs have also been found and various text-based and image-based CAPTCHAs have been broken [7, 11, 33]. For example, Mori and Malik from the University of California at Berkeley demonstrate how they were able to break the EZ-Gimpy CAPTCHA with a 92% rate of success and Gimpy with a 33% rate of success [17].

It is the need for a more efficient CAPTCHA that is both usable for humans while secure against machines, along with the insights to be gained from persistent problems in computer recognition of handwritten text and graphics [5, 26] that motivates our approach. We combine a randomly generated tree structure with random test questions and mandatory interpretation of handwritten words transformed according to the Gestalt and Geon principles (Figure 1). This new challenge meets the criteria of being a CAPTCHA in that large quantities of human-like handwritten images can be automatically created via a synthetic handwriting tool [25] and then transformed and rendered on a tree structure. Our stringent adherence to the Gestalt and Geon principles related to human perception of objects, including letters and words, to create our transformations ensures that our new CAPTCHA is still easily solvable by humans while presenting challenges to computer programs. The tree-based CAPTCHA featuring deformed handwritten images (Figure 1) described in this paper addresses both security and usability aspects to create a viable alternative CAPTCHA to those in existence, while at the same time having the potential to add insights into the overlooked area of real time handwriting recognition and interpretation of complex multi-layer image based documents.

2. Technical Background

Challenge-response tests using visual recognition have been the most widely used type of CAPTCHA employed online to protect web services from abuse by automated programs. The purpose of this study and the approach developed is to expand the use of visual CAPTCHAs by inserting additional complexity for computers, while keeping the tests easy for humans to pass (again, if success rates for machines decrease but success rates for humans also decrease for a CAPTCHA, it is not viable [6]). In this context we discuss several factors that we have leveraged in our system.

2.1 Gestalt Principles and Geon Theory Factor

We have studied the Gestalt laws of perception and Geon theory and have used guiding principles of each to determine which very specific transformations can be applied to our handwriting samples to both assist human interpretation and present unique challenges to machine recognition. According to Gestalt principles, humans have a unique ability to make sense of pictures, even those that are incomplete or are marred in some way [15]. Humans are able to make sense of images presented to them by relying on their senses, past experience, which shapes how they view data currently, and what they are expecting to see. Humans are able to filter out irrelevant data such as noise or extra pieces in an image in order to interpret it. Gestalt principles are based on the fact that humans typically experience things that are outside of the range of simple perception. Humans tend to group information and interpret the whole rather than looking at individual pieces and then combining them. This is similar to the theory of holistic word recognition where the word is seen as an indivisible unit rather than as a series of individual parts which can be interpreted separately and then reassembled for recognition [16].

The Gestalt laws that aid human recognition of objects with transformations applied include proximity, similarity, symmetry, continuity, closure, familiarity and figure-ground as follows:

- Proximity: how objects are grouped together by distance from or location to each other.
- Similarity: how elements that are similar to each other tend to be viewed as part of a singular group.
- Symmetry: how objects are grouped into figures according to symmetry and meaning.
- Continuity: how objects are grouped according to flow of lines or alignment.

- Closure: how elements are grouped together if they tend to complete some pattern, allowing perception of objects that are visually absent.

- Familiarity: how elements are more likely to be interpreted as part of a group if they appear familiar to the viewer.

- Figure-ground distinction: how a scene is broken up into foreground (the object of interest) and background (the rest of the scene) which is what allows an object to be distinguished from its surroundings.

Other human cognitive factors at play in recognition are memory, internal metrics, familiarity of letters and letter orientation [22, 23, 24].

Human perception relies, in the end, on all of the Gestalt principles working together. In addition to using the Gestalt laws of perception to determine which transformations may be applied to CAPTCHAs to capitalize on machine recognition weaknesses and simultaneous human strengths, the Geon theory of pattern recognition is also useful to determine which core components must be present in a transformed image so that it is still interpretable by humans. Two key aspects of geons are edges and intersections. The importance of these has been tested on images where various parts were deleted [3]. Recognition for humans is easy if an object's geons can be recognized and edges and intersections are a critical part in recognition. We have made use of the Gestalt principles and Geon theory in development of our CAPTCHA through specific handwriting image transformations to ensure human legibility while foiling machines. Similar transformations can successfully be applied to the tree structure as well as any shape or object in general.

2.2 Handwriting Recognition Factor

Recognition of unconstrained handwriting, especially when it has certain transformations applied to it, continues to be a challenge for automatic recognition systems [26] while humans maintain a superior ability due to the Gestalt laws of perception [15] and Geon theory [2, 3]. Part of the problem for machines is that natural variability in handwriting exists at a level that does not exist in machine-printed text [26]. Our tests show that natural handwriting variability, as well as defects applied such as occlusions or fragmentations (Figure 4b), can currently be overcome by humans due to cognitive factors, but not by machines. Segmentation, or the ability for a machine to determine character and word boundaries, continues to be a problem [7, 18, 26]. Handwriting presents more segmentation issues for machines than machine-printed text making handwriting arguably superior to machine-printed text for use in a secure CAPTCHA.

While advances have been made in handwriting recognition and applications have found their place in certain contexts such as the US postal services [27], or bank check reading, these contexts are usually well known in the sense that a relatively small set of words is being used in a familiar and narrow context. Existing handwriting recognition approaches require a lexicon (as a dictionary or pre-determined list of words and expressions in a particular language used by a particular application), for high recognition accuracy [13, 14, 31, 32]. In our application to CAPTCHA, the use of words is infinite with no specific context, thus the required lexicon would have to be extremely large with consequently extremely poor accuracy by recognizers. Moreover, by applying very specific transformations that exploit the weaknesses of state-of-the-art recognition systems to our image samples on purpose, we add extra difficulty for machine recognition.

2.3 Graphics Recognition Factor

While advances have been made in the area of document image analysis, various open problems of interest remain. One key open issue is the lack of a general purpose cross-domain recognition tool. Most tools are very domain specific and require domain context [5] or a case-based approach [35] to interpret a particular graphic. For example, tools used to recognize domain-specific graphics such as electrical diagrams rely on primitives in the graphic that have intrinsic meaning. In musical scores, for example, the musical notes comprise a finite set of primitives that can be extracted and interpreted because they have a meaning apart from the whole graphic [5]. Once the primitives are interpreted, the graphic as a whole can be interpreted. We will discuss later how our tree drawing does not rely on any particular domain context and thus would be hard for machines to interpret. No parts of our tree have any intrinsic meaning and are always interpreted in the context of reading of deformed handwritten images.

3. Generation of Tree-based Handwritten CAPTCHA

The development of our CAPTCHA has focused on using transformed handwriting samples due to the aid provided by cognitive principles that humans can make use of and the challenges presented to machines. We have increased difficulty for machines as well as the potential for insights in the area of graphics recognition by arranging our handwritten images into a tree structure. It should be noted that in our CAPTCHA the interpretation of the tree structure will always be combined with interpreting our handwritten images. There are two main parts to the creation of our combined CAPTCHA. First, images featuring synthetic handwritten are generated using a handwriting generator [25]. Gestalt and Geon-motivated transformations are then applied to the images to take advantage of human perception abilities and to create more difficulty for machines. Second, there is the random generation of a tree structure and tree elements and arrangement of the deformed handwritten images in the tree, making spatial recognition, which is a common task for humans, also part of the challenge. The overall architecture for our HIP system is shown in Figure 3.

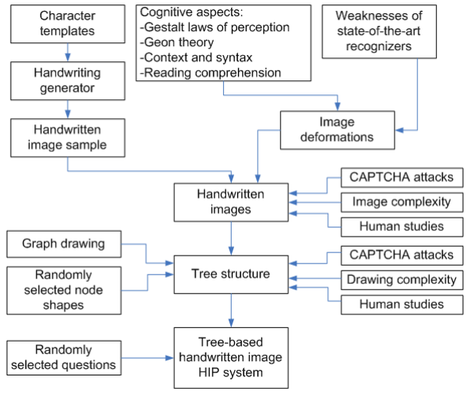
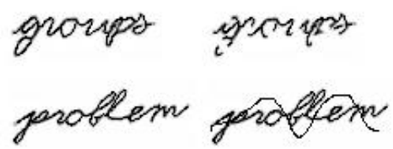


Figure 3. Overall architecture for tree-based handwritten CAPTCHA.

3.1 Transformed Handwritten Images

For testing our approach we have used handwritten word image samples. We have also developed a method to generate virtually infinite quantities of synthetic handwritten images based on real character templates [25] and to transform them on the fly. We note that a somewhat narrow set of words and their corresponding handwritten images was used for testing in order to provide machines with a lexicon to give them a fair chance at solving. Synthetic handwritten images that are generated and transformed on the fly for use in the CAPTCHA application are shown in Figure 4, before and after applying deformations that defeat state-of-the-art handwriting recognizers.



a) b)

Figure 4. Synthetically generated handwritten images: a) original; b) transformed.

To create our CAPTCHA, the number, type and severity of transformations are randomly chosen and applied, with some basic rules applied to ensure the images remain at once unreadable to machines and understandable by humans related to both Gestalt principles and Geon theory [2, 3, 15]. To the best of our knowledge, tying particular Gestalt principles to specific image transformations is novel and helps ensure that we maintain legibility for humans while foiling machine recognition.

Transformations that can be applied to handwritten images include horizontal or vertical overlaps with the principles guiding human interpretation proximity, symmetry, familiarity, continuity and figure-ground; occlusions such as by circles, rectangles or lines in the same color as the background pixels (Figure 4b, top) with the principles guiding human reconstruction closure, proximity, continuity and familiarity; occlusions by waves from left to right in the same color as the background with the principles guiding human reconstruction closure, proximity and continuity; occlusions using the same color pixels as the foreground with the principles guiding human reconstruction familiarity and figure-ground; adding extra strokes (Figure 4b, bottom) with the principles guiding human reconstruction familiarity and figure-ground; using empty letters, broken letters, rough contours, fragmentations, etc. with the principles guiding human reconstruction closure, proximity, continuity and figure-ground; splitting the image into parts and offsetting them from each other or splitting the image into parts and spreading the parts out (mosaic effect) with the principles guiding human reconstruction closure, proximity, continuity and symmetry; changing word orientation or stretching or compressing with the principles guiding human reconstruction memory, internal metrics, familiarity of

objects and orientation. The synthetic handwriting generation [25], as well as the transformations applied, ensures that infinitely many variations in samples can be produced to prevent machine learning.

3.2 Creation of a Tree Structure with Transformed Handwritten Images

Trees can be used for a wide range of tasks such as manipulating hierarchical data, making data easy to search, or as in our case, for presenting visual elements. Trees are drawn from the top down. A node is an element of the tree while a branch is a line that connects elements. We have used a binary tree in our CAPTCHA (Figure 1) to ensure the drawing does not become too large or unnecessarily complex for human readers. Each node in this tree has at most two branches. Our use of tree structures to add complexity to a handwritten CAPTCHA is motivated by several factors. First, we leverage currently superior human skills not only in reading carefully transformed handwritten images but in interpreting them inside a graphic with additional elements. Our CAPTCHA in all cases requires the interpretation of handwriting and the tree merely adds more complexity. Second, we take into account the various open issues in graphics recognition and in document analysis and recognition generally. Our use of tree drawings is further encouraged by early user studies, which indicate that the trees do not present additional complexity to humans beyond handwritten CAPTCHAs alone.

The tree generation algorithm for our CAPTCHA uses randomness whenever possible to create a random tree overlaid with transformed synthetic handwritten images. The program begins with generating a random number of nodes. Once the number of nodes has been determined, the algorithm begins building a binary tree data structure. In addition to the random makeup of the tree, a randomly selected scaling and sizing algorithm ensures that the visual representation of the tree never looks the same. During the drawing phase, when each node is created, a randomly chosen generated image of handwritten text will be placed next to it. Also, for each tree branch that is created, a randomly selected symbol is placed in the middle of the branch. This could be extended to use almost any shape, drawing, or another type of symbols or pictures. As noted previously, deformations may be applied not only to the handwritten images but to the tree itself to further discourage machine learning.

To complete the generation of our CAPTCHA, the program selects a node, or set of nodes in the rendered tree at random about which to ask the question. Once the tree and proper placement of images has been completed, a question will be selected at random from a list of potential questions about the tree such as "Which {word} is connected to {some other word} by a {shape}? (Figure 1). In all cases the user must quickly scan through and interpret all handwritten samples in the tree to correctly answer the question. Alternatively, the user may be asked to name two words that are connected. For machines to break our CAPTCHA, much more than one handwritten image would need to be interpreted. In addition, to solve our CAPTCHA machines would need to segment out the various objects in our CAPTCHA, including tree elements, shapes or other graphics, and handwritten images. Thus with our CAPTCHA we exploit the open problem in document image analysis of analyzing, segmenting and recognizing the various elements in a digital document or image [5]. To complete the generation of the test, the handwritten images and their truth words (correct answers) are passed to the verifier. Upon challenge submittal, the user response is verified and the application determines whether the user passes or fails. If the user passes by interpreting all of the letters in the handwritten image correctly, they would then be given access to the Web resource in question, otherwise they would be given another different challenge. Since the difficulty level of recognition needed to answer the question is greater and also has a higher level of randomness, our tree-based handwritten CAPTCHA poses more difficulty for machines, while still remaining simple for human users based on their cognitive abilities.

4. HIP System Evaluation

We have designed the tree-based handwritten HIP system as a challenge-response protocol for Web security. Experimental tests on word legibility have been conducted with human subjects and state-of-the-art handwriting recognizers. We have tested large sets of images on machines. To make it a fair test for machines, we have assisted the word recognizers with lexicons that contain all the truth words of the test images. For testing we used scanned handwritten image samples of US city names which we had readily available from postal applications, in order to provide samples corresponding to a known, finite lexicon (size 40,000, roughly the number of US city names) to help machine recognition, as well as synthetically generated samples. In reality, in actual applications such as our CAPTCHA having no context-specific dictionary, the number of entries in the lexicon will be much larger which will affect recognition accuracy drastically, as indicated by researchers [13, 32].

4.1 Machine Testing

The handwritten CAPTCHAs have been tested by several state-of-the-art handwriting recognizers (Word Model Recognizer (WMR), Character Model Recognizer (CMR), and Accuscript (HMM)) [14, 31]. We have tested several sets of images using human-written scanned samples, synthetically generated samples, and tree-based handwritten images. Transformations were applied to human-written and scanned image samples based on the Gestalt principles and Geon theory. Several sets, each of them with over 4,000 handwritten city name images, were used, one set for each transformation. Parameter values for transformations were randomly chosen and successively applied to the handwritten word images. The individual transformations we were concerned with were less vs. more fragmentation, empty letters, displacement, mosaic effect, adding jaws, arcs, or extra strokes, occlusion by circles, occlusion by waves (white vs. black), vertical or horizontal overlap, and overlap of different words. We note that with the exception of occlusion by circle transformation, machine recognition rates were low for Gestalt-based transformations, even though the lexicon used to help machine accuracy was of a relatively small size. We observed that for occlusions by circle, machine recognition could be affected if the transformation did not adequately affect the foreground. We encountered overall accuracy of 5.74% for WMR, 1.21% for Accuscript and 3.8% for CMR, with accuracies approaching 0% for individual transformations such as letter fragmentations, overlaps, and adding extra strokes, when recognizers were aided by a relatively small lexicon of 4,000 words. On the other hand, these presented the least difficulty for human subjects, based on the Gestalt laws of closure and continuity [23, 24].

We also tested 300 synthetic handwriting samples that were generated corresponding to US city names, US states and world-wide countries. Similar Gestalt and Geon-motivated transformations were applied. For these images we saw an accuracy rate of only 1.00% for WMR, 0.7% for Accuscript, and 0.3% for CMR. This extremely low machine recognition rate even with a small word set and a provided small lexicon suggests that synthetic handwritten images are an excellent choice for generating infinitely many CAPTCHAs.

To the best of our knowledge, there is not yet any commercial program which can fully interact with or decipher our tree-based handwritten CAPTCHA. As we have noted previously, our use of tree structures is motivated by the fact that they can provide complexity for machine recognition beyond interpretation of handwritten images. In addition, we leverage open problems in graphics recognition as well as in the wider field of Document Analysis and Recognition. In our tree structure, symbols have no intrinsic meaning outside of our CAPTCHA. Thus, segmenting parts of our drawing would not assist in solving the combined CAPTCHA since the task of handwriting interpretation would still remain difficult. Our drawing only makes sense as a complete entity which requires the interpretation of symbols with no inherent meaning on top of interpreting handwritten transformed images with their previously mentioned difficulties for machines.

4.2 Usability Testing

Our usability testing focused on understanding the viability of our CAPTCHA both from a user experience perspective and based on how often users were able to interpret our CAPTCHA. A key area of focus was on determining whether our tree structure in combination with handwritten images presented any additional difficulty as compared to a handwriting only CAPTCHA. User tests were conducted both for handwritten images alone and for our tree-based handwritten CAPTCHA.

To test handwritten images alone that were transformed according to Gestalt and Geon principles, random sets of 90 images were given to be recognized by 9 volunteers. The test consisted of 10 handwritten word images for each of the 9 types of transformations. For the purposes of testing, to ensure human results could be fairly compared to machine results, images were chosen at random from transformed US city name images. The actual application will feature virtually infinite-many different synthetically generated and transformed word images with an unrestricted domain to foil machine recognition. We note that most of the human errors came from poor original images rather than being related to the transformations applied. Success rates for humans averaged 80%. As noted earlier, we believe that through a careful process of parameter selection and good quality starting samples, which can best be guaranteed by using synthetic samples, we can achieve a higher success rate for human recognition for our images. We have also compared human recognition on a set of human

handwritten US city name images to a set containing 79 synthetic US city name, state or country name images automatically generated by our handwriting generator program. Similar high human recognition of 80% or better for both human-written and synthetic image sets suggests that synthetic images pose no additional problems to humans.

Our first round of user studies on our tree-based handwritten CAPTCHA included 15 volunteer graduate and undergraduate students. Approximately 30% of the volunteers were non-native English speakers which suggests that our CAPTCHA may be useful to a large audience. Subsequent tests will be run with a larger set of participants and tests featuring a larger set of words. During our initial test, 190 challenges were completed and a series of survey questions were administered on a volunteer basis to all participants. Each participant was asked to take 20 tree-based handwritten CAPTCHA challenges. The tests were self-administered and were taken at a testing Web site. Participants were given only very basic information on the concept of CAPTCHA and no prior knowledge of the field was assumed. Users were asked to solve the CAPTCHA presented (Figure 1) and to rate each CAPTCHA on a scale of 1-5, with 1 being "least difficult" and 5 being "most difficult". At the end users were asked to provide general comments about our CAPTCHA as well as responses to specific questions about their Web usage and exposure to CAPTCHAs.

Users were able to interpret the tree-based handwritten CAPTCHA 80.6% of the time which was no less often than the 80% for handwritten CAPTCHA trials. The most common rating given for the tree-based handwritten CAPTCHA trials was 2, although the presence of trials rated as 4 or 5 made the average (2.8) slightly higher. We have observed that generally the samples given a higher rating were those with poor image quality from scanned samples. We feel that the average rating is acceptable given the current sample set and will only be improved with a cleaner, all synthetic set to be used in the next round of testing. One interesting observation is that in many cases where users rated an image as a 4 or 5, they were still able to correctly guess the image. We believe that recognizing additional elements in the tree allowed users to fill in the blanks and interpret images that they may not have otherwise been able to read. Several general comments collected in the survey support this assumption. The ability to guess words even when they were hard to read once again highlights the importance of human perception factors involved in reading, including those of local context and Gestalt and Geon principles. The same assistance provided to humans by the tree structure provided more difficulty for machines due to problems in graphics segmentation of complex multi-layer images. Of the individual transformations, more correct answers were for word images transformed using overlaps, mosaic effect and extra strokes.

5. Conclusions and Future Work

This paper furthers the work on handwritten CAPTCHA [22, 23, 24] and provides insights into the fields of AI, Handwriting and Graphics Recognition. A new approach is presented for a HIP system that combines handwritten word images in a randomly generated tree-based structure with randomly selected test questions. Our approach leverages currently superior human ability over machines in interpreting graphics and reading unconstrained handwriting, especially when various transformations such as fragmentations or occlusions are applied. The Gestalt laws of perception and Geon theory and the weaknesses of handwriting recognizers were closely studied to determine the most effective transformations to apply to keep images legible for humans while making them difficult to read by machines. We add the novel element of a randomly drawn tree structure with randomly drawn node elements in addition to handwritten images to further leverage human cognitive strengths over machines.

Experimental results show a high degree of human success even with inconsistent quality scanned handwriting samples, and user feedback has also been largely positive indicating that our CAPTCHA is human-friendly. At the same time, our tests using state of the art recognizers prove that machine success rates with the same tests are low to non-existent. Testing both real human handwriting and synthetically generated samples allowed us to compare the results at machine and human level and conclude that synthetic handwriting is at least as good as real handwriting for CAPTCHA purposes. All these aspects indicate that our CAPTCHA can successfully be used to protect online services as a viable alternative solution for Cyber security. Additionally, our CAPTCHA provides fertile ground for work on important problems in other areas such as AI, Image Analysis, Machine Learning, Security, etc., and invite researchers to work on breaking our CAPTCHA and further advance knowledge in those fields.

We are considering several improvements to our application based on user feedback. We will run additional user tests using a larger set of non-domain specific words and synthetic samples generated and transformed on the fly by our handwriting generator. A wider range of participants will be considered and metrics on human time to solve the challenges by transformation type collected. More testing will be conducted to understand the processing load that our application might present during real-time use by many concurrent users. We also plan to have researchers create custom attacks to understand any potential vulnerabilities of our approach or will release our application to the public using some of our university online services in order to further understand how machines and a wider set of human users might interact with it. Combining handwritten text images with images of objects is another possible extension for our CAPTCHA. Last but not least, an alternative CAPTCHA for visually impaired users will be considered as well.

6. References

[1] Baird, H. S. and Popat, K. Human Interactive Proofs and Document Image Analysis. In Proc. IAPR Workshop on Document Analysis Systems, 2002.

[2] Biederman, I. Recognition-by-components: A theory of human image understanding. Psychological Review, 94, 2, 115-147, 1987.

[3] Biederman, I. and Blickle, T. The perception of objects with deleted contours. Unpublished manuscript, 1985.

[4] Biederman, I. and Gerhardstein, P.C.Recognizing Depth-Rotated Objects: Evidence and Conditions for Three-Dimensional Viewpoint Invariance. Journal of Experimental Psychology: Human Perception and Performance, 19, 1162-1182, 1993.

[5] Chaudhuri, B. Digital Document Processing: Major Directions and Recent Advances. Springer London, 2007.

[6] Chellapilla, K, Larson, K, Simard, P and Czerwinski, M. Designing Human Friendly Human Interaction Proofs (HIPs). In Proc. CHI 2005, ACM Press, 711-720, 2005.

[7] Chellapilla, K. and Simard, P. Using Machine Learning to Break Visual Human Interaction Proofs (HIPs). Advances in Neural Information Processing Systems, 17, 2004.

[8] Chew, M. and Baird, H.S. BaffleText: A Human Interactive Proof. In Proc. 10th IS&T/SPIE Document Recognition and Retrieval Conference, 2003.

[9] Freyd, J.J. Dynamic Mental Representation. Psychological Review, 94, American Psychological Assoc, 427-438, 1987.

[10] Gimpy web site: http://www.captcha.net/captchas/gimpy/

[11] Golle, P. Machine Learning Attacks Against the Asirra CAPTCHA. Proc. CCS '08, ACM, New York, 535-542, 2008.

[12] Goodman, J., Cormack, G.V., and Heckerman, D. Spam and the Ongoing Battle for the Inbox. Commun. ACM, 50:2, ACM, New York, 25-33, 2007.

[13] Govindaraju, V., Slavik, P. and Xue, H. Use of Lexicon Density in Evaluating Word Recognizers. IEEE Transactions on Pattern Analysis and Machine Intelligence, 24, 6, 789-800, 2002.

[14] Kim, G. and Govindaraju, V. A Lexicon Driven Approach to Handwritten Word Recognition for Real-Time Applications. IEEE Transactions on Pattern Analysis and Machine Intelligence, 19, 4, 366-379, 1997.

[15] Koffka, K. Principles of Gestalt Psychology. New York: Harcourt, Brace and Company, New York, 1935.

[16] Madhvanath, S. and Govindaraju, V. The Role of Holistic Paradigms in Handwritten Word Recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence, 23, 2, 149 – 164, 2001.

[17] Mori, G. and Malik, J. Recognizing objects in adversarial clutter: breaking a visual CAPTCHA. Proc. Computer Vision and Pattern Recognition, 1, I-134-I-141, 2003.

[18] Plamondon, R. and Srihari, S.N. Online and Off-Line Handwriting Recognition: A Comprehensive Survey., IEEE Transactions on Pattern Analysis and Machine Intelligence, 22, 1, 63-84, 2000.

[19] reCAPTCHA project – a CAPTCHA implementation. http://www.recaptcha.net/

[20] Rice, S., Nagy, G., and Nartker, T. Optical Character Recognition: An Illustrated Guide to the Frontier, Kluwer, Dordrecht, 1999.

[21] Rui, Y. and Liu, Z. ARTiFACIAL: Automated Reverse Turing Test Using Facial Features. Proc. MM '03, ACM, New York, 295-298, 2003.

[22] Rusu, A. and Govindaraju, V. Handwritten CAPTCHA: Using the difference in the abilities of humans and machines in reading handwritten words. In Proc. Ninth International Workshop on Frontiers in Handwriting Recognition, 226-231, 2004.

[23] Rusu, A. and Govindaraju, V. Visual CAPTCHA with Handwritten Image Analysis. Human Interactive Proofs: Second International Workshop, HIP 2005, Bethlehem, PA, USA, May 19-20, 2005: Proceedings, Springer, 42-51, 2005.

[24] Rusu, A. and Govindaraju, V. A Human Interactive Proof Algorithm Using Handwriting Recognition. In Proc. Eighth International Conference on Document Analysis and Recognition, 2, 29, 967-971, 2005.

[25] Rusu, A., Midic, U and Govindaraju, V. Synthetic Handwriting Generator for Cyber Security. In Proc. 13th Conference of the International Graphonomics Society, 2007.

[26] Senior, A.W. and Robinson, A.J. An Off-Line Cursive Handwriting Recognition System. IEEE Transactions on Pattern Analysis and Machine Intelligence, 20, 3, 309-321, 1998.

[27] Srihari, S.N. and Kuebert, E.J. Integration of Hand-Written Address Interpretation Technology into the United States Postal Service Remote Computer Reader System. In Proc. Fourth International Conference on Document Analysis and Recognition, 2, 892-896, 1997.

[28] Turing, A. Computing machinery and intelligence. Mind 59, 236, 433-460, 1950.

[29] von Ahn, L., Blum, M., Hopper, N. and Langford, J. CAPTCHA: Using Hard AI Problems for Security. LNCS, Springer Berlin / Heidelberg, Vol. 2656, 294-311, 2003.

[30] von Ahn, L., Blum, M., and Langford, J. Telling humans and computers apart automatically. Communications ACM 47, 2, 56-60, 2004.

[31] Xue, H. and Govindaraju, V. A stochastic model combining discrete symbols and continuous attributes and its application to handwriting recognition. International Workshop of Document Analysis and Systems, 70-81, 2002.

[32] Xue, H. and Govindaraju, V. On the dependence of handwritten word recognizers on lexicons. IEEE Transactions on Pattern Analysis and Machine Intelligence, 24, 12, 1553-1564, 2002.

[33] Yan, J. and El Ahmad, A.S. A Low-cost Attack on a Microsoft CAPTCHA. Proc. CCS '08, ACM, New York, 543-554, 2008.

[34] Yan, J. and El Ahmad, A.S. Usability of CAPTCHAs Or usability issues in CAPTCHA design. *Proc. SOUPS 2008,* ACM, New York, 44-52, 2008.

[35] Yan, L. and Wenyin, L. Engineering Drawings Recognition Using a Case-based Approach. Proc. Seventh Int'l Conf. on Document Analysis and Recognition, IEEE, Washington, D.C., 1-5, 2003.

GULFSTREAM: Staged Static Analysis for Streaming JavaScript Applications

Salvatore Guarnieri
University of Washington

Benjamin Livshits
Microsoft Research

Abstract

The advent of Web 2.0 has led to the proliferation of client-side code that is typically written in JavaScript. Recently, there has been an upsurge of interest in static analysis of client-side JavaScript for applications such as bug finding and optimization. However, most approaches in static analysis literature assume that the *entire program* is available to analysis. This, however, is in direct contradiction with the nature of Web 2.0 programs that are essentially being streamed at the user's browser. Users can see data being streamed to pages in the form of page updates, but the same thing can be done with code, essentially delaying the downloading of code until it is needed. In essence, the entire program is never completely available. Interacting with the application causes more code to be sent to the browser.

This paper explores *staged static analysis* as a way to analyze streaming JavaScript programs. We observe while there is variance in terms of the code that gets sent to the client, much of the code of a typical JavaScript application can be determined statically. As a result, we advocate the use of combined offline-online static analysis as a way to accomplish fast, browser-based client-side online analysis at the expense of a more thorough and costly server-based offline analysis on the static code. We find that in normal use, where updates to the code are small, we can update static analysis results quickly enough in the browser to be acceptable for everyday use. We demonstrate the staged analysis approach to be advantageous especially in mobile devices, by experimenting on popular applications such as Facebook.

1 Introduction

The advent of Web 2.0 has led to the proliferation of client-side code that is typically written in JavaScript. This code is often combined or *mashed-up* with other code and content from different third-party servers, making the application only fully available within the user's browser. Recently, there has been an upsurge of interest in static analysis of client-side JavaScript. However, most approaches in the static analysis literature assume that the entire program is available for analysis. This, however, is in direct contradiction with the nature of Web 2.0 programs that are essentially being *streamed* to the user's browser. In essence, the JavaScript application is never available in its entirety: as the user interacts with the application, more code is sent to the browser.

A pattern that emerged in our experiments with static analysis to enforce security properties [14], is that while most of the application can be analyzed offline, some parts of it will need to be analyzed on-demand, in the browser. In one of our experiments, while 157 KB (71%) of Facebook JavaScript code is downloaded right away, an additional 62 KB of code is downloaded when visiting event pages, etc. Similarly, Bing Maps downloads most of the code right away; however, requesting traffic requires additional code downloads. Moreover, often the parts of the application that are downloaded later are composed on the client by referencing a third-party library at a fixed CDN URL; common libraries are jQuery and `prototype.js`. Since these libraries change relatively frequently, analyzing this code ahead of time may be inefficient or even impossible.

The dynamic nature of JavaScript, combined with the incremental nature of code downloading in the browser leads to some unique challenges. For instance, consider the piece of HTML in Figure 1. Suppose we want to statically determine what code may be called from the `onclick` handler to ensure that none of the invoked functions may block. If we only consider the first `SCRIPT` block, we will conclude that the `onclick` handler may only call function `foo`. Including the second `SCRIPT` block adds function `bar` as a possible function that may be called. Furthermore, if the browser proceeds to download more code, either through more `SCRIPT` blocks or `XmlHttpRequests`, more code might need to be consid-

```
<HTML>
    <HEAD>
        <SCRIPT>
            function foo(){...}
            var f = foo;
        </SCRIPT>

        <SCRIPT>
            function bar(){...}
            if (...) f = bar;
        </SCRIPT>
    </HEAD>
    <BODY onclick="f();">
        ...
    </BODY>
</HTML>
```

Figure 1: Example of adding JavaScript code over time.

red to find all possible targets of the `onclick` handler.

While it is somewhat of an artificial example, the code n Figure 1 demonstrates that JavaScript in the browser ssentially has a *streaming programming* model: sites in- ert JavaScript into the HTML sent to the user, and the rowser is happy to execute any code that comes its way.

GULFSTREAM advocates performing *staged static nalysis* within a Web browser. We explore the trade-off etween offline static analysis performed on the server nd fast, staged analysis performed in the browser. We onclude that staged analysis is fast enough, especially n small incremental updates, to be made part of the verall browser infrastructure. While our focus is on nalyzing large, modern AJAX applications that use avaScript, we believe that a similar approach can be ap- lied to other platforms such as Silverlight and Flash.

.1 Contributions

'his paper makes the following contributions:

- **Staged analysis.** With GULFSTREAM, we demonstrate how to build a staged version of a points-to analysis, which is a building block for implementing static analysis for a wide range of applications, including security and reliability checkers as well as optimizations. Our analysis is staged: the server first performs *offline* analysis on the statically available code, serializes the results, and sends them to a client which performs analysis on code *deltas* and updates the results from the offline analysis. To our knowledge, GULFSTREAM is the first static analysis to be staged across across multiple machines.
- **Trade-offs.** We use a wide range of JavaScript inputs of various sizes to estimate the overhead of staged computation. We propose strategies for choosing between staging analysis and full analysis for various network settings. We explore the trade-off between computation and network data transfer and suggest strategies for different use scenarios.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 provides background on both client-side Web applications and static analysis. Section 3 provides an overview of our approach. Section 4 gives a description of our implementation. Section 5 discusses our experimental results. Finally, Sections 6 and 7 describe related work and outline our conclusions.

2 Background

This section first provides a background on static analysis and its most common applications, and then talks about code loading in Web applications.

2.1 Static Analysis

Static analysis has long been recognized as an important building block for achieving reliability, security, and performance. Static analysis may be used to find violations of important reliability properties; in the context of JavaScript, tools such as JSLint [9] fulfill such a role. Soundness in the context of static analysis gives us a chance to provide guarantees on the analysis results, which is especially important in the context of checking security properties. In other words, lack of warnings of a static analyzer implies that no security violations are possible at runtime; several projects have explored this avenue of research for client-side JavaScript [8, 14]. Finally, static analysis may be used for optimization: statically-computed information can be used to optimize runtime execution. For instance, in the context of JavaScript, static knowledge of runtime types [18] may be used to improve the performance of runtime interpretation or tracing [13] within the JavaScript runtime.

Several broad approaches exist in the space of static analysis. While some recent static analysis in type inference have been made for JavaScript [16], the focus of this paper is on *pointer analysis*, long recognized as a key building block for a variety of static analysis tasks. Because function closures can be easily passed around in JavaScript, pointer analysis is even necessary for something as ostensibly simple as call graph construction.

The goal of pointer analysis is to answer the question "given a variable, what heap objects may it point to?" While a great variety of techniques exist in the pointer analysis space, resulting in widely divergent trade-offs between scalability and precision, a popular choice is to represent heap objects by their allocation site. For instance, for the following program


```
1. var v = null;
2. for (...) {
3.     var o1 = new Object();
4.     var o2 = new Object();
5.     if (...)
6.         v = o1;
7.     else
8.         v = o2;
9. }
```

variables `o1` and `o2` point to objects allocated on lines 3 and 4, respectively. Variable `v` may point to either object, depending on the outcome of the `if` on line 5. Note that all objects allocated on line 3 within the loop are represented by the same allocation site, potentially leading to imprecision. However, imprecision is inevitable in static analysis, as it needs to represent a potentially unbounded number of runtime objects with a constant number of static representations.

In this paper, we focus on the points-to analysis formulation proposed by the Gatekeeper project [14]. Gatekeeper implements a form of inclusion-based Andersen-style context-insensitive pointer analysis [2], which shows good scalability properties, potentially with a loss of precision due to context insensitivity. However, for many applications, such as computing the call graph for the program, context sensitivity has not been shown to be necessary [21].

Static analysis is generally used to answer questions about what the program might do at runtime. For instance, a typical query may ask if it is possible for the program to call function `alert`, which might be desirable to avoid code leading to annoying popup windows. Similarly, points-to information can be used to check heap isolation properties such as "there is no way to access the containing page without going through proper APIs" in the context of Facebook's FBJS [12]. Properties such as these can be formulated as statically resolved heap reachability queries.

2.2 Code Loading in Web Applications

As we described above, Web 2.0 programs are inherently streaming, which is to say that they are downloaded over time. Below we describe a small study we performed of two large-scale representative AJAX applications. Figure 2 summarizes the results of our experiments. We start by visiting the main page of each application and then attempt to use more application features, paying attention to how much extra JavaScript code is downloaded to the user's browser. Code download is cumulative: we take care not to change the browser location URL, which would invalidate the current JavaScript context.

As Figure 2 demonstrates, much of the code is downloaded initially. However, as the application is used, quite a bit of extra code, spanning multiple potentially

Page visited or action performed	Added JavaScript files	Added JavaScript KB
FACEBOOK FRONT PAGE		
Home page	19	157
Friends	7	29
Inbox	1	20
Profile	1	13
FACEBOOK SETTINGS PAGE		
Settings: Network	13	136
Settings: Notifications	1	1
Settings: Mobile	3	14
Settings: Language	1	1
Settings: Payments	0	0
OUTLOOK WEB ACCESS (OWA)		
Inbox page	7	1,680
Expand an email thread	1	95
Respond to email	2	134
New meeting request	2	168

Figure 2: Incremental loading of Facebook and OWA JavaScript code.

independently changing files, is sent to the browser. In the case of Facebook, the JavaScript code size grows by about 30% (9 files) once we have used the application for a while. OWA, in contrast, is a little more monolithic, growing by about 23% (5 files) over the time of our use session. Moreover, the code that is downloaded on demand is highly workload-driven. Only some users will need certain features, leading much of the code to be used quite rarely. As such, analyzing the "initial" portion of the application on the server and analyzing the rest of the code on-the-fly is a good fit in this highly dynamic environment.

3 Overview

In this paper, we consider two implementation strategies for points-to analysis. The first one is based on in-memory graph data structures that may optionally be serialized to be transmitted from the server to the client. The second one is Gatekeeper, a BDD-based implementation described by Guarnieri and Livshits in [14]. Somewhat surprisingly, for small code bases, we conclude that there is relatively little difference between the two implementations, both in terms of running time as well as in terms of the size of result representation they produce. In some cases, for small incremental updates, a graph-based representation is more efficient than the `bddbddb`-based one. The declarative approach is more scalable, however, as shown by our analysis of Facebook in Section 5.4. Figure 3 summarizes the GULFSTREAM approach and shows how it compares to the Gatekeeper strategy.

Staged analysis. As the user interacts with the Web site, updates to the JavaScript are sent to the user's browser

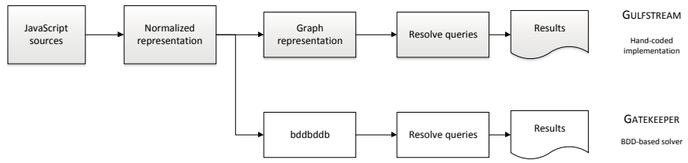


Figure 3: GULFSTREAM architecture and a comparison with the Gatekeeper project.

that in turn update the Web site. If the updates to the Web site's JavaScript are small, it would make sense that an staged analysis would perform better than a full program analysis. We looked at range of update sizes to identify when an staged analysis is faster than recomputing the full program analysis. Full program analysis might be faster because there is book keeping and graph transfer time in the staged analysis that is not present in the full program analysis. Section 5 talks about advantages of staged analysis in detail. In general, we find it to be advantageous in most settings, especially on slower mobile connections with slower mobile hardware.

Soundness. In this paper we do not explicitly focus on the issue of analysis soundness. Soundness would be especially important for a tool designed to look for security vulnerabilities, for instance, or applications of static analysis to runtime optimizations. Generally, sound static analysis of JavaScript only has been shown possible for *subsets* of the language. If the program under analysis belongs to a particular language subset, such as JavaScript$_{\text{SAFE}}$ advocated by Guarnieri et al. [14], the analysis results are sound. However, even if it does *not*, analysis results can still be used for bug finding, without necessarily guaranteeing that all the bugs will be found. In the remainder of the paper, we ignore the issues of soundness and subsetting, as we consider them to be orthogonal to staged analysis challenges.

Client analyses as queries. In addition to the pointer analysis, we also show how GULFSTREAM can be used to resolve two typical queries that take advantage of points-to analysis results. The first query looks for calls to `alert`, which might be an undesirable annoyance to the user and, as such, need to be prevented in third-party code. The second looks for calls to `setInterval`[1] with non-function parameters.

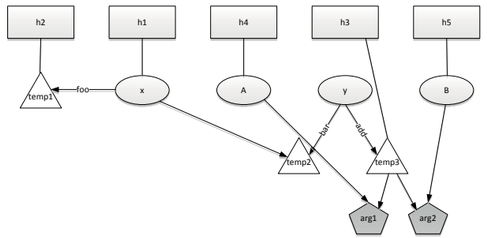
4 Techniques

The static analysis process in GULFSTREAM proceeds in stages, as is typical for a declarative style of program analysis. On a high level, the program is first represented as a database of facts. Next, a *solver* is used to derive new information about the program on the basis of initial facts and *inference rules*.

[1]Function `setInterval` is effectively a commonly overlooked form of dynamic code loading similar to `eval`.

```
1.  var A = new Object();
2.  var B = new Object();
3.  x     = new Object();
4.  x.foo = new Object();
5.  y     = new Object();
6.  y.bar = x;
7.  y.add = function(a, b) {}
8.  y.add(A, B)
```

(a) Input JavaScript program.



(b) Resulting graph.

Figure 4: Example of a program with a function call.

In GULFSTREAM, the first analysis stage is normalizing the program representation. Based on this normalized representation, we built two analyses. The first is the declarative, bddbddb-based points-to analysis described in Gatekeeper [14]. The second is a hand-coded implementation of points-to information using graphs as described in the rest of this section.

The graph-based representation also produces graphs that can efficiently compressed and transferred to the browser from the server. To our surprise, we find that at least for small programs, the graph-based representation performs at least as well as the bddbddb-based approach often advocated in the past; bddbddb-based analysis, however, performs faster on larger code bases, as discussed in Section 5.4.

Node type	Description	Node shape
Variable	The basic node is a simple variable node. It represents variables from the program or manufactured during normalization. Line 1 in Figure 4 has two variable nodes, `A` and `Object`.	Oval
Heap	These nodes represent memory locations and are sinks in the graph: they do not have any outgoing edges. Heap nodes are created when new memory is created like in line 1 in Figure 4 when a new `Object` is created.	Rectangle
Field	These nodes represent fields of objects. They are similar to variable nodes, except they know their object parent and they know the field name used to access them from their object parent. Conversely, variables that have fields contain a list of the field nodes for which they are the parent. Field nodes are represented by a triangular node connected to the object parent by a named edge. Line 4 shows the use of a field access. The name of the edge is the name of the field.	Triangle
Argument	The fourth type of node is a special node called an argument node. These nodes are created for functions and are used to link formals and actuals. The argument nodes contain edges to their respective argument variables in the function body and when a function is called, the parameter being passed in gets an edge to the respective argument node. In the graph, Argument nodes are represented by pentagons. Lines 7 and 8 from Figure 4 show a function node being created and used. Return values are also represented by this type of node.	Pentagon

Figure 5: Description of nodes types in the graph.

4.1 Normalization

The first analysis stage is normalizing the program representation and is borrowed from the Gatekeeper [14] project. The original program statements are broken down to their respective normalized versions, with temporaries introduced as necessary. Here is a normalization example that demonstrates variable introduction:

```
var x = new Date();      x = new Date();
var y = 17;              y = ⊥;
h.f   = h.g;             t = h.g; h.f = t;
```

Variable `t` has been introduced to hold the value of field `h.g`. Since we are not concerned with primitive values such as `17`, we see it represented as ⊥.

4.2 Graph Representation

The points-to information is calculated from a graph representing the program stored in memory. The graph is generated from the normalized program. Assignments turn into edges, field accesses turn into named edges, constructor calls create new sinks that represent the heap, and so on. The graph fully captures the points-to information for the program. One important note is that this graph is not transitively closed. If the program states that `A` flows to `B` and `B` flows to `C`, the graph does not contain an edge from `A` to `C` even though `A` flows to `C`. The graph must be traversed to conclude that `A` points to `C`.

The full graph consists of several different types of nodes, as summarized in Figure 5. We use the program and corresponding graph in Figure 4 as an example for our program representation. In lines 1-5, the program is creating new objects which creates new heap nodes in the graph. In lines 4, 6, and 7, the program is accessing a field of an object which makes use of a field edge to connect the base object's node to the field's field node. The first use of a field creates this new edge and field node. Line 7 creates a new function, which is similar to creating a new object. It creates a new heap node, but the function automatically contains argument nodes for each of its arguments. These nodes act as a connection between actuals and formals. All actuals must flow through these argument nodes to reach the formal nodes inside the function body. Line 8 calls the function created in line 7. This line creates assignment edges from the actuals (`A` and `B`) to the argument nodes, which already have flow edges to the formal.

4.3 Serialized Graph Representation

The output of each stage of analysis is also the input to the next stage of analysis, so the size and transfer time of this data must be examined when looking at our staged analysis. We compare the sizes of two simple file formats that we implemented and a third that is the bddbddb graph output, which is a a serialized BDD.

The first format from our analysis is based on the graphviz DOT file format [11]. This format maintains variable names for each node as well as annotated edges. The second format from our analysis is efficient for directed graphs and removes all non-graph related data like names. This format is output in binary is as follows:

```
[nodeid];[field_id1],[field_id2],...;[arg_id1],...;
[forward_edge_node_id1],[forward_edge_node_id2],...;
[backward_edge_node_id1],[backward_edge_node_id2],...;
[nodeid]...
```

where `nodeid`, `field_id1`, etc. are uniquely chosen integer identifiers given to nodes within the graph. Finally, the third format is a serialized BDD-based representation of bddbddb.

Overall, the sizes of the different formats of the staged-results graph vary widely. The DOT format is the largest, and this is to be expected since it is a simple text file describing how to draw the graph. The binary format and bddbddb output are closer in size, with the binary format being marginally smaller. A more detailed comparison of graph representation sizes is presented in Section 5.

```
pointsTo = ∅ ↦ ∅ // points-to map
reversePointsTo = ∅ ↦ ∅ // reverse version of points-to map
inc_insert(G, e) // incrementally update points-to map

 1: invalid = ∅
 2: if e.src ∈ G then
 3:     invalidate(e.src, invalid)
 4: end if
 5: if e.dst ∈ G then
 6:     invalidate(e.dst, invalid)
 7: end if
 8: G = ⟨G_N ∪ {e_src, e_dst}, G_E ∪ {e}⟩
 9: for all n ∈ invalid do
10:     ans =compute-points-to(n, ∅)
11:     pointsTo[n] = pointsTo[n] ∪ ans
12:     for all h ∈ ans do
13:         reversePointsTo[h] = reversePointsTo[h] ∪ n
14:     end for
15: end for

invalidate(n ∈ G_N, invalid) // recursively invalidate following flow edges

 1: if n ∈ invalid then
 2:     return
 3: end if
 4: invalid = invalid ∪{n}
 5: if n is FieldNode then
 6:     toVisit = compute-field-aliases(n.parent, n.fieldname)
 7: end if
 8: for all n' adjacent to n do
 9:     if n → n' is an assignment edge then
10:         toVisit = toVisit ∪ n'
11:     end if
12: end for
13: for all n' ∈ toVisit do
14:     invalidate(n')
15: end for
```

Figure 6: Routines inc_insert and invalidate.

Since our main focus was not to develop a new efficient graph storage format, we `gzip` all the graph output formats to see how their sizes compared under an industry-standard compression scheme. Since BDDs are highly optimized to minimize space usage, one would expect their zipped size to be similar to their unzipped size. As expected, the DOT format receives huge gains from being zipped, but it is still the largest file format. The difference between the three formats is minimal once they are all zipped. Since this data must be transferred from the server to the client to perform the staged analysis, these figures indicate that the graph output format does not make much of a difference on the staged analysis time on a fast link assuming `gzip` times do not vary much from one format to another. We leave more detailed measurements that take decompression time into account for future work.

4.4 Points-to Analysis Implementation

Our system normalizes JavaScript into a representation that we can easily output for analysis. This means it is straightforward for us to try several different analysis techniques. We have two outputs of our representation at the moment, an output to Datalog facts that is used by bddbddb and an output to a graph representing the program which is used by our implementation of a points-to analysis. The reader is referred to prior work for more information about bddbddb-based analyses [6, 14, 25].

```
compute-points-to(n, visitedNodes)
 1: if n ∈ visitedNodes then
 2:     return ∅
 3: else
 4:     visitedNodes = visitedNodes ∪ {n}
 5: end if
 6: toVisit = ∅
 7: ans = ∅
 8: if n is HeapNode then
 9:     return n
10: end if
11: if n is FieldNode then
12:     toVisit = toVisit ∪
            compute-field-aliases(n.parent, n.fieldname)
13: end if
14: for assignment-edge e leaving n do
15:     toVisit = toVisit ∪ {e.sink}
16: end for
17: for node n' ∈ toVisit do
18:     ans = ans ∪ compute-points-to(n', visitedNodes)
19: end for
20: return ans

compute-field-aliases(parent, fieldname)
 1: toVisit = ∅
 2: if parent is FieldNode then
 3:     toVisit = toVisit ∪
            compute-field-aliases(parent.parent, parent.fieldname)
 4: end if
 5: toVisit = toVisit∪ compute-aliases(parent)
 6: for n ∈ toVisit do
 7:     if n has field fieldname then
 8:         ans = ans ∪ {n.fieldname}
 9:     end if
10: end for
11: return ans

compute-aliases(n, visitedNodes)
 1: ans = n
 2: if n ∈ visitedNodes then
 3:     return ∅
 4: else
 5:     visitedNodes = visitedNodes ∪ {n}
 6: end if
 7: for edge e leaving n do
 8:     ans = ans ∪ compute-aliases(e.sink, visitedNodes)
 9: end for
10: return ans
```

Figure 7: Points-to computation algorithm.

GULFSTREAM maintains a graph representation that is updated as more of the program is processed. Figure 6 shows a pseudo-code version of the graph update algorithm that we use. In addition to maintaining a graph G, we also save two maps: $pointsTo$, mapping variables to heap locations and its reverse version for fast lookup, $reversePointsTo$. Function inc_insert processes every edge e inserted into the graph. If the edge is not adjacent to any of the existing edges, we update G with edge e. If it is, we add the set of nodes that are adjacent to the edge, together with a list of all nodes from which they flow to a worklist called $invalid$. Next, for all nodes in that worklist, we proceed to recompute their points-to values.

The points-to values are recomputed using a flow based algorithm. Figure 7 shows the pseudo-code version of our points-to algorithm, including helper functions. For standard nodes and edges, it works by recursively following all reverse flow edges leaving a node until it reaches a heap node. If a cycle is detected, that recursion fork is killed as all nodes in that cycle will point to the same thing and that is being discovered by the other recursion forks. Since flows are created to argument nodes when functions are called, this flow analysis will pass through function boundaries.

Field nodes and argument nodes require special attention. Since these nodes can be indirectly aliased by accessing them through their parent object, they might not have direct flows to all their aliases. When a field node is reached in our algorithm, all the aliases of this field node are discovered, and all edges leaving them are added to our flow exploration. This is done by recording the name of the field for the current field node, finding all aliases of the parent to the field node, and getting their copy of a field node representing the field we are interested in. In essence, we are popping up one level in our flow graph, finding all aliases of this node, and descending these aliases to reach an alias of our field node. This process may be repeated recursively if the parent of a field node is itself a field node. The exact same procedure is done for argument nodes for the case when function aliases are made.

Note that the full analysis is a special, albeit more inefficient, case of the staged analysis where the $invalid$ worklist is set to be all nodes in the graph G_N. Figure 7 shows pseudo-code for computing points-to values for a particular graph node n.

4.5 Queries

The points-to information is essentially a mapping from variable to heap locations. Users can take advantage of this mapping to run queries against the program being loaded. In this paper, we explore two representative queries and show how they can be expressed and resolving using points-to results.

Not calling `alert`. It might be undesirable to bring up popup boxes, especially in library code designed to be integrated into large Web sites. This is typically accomplished with function `alert` in JavaScript. This query checks for the presence of `alert` calls.

Not calling `setInterval` with a dynamic function parameter. In JavaScript, `setInterval` is one of the dynamic code execution constructs that may be used to invoke arbitrary JavaScript code. This "cousin of `eval`" may be used as follows:

```
setInterval(
  new Function(
        "document.location='http://evil.com';"),
      500);
```

In this case, the first parameter is dynamically constructed function that will be passed to the JavaScript interpreter for execution. Alternatively, it may be a reference to a function statically defined in the code. In order to prevent arbitrary code injection and simplify analysis, it is desirable to limit the first parameter of `setInterval` to be a statically defined function, not a dynamically constructed function.

Figure 8 shows our formulation of the queries. The `detect-alert-calls` query looks for any calls to `alert`. It does this by first finding all the nodes that point to `alert`, then examining them to see if they are called (which is determined during normalization). The `detect-set-interval-calls` is somewhat more complicated. It cares if `setInterval` is called, but only if the first parameter comes from the return value of the `Function` constructor. So, all the source nodes from edges entering the first argument's node in `setInterval` must be examined to see if it has an edge to the return node of the `Function` constructor. In addition, all aliases of these nodes must also be examined to see if they have a flow edge to the return node of the `Function` constructor.

The results to these queries are updated when updates are made to the points-to information. This ensures that the results are kept current on the client machine. A policy is a set of queries and expected results to those queries. A simple policy would be to disallow any calls to `alert`, so it would expect `detect-alert-calls` from Figure 8 to return `false`. If `detect-alert-calls` ever returns true, the analysis engine could either notify the user or stop the offending page from executing.

4.6 Other Uses of Static Analysis

In addition to the two queries detailed above, there are many other uses that could be very useful. One other useful query would be to use the points-to results to identify when updates to pages modify important global variables. The points-to results can be traversed to identify when any aliases to global variables, like `Array` or common global library names, are modified which could lead to unexpected behavior.

Another use of the staged static analysis is helping to improve performance through optimization. Traditionally Just-In-Time (JIT) compilers have been used to improve the performance of dynamic languages like JavaScript [5]. These JITs have the benefit of actually seeing code paths during execution and optimizing them, but they must run on the client and thus have some amount of performance impact. Any complex analysis done by a JIT would negatively affect performance, which is what the JIT is trying to improve in the first place. Performing some amount of static analysis be-

```
detect-alert-calls()
 1: nodes = reversePointsTo[alert ]
 2: for all n    nodes do
 3:     if n.isCalled() then
 4:         return true
 5:     end if
 6: end for
 7: return false
```

```
detect-set-interval-calls()
 1: n = setInterval  .arg1
 2: for all edge e entering n do
 3:     if e.src == Function  .return then
 4:         return true
 5:     else
 6:         p = p    find-all-aliases(e.src)
 7:     end if
 8: end for
 9: for all node n2 in p do
10:     for all edge e2 entering n2 do
11:         if e2.src == Function  .return then
12:             return true
13:         end if
14:     end for
15: end for
```

```
find-all-aliases(node)
1: aliases = empty
2: heapNodes = pointsTo[node]
3: for all n    heapNodes do
4:     aliases = aliases    reversePointsTo[n]
5: end for
6: return aliases
```

Figure 8: Queries detect-alert-calls and detect-set-interval-calls.

fore running JavaScript through a JIT could empower the JIT to better optimize code [18]. GULFSTREAM would permit this without having to do an expensive analysis while the JIT is running. GULFSTREAM is especially well suited to this situation because the majority of the staged analysis is done offline and only updates to the code are analyzed on the client. Static analysis enables many analyses that can be semantically driven rather than syntactically driven and possibly fragile.

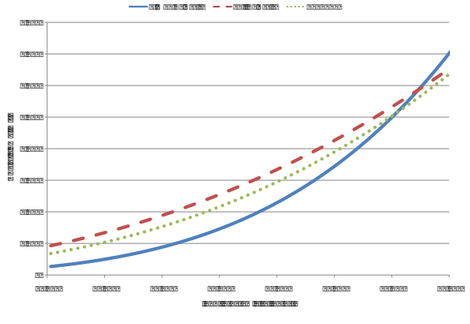


Figure 9: Trend lines for running times for full and staged analyses as well as the bddbddb-based implementation

5 Experimental Evaluation

This section is organized as follows. We first discuss analysis time and the space required to represent analysis results in Sections 5.1 and 5.2. Section 5.3 explores the tradeoff between computing results on the client and transferring them over the wire.

Measurements reported in this paper were performed on a MacBook Pro 2.4 GHz Dual Core machine running Windows 7. The JavaScript files we used during testing were a mix of hand crafted test files, procedurally generated files, and files obtained from Google code search, looking for JavaScript files. GULFSTREAM uses two bootstrap JavaScript files. The first introduces the native environment, where `Object`, `Array`, and other globals are defined. The second introduces a set of browser-provided globals such as `document` and `window`. Together these files are approximately 30 KB in size.

5.1 Analysis Running Time

Figure 9 shows full, staged and the bddbddb-based analyses on the same scale . For this experiment, we used our bootstrap file for the base in the staged analysis. We ran various sized JavaScript files through the full, staged, and bddbddb-based analyses. The full and bddbddb-based analyses processed the JavaScript file concatenated with the bootstrap file. The staged analysis processed the JavaScript file as an update to the already computed analysis on the bootstrap file.

We see that staged analysis is consistently faster than full analysis. In the cases of smaller code updates, the difference in running times can be as significant as a couple of orders of magnitude. We also see that for small updates, the staged analysis performs better than the bddbddb-based analysis. This is encouraging: it means that we can implement the staged analysis within the browser without the need for heavyweight BDD machinery, without sacrificing performance in the process. In the next section, we show that our space overhead is also generally less than that of BDDs.

5.2 Space Considerations

Figure 10 shows the sizes of three representations for points-to analysis results and how they compare to each other. The representations are DOT, the text-based graph format used by the Graphviz family of tools, bddbddb, a compact, BDD-based representation, as well as BIN, our graph representation described in Section 4.3. All numbers presented in the figure are after applying the industry-standard `gzip` compression.

We were not surprised to discover that the DOT version is most verbose. To our surprise, our simple binary format beats the compact bddbddb format in most cases, making us believe that a lightweight staged analysis implementation is a good candidate for being integrated within a Web browser.

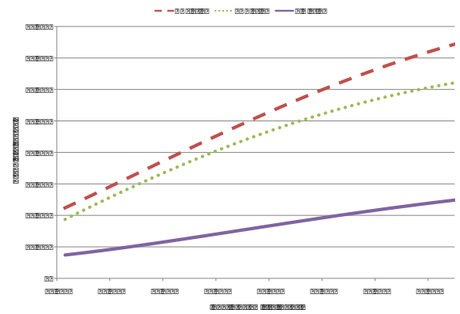


Figure 10: Trend lines for pointer analysis graph size as a function of the input JavaScript file size (`gzip`-ed).

5.3 Staged vs. Full Analysis Tradeoff

To fully explore the tradeoff between computing the full analysis on the client and computing part of the analysis on the server and transferring it over the wire to the client, we consider 10 device configurations. These configuration vary significantly in terms of the CPU speed as well as network connectivity parameters. We believe that these cover a wide range of devices available today, from the most underpowered: mobile phones connected over a slow EDGE network, to the fastest: desktops connected over a T1 link.

A summary of information about the 10 device configurations is shown in Figure 11. We based our estimates of CPU multipliers on a report comparing the performance of SunSpider benchmarks on a variety of mobile, laptop, and desktop devices [1]. While not necessarily representative of Web 2.0 application performance [23], we believe these benchmark numbers to be a reasonable proxy for the computing capacity of a particular device.

We compare between two options: 1) performing full analysis on the client and 2) transferring a partial result over the wire and performing the staged analysis on the client. The equation below summarizes this comparison. On the left is the overall time for the full analysis and on the right is the overall time for the staged analysis. B is the bandwidth, L is the latency, b is the main page size, is the incremental JavaScript update size, size is the size of the points-to data needed to run the staged analysis, and F and I are the full and staged analysis times respectively. c is the CPU coefficient from Figure 11:

$$c \quad F(b+ \quad) \; ? \; L + \frac{size}{B} + c \quad I(\quad)$$

ID	Configuration Name	CPU coef. c	Link type	Latency L in ms	Bandwidth B in kbps
1	G1	67.0	EDGE	500	2.5
2	Palm Pre	36.0	Slow 3G	500	3.75
3	iPhone 3G	36.0	Fast 3G	300	12.5
4	iPhone 3GS 3G	15.0	Slow 3G	500	3.75
5	iPhone 3GS WiFi	15.0	Fast WiFi	10	75.0
6	MacBook Pro 3G	1	Slow 3G	500	3.75
7	MacBook Pro WiFi	1	Slow WiFi	100	12.5
8	Netbook	2.0	Fast 3G	300	12.5
9	Desktop WiFi	0.8	Slow WiFi	100	12.5
10	Desktop T1	0.8	T1	5	1,250.0

Figure 11: Device settings used for experiments across CPU speeds and network parameters. Devices are roughly ordered in by computing and network capacity. Configurations 1–5 correspond to a mobile setting; configurations 6–10 describe a desktop setting.

Figure 12 summarizes the results of this comparison over our range of 10 configurations. The code analyzed at the server is the bootstrap code from before. Therefore, the points-to data sent to the client was always the same while the size of the incremental code update var-

Incremental size	Configuration (from Figure 11) 1	2	3	4	5	6	7	8	9	10
88	+	+	+	+	+	-	+	+	-	+
619	+	+	+	+	+	-	+	+	-	+
1,138	+	+	+	+	+	-	+	+	-	+
1,644	+	+	+	+	+	-	-	+	-	+
2,186	+	+	+	+	+	-	-	+	-	+
2,767	+	+	+	+	+	-	-	+	-	+
3,293	+	+	+	+	+	-	-	+	-	+
3,846	+	+	+	+	+	-	-	+	-	+
4,406	+	+	+	+	+	-	-	-	-	+
5,008	+	+	+	+	+	-	-	+	-	+
5,559	+	+	+	+	+	-	-	+	-	+
6,087	+	+	+	+	+	-	-	+	-	+
6,668	+	+	+	+	+	-	-	+	-	+
7,249	+	+	+	+	+	-	-	+	-	+
7,830	+	+	+	+	+	-	-	+	-	+
8,333	+	+	+	+	+	-	-	+	-	+
8,861	+	+	+	+	+	-	-	-	-	+
9,389	+	+	+	+	+	-	-	-	-	+
9,917	+	+	+	+	+	-	-	-	-	+
10,445	+	+	+	+	+	-	-	-	-	+
10,973	+	+	+	+	+	-	-	-	-	+
11,501	+	+	+	+	+	-	-	-	-	+
12,029	+	+	+	+	+	-	-	-	-	+
12,557	+	+	+	-	+	-	-	-	-	+
14,816	+	+	+	+	+	-	+	+	+	+
16,485	-	-	-	-	-	-	-	-	-	-
17,103	+	+	+	+	+	-	-	-	-	+
17,909	-	-	-	-	-	-	-	-	-	-
20,197	-	-	-	-	-	-	-	-	-	-
25,566	-	-	-	-	-	-	-	-	-	-
31,465	-	-	-	-	-	-	-	-	-	-
37,689	-	-	-	-	-	-	-	-	-	-
38,986	-	-	-	-	-	-	-	-	-	-
57,254	+	+	+	+	+	-	-	+	-	+
77,074	+	+	+	+	+	-	+	+	-	+
124,136	-	-	-	-	-	-	-	-	-	-
129,739	-	-	-	-	-	-	-	-	-	-

Figure 12: Analysis tradeoff in different environments. "+" means that staged incremental analysis is advantageous compared to full analysis on the client.

Page	Lines of code		Analysis time (seconds)			Full/
	Total	Inc.	Inc.	Full	bddbddb	bddbddb
home	965	339	591	599	4	148
friends	1,274	309	1,297	1,866	6	324
inbox	1,291	17	800	1,840	6	313
profile	1,308	17	851	4,180	6	716

Figure 13: Analysis times for an incrementally loading site (Facebook.com). Each page sends an update to the JavaScript that adds to the previous page's JavaScript.

ied. A + in the table indicates that staged analysis is faster. Overall, we see that for all configurations except 6, 7, and 9, staged analysis is generally the right strategy. "High-end" configurations 6, 7, and 9 have the distinction of having a relatively fast CPU and a slow network; clearly, in this case, computing analysis results from scratch is better than waiting for them to arrive over the wire. Unsurprisingly, the staged approach advocate by GULFSTREAM excels on mobile devices and underpowered laptops. Given the growing popularity of Web-connected mobile devices, we believe that the staged analysis approach advocated in this paper will become increasingly important in the future.

5.4 Facebook Analysis Experiment

Thus far, we have shown how GULFSTREAM performs when run on smaller JavaScript fragments, chosen to simulate incremental code updates of varying sizes. To see how GULFSTREAM handles a real incrementally loading JavaScript program, we captured an interactive session of Facebook usage. JavaScript code was incrementally loaded as the user interacted with the page. We fed each JavaScript update through GULFSTREAM to simulate a user navigating Facebook with GULFSTREAM updating analysis results, as more JavaScript is loaded into the browser.

The navigation session we recorded and used comprises a particular interaction with the main Facebook page. We were careful to use actions such as link clicks that do *not* take the user away from the current page (that would create a new, clean JavaScript engine context). The recorded interaction is clicking several links in a row that keeps the user at the same page. The user starts at the homepage where a large amount of JavaScript is downloaded. Then the user clicks on their friends link, which causes more JavaScript to be downloaded and the page to be updated. The same happens when the user then clicks on their inbox link, and their profile link. These four pages: the *homepage*, *friends*, *inbox*, and *profile* make up our Facebook staged analysis experiment.

In the experiment, each of the four pages was processed by the staged analysis, the full analysis, and the bddbddb analysis. For the staged analysis, the code from the initial page was considered an incremental update upon the bootstrap JavaScript that includes our native environment definition and our browser environment definition. Then in all subsequent pages, the code downloaded for that page was considered an incremental update upon the already computed results from the previous page. For the full analysis and the bddbddb analysis, each of the four pages was analyzed in isolation.

Figure 13 contains the sizes of the pages used in this experiment. The lines of code reported is the line count from the files as they are when downloaded. Note that in many cases, code includes long `eval` statements that expand to many hundreds to thousands of lines of code. For example, the incremental part of the first page expands to over 15,000 lines of code and the incremental part of the last page expands to over 1,400 lines of code.

The goal for the staged analysis is to perform updates to the analysis faster than it takes to recompute the entire analysis. This is highly desirable since the updates in this case are small and rerunning a full analysis to analyze just a few new lines of code is highly wasteful. Figure 13 confirms this intuition by comparing staged analysis against full analysis. Additionally Figure 14 shows the time savings in seconds for each of the device and network configurations from Figure 11. In every configuration the staged analysis fares better, leading to savings on the order of 5 minutes or more.

However, the bddbddb full analysis outperforms the staged analysis by several orders of magnitude, as shown in the last column in Figure 13. This is because the bddbddb analysis solver is highly optimized to scale well and because of our choice of an efficient variable order for BDD processing [25]. While this experiment shows that our staged analysis is better than our full analysis, it also shows that the highly optimized bddbddb-based technique is significantly better for analyzing the code quickly; this is in line with what has been previously observed for Java and C, when comparing declarative vs. hand-written implementations. It should also be noted that the JavaScript for these pages is more complex than in our hand-crafted and procedurally generated files used for other experiments, which produces more complex constraints and favors the more scalable bddbddb-based approach. However, running a highly optimized Datalog solver such as bddbddb within the browser might prove cumbersome for other reasons such as the size and complexity of the code added to the browser code base.

6 Related Work

In this section, we focus on static and runtime analysis approaches for JavaScript.

	Configuration (from Figure 11)									
Page	1	2	3	4	5	6	7	8	9	10
home	541	290	291	119	121	5	7	15	5	6
friends	38,083	20,460	20,469	8,516	8,530	554	564	1,133	450	454
inbox	69,685	37,439	37,451	15,589	15,606	1,022	1,035	2,075	827	832
profile	223,029	119,833	119,845	49,920	49,937	3,311	3,323	6,652	2,658	2,663

Figure 14: Time savings in ms from using staged analysis compared to full analysis on Facebook pages (device and network settings are from Figure 11).

6.1 Static Safety Checks

JavaScript is a highly dynamic language which makes it difficult to reason about programs written in it. However, with certain expressiveness restrictions, desirable security properties can be achieved. ADSafe and Facebook both implement a form of static checking to ensure a form of safety in JavaScript code. ADSafe [10] disallows dynamic content, such as `eval`, and performs static checking to ensure the JavaScript in question is safe. Facebook uses a JavaScript language variant called FBJS [12], that is like JavaScript in many ways, but DOM access is restricted and all variable names are prefixed with a unique identifier to prevent name clashes.

A project by Chugh et al. focuses on staged analysis of JavaScript and finding information flow violations in client-side code [8]. Chugh et al. focus on information flow properties such as reading document cookies and changing the locations. A valuable feature of that work is its support for dynamically loaded and generated JavaScript in the context of what is generally thought of as whole-program analysis. Gatekeeper project [14] proposes a points-to analysis based on bddbddb together with a range of queries for security and reliability. GULFSTREAM is in many way a successor of the Gatekeeper project; while the formalism and analysis approaches are similar, GULFSTREAM's focus is on staged analysis.

Researchers have noticed that a more useful type system in JavaScript could prevent errors or safety violations. Since JavaScript does not have a rich type system to begin with, the work here is devising a correct type system for JavaScript and then building on the proposed type system. Soft typing [7] might be one of the more logical first steps in a type system for JavaScript. Much like dynamic rewriters insert code that must be executed to ensure safety, soft typing must insert runtime checks to ensure type safety.

Other work has been done to devise a static type system that describes the JavaScript language [3, 4, 24]. These works focus on a subset of JavaScript and provide sound type systems and semantics for their restricted subsets of JavaScript. As far as we can tell, none of these approaches have been applied to realistic bodies of code. GULFSTREAM uses a pointer analysis to reason about the JavaScript program in contrast to the type systems and analyses of these works. We feel that the ability to reason about pointers and the program call graph allows us to express more interesting security policies than we would be able otherwise.

This work presents staged analysis done on the client's machine to perform analysis on JavaScript that is loaded as the user interacts with the page. A similar problem is present in Java with dynamic code loading and reflection. Hirzel et al. solved this problem with a offline-online algorithm [15]. The analysis has two phases, an offline phase that is done on statically known content, and an online phase done when new code is introduced while the program is running. They utilize their pointer analysis results in the JIT. We use a similar offline-online analysis to compute information about statically known code, then perform an online analysis when more code is loaded. To our knowledge, GULFSTREAM is the first project to perform staged static analysis on multiple tiers.

6.2 Rewriting and Instrumentation

A practical alternative to static language restrictions is instrumentation. Caja [22] is one such attempt at limiting capabilities of JavaScript programs and enforcing this through the use of runtime checks. WebSandbox is another project with similar goals that also attempts to enforce reliability and resource restrictions in addition to security properties [20].

Yu et al. [26] traverse the JavaScript document and rewrite based on a security policy. Unlike Caja and WebSandbox, they prove the correctness of their rewriting with operational semantics for a subset of JavaScript called CoreScript. Instrumentation can be used for more than just enforcing security policies. AjaxScope [17] rewrites JavaScript to insert instrumentation that sends runtime information, such as error reporting and memory leak detection, back to the content provider. Static analysis may often be used to reduce the amount of instrumentation, both in the case of enforcement techniques such as ConScript [19] and regular code execution.

7 Conclusions

Static analysis is a useful technique for applications ranging from program optimization to bug finding. This paper explores staged static analysis as a way to analyze

streaming JavaScript programs. In particular, we advocate the use of combined offline-online static analysis as a way to accomplish fast, online analysis at the expense of a more thorough and costly offline analysis on the static code. The offline stage may be performed on a server ahead of time, whereas the online analysis would be integrated into the web browser. Through a wide range of experiments on both synthetic and real-life JavaScript code, we find that in normal use, where updates to the code are small, we can update static analysis results within the browser quickly enough to be acceptable for everyday use. We demonstrate this form of staged analysis approach to be advantageous in a wide variety of settings, especially in the context of mobile devices.

References

[1] Ajaxian. iPhone 3GS runs faster than claims, if you go by SunSpider. `http://bit.ly/RHHgO`, June 2009.

[2] L. O. Andersen. Program analysis and specialization for the C programming language. Technical report, University of Copenhagen, 1994.

[3] C. Anderson and P. Giannini. Type checking for JavaScript. In *In WOOD 04, volume WOOD of ENTCS. Elsevier, 2004. http://www.binarylord.com/work/js0wood.pdf*, 2004.

[4] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 429–452, July 2005.

[5] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A trace-based JIT compiler for CIL. Technical Report MSR-TR-2010-27, Microsoft Research, March 2010.

[6] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

[7] R. Cartwright and M. Fagan. Soft typing. *ACM SIGPLAN Notices*, 39(4):412–428, 2004.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2009.

[9] D. Crockford. The JavaScript code quality tool. `http://www.jslint.com/`, 2002.

[10] D. Crockford. AdSafe: Making JavaScript safe for advertising. `http://www.adsafe.org`, 2009.

[11] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz - open source graph drawing tools. *Graph Drawing*, pages 483–484, 2001.

[12] Facebook, Inc. FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`, 2007.

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 465–478, 2009.

[14] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[15] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2):11, 2007.

[16] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proceedings of the International Static Analysis Symposium*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[17] E. Kıcıman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proceedings of Symposium on Operating Systems Principles*, Oct. 2007.

[18] F. Logozzo and H. Venter. RATA: Rapid atomic type analysis by abstract interpretation- application to JavaScript optimization. In *Proceedings of the International Conference on Compiler Construction*, pages 66–83, 2010.

[19] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[20] Microsoft Live Labs. Live Labs Websandbox. `http://websandbox.org`, 2008.

[21] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graphs for C programs with function pointers. *Automated Software Engineering*, 11(1):7–26, 2004.

[22] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. `http://google-caja.googlecode.com/files/caja-2007.pdf`, 2007.

[23] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the USENIX Conference on Web Application Development*, June 2010.

[24] P. Thiemann. Towards a type system for analyzing JavaScript programs. *European Symposium On Programming*, 2005.

[25] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog and binary decision diagrams for program analysis. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, Nov. 2005.

[26] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of Conference on Principles of Programming Languages*, Jan. 2007.

Managing State for Ajax-Driven Web Components

John Ousterhout and Eric Stratmann
Department of Computer Science
Stanford University
{ouster,estrat}@cs.stanford.edu

Abstract

Ajax-driven Web applications require state to be maintained across a series of server requests related to a single Web page. This conflicts with the stateless approach used in most Web servers and makes it difficult to create modular components that use Ajax. We implemented and evaluated two approaches to managing component state: one, called *reminders*, stores the state on the browser, and another, called *page properties*, stores the state on the server. Both of these approaches enable modular Ajax-driven components but they both introduce overhead for managing the state; in addition the reminder approach creates security issues and the page property approach introduces storage reclamation problems. Because of the subtlety and severity of the security issues with the reminder approach, we argue that it is better to store Ajax state on the server.

1 Introduction

Ajax (shorthand for "Asynchronous Javascript And XML") is a mechanism that allows Javascript code running in a Web browser to communicate with a Web server without replacing the page that is currently displayed [6]. Ajax first became available in 1999 when Microsoft introduced the XMLHTTP object in Internet Explorer version 5, and it is now supported by all Web browsers. In recent years more and more Web applications have begun using Ajax because it permits incremental and fine-grained updates to Web pages, resulting in a more interactive user experience. Notable examples of Ajax are Google Maps and the autocompletion menus that appear in many search engines.

Unfortunately, Ajax requests conflict with the stateless approach to application development that is normally used in Web servers. In order to handle an Ajax request, the server often needs access to state information that was available when the original page was rendered but discarded upon completion of that request. Current applications and frameworks use *ad hoc* approaches to reconstruct the state during Ajax requests, resulting in code that is neither modular nor scalable.

We set out to devise a systematic approach for managing Ajax state, hoping to enable simpler and more modular code for Ajax-driven applications. We implemented and evaluated two alternative mechanisms. The first approach, which we call *reminders*, stores the state information on the browser with the page and returns the information to the server in subsequent Ajax requests. The second approach, which we call *page properties*, stores the state information on the server as part of the session. Both of these approaches allow the creation of reusable components that encapsulate their Ajax interactions, so that Web pages can use the components without being aware of or participating in the Ajax interactions. Although each approach has disadvantages, we believe that the page property mechanism is the better of the two because it scales better and has fewer security issues.

The rest of this paper is organized as follows. Section 2 introduces the Ajax mechanism and its benefits. Section 3 describes our modularity goal and presents an example component that is used in the rest of the paper. Section 4 describes the problems with managing Ajax state, and how they impact the structure of applications. Sections 5 and 6 introduce the reminder and page property mechanisms, and Section 7 compares them. Section 8 presents examples of Ajax-driven components using these mechanisms, Section 9 describes related work, and Section 10 concludes.

2 Ajax Background

Ajax allows a Web page to communicate with its originating Web server as shown in Figure 1. An Ajax-driven page is initially rendered using the normal mechanism where the browser issues an HTTP request

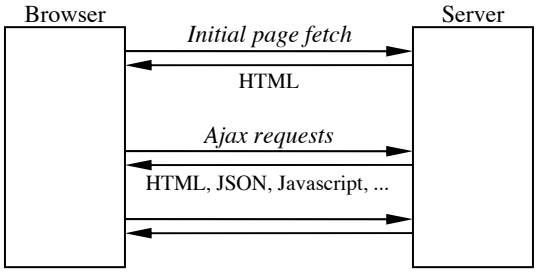


Figure 1. After the initial rendering of a Web page, Ajax requests can be issued to retrieve additional data from the server, which can be used to make incremental modifications to the page displayed in the browser.

to the server and the server responds with HTML for the page contents. Once the page has been loaded, Javascript event handlers running in that page can issue Ajax requests. Each Ajax request generates another HTTP request back to the server that rendered the original page. The response to the Ajax request is passed to another Javascript event handler, which can use the information however it pleases.

Ajax responses can contain information in any format, but in practice the response payload usually consists of one of three things:

- An HTML snippet, which the Javascript event handler assigns to the `innerHTML` property of a page element in order to replace its contents.
- Structured data in a format such as JSON [4], which the Javascript event handler uses to update the page by manipulating the DOM.
- Javascript code, which is evaluated in the browser (this form is general enough to emulate either of the other forms, since the Javascript can include literals containing HTML or any other kind of data).

The power of Ajax stems from the fact that the response is passed to a Javascript event handler rather than replacing the entire page. This allows Web pages to be updated in an incremental and fine-grained fashion using new information from the server, resulting in a more interactive user experience. One popular example is Google Maps, which allows a map to be dragged with the mouse. As the map is dragged, Ajax requests fetch additional images that extend the map's coverage, creating the illusion of a map that extends infinitely in all directions. Another example is an auto-completion menu that appears underneath the text entry for a search engine. As the user types a search term the auto-completion menu updates itself using Ajax requests to display popular completions of the search term the user is typing.

Although the term “Ajax” typically refers to a specific mechanism based on Javascript `XMLHttpRequest` objects, there are several other ways to achieve the same effect in modern browsers. One alternative is to create a new `<script>` element in the document, which causes Javascript to be fetched and executed. Another approach is to post a form, using the `target` attribute to direct the results to an invisible frame; the results can contain Javascript code that updates the main page. In this paper we will use the term “Ajax” broadly to refer to any mechanism that allows an existing page to interact with a server and update itself incrementally.

3 Encapsulation Goal

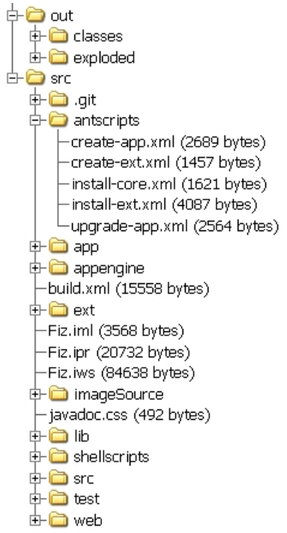
The basic Ajax mechanism is quite simple and flexible, but it is difficult to incorporate cleanly into Web application frameworks. Our work with Ajax occurred in the context of Fiz [10], an experimental server-side Web application framework under development at Stanford University. The goal for Fiz is to raise the level of programming for Web applications by encouraging a component-based approach, where developers create applications by assembling pre-existing components. Each component manages a portion of the Web page, such as:

- A form field that displays a calendar from which a user can select a particular date.
- A general-purpose table that can be sorted based on the values one or more column(s).
- A catalog display tailored to the interests of the current user.
- A shopping cart.

Ideally, a component-based approach should simplify development by encouraging reusability and by hiding inside the components many of the complexities that developers must manage explicitly today, such as the quirks of HTML, Ajax requests, and a variety of security issues.

For a component framework to succeed it must have several properties, one of the most important of which is *encapsulation*: it must be possible to use a component without understanding the details of its implementation, and it must be possible to modify a component without modifying all of the applications that use the component. For example, consider a large Web site with complex pages, such as Amazon. Teams of developers manage different components that are used on various Web pages, such as sponsored advertisements, user-directed catalog listings, and search bars. Each team should be able to modify and improve its own components (e.g., by adding Ajax interactions) without requiring changes in the pages that use those components. Thus, one of our goals for Fiz is that a component should be able to use Ajax requests in its implementation without those requests being visible outside the component.

In this paper we will use the TreeSection component from Fiz to illustrate the problems with Ajax components and the potential solutions. TreeSection is a class that provides a general-purpose mechanism for browsing hierarchical data as shown in Figure 2(a). It displays hierarchically-organized data using icons and indentation; users can click on icons to expand or hide subtrees. In order to support the display of large structures, a TreeSection does not download the entire tree



(a)

```
new TreeSection("FS.filesInDir",
                "code/Fiz");
```

(b)

Figure 2. The Fiz TreeSection component displays hierarchical information using nested indentation and allows the structure to be browsed by clicking on + and - icons: (a) the appearance of a TreeSection that displays the contents of a directory; (b) Java code to construct the TreeSection as part of a Web page.

to the browser. Instead, it initially displays only the top level of the tree; Ajax requests are used to fill in the contents of subtrees incrementally when they are expanded.

The TreeSection class automatically handles a variety of issues, such as the tree layout, Javascript event handlers to allow interactive expansion and collapsing, and the Ajax-based mechanism for filling in the tree structure on demand. It also provides options for customizing the display with different icons, node formats, and graphical effects.

In order to maximize its range of use, the TreeSection does not manage the data that it displays. Instead, whenever it needs information about the contents of the tree it invokes an external *data source*. The data source is passed the name of a node and returns information about the children of the node. When a TreeSection is constructed it is provided with the name of the data source method (`FS.filesInDir` in Figure 2(b)), along with the name of the root node of the tree (`code/Fiz` in Figure 2(b)). In the example of Figure 2 the data source reads information from the file system, but different data sources can be used to browse different structures. The TreeSection invokes the data source once to display the top level of the tree during the generation of the original Web page, then again during Ajax requests to expand nodes.

The challenge we will address in the rest of this paper is how to manage the state of components such as TreeSection in a way that is convenient for developers and preserves the encapsulation property.

4 The Ajax State Problem

Using Ajax today tends to result in complex, non-modular application structures. We will illustrate this problem for servers based on the model-view-controller (MVC) pattern [11,12]. MVC is becoming increasingly popular because it provides a clean decomposition of application functionality and is supported in almost all Web development frameworks. Similar problems with Ajax state arise for Web servers not based on MVC.

When an HTTP request arrives at a Web server based on MVC it is dispatched by the application framework to a particular method in a particular controller class, based on the URL in the request, as shown in Figure 3(a). The controller method collects data for the page from model classes and then invokes one or more view classes to render the page's HTML. If the page subsequently makes an Ajax request, the request is dispatched to another method in the same controller. The Ajax service method invokes model classes to collect data and then view classes to format a response.

Unfortunately, with this approach the controller for a page must be involved in every Ajax request emanating from the page, which breaks the application's modularity. If one of the views used in a page introduces new Ajax requests, every controller using that view must be modified to mediate those requests. As a result, it is not possible to create reusable components that encapsulate Ajax, and Ajax-driven applications tend to have complex and brittle structures.

The first step in solving this problem is to bypass the controller when handling an Ajax request and dispatch directly to the class that implements the component, as shown in Figure 3(b). Virtually all frameworks have dispatchers that can be customized to implement this behavior.

Dispatching directly to the component creates two additional issues. First, the controller is no longer present to collect data for the component, so the component must

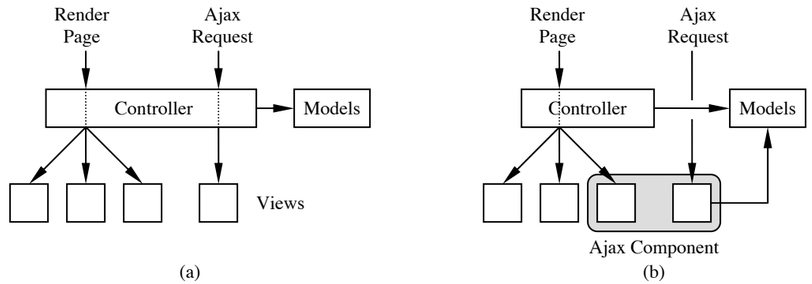


Figure 3. (a) The structure of a typical Web application today, where all HTTP requests for a page (including Ajax requests) are mediated by the page's controller class; (b) A component-oriented implementation in which all aspects of the Ajax-driven element, including both its original rendering and subsequent Ajax requests, are encapsulated in a reusable component. In (b) Ajax requests are dispatched directly to the component, bypassing the controller, and the component fetches its own data from model classes

invoke models itself to gather any data it needs. This is not difficult for the component to do, but it goes against the traditional structure for Web applications, where views receive all of their data from controllers.

The second problem created by direct dispatching relates to the state for the Ajax request. In order to handle the Ajax request, the component needs access to configuration information that was generated during the original rendering of the page. In the TreeSection example of Figure 2, the component needs the name of the data source method in order to fetch the contents of the node being expanded. This information was available at the time the component generated HTML for the original page, but the stateless nature of most Web servers causes data like this to be discarded at the end of each HTTP request; thus Ajax requests begin processing with a clean slate.

One of the advantages of dispatching Ajax requests through the controller is that it can regenerate state such as the name of the data source method. This information is known to the controller, whose code is page-specific, but not to the component, which must support many different pages with different data sources. If Ajax requests are dispatched directly to the component without passing through the controller, then there must be some other mechanism to provide the required state to the component.

Solutions to the state management problems fall into two classes: those that store state on the browser and those that store state on the server. We implemented one solution from each class in Fiz and compared them. The next section describes our browser-based approach, which we call *reminders*; the server-based approach, which we call *page properties*, is described in the following section.

5 Reminders

Our first attempt at managing Ajax state was to store the state in the browser so the server can remain stateless. When an Ajax-driven component renders its portion of the initial page it can specify information called *reminders* that it will need later when processing Ajax requests. This information is transmitted to the browser along with the initial page and stored in the browser using Javascript objects (see Figure 4). Later, when Ajax requests are issued, relevant reminders are automatically included with each Ajax request. The reminders are unpacked on the server and made available to the Ajax handler. An Ajax handler can create additional reminders and/or modify existing reminders; this

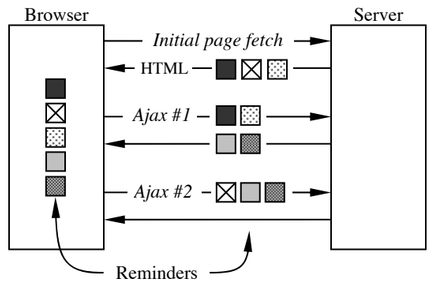


Figure 4. Reminders are pieces of state that are generated by the server, stored in the browser, and returned in later Ajax requests. Additional reminders can be generated while processing Ajax requests.

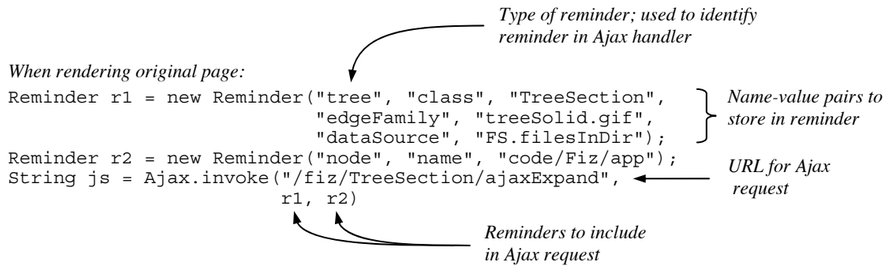


Figure 5. Examples of the APIs for creating and using reminders in the Fiz TreeSection.

information is returned to the browser along with the Ajax response, and will be made available in future Ajax requests. Reminders are similar to the View State mechanism provided by Microsoft's ASP.NET framework; see Section 9 for a comparison.

Figure 5 illustrates the APIs provided by Fiz for managing reminders. Each Reminder object consists of a type and a collection of name-value pairs. The type is used later by Ajax request handlers to select individual reminders among several that may be included with each request. For example, the TreeSection creates one reminder of type `tree` containing overall information about the tree, such as its data source and information needed to format HTML for the tree. It also creates one reminder of type `node` for each expandable node in the tree, which contains information about that particular node. When the user clicks on a node to expand it, the resulting Ajax request includes two reminders: the overall `tree` reminder for the tree, plus the `node` reminder for the particular node that was clicked.

Fiz automatically serializes Reminder objects as Javascript strings and transmits them to the browser. Fiz also provides a helper method `Ajax.invoke`, for use in generating Ajax requests. `Ajax.invoke` will create a Javascript statement that invokes an Ajax request for a given URL and includes the data for one or more reminders. The result of `Ajax.invoke` can be incorporated into the page's HTML; for example, TreeSection calls `Ajax.invoke` once for each expandable node and uses the result as the value of an `onclick` attribute for the HTML element displaying the + icon.

When an Ajax request arrives at the Web server, Fiz dispatches it directly to a method in the TreeSection class. Fiz automatically deserializes any reminders attached to the incoming request and makes them available to the request handler via the `getReminder` method. In the TreeSection example the request handler collects information about the node being expanded (by calling the data source for the tree), generates HTML to represent the node's contents, and returns the HTML to the browser, where it is added to the existing page. If the node's contents include expandable sub-nodes, an additional `node` reminder is created for each of those sub-nodes and included with the Ajax response.

5.1 Evaluation of reminders

The reminder mechanism makes it possible to encapsulate Ajax interactions within components, and it does so without storing any additional information on the server. Ajax interactions are not affected by server crashes and reboots, since their state is in the browser.

Reminders have two disadvantages. First, they introduce additional overhead for transmitting reminder data to the browser and returning it back to the server. In order to minimize this overhead we chose a granular approach with multiple reminders per page: each Ajax request includes only the reminders needed for that request. In our experience implementing Ajax components we have not yet needed reminders with more than a few dozen bytes of data, so the overhead has not been a problem.

The second disadvantage of reminders is that they introduce security issues. The data stored in reminders

represents internal state of the Web server and thus may need to be protected from hostile clients. For example, the reminders for the TreeSection include the name of the data source method and the name of the directory represented by each node. If a client modifies reminders it could potentially invoke any method in the server and/or view the contents of any directory.

Fiz uses message authentication codes (MACs) to ensure the integrity of reminders. Each reminder includes a SHA-256 MAC computed from the contents of the reminder using a secret key. There is one secret key for each session, which is stored in the session and used for all reminders associated with that session. When the reminder is returned in an Ajax request the MAC is verified to ensure that the reminder has not been modified by the client.

MACs prevent clients from modifying reminders, but they don't prevent clients from reading the contents of reminders. This could expose the server to a variety of attacks, depending on the content of reminders. For example, if passwords or secret keys were stored in reminders then hostile clients could extract them. It is unlikely that an application would need to include such information in reminders, but even information that is not obviously sensitive (such as the name of the data source method for the TreeSection) exposes the internal structure of the server, which could enable attackers to identify other security vulnerabilities. Unfortunately, it is difficult to predict the consequences of exposing internal server information. In order to guarantee the privacy of reminders they must be encrypted before computing the MAC. Fiz does not currently perform this encryption.

The granular nature of reminders also compromises security by enabling mix-and-match replay attacks. For example, in the TreeSection each Ajax request includes two separate reminders. The first reminder contains overall information about the tree, such as the name of the data source method. The second reminder contains information about a particular node being expanded, including the pathname for the node's directory. A hostile client could synthesize AjaxRequests using the tree reminder for one tree and the node reminder for another; this might allow the client to access information in ways that the server would not normally allow.

One solution to the replay problem is to combine all of the reminders for each page into a single structure as is done by View State in ASP.NET. This would prevent mix-and-match attacks but would increase the mechanism's overhead since all of the reminders for the page would need to be included in every request. Another approach is to limit each Ajax request to a single reminder. Each component would need to aggregate all of the information it needs into one reminder; for example, the TreeSection would duplicate the information about the tree in each node reminder. This approach would make it difficult to manage mutable state: if, for example, some overall information about the tree were modified during an Ajax request then every node reminder would need to be updated. Yet another approach is to use unique identifiers to link related reminders. For example, the tree reminder for each TreeSection might contain a unique identifier, and the server might require that each node reminder contains the same unique identifier as its tree reminder. This would prevent a node reminder from being used with a different tree reminder, but it adds to the complexity of the mechanism.

As we gained more experience with reminders we became concerned that it would be difficult to use them in a safe and efficient fashion, and that these problems will increase as Ajax usage becomes more pervasive and sophisticated. If the framework handles all of the security issues automatically it will require a heavyweight approach such as aggregating all state into a single reminder that is both encrypted and MAC-protected. However, this would probably not provide acceptable performance for complex pages with many Ajax-driven components. On the other hand, if developers are given more granular control over the mechanism they could probably achieve better performance, but at a high risk for security vulnerabilities. Even highly skilled developers are unlikely to recognize all of the potential loopholes, particularly when working under pressure to bring new features to market. For example, when asked to create an alternative implementation of TreeSection in Ruby on Rails for comparison with Fiz, a Stanford undergraduate with experience developing Web applications did not recognize that the node names need to be protected from tampering. The prevalence of SQL injection attacks [1] also indicates how difficult it is for Web developers to recognize the security implications of their actions.

6 Page Properties

Because of the problems with the reminder mechanism we decided to implement a different approach where component state is kept in the server instead of the browser. This eliminated the security and overhead issues with reminders but introduced a different problem related to garbage collection.

The server-based approach is called *page properties*. A page property consists of a name-value pair that is accessible throughout the lifetime of a particular Web page. The name must be unique within the page and the value may be any serializable Java object. Page properties may be created, examined, and updated at any time using a simple API consisting of `getPageProperty` and `setPageProperty` methods. For example, the TreeSection creates a page property for each tree when it renders the top level of the tree during initial page display. The page property contains overall information about the tree, such as the name of the data source method, plus information about each node that has been rendered in the tree.

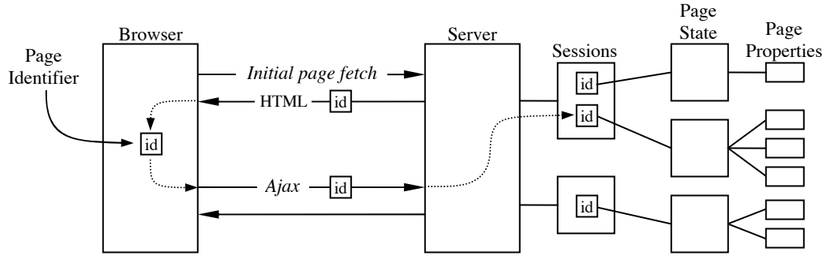


Figure 6. Fiz stores page properties on the server as part of the session. Each Web page is assigned a unique identifier, which is included in the page and returned to the server as part of each Ajax request; this allows the server to locate the properties for the page.

When an Ajax request arrives to expand a TreeSection node, it is dispatched directly to the TreeSection class just as in the reminder approach. The Ajax request includes an identifier for a particular tree instance (in case there are several trees in a single page) and an identifier for the node that is being expanded. The Ajax handler in TreeSection retrieves the page property for the tree instance, looks up the node identifier in the page property object, and uses that information to retrieve information about the children of the expanded node; this information is used to generate HTML to return to the browser, and also to augment the page property object with information about children of the expanded node.

The names of page properties are only unique within a page, so Fiz associates a unique *page identifier* with each distinct Web page and uses it to separate the page properties for different pages. A page identifier is assigned during the initial rendering of each page and is stored in the page using a Javascript variable (see Figure 6). Subsequent Ajax requests and form posts coming from that page automatically include the page identifier as an argument. Operations on page properties apply to the properties associated with the current page.

Fiz stores page properties using the session mechanism: all of the properties for each page are collected into a PageState object, and each session can contain multiple PageState objects, indexed by their page identifiers (see Figure 6). Storing page properties in the session ensures that they are preserved across the various requests associated with a page, even though the individual requests are implemented in a stateless fashion. Page properties have the same level of durability as other session information.

6.1 Evaluation of page properties

Page properties avoid the issues that concerned us with reminders: the only information sent to the browser is the page identifier, so page properties reduce the overhead of transmitting data back and forth across the network. Page properties also avoid the security issues associated with reminders, since state information never leaves the server. The use of sessions to store page properties ensures isolation between different users and sessions.

However, page properties introduce new issues of their own. First, in order for page properties to survive server crashes they must be written to stable storage after each request along with the rest of the session data. If page properties contain large amounts of information then they could still result in substantial overhead (e.g. for a TreeSection displaying hundreds of expandable nodes there could be several kilobytes of page properties). However, the overhead for saving page properties on the server is likely to be less than the overhead for transmitting reminders back and forth over the Internet to browsers.

Fiz currently stores page properties using the standard session facilities provided by the underlying Java servlets framework. However, it may ultimately be better

to implement a separate storage mechanism for page properties that is optimized for their access patterns. For example, many session implementations read and write the entire session monolithically; however, a given request will only use the page properties for its particular page, so it may be inefficient to read and write all of the properties for other pages at the same time. In addition, the standard session mechanisms for reflecting session data across a server pool may not be ideal for page properties.

A second, and more significant, problem concerns the garbage collection of page properties: when is it safe to delete old page properties? Unfortunately the lifetime of a Web page is not well-defined: the server is not notified when the user switches to a different page; even if it were notified, the user can return to an old page at any time by clicking the "Back" browser button. There is no limit on how far back a user can return. If the server deletes the page properties for a page and the user then returns to that page, Ajax requests from the page will not function correctly.

To be totally safe, page properties must be retained for the lifetime of the session. However, this would bloat the size of session data and result in high overheads for reading and writing sessions (most frameworks read and write all of the data for a session monolithically, so all page properties for all pages will be read and written during each request).

For the Fiz implementation of page properties we have chosen to limit the number of pages in each session for which properties are retained. If the number of PageState objects for a session exceeds the limit, the least recently used PageState for that session is discarded. If a user invokes an Ajax operation on a page whose properties have been deleted, Fiz will not find the PageState object corresponding to the page identifier in the request. Fiz then generates an Ajax response that displays a warning message in the browser indicating that the page state is stale and suggesting that the user refresh the page. If the user refreshes the page a fresh page identifier will be allocated and Ajax operations will work once again; however, the act of refreshing the page will reset the page display (in the case of the TreeSection the tree will revert to its original display showing only the top-level nodes). We call this situation a *broken page*. Broken pages will be annoying for users so it is important that they not occur very frequently (of course, users will not notice that a page is broken unless they invoke an Ajax operation that requires page properties).

The frequency of broken pages can be reduced by retaining more PageState objects for each session, but this will increase the overhead for storing page properties.

In order to estimate the frequency of broken pages with LRU replacement, we ran a trace-driven simulation experiment. We wrote a Firefox add-on that records all operations that change the current Web page being displayed and transmits that information to a central server at regular intervals. The information logged includes new pages, "back" and "forward" history operations, redisplays, and tab switches. We used the add-on to collect data from about thirty people (mostly Stanford students and faculty) over a period of two months (approximately 200,000 page views in total). We then used the data to drive two simulations of the page property mechanism.

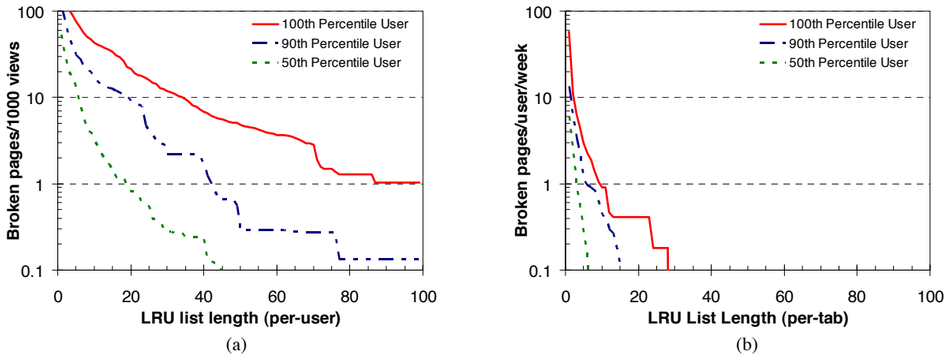


Figure 7. A trace-driven simulation of LRU lists for page properties assuming a single LRU list for each user's interaction with each server host (a) and separate LRU lists for each tab (b). The top curve in each figure shows the rate of broken pages for the worst-case user for each LRU list size, and the bottom curve shows behavior for the median user. 1000 page views represents roughly one week's worth of activity for a typical user.

In the first simulation (Figure 7 (a)) we assumed one LRU list of page properties for each session (a particular user accessing a particular server host). The figure shows the rate of broken pages as a function of LRU list length, both for "typical" users and for more pathological users. It assumes that every page uses Ajax and requires page properties. For the trace data we collected, an LRU limit of 50 pages per session results in less than one broken page per thousand page views for most users. The actual frequency of broken pages today would be less than suggested by Figure 7, since many pages do not use Ajax, but if Ajax usage increases in the future, as we expect, then the frequency of broken pages could approach that of Figure 7.

The primary reason for broken pages in the simulations is switches between tabs. For example, if a user opens an Ajax-driven page in one tab, then opens a second tab on the same application and visits several Ajax-driven pages, these pages may flush the page properties for the first tab, since all tabs share the same session. If the user switches back to the first tab its page will be broken.

Figure 7(b) shows the frequency of broken pages if a separate LRU list is maintained for each tab in each session; in this scenario LRU lists with 10 entries would eliminate almost all broken pages. Per-tab LRU lists will result in more pages cached than per-session LRU lists of the same size, since there can be multiple tabs in a session. In our trace data there were about 2 tabs per session on average; per-tab LRU lists used roughly the same memory as per-session LRU lists 2.5-3x as long. Overall, per-tab LRU lists would result in fewer broken pages with less memory utilization than per-user LRU lists, and they also improve the worst-case behavior.

Unfortunately, today's browsers do not provide any identifying information for the window or tab responsible for a given HTTP request: all tabs and windows participate indistinguishably in a single session. Such information would be easy for a browser to provide in a backwards-compatible fashion: it could consist of an HTTP header that uniquely identifies the window or tab for the request; ideally it would also include information indicating when tabs have been closed. Window/tab information also has other uses: for example, it would enable applications to implement sub-sessions for each tab or window so that the interaction stream for each tab/window can be handled independently while still providing shared state among all of the tabs and windows. Without this information, some existing Web applications behave poorly when a single user has multiple tabs open on the same application, because interactions on the different tabs get confused.

In the absence of tab identifiers, and assuming that Ajax becomes pervasive, so that virtually all Web pages need Ajax state, we conclude that servers would need to retain state for about 50 pages per session in order to reduce the frequency of broken pages to an acceptable level. In our current uses of page properties the amount of state per Ajax component is typically only a few tens of bytes (see Section 8), so storing state for dozens of pages would not create a large burden for servers.

It would also be useful to add priorities to the page property mechanism; the state for higher priority pages would be retained in preference to that for lower priority pages. It is particularly annoying for users to lose partially entered form data, so a priority mechanism could be used to preserve the state for pages containing unsubmitted forms. Once the form has been submitted successfully, the priority of its data could be reduced to allow reclamation.

7 Comparisons

After implementing and using both page properties and reminders, our conclusion is that the page property approach is the better of the two. Both mechanisms introduce overhead to transmit or store state, but the overheads for page properties are likely to be lower, since the state can be stored locally on the server without transmitting it over the Internet. The reminder approach has unique problems related to security, and the page property approach has unique problems related to garbage collection. However, we believe that the garbage collection issues for page properties are manageable and that the subtle security loopholes that can occur in the reminder mechanism will cause more catastrophic problems.

Ideally, the state management mechanism should scale up gracefully as Web applications make more intensive use of Ajax interactions and develop more complex state. We believe that page properties are likely to handle such scaling better than reminders. Consider a Web page with a collection of related Ajax-driven components. It is possible that multiple Ajax requests might be issued from different components simultaneously; for example, one component might be refreshing itself periodically based on a timer, while another component issues an Ajax request because of a user interaction. If the components are related then they may also share state (reminders or page properties). With the reminder approach each request will receive a separate copy of the relevant reminders and there is no obvious way for

the concurrent requests to serialize updates to their reminders. With the page property approach the concurrent requests will access the same page properties, so they can synchronize and serialize their updates to those properties using standard mechanisms for manipulating concurrent data structures.

In both the reminder and page property mechanisms the state must be serializable. For reminders the state must be serialized so it can be transmitted to and from the browser; for page properties the state must be serialized to save it as part of the session.

8 Component Examples in Fiz

We have implemented three components in Fiz that take advantage of the page property mechanism; they illustrate the kinds of state that must be managed for Ajax requests.

The first component is the TreeSection that has already been described. The state for this component divides into two parts. The first part consists of overall state for the entire tree; it includes the name of the data source method that supplies data about the contents of the tree, the `id` attribute for the HTML element containing the tree, and four other string values containing parameters that determine how tree nodes are rendered into HTML. The second part of the state for a TreeSection consists of information for each node that has been displayed so far; it includes the `id` attribute for the node's HTML element and an internal name for the node, which is passed to the data source in order to expand that node.

The second component is an auto-complete form element. The component renders a normal `<input type="text">` form element in the page but attaches event handlers to it. As the user types text into the element, Ajax requests are issued back to the server with the partial text in the form element. The server computes the most likely completions based on the partial text and returns them back to the browser where they are displayed in a menu underneath the form element. The auto-complete component handles relevant mouse and keyboard events, issues Ajax requests, displays and undisplays the menu of possible completions, and allows the user to select completions from the menu. However, the auto-complete component does not contain code to compute the completions since this would restrict its reusability. Instead, it calls out to an external method to compute the completions during Ajax requests. The name of this method is provided as a parameter during the rendering of the original page, and it is saved in a page property along with the `id` attribute of the HTML form element. The auto-complete component is general-purpose and reusable: there can be multiple auto-complete form elements on the same page, and each auto-complete element can use a different mechanism to compute completions.

The third usage of page properties in Fiz is for form validation. When a form is initially rendered in Fiz, one or more validators can be associated with each field in the form. Information about these validators is saved in a page property, including the name of a method that will perform the validation, one or more parameters to pass to that method (for example, the `validateRange` method takes parameters specifying the end points of the valid range), and additional information used to customize the HTML formatting of error messages when validation fails. When the form is posted Fiz uses the information in the page property to validate the information in the form; if any validations fail then Fiz automatically returns error messages for display in the browser. In this case the browser-server communication mechanism is a form post rather than an Ajax request, but it uses the same page property mechanism. The form validation mechanism also allows forms to be validated dynamically as the user fills them in, using Ajax requests to request validation.

9 Related Work

9.1 View State

Most existing Web frameworks provide little or no support for managing the state of Ajax requests. One exception is Microsoft's ASP.NET framework, which includes a mechanism called View State that is similar to the reminder mechanism in Fiz. View State is used extensively by components in ASP.NET [5,8]. The View State mechanism provides each control with a ViewState property in which it can store key-value pairs. ASP.NET automatically packages all of the ViewState properties for all components into a single string, which is sent to the browser as part of the HTML for a page. Any "postbacks" for that page (which include both Ajax requests and form posts) include the View State, which is deserialized and made available to all of the components that handle the postback. Components can modify their View State while handling the request, and a new version of the View State is returned to the browser as part of the response.

The primary difference between View State and reminders is that View State is monolithic: all of the state for the entire page is transmitted in every interaction. In contrast, reminders are more granular: each request includes only the reminders needed for that request. The View State approach avoids the complexity of determining what information is needed for each request, but it results in larger data transfers: every request and every response contains a complete copy of the View State for the entire page. Many of the built-in ASP.NET components make heavy use of View State, so it is not unusual for a page to have 10's of Kbytes of View State [8]. Complaints about the size of View State are common, and numerous techniques have been discussed for reducing its size, such as selectively disabling View State for some components (which may impact their behavior).

The security properties of View State are similar to those for reminders. View State is base-64 encoded to make it difficult to read, and by default a MAC is attached and checked to prevent tampering. However, the MAC is not session-specific so applications must attach an additional "salt" to prevent replay attacks between sessions; reminders eliminate this problem by using a different MAC for each session. View State also supports encryption, but by default it is disabled.

It is possible for applications to modify the View State mechanism so that View State is stored on local disk instead of being transmitted back and forth to the browser. This provides a solution somewhat like page properties. However, the approach of storing View State on disk does not appear to be heavily used or well supported. For example, the problem of garbage collecting old View State is left up to individual applications.

9.2 Query Values

A simpler alternative than either reminders or page properties is to use URL query values: if a component needs state information to handle an Ajax request, it can serialize that state when rendering the original page and incorporate it into the URL for the Ajax request as a query value; when the request is received, the component can read the query value and deserialize its state. This approach works well for simple state values where security is not an issue, and it is probably the most common approach used for Ajax requests today. However, it leaves serialization and deserialization up to the application and does not handle any of the security issues associated with Ajax state. In addition, the query-value approach does not easily accommodate changes to the state, since this would require modifying the URLs in the browser. The reminder mechanism handles all of these issues automatically. If the Ajax state becomes large, then query values will have overhead problems similar to those of reminders and View State.

9.3 Alternative application architectures

The issues in managing Ajax state arise because the functionality of a Web application is split between the server and the browser. In most of today's popular Web development frameworks this split must be managed explicitly by application developers. However, there exist alternative architectures for Web applications that change this split and the associated state management issues.

One approach is to change the application structure so that it is driven by Javascript in the browser. In this approach the server does not generate HTML; its sole purpose is to provide data to the browser. The application exists entirely as Javascript running in the browser. The initial page fetch for the application returns an empty HTML page, plus `<script>` elements to download Javascript. The Javascript code makes Ajax requests to the server to fetch any data needed for the page; the server returns the raw data, then Javascript code updates the page. As the user interacts with the page additional Ajax requests are made, which return additional data that is formatted into HTML by Javascript or used to modify the DOM. Google's Gmail is one example of such an application.

In a Javascript-driven application there is no need for the server to maintain state between Ajax requests: all of the interesting state is maintained in Javascript structures in the browser. For example, if a Javascript-driven application contains a component like the TreeSection, parameters for the TreeSection such as the data source method and HTML formatting information are maintained in Javascript variables on the browser. The server can process each incoming request independently and there is no state carried over from one request to another. There are still security issues with this approach, but they are simpler and more obvious: the server treats each request as potentially hostile and validates all arguments included with the request.

The biggest disadvantage of Javascript-driven applications is the overhead of downloading all of the Javascript code for the application; for complex applications the Javascript code could be quite large [9]. This approach also requires the entire application to be written in Javascript, whereas the traditional server-oriented approach permits a broader selection of languages and frameworks. There exist a few frameworks, such as Google's GWT [7], where application code can be written in other languages (Java in the case of GWT) and the framework automatically translates the code to Javascript.

Javascript-driven applications also have the disadvantage of exposing the application's intellectual property, since essentially the entire application resides on the browser where it can be examined. To reduce this exposure some applications use Javascript obfuscators that translate the Javascript code into a form that minimizes its readability. These obfuscators can also compact the code to reduce the download overheads.

Another alternative architecture is one where the partitioning of functionality between server and browser is handled automatically. The developer writes Web application code without concern for where it will execute, and the framework automatically decides how to partition the code between server and browser. In this approach the framework handles all of the issues of state management, including the related security issues. Automatic partitioning has been implemented in several experimental systems, such as Swift [2]. Although developers can use these systems without worrying about Ajax state issues, the framework implementers will still have to face the issues addressed by this paper.

9.4 Dynamic application state

The Ajax state discussed in this paper consists of information used to manage a Web user interface, such as information about the source(s) of data and how to render that data in the Web page. However, Ajax is often used to address another state management problem, namely the issue of dynamic application state. If the state underlying an application, such as a database or a system being monitored, changes while a Web page is being displayed, Ajax can be used to reflect those changes immediately on the Web page. A variety of mechanisms have been implemented for "pushing" changes from Web servers to browsers, such as Comet [3,13]. The issue of managing dynamic application state is orthogonal to that of managing Ajax state: for example, the techniques described in this paper could be used in a Comet-based application to keep track of the data sources that are changing dynamically.

10 Conclusion

Managing the state of Web applications has always been complex because the application state is split between server and browser. The introduction of Ajax requests requires additional state to maintain continuity across requests related to a single page, yet the stateless nature of most servers makes it difficult to maintain this state. Furthermore, if Ajax-driven interactions are to be implemented by reusable components then even more state is needed to maintain the modularity of the system.

This paper has explored two possible approaches to maintaining Ajax state, one that stores the state on the browser (reminders) and one that stores state on the server (page properties). Although both approaches meet the basic needs of Ajax-driven components, each of them has significant drawbacks. The browser-based approach introduces overheads for shipping state between browser and server, and it creates potential security loopholes by allowing sensitive server state to be stored in the browser. The server-based approach introduces overheads for saving state as part of sessions, and it has garbage-collection issues that can result in the loss of state needed to handle Ajax requests if the user returns to old pages. Based on our experiences we believe that the disadvantages of the server-based approach are preferable to those of the browser-based approach.

In the future we expect to see the use of Ajax increase, and we expect to see pages with more, and more complex, Ajax components. As a result, the issues of managing Web application state will probably become even more challenging in the future.

11 Acknowledgments

Jeff Hammerbacher and the anonymous referees made many useful comments that improved the presentation of the paper. This work was supported in part by a grant from the Google Research Awards program.

12 References

[1] Anley, Chris, *Advanced SQL Injection in SQL Server Applications* (http://www.nextgenss.com/papers/advanced_sql_injection.pdf).

[2] Chong, S., Liu, J., Myers, A., Qi, X., Zheng, L., and Zheng, X., "Secure Web Applications via Automatic Partitioning," *Proc. 21st ACM Symposium on Operating System Principles*, October 2007, pp. 31-44.

[3] Cometd project home page (http://cometd.org/).

[4] Crockford, D., *The application/json Media Type for JavaScript Object Notation (JSON)*, IETF RFC 4627, http://tools.ietf.org/html/rfc4627, July 2006.

[5] Esposito, D., "The ASP.NET View State," *MSDN Magazine*, February 2003,

[6] Garrett, Jesse James, *Ajax: a New Approach to Web Applications*, http://www.adaptivepath.com/ideas/essays/archives/000385.php.

[7] Google Web Toolkit home page (http://code.google.com/webtoolkit/).

[8] Mitchell, S., *Understanding ASP.NET View State*, http://msdn.microsoft.com/en-us/library/ms972976.aspx, May 2004.

[9] Optimize a GWT Application (http://code.google.com/webtoolkit/doc/latest/DevGuideOptimizing.html).

[10] Ousterhout, J., *Fiz: A Component Framework for Web Applications*, Stanford CSD technical report, http://www.stanford.edu/~ouster/cgi-bin/papers/fiz.pdf, January 2009.

[11] Reenskaug, Trygve, *Models-Views-Controllers*, Xerox PARC technical notes, December, 1979 (http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf).

[12] Reenskaug, Trygve, *Thing-Model-View*-Editor, Xerox PARC technical note, May 1979 (http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf).

[13] Russell, Alex, *Comet: Low Latency Data for the Browser*, March 2006 (http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/).

SVC: Selector-based View Composition for Web Frameworks

William P. Zeller and Edward W. Felten
{wzeller,felten}@cs.princeton.edu
Princeton University

Abstract

We present *Selector-based View Composition* (SVC), a new programming style for web application development. Using SVC, a developer defines a web page as a series of transformations on an initial state. Each transformation consists of a selector (used to select parts of the page) and an action (used to modify content matched by the selector). SVC applies these transformations on either the client or the server to generate the complete web page.

Developers gain two advantages by programming in this style. First, SVC can automatically add Ajax support to sites, allowing a developer to write interactive web applications without writing any JavaScript. Second, the developer can reason about the structure of the page and write code to exploit that structure, increasing the power and reducing the complexity of code that manipulates the page's content.

We introduce SVC as a stateless, framework-agnostic development style. We also describe the design, implementation and evaluation of a prototype, consisting of a PHP extension using the WebKit browser engine [37] and a plugin to the popular PHP MVC framework Code Igniter [8]. To illustrate the general usefulness of SVC, we describe the implementation of three example applications consisting of common Ajax patterns. Finally, we describe the implementation of three post-processing filters and compare them to currently existing solutions.

1 Introduction

The growth of Ajax has resulted in increased interactivity on the web but imposes additional developmental costs on web developers. Many browsers do not support JavaScript (and therefore Ajax), including search engine crawlers, older browsers, browsers with JavaScript (JS) disabled, and screen readers. These browsers will not be compatible with portions of a site which exclusively use Ajax.

The standard "best practice" when creating a site that supports both JS and non-JS browsers is to use a technique called *progressive enhancement* [5]. A developer first creates a site that works in non-JS browsers and then uses JavaScript to add interactivity. For example, a page might include a link titled "Click here for more" which loads a new page when clicked. Progressive enhancement might involve modifying that link to load and insert additional content in-line using Ajax, without navigating to a new page. Progressively enhanced sites work in both JS and non-JS browsers, but require the developer to duplicate much of the site's functionality in order to respond appropriately to both Ajax and non-Ajax requests. (We use the term *non-Ajax request* to refer to a request that loads an entirely new page and *Ajax request* to refer to a request made by an existing page which completes without leaving the current page).

Alternatively, a developer may choose to only support JS browsers. With this approach, a developer directly implements functionality using JavaScript or uses a framework such as Google Web Toolkit (GWT) [20], which provides server-side support for generating client-side JavaScript, but provides no easy way of supporting non-JS browsers.

Currently, developers face a choice. They can either support both JS and non-JS browsers by duplicating much of a site's functionality or risk preventing certain users and search engines from accessing and indexing their sites.

We believe this tradeoff is unnecessary. We present a new programming style, SVC, which allows a site to be constructed in a way that can support Ajax and non-Ajax requests automatically, providing both interactivity and backwards compatibility.

1.1 Background: MVC

The model/view/controller (MVC) architectural pattern is commonly used in web frameworks (e.g., Django, Ruby on Rails, Code Igniter, Struts, etc. [11, 29, 8, 34]). In web MVC (which differs somewhat from traditional MVC), a model encapsulates application data in a way that is independent of how that data is rendered by the application. The view accepts some number of models as input and transforms them into appropriate output that will be sent to the browser. The controller connects the models and views, typically by bundling up model data and sending it to the view for rendering.

The view accepts data as input and produces a string as output. This output may include information about its type (e.g., HTML, XML, or JSON [23]), but is otherwise treated as an opaque string by the framework. After being manipulated by some number of post-processing filters, the string is sent directly to the browser. Because the view's output is treated as an opaque string, it is difficult for the framework to reason about the structure of the content. These views are complicated by the need to provide both Ajax and non-Ajax versions of a site.

1.2 Our Approach: SVC

SVC is a programming style that changes how views construct their output. Instead of returning an opaque string, a view describes the output as a sequence of transformations. Each transformation consists of a selector, used to query the document, and an action, used to modify the DOM [2] nodes matched by the selector. The framework, which previously operated on an opaque string, now has knowledge of both the structure of the page as well as how the page was composed. SVC expresses page content in a manner that is succinct, powerful, and portable.

A key benefit of SVC is that the framework can choose whether to apply the transformations on the server or on the client (where possible). When called on to respond to an Ajax request, it returns a list of transformations needed to convert the current page into the new page, using client-side JS. The use of selectors allows both client- and server-side code to convert the list of transformations into the same complete document, allowing SVC to provide automatic Ajax support and automatic progressive enhancement to pages written in this style. This benefit relies on the portability (from server to client) of SVC's transformation rules.

SVC does not attempt to replace existing template systems. Instead, a developer *composes* different *views* (which may be the output of a template). The developer describes where the composition should occur using *selectors* (described in Section 3.1.1). Developers may continue to use any template language with SVC.

Additionally, SVC does not interfere with existing code. Both JavaScript and existing controllers not using SVC will continue to work without modification. This allows SVC to be added to an existing site and used only when necessary (without the need to immediately refactor legacy code).

Finally, SVC is able to use its knowledge of a page's structure to provide developers with a succinct and familiar post-processing mechanism. This allows developers to write code to filter a page based on its content without forcing them to parse the page themselves.

1.3 Contributions and Organization

This paper makes the following contributions:

- §2 We describe the architecture of SVC and discuss how it differs from the traditional MVC model.
- §3 We describe the server-side and client-side components that make up SVC and discuss our design decisions.
- §4 We describe the implementation of an SVC prototype. The prototype consists of a PHP extension written in C++ using the WebKit engine [37], a PHP plugin for the MVC framework Code Igniter [8], and a client-side JavaScript plugin that handles Ajax responses.
- §5 We present a complete minimal example of a site implemented with and without SVC to show how our approach differs. We also briefly describe a number of example sites and filters we implemented using SVC.

We evaluate the performance of our implementation in Section 6, before discussing related work and concluding.

2 Architecture

SVC extends the model/view/controller (MVC) pattern to manage responsibilities currently handled by the developer. Fig. 1(a) shows how a request travels through a traditional MVC application which supports both Ajax and non-Ajax requests. (The "model" in MVC is irrelevant to this discussion and omitted.)

The request is sent to a controller which calls the appropriate view depending on the type of request. If the request is a non-Ajax request, the non-Ajax view is called. This view outputs the HTML document which is rendered by the browser.

In the case of an Ajax request, the controller calls the Ajax view which outputs data in a format that can be read

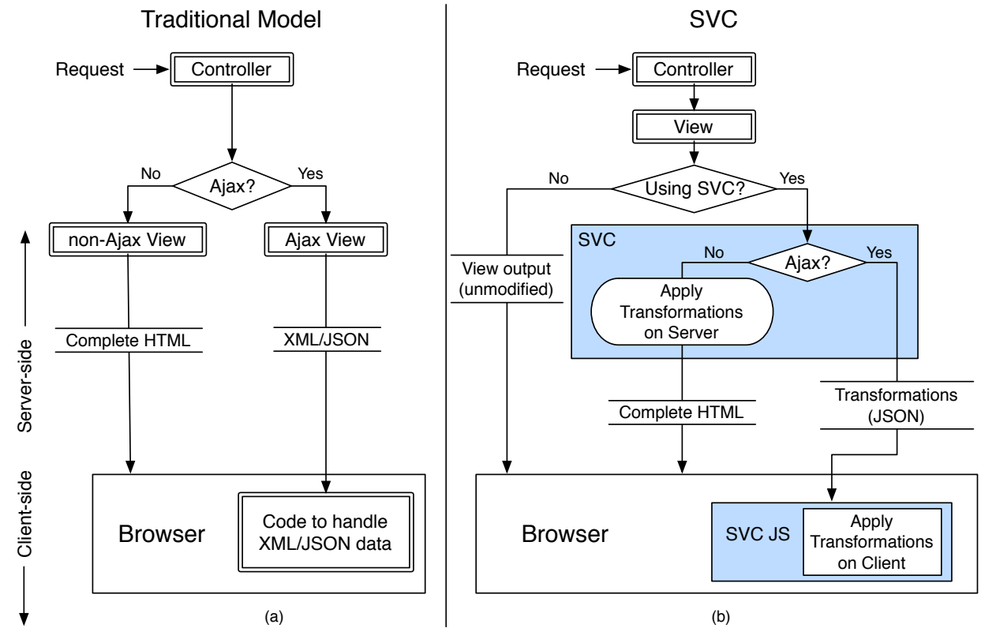


Figure 1: Architecture of SVC. Boxes surrounded by a double line represent components that are the developer's responsibility. Fig. 1(a) shows the traditional MVC architecture (with models omitted) for sites supporting both Ajax and non-Ajax requests. A request is sent to the controller which loads either an Ajax or non-Ajax view. The developer must write both views, as well as client-side code which handles the output of her Ajax view. Fig. 1(b) shows the SVC model. A request is sent to a single view, which can decide to use SVC or not use SVC. The view uses SVC by sending it a transformation list. SVC responds to a non-Ajax request by applying the transformations on the server and returning a complete HTML page. SVC responds to an Ajax request by simply returning the transformation list as JSON. The transformation list is applied by the client-side SVC JS to the current document in the browser. In Fig. 1(b), SVC allows the developer to write only one view and no client-side code.

by client-side code (typically XML or JSON). While this output may contain HTML snippets, custom client-side JS is required to insert the snippets in the page at the appropriate positions. Both views must be created by the developer, who must duplicate functionality to provide both Ajax and non-Ajax output. In addition, the developer needs to write client-side code to handle the output created by her Ajax view. Because the framework knows nothing about the view's structure, it cannot assist in this process.

SVC's extension to MVC is shown in Fig. 1(b). The request is sent to a controller which now calls only one view regardless of the type of request. Instead of outputting HTML for a non-Ajax request and XML/JSON for an Ajax request, the developer describes the page as a series of transformation rules as well as any items on which the transformations depend. The transformation list is sent to the SVC, which decides how to act based on the type of request.

In the case of a non-Ajax request, SVC applies the transformation list on the server-side, creating a complete HTML document on the server. This document is then sent to the browser.

In the case of an Ajax request, SVC converts the transformation list to a form readable by client-side code. This serialized list is sent to the client where the transformations are applied directly to the current document in the browser. As SVC includes all the client-side code necessary to apply the transformations to the document, the developer does not need to write any client-side JS when creating an Ajax-compatible site.

Using the architecture in Fig. 1, SVC is able to automatically generate both Ajax and non-Ajax versions of a site. In addition, SVC needs to progressively enhance the site and inject a script into all pages to handle client-side actions. Specifically, SVC adds a `CLASS` attribute

to each element needing to be enhanced and inserts a `SCRIPT` tag into the head of the document which points to the client-side SVC JS. This script adds a *click* event handler to each element with the above class and also manages requesting and applying the transformation list. Progressive enhancement and script injection are made possible by a post-processing filtering system that SVC uses internally and exposes to developers.

SVC is stateless–it operates only on the transformation list provided by the view. It does not store HTML or output from previous requests, nor does it depend on the state of the browser (i.e., which requests have previously occurred).

SVC consists of a server-side API, server-side code to handle both Ajax and non-Ajax requests, client-side code to handle the output of Ajax requests, and a filtering module, both used internally and made available to developers.

3 Design

3.1 Server-side Components

Developers use the SVC server-side API to describe how a page should be constructed.

3.1.1 Selectors and Actions

SVC asks a developer to construct a page by defining a list of transformation rules. These rules consist of a selector, used to locate one or more nodes in the DOM, and an action, used to modify nodes matched by the selector. By defining a page as a series of transformation rules, SVC is able to respond to both Ajax and non-Ajax requests appropriately. SVC is able to decide whether to send the list of transformations to the client for conversion using JS or to convert the list on the server (by applying each transformation, in order) and return a full HTML document to the browser.

We consider next how a transformation rule is expressed: how the web programmer specifies the *selection* of portions of a page, and how the programmer expresses which *action* to take on the selected portions.

Selectors Selectors provide a way to query HTML documents. Selectors were introduced as part of the CSS1 Specification [24], where they were used to identify the DOM nodes to which styling rules would be applied. Table 1 shows a few examples of selectors.

The selector syntax is simple and includes shortcuts that make HTML queries more succinct. Additionally, developers have grown accustomed to selectors due to their ubiquity in CSS. These benefits resulted in a number of JS libraries adopting selectors as a query language,

Selector Examples

Selector	Description
`*`	All elements
`#foo`	Elements with id `foo`
`.bar`	Elements with class `bar`
`div`	All `div` elements
`div[f="b"]`	`div` elements with attribute `f` = `b`
`div > a`	`a` elements that are children of `div` elements

Table 1: A few examples of selectors. The complete selector syntax can be found in the W3C Selectors Level 3 Recommendation [4].

including Closure, Dojo, jQuery, MooTools, Prototype, and YUI [7, 13, 22, 25, 26, 39]. Initially, JS libraries were forced to re-implement much of the selector syntax due to a lack of browser support, but recent proposals (i.e, the Selectors API [36]) have led to increased support in browsers.

We chose selectors because they are expressive, succinct, familiar to developers, designed with HTML in mind, and supported by a growing number of browsers.

An alternative choice for a querying mechanism might be XPath [10] or a custom template language. XPath is more expressive than selectors but is more verbose and designed for XML, not HTML, so HTML shortcuts do not exist. Also, developers are less familiar with XPath, because it is not as widely used as selectors in front-end web development.

Another option might be to offer a custom template language. A custom template language would force developers to annotate their code with template instructions, which could conflict with existing template systems in use. A template language would also need to be implemented on both the client and server-side and would require developers to learn a new layout language. We chose not to take this approach, as selectors already meet our needs.

Actions The second part of a rule is an action. Actions define the set of operations a developer can perform on a document during composition. We modeled our API after the jQuery manipulation functions. We consider the jQuery manipulation functions a reasonable approximation of the actions needed by developers. Table 2 shows the actions our SVC implementation makes available.

SVC allows developers to define additional actions. These actions only need to be written once and could be distributed with SVC or as a set of plugins. Creating a new action consists of writing client- and server-side code that implements the action on the client and server, respectively. Since the only requirement for an action is that it can be executed on the server and client, actions could manipulate the DOM or provide unrelated functionality. For example, a `location` action could be written to redirect a page, which would set an HTTP `Location` header on the server and use `window.location` to redirect on the client.

Examples of Actions

Action	Result
`addClass(s,c)`	Add class `c`
`append(s,el)`	Append `el` inside
`attr(s,a,b)`	Set attribute `a` to `b`
`css(s,a,b)`	Set css property `a` to `b`
`html(s,el)`	Set inner HTML to `el`
`prepend(s,el)`	Prepend `el` inside
`remove(s)`	Remove from DOM
`removeClass(s,c)`	Remove class `c`
`text(s,t)`	Set inner text value to `t`

Table 2: Actions supported by our SVC prototype. Each action is passed a selector `s` as the first argument. The result of each action is performed on all nodes matching the selector. More actions could be added to an SVC implementation, with the only requirement that they be implemented in both server and client-side code.

3.1.2 Example

To illustrate how a developer would use selectors and actions to modify a page, we show a few example transformations in Table 3. Our implementation of SVC provides a class called SVCList which represents a list of transformations. Each command consists of an action, which is a method that accepts a selector as the first argument along with any additional arguments needed by the action.

We now describe how SVC responds to a request using commands 1-5 from Table 3 (for brevity, we ignore commands 6-10).

If SVC receives a non-Ajax request, it needs to construct the entire page and send it to the browser. SVC does this by taking the initial page (here, simply the string "`<a></a>`") and applying each transformation to it, in order. The result of applying the transformations can be seen in in the "Output" column of Table 3.

If SVC receives an Ajax request, it sends only the transformation list to the client. This list is encoded as JSON [23], so the actual data sent to the client is:

```
// action, selector, arguments
[["text",    ["a", "x"]],
 ["append",  ["a", "<b>y</b>"]],
 ["html",    ["b", "t <s>u</s> u"]],
 ["prepend", ["b", "<i>z</i>"]]]
```

This JSON list of transformations is applied to the current document in the browser by the SVC client-side JS.

3.1.3 Initial Content

SVC responds to an Ajax request with a list of transformations that operate on an existing document (residing in the browser). However, when a non-Ajax request is made, SVC must respond with a complete page. One option would be to always generate a complete page and discard the unnecessary portion when responding to Ajax requests. This would require generating the complete initial page for each request. Instead, we provide the method `initial`, which allows a developer to define the initial page on which to apply transformations. If the argument to this method is the name of a controller, that controller is called. If the argument is a text string, that text is used as the initial page.

```
function page() {
  $this->svc->initial('base');
  $this->svc->text('title', 'Page title');
  $this->svc->html('body', '<b>The body<b/>');
}
```

When an Ajax request is made to `page`, the controller `base` does not run and only the list of transformations (`text` and `html`) is returned to the client. When a non-Ajax request is made, SVC (on the server side) applies the transformation list to the output of the `base` controller.

The separation of the initial page from the transformation list allows SVC to only run the necessary code in response to an Ajax request.

3.1.4 Progressive Enhancement

Once the developer has specified a list of dependencies and transformations, SVC is able to respond to both Ajax and non-Ajax requests correctly. Remember, however, that the developer has written the site without Ajax in mind, so no links on the site cause an Ajax request to occur. SVC needs to progressively enhance pages to use Ajax where appropriate. The developer is in no position to do this herself, since she does not know (and should not need to know) the internals of our client-side library.

Links should only be enhanced to use Ajax if an Ajax request makes sense in the context of the current page. If no selector in the transformation list matches an element on the page, the transformation list will have no

Using Selectors and Actions to Modify HTML

	Command	Output of `$s->toHTML()`
1	`$s = new SVCList('<a></a>');`	`<a></a>`
2	`$s->text('a', 'x');`	`<a>x</a>`
3	`$s->append('a', '<b>y</b>');`	`<a>x<b>y</b></a>`
4	`$s->html('b', 't <s>u</s> u');`	`<a>x<b>t <s>u</s> u</b></a>`
5	`$s->prepend('b', '<i>z</i>');`	`<a>x<b><i>z</i>t <s>u</s> u</b></a>`
6	`$s->remove('i, s');`	`<a>x<b>t u</b></a>`
7	`$s->css('a', 'color', 'red');`	`<a style="color:red;">x<b>t u</b></a>`
8	`$s->attr('[style]', 'style', '');`	`<a style="">x<b>t u</b></a>`
9	`$s->addClass('b', 'c1');`	`<a style="">x<b class="c1">t u</b></a>`
10	`$s->remove('.c1');`	`<a style="">x</a>`

Table 3: Each command is run in order, from top to bottom. The output (on the right) shows the HTML output if the SVCList were converted to a complete HTML document at each step.

effect. However, a developer may define transformations that may happen (if the selector matches) but do not need to happen. These will effectively be ignored when the transformations are applied on the client.

Our SVC implementation provides the method `rewrite(foo,bar)` which specifies that all links pointing to controller `foo` should be rewritten to use Ajax if the current controller or any dependencies of the current controller is `bar`. Providing `bar` is necessary because Ajax requests may only be appropriate in the context of a specific page. For example, loading a new tab using Ajax would only work if the element containing the tab exists in the current page.

3.1.5 Filters

SVC also provides *filters*, which allow the modification of the output of a view. Filters exist in other frameworks (e.g., Django Middleware [12]) and are used to rewrite pages in a variety of ways (see Sec. 5.2.2 for examples of filters). Our implementation of SVC provides the class SVCFilter to manage filter registration.

SVC uses filters internally to rewrite links on a page and to automatically inject the SVC client-side JS into the page. To illustrate the use of filters, we show how SVC injects its client-side code into the page.

```
function insert_svc_js($head) {
  $svc_js = '<script src="SVC.js"></script>';
  $head->append($svc_js);
}

// converts ...<head></head>... to:
// <head><script src="SVC.js"></script></head>
$svcfilter->register('head', 'insert_svc_js');
```

SVCFilter also supports action shortcuts, allowing this to be written more succinctly:

```
$svc_js = '<script src="SVC.js"></script>';
$svcfilter->append('head', $svc_js);
```

Filters are similar to view transformations with two important differences. First, filters are always applied on the server-side and never on the client. This is necessary in certain situations, such as in the above script example. It is also necessary if the developer wants to prevent data from ever appearing on the client site. For example, a transformation list would not be the appropriate place to sanitize user input, because the user input would be sent to the client as part of the transformation list. Writing code as a filter ensures that conversion happens on the server.

The second difference between filters and view transformations is that filters can run on pieces of a document before the pieces are converted into the full document. For example, take commands 3-5 from Table 3. In an Ajax request, the server returns a list of transformations. This list contains three snippets of HTML with may match a selector (i.e., "`<b>y</b>`", "`<s>u</s> u`", and "`<i>z</i>`"). Filters are run on each snippet independently, allowing post-processing to occur on Ajax output. In a non-Ajax request, filters are run on the complete document before being sent to the browser.

To allow filters to run on pieces of a document before they are converted to a full document, we permit only a limited subset of selectors to be used in filters. Specifically, we only allow *simple selectors*, which are defined by the W3C Selectors spec [4]. Simple selectors may only refer to specific nodes and not their position in a document. For example, "`title`" and "`a[href$=jpg]`" are examples of simple selectors, while "`div > a`" is not. Simple selectors are necessary because filters run on pieces of the document in response to an Ajax request, independent of where they may later reside in the DOM. SVC is unable to determine where those pieces will exist in the complete document after they are composed on the client-side.

3.2 Client-side Components

The client-side component of SVC consists of a JS script. This script has two responsibilities. The first is to progressively enhance the page to make Ajax calls. The second is to load a transformation list from the server using Ajax and apply that transformation list to the current document in the browser. The client-side script must apply this transformation list in the same way it would be applied on the server. This ensures that the resulting document is the same, regardless of where the conversion occurred.

4 Implementation

The server-side API consists of four classes that are needed by compatible implementations.

SVCList SVCList represents a transformation list. This class provides action methods (append, addClass, etc), each of which pushes a new transformation to the end of the list. SVCList provides the method `toHTML` which applies each transformation in order and then returns the HTML representation of the final document. Also provided is `toJSON`, which serializes the transformation list as JSON.

SVCManager SVCManager provides two methods to developers. The first is `initial`, which accepts one or more controller names as arguments. The second is `rewrite(foo, bar)`, which rewrites all links to the controller `foo` when the controller `bar` has been executed.

SVCFilter SVCFilter allows the developer to register post-processing actions. These actions will run regardless of the controller (or view) called. SVCFilter provides the method `register(simple_selector, callback)`, which runs the callback function `callback` with an argument of each node matched by `simple_selector`. SVCFilter also provides action methods as a convenience, allowing simple filters to be easily created.

Snippet The Snippet class represents a parsed HTML fragment. Snippet objects are used internally by SVCList to actually perform transformations. Note that the site will only output the result of the parser used by Snippet, which means the developer is constrained to the HTML supported by the parser. Implementations of SVC should use liberal parsers. The Snippet class is typically hidden from developers by accepting strings as input to most action methods. For example, the following two function calls are equivalent:

```
$svclist->html('body', 'foo <b>bar</b>');
$svclist->html('body',
               new Snippet('foo <b>bar</b>'));
```

The Snippet class supports all action methods. These methods can accept a selector as the first argument, in which case the selector runs on the nodes in the snippet matched by the selector. Snippet action methods can also be called on the snippet itself, which runs the actions on all top-level nodes.

In addition to these classes, the implementation has the following responsibilities:

Script Injection Our SVC implementation injects its client-side JS into the current page using a filter which appends a `script` tag to the `head` element of the page.

Progressive Enhancement Progressive enhancement involves a step on the server and the client. On the server, each link that should be rewritten (checked by comparing `rewrite` calls to the current controller and dependency list) is annotated with a special `CLASS` attribute using a filter.

When the page loads, the SVC JS searches for any element having this special class and adds a `click` or `submit` event to this element. When this event occurs, the script uses Ajax to load a transformation list from the server.

Client-side Code The client-side component of SVC consists of a single JS script. This script progressively enhances the site, loads a transformation list from the server, and applies this list to the current document.

4.1 Implementation Details

We implemented a prototype of SVC as a plugin for the PHP MVC framework Code Igniter (1.7.2). We implemented the Snippet class as a PHP extension written in C++. We used the WebKit engine (used by the Safari [30] and Chrome [6] browsers) to parse HTML and WebKit's `querySelectorAll` function (defined as part of the Selectors API [36]) to perform selector matching. Specifically, we used WebCore (which is a component of WebKit), from the WebKitGTK+ 1.1.15.3 distribution [38]. WebKitGTK+ was chosen due to its compatibility with Ubuntu (no GUI code was used). SVCList, SVCFilter and SVCManager were written in PHP. We implemented Snippet actions in C++.

The client-side code consists of a 4.3/1.7KB (uncompressed/compressed with YUI Compressor [40]) JS file. This code uses jQuery (1.3.2) to handle actions and selectors. We chose jQuery because it supports the same CSS 3 Selector syntax supported by WebKit's `querySelectorAll`, and because many of our actions exist as functions in jQuery. The file size above does not include the size of jQuery, which is 19KB (compressed and Gzipped).

The client-side SVC library has been successfully tested in Internet Explorer (6,7,8), Safari (4), Chrome (3.0) and Firefox (3.5).

5 Examples

5.1 Complete Minimal Example

To make SVC more complete, we give a full example of a minimal site implemented with and without SVC in Figure 3.

This site consists of a page containing the text "A brief overview" and a link, titled "Show more", which loads more content when clicked. This link should load more content inline using Ajax when available but continue to work in non-JS browsers. Figure 3 (a)-(d) shows this site implemented without SVC. A controller contains the methods `index` and `more` which are executed when `/` and `/more` are called, respectively.

Both controllers pass an array consisting of a title, a boolean (whether to show more content) and an optional content string to the template (b). This template replaces the title with the passed value and either outputs a link pointing to additional content or the content itself. The `more` method responds to an Ajax request with a JSON version of the data array.

On the client, custom JS is required to interpret the JSON (c). Note that this JavaScript performs much the same function as the template (setting the title, inserting content, etc).

An SVC version of this site can be seen in Figure 3 (e)-(g). The template (f) consists of the initial page, which is loaded by the `index` method in (e). The method `more` defines three transformations which are either returned as JSON (in response to an Ajax request) or applied to the output of `index` method (in response to a non-Ajax request). Note that no custom JS is necessary because SVC JS will apply the transformations on the client-side automatically. Figure 3 (h) shows how SVC transforms the `index` page (the `more` page is omitted for brevity) by inserting its client-side script and added the class `svc_rewrite` to the appropriate links.

5.2 Additional Examples

5.2.1 Sites

We implemented three sites to illustrate the usefulness of SVC. We briefly describe these sites below.

Tabs We created a site consisting of a number of tabbed pages. The transformation list of each page changes the title of the document, sets the correct tab header (by setting a class), and sets the content of the page. The tab content is positioned inside of a tab-specific `div`, which allows us to rewrite links to use Ajax if that tab-specific `div` exists.

Status Feed We implemented a site to allow users of a fictional social network to update their status, which is combined on a single page. The transformation list consists of new status updates. These status updates are prepended to the current list of updates in response to an Ajax request or set as the current list of updates in response to a non-Ajax request.

Form Validation We implemented an example of form validation using SVC. Form validation is an Ajax pattern which must also be implemented in server-side code to prevent a malicious user from inserting invalid data into the application.

Each `input` element is given an associated `span` element to hold errors (e.g., Fig. 2(a)).

```
<input type="text" name="name"/>
  <span id="name_error"></span>
<input type="text" name="email"/>
  <span id="email_error"></span>
```

(a) Two inputs with associated error `spans`

```
$svclist->text('span#email_error',
               'Email error');
```

(b) Setting email error text

Figure 2: Setting an error message with SVC.

The form controller depends on a method which generates the form. The controller then sets the appropriate error message (e.g., Fig. 2(b)). If the form is submitted using Ajax, the error message will be updated in place. If it is submitted without Ajax, the error message will be properly set when the page reloads.

5.2.2 Filter Examples

We implemented three filters to illustrate the ease with which they allow developers to post-process pages. An example of filter code can be found in Appendix A.

Example not using SVC

(a) Controller

```
<?php
class Article extends Controller {

function index() {
    $data = array('title' => 'Brief Overview',
                  'showing_more' => FALSE);

    echo view('index.php', $data);
}

function more() {
    $data = array('title' => 'More Content',
        'content' => view('morecontent.php'),
        'showing_more' => TRUE);

    if (is_ajax()) {
        echo json_encode($data);
    } else {
        echo view('index.php', $data);
    }
}

}
```

(b) index.php

```
<html>
<head><title><?=$title?></title>
<script src="/js/jquery.js"></script>
<script src="/js/morecontent.js"></script>
</head><body>

A brief overview.
<? if (!$showing_more) { ?>
  <a href="/more" id="show_more">Show More</a>
  <div id="more_content"></div>
<? } else {?>
  <div id="more_content"><?=$content?></div>
<? } ?>

</body></html>
```

(c) morecontent.js

```
function more_content_clk(json) {
   $('a#show_more').remove();
   document.title = json.title;
   $('#more_content').html(json.content);
}

$(function() {

   $('#show_more').click(function() {
       $.getJSON('/more', more_content_clk);
       return false;
   });

});
```

(d) morecontent.php

```
Some <b>more</b> content
```

Example using SVC

(e) Controller

```
<?php
class Article extends Controller {

function __construct() {
    $this->svc->rewrite('more', 'index');
}

function index() {
    $this->svc->initial(view('index_svc.php'));
}

function more() {
    $this->svc->initial('index');
    $this->svc->text('title', 'More Content');
    $this->svc->remove('a#show_more');
    $this->svc->html('#more_content',
        view('morecontent.php'));
}

}
```

(f) index_svc.php

```
<html>
<head><title>Brief Overview</title></head>
<body>

A brief overview.

<a href="/more" id="show_more">Show More</a>

<div id="more_content"></div>

</body></html>
```

(g) morecontent.php

```
Some <b>more</b> content
```

SVC Output Example

(h) SVC generated index

```
<html>
<head>
<title>Brief Overview</title>
<script src="svc.js"></script>
</head>
<body>

A brief overview.

<a href="/more" id="show_more"
    class="svc_rewrite">Show More</a>

<div id="more_content"></div>

</body></html>
```

Figure 3: Two implementations of a web page containing the link "Show More". When clicked, additional content is loaded. Both Ajax and non-Ajax calls are supported. The left column ((a)-(d)) shows this page implemented without SVC. The right column ((e)-(g)) implements the page using SVC. The SVC output of the index page can be seen in (h). SVC has inserted its client-side script and added the class `svc_rewrite` to the appropriate links.

CoralCDN Filter CoralCDN [18] is a free peer-to-peer content distribution network. It provides an elegant URL structure that allows a resource to be turned into a "coralized" resource by simply appending `.nyud.net` to the hostname. Fig. 6 shows a complete SVC filter which rewrites all links and images in a page having the class "coralize". Fig. 7 shows the regular expressions used by a WordPress plugin [9] to perform the same search.

CSRF Form Filter Cross-Site Request Forgery vulnerabilities allows an adversary to perform actions on behalf of a user by maliciously submitting data to a trusted site [31]. A complete description of CSRF vulnerabilities is outside the scope of this paper, but a common defense against CSRF attacks is to insert a *CSRF token* into all `form` elements which will be submitted to the current site using the `POST` method. This token can be used by the server to verify that requests were sent by the user and not a malicious site. We wrote a filter that uses SVC to insert a token as a child of the appropriate `form` elements.

Script Coalescing Filter JavaScript can be compressed to reduce its size. A web service, called Reducisaurus [27], accepts URL arguments consisting of one or more JS files or JS code. Reducisaurus combines all scripts into a single, compressed file. We wrote a filter to convert all script tags to the appropriate URL arguments. The filter removes these scripts from the page and appends a link to the compressed JS file to the document's `head` element.

6 Performance Evaluation

Three aspects of SVC impose performance costs on the application. SVC needs to parse each snippet, run selectors, and perform actions. We evaluate these costs below and show the total performance cost of an example site. All tests were run on an desktop Dell XPS Dimension 9150 (Pentium D 2.8Ghz) machine with 2GB of RAM running Ubuntu 9.10 (Linux Kernel 2.6.31-16-generic). We used PHP 5.2.10 and Apache/2.2.12 (where applicable).

Parsing time To evaluate the parsing speed of our SVC implementation, we parsed the content of 10,000 popular websites according to Alexa [1]. The Alexa list of the most popular million sites was downloaded on 21-Nov-2009 and the first page of each site was downloaded on Nov 24, 2009. We parsed the content of the first 10,000 sites, skipping any site that did not return an HTTP 200 status. We compare our parsing speed to the speed of DOMDocument [14], which is an HTML parser built into PHP (we did not use DOMDocument for our implementation because it does not support selectors). DOMDocument does not parse JavaScript or CSS, while WebKit (used by our implementation) does. See Fig. 4 for the results of our parsing tests. The cost to parse a page is a cost not imposed on traditional PHP sites, which can output their content directly.

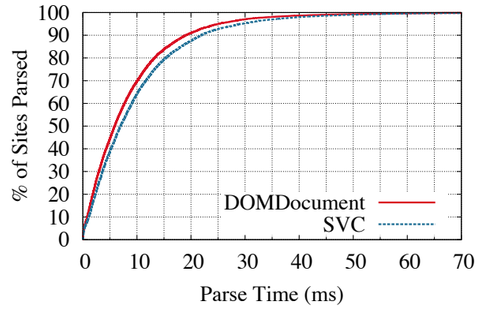


Figure 4: Parse time of 10,000 popular sites. SVC is compared to DOMDocument, an HTML parser that comes with PHP. This graph shows that, e.g., 80% of sites were parsed in 15ms.

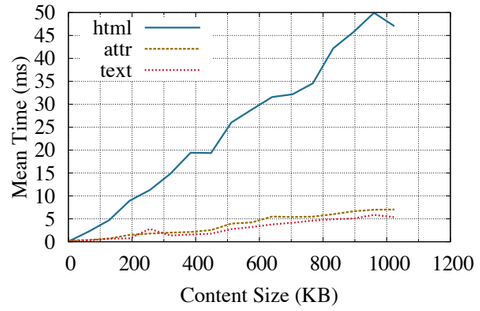


Figure 5: Mean DOM manipulation time as a function of content size. Each action (`html`, `attr`, and `text`) was called 100 times on each content size, which varied from 0 to 1 MB (incrementing by 64 KB).

Selector query time The cost of running a selector on a document depends on the complexity of the selector and complexity of the document. We test selector performance by implementing MooTool's SlickSpeed test suite [32] in SVC. The SlickSpeed test suite runs 40 selectors of varying complexity against a standard 108KB web page.

We ran the test suite 1000 times and calculated the total time spent running each test. The result was a mean time of 59.180ms ($std.dev = 2.906$), giving a mean of 1.498ms per selector.

Actions In our implementation of SVC, actions are mapped to manipulations of WebKit's internal DOM tree. The cost of DOM manipulation is dependent on the type of action performed, which largely depends on WebKit's internal performance. We measured the performance of three actions (`html`, `attr`, and `text`) in Figure 5. Each action was called with a varying amount of random data, from 0 to 1024KB. Content was escaped to ensure that additional HTML tags were not introduced.

These costs are only imposed on non-Ajax requests. For the cost of client-side DOM manipulation, see Dromaeo [28, 16].

7 Related Work

We are aware of no tool that allows for automatic progressive enhancement of non-JS sites, which SVC allows. Various frameworks allow Ajax code to be automatically created (e.g., Cappuccino, GWT, RJS (a Ruby on Rails feature) and SproutCore [3, 20, 21, 33]). Although these frameworks provide a convenient means of generating JavaScript, they do not generate code that supports both JS and non-JS browsers. In fact, these frameworks make it difficult or impossible to support non-JS browsers when using their Ajax capabilities.

Various server-side frameworks allow DOM manipulation (e.g., Genshi, DOMTemplate, GWT [19, 15, 20]). We are not aware of any framework that allows manipulation to happen on either the client or server depending on the type of request, or any framework that uses DOM manipulation as a means to allow for automatic Ajax instrumentation.

FlyingTemplates [35] proposes a system where templates are sent to the client along with data to replace variables in the template. The key idea of FlyingTemplates is to allow templates to be static and to send the data that will be substituted in the template separately. If templates are static, they can be cached by the browser and served statically by the website. SVC differs significantly from FlyingTemplates. FlyingTemplates requires a template parser to exist on the client. Additionally, it has no notion of conditional template replacement (all replacement happens on the client). FlyingTemplates also only runs on an initial template. Once it has performed replacement on a template, it cannot operate on that replaced data. This makes it unable to assist in automatic Ajax instrumentation. Also, FlyingTemplate only works in JS browsers because it relies on the client to do template substitution.

Post-processing filters exist in a number of frameworks. The main advantages of SVC over filters in other frameworks is the ability to register a filter using selectors and the ability to run filters on pieces of the document. The most similar filtering mechanism found is Genshi's [19], which allows XPath expressions to be used in filters.

8 Conclusions

The paper presents SVC as a novel programming style that can reduce development costs related to web programming. SVC allows developers to create both Ajax and non-Ajax versions of a site by composing different pieces of a page together using selectors. Developers can create Ajax sites without writing JavaScript, while also supporting non-JS browsers. SVC can be integrated with existing sites and does not interfere with previously written controllers, JavaScript, or template systems.

SVC also provides a succinct filtering mechanism that allows post-processing to be expressed more clearly than existing solutions.

We implemented a prototype of SVC for PHP and the Code Igniter framework, but it could easily be extended to other languages and frameworks. The only requirement is an HTML parser and selector support. In Python, for example, the lxml library could be used directly, without the need to compile a separate HTML parser (also, lxml supports selectors directly) [17].

Since SVC manages Ajax calls, client-side JS plugins could be written which implement common Ajax patterns. For example, supporting the "back" button (which can be tricky in Ajax applications) could be handled automatically.

Acknowledgements

We would like to thank Joe Calandrino, Will Clarkson, Thorsten von Eicken, Ari Feldman, J. Alex Halderman, Jon Howell, Tim Lee, and the anonymous referees for their helpful comments and suggestions.

References

[1] Alexa Top 1,000,000 Sites (Updated Daily). http://www.alexa.com/topsites.

[2] Byrne, S., Hors, A. L., Hégaret, P. L., Champion, M., Nicol, G., Robie, J., and Wood, L. Document object model (DOM) level 3 core specification. W3C recommendation, W3C, Apr. 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407.

[3] Cappuccino. http://cappuccino.org.

[4] Çelik, T., Etemad, E. J., Glazman, D., Hickson, I., Linss, P., and Williams, J. Selectors Level 3. W3C proposed recommendation, W3C, Dec. 2009. 3http://www.w3.org/TR/2009/PR-css3-selectors-20091215/3.

[5] CHAMPEON, S., AND FINCK, N. Inclusive Web Design For the Future. http://www.hesketh.com/publications/inclusive_web_design_for_the_future/, Mar. 2003.

[6] Chrome. http://google.com/chrome.

[7] Closure. http://code.google.com/closure/library/.

[8] Code Igniter. http://codeigniter.com.

[9] Coralize for wordpress (v.08b). http://theblogthatnoonereads.davegrijalva.com/2006/02/12/coralize/.

[10] DEROSE, S., AND CLARK, J. XML path language (XPath) version 1.0. W3C recommendation, W3C, Nov. 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[11] Django. http://www.djangoproject.com.

[12] Django Middleware. http://docs.djangoproject.com/en/1.1/topics/http/middleware/.

[13] Dojo. http://dojotoolkit.org.

[14] DOMDocument. http://php.net/manual/en/class.domdocument.php.

[15] DOMTemplate. http://www.domtemplate.com.

[16] Dromaeo. http://dromaeo.com.

[17] FAASSEN, M. lxml. http://codespeak.net/lxml/.

[18] FREEDMAN, M. J., FREUDENTHAL, E., AND MAZIÈRES, D. Democratizing content publication with coral. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2004), USENIX Association, pp. 18–18.

[19] Genshi. http://genshi.edgewall.org.

[20] Google Web Toolkit. http://code.google.com/webtoolkit/.

[21] HANSSON, D. H. RJS. http://wiki.rubyonrails.org/howtos/rjs-templates.

[22] jQuery. http://jquery.com.

[23] JSON (JavaScript Object Notation). http://www.json.org.

[24] LIE, H. W., AND BOS, B. Cascading style sheets, level 1 recommendation. first edition of a recommendation, W3C, Dec. 1996. http://www.w3.org/TR/REC-CSS1-961217.

[25] MooTools. http://mootools.net.

[26] Prototype. http://prototypejs.org.

[27] Reducisaurus. http://code.google.com/p/reducisaurus/.

[28] RESIG, J. JavaScript Performance Rundown. http://ejohn.org/blog/javascript-performance-rundown/.

[29] Ruby on Rails. http://rubyonrails.org.

[30] Safari. http://apple.com/safari/.

[31] SHIFLETT, C. Security Corner: Cross-Site Request Forgeries. http://shiflett.org/articles/cross-site-request-forgeries.

[32] SlickSpeed (MooTools). http://mootools.net/slickspeed/.

[33] SproutCore. http://sproutcore.com.

[34] Struts. http://struts.apache.org.

[35] TATSUBORI, M., AND SUZUMURA, T. Html templates that fly: a template engine approach to automated offloading from server to client. In *WWW '09: Proceedings of the 18th international conference on World wide web* (New York, NY, USA, 2009), ACM, pp. 951–960.

[36] VAN KESTEREN, A., AND HUNT, L. Selectors api level 1. Tech. rep., W3C, Dec. 2009. http://www.w3.org/TR/2009/CR-selectors-api-20091222/.

[37] The WebKit Open Source Project. http://webkit.org.

[38] WebKitGTK+. http://webkitgtk.org.

[39] The YUI Library. http://developer.yahoo.com/yui.

[40] YUI Compressor. http://developer.yahoo.com/yui/compressor/.

A Filter Examples

Below is an example filter, described in Section 5.2.2.

```
<?php
// point URL to CoralCDN
function coralize($url) {
  $host = parse_url($url, PHP_URL_HOST);
  $s = '://'.$host;
  // append '.nyud.net:8080' to host
  return str_replace($s, $s.'.nyud.net:8080',
                     $url);
}

function c_img($s) {
  $url = coralize($s->attr('src'));
  $s->attr('src', $url);
}

function c_a($s) {
  $url = coralize($s->attr('href'));
  $s->attr('href', $url);
}

// Coralize all elements with class 'coralize'
$svcfilter->register('img.coralize', 'c_img');
$svcfilter->register('a.coralize', 'c_a');
```

Figure 6: A filter which rewrites all links and images (having the class "coralize") to use the CoralCDN web service.

```
$content = preg_replace('/(\<(img|a)\s+.*?cla
  ss\=["\'].*?coralize.*?["\'].*?(src|href)\=
  ["\']http\:\/\/.*?)(\/.*?["\'].*?\>)/i', '$
  1.nyud.net:8080$4', $content);
$content = preg_replace('/(\<(img|a)\s+.*?(sr
  c|href)\=["\']http\:\/\/.*?)(\/.*?["\'].*?c
  lass\=["\'].*?coralize.*?["\'].*?\>)/i', '$
  1.nyud.net:8080$4', $content);
$content = preg_replace('/(\<(img|a)\s+.*?cla
  ss\=["\'].*?coralize.*?["\'].*?(src|href)\=
  ["\'])(\/.*?["\'].*?\>)/i', '$1http://'.$_S
  ERVER['HTTP_HOST'].'.nyud.net:8080$4',
  $content);
$content = preg_replace('/(\<(img|a)\s+.*?(sr
  c|href)\=["\'])(\/.*?["\'].*?class\=["\'].*
  ?coralize.*?["\'].*?\>)/i', '$1http://'.$_S
  ERVER['HTTP_HOST'].'.nyud.net:8080$4',
  $content);
```

Figure 7: Taken verbatim from the Coralize for Wordpress plugin [9]. In addition to being difficult to read, these regular expressions would incorrectly match an `img` element with the class "donotcoralize" and would not match elements with irregular spacing.

Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads

James Mickens
Microsoft Research
mickens@microsoft.com

Abstract

A modern web page contains many objects, and fetching these objects requires many network round trips—establishing each HTTP connection requires a TCP handshake, and each HTTP request/response pair requires at least one round trip. To decrease a page's load time, designers try to minimize the number of HTTP requests needed to fetch the constituent objects. A common strategy is to inline the page's JavaScript and CSS files instead of using external links (and thus separate HTTP fetches). Unfortunately, browsers only cache externally named objects, so inlining trades fewer HTTP requests now for greater bandwidth consumption later if a user revisits a page and must refetch uncacheable files.

Our new system, called Silo, leverages JavaScript and DOM storage to reduce both the number of HTTP requests *and* the bandwidth required to construct a page. DOM storage allows a web page to maintain a key-value database on a client machine. A Silo-enabled page uses this local storage as an LBFS-style chunkstore. When a browser requests a Silo-enabled page, the server returns a small JavaScript shim which sends the ids of locally available chunks to the server. The server responds with a list of the chunks in the *inlined* page, and the raw data for chunks missing on the client. Like standard inlining, Silo reduces the number of HTTP requests; however, it facilitates finer-grained caching, since each chunk corresponds to a small piece of JavaScript or CSS. The client-side portion of Silo is written in standard JavaScript, so it runs on unmodified browsers and does not require users to install special plugins.

1 Introduction

Users avoid slow web sites and flock towards fast ones. A recent study found that users expect a page to load in two seconds or less, and 40% of users will wait for no more than three seconds before leaving a site [1]. Thus, fast-loading pages result in happier users, longer visit times, and higher revenues for page owners. For example, when the e-commerce site Shopzilla reduced its average load time from 5 seconds to 1.5 seconds, it boosted page views by 25% and revenue by 10% [4]. Faster loads also lead to more advertising impact, since Google's AdWords system preferentially displays ads whose target pages load quickly [9]. Search engines also make loading speed a factor in their page rankings [22].

Given all of this, web designers have accumulated a series of techniques for decreasing load times. Souder's influential book *High Performance Web Sites* lists 14 of these techniques [23], with the most important one being to minimize the number of HTTP requests needed to construct a page. Over 80% of user-perceived load time is spent downloading HTML, JavaScript, images, and other objects, and 40–60% of page visitors arrive with an empty cache [25]. By minimizing the number of HTTP requests needed to build a page, developers reduce the number of round-trips needed to fetch the page's objects, and they avoid TCP slow starts for now-superfluous HTTP connections.

1.1 The Costs and Benefits of Inlining

An obvious way to eliminate HTTP requests is to make pages contain fewer objects. Unfortunately, this approach results in less interesting pages, so developers instead use *object inlining* [23]. Multiple JavaScript files are concatenated to form a smaller number of JavaScript files; similarly, multiple style sheets are combined into a single CSS file. In both cases, the number of HTTP requests decreases.

The greatest savings occur when *all* of the JavaScript and CSS is directly inserted into the page's HTML. Such aggressive inlining delivers all of the page's HTML, CSS, and JavaScript in a single HTTP fetch. Unfortunately, the significant reduction in load time comes with a price—since the browser cache can only store URL-addressable objects, the individual HTML, CSS, and

JavaScript files cannot be cached. Instead, the browser caches the aggregate inlined HTML, and if any of the embedded objects change, the browser must refetch the bytes for *all* of the objects. Ideally, we would like the best of both worlds: aggressive inlining which maintains the cacheability of the constituent objects.

1.2 Our Solution: Silo

Silo is our new framework for deploying fast-loading web applications. Silo exploits JavaScript to implement a delta-encoding HTTP protocol between an unmodified web browser and a Silo-aware web server. A Silo web server aggressively inlines JavaScript and CSS, and breaks the inlined HTML into chunks using Rabin fingerprints [18]. When a browser requests the page, the server does not return the inlined HTML—instead, it returns a small JavaScript shim which uses the browser's DOM storage [26] as a chunk cache. The shim informs the server of locally available chunks. The server responds with a list of chunk ids in the page, as well as the raw data for any chunks that do not reside on the client.

By aggressively inlining, browsers can fetch the HTML, CSS, and JavaScript for a Silo-enabled page in at most two HTTP round trips (§3.1). However, using chunking, Silo restores the cacheability that was previously destroyed by aggressive inlining. Indeed, Silo introduces a *finer* granularity of caching, since data is cached at the level of 2 KB chunks instead of entire files. This reduces bandwidth requirements when updating already cached HTML, JavaScript, and CSS files that have changed, but that retain some of their old content. Since client-side chunk data is associated with an entire domain, chunks downloaded from one page in a domain can be used to reconstruct a sibling. Thus, Silo can exploit the fact that different pages in the same domain often share content [5, 21].

1.3 Our Contributions

This paper makes the following contributions:

- We show how unmodified browsers can exploit JavaScript and DOM storage to implement a delta-encoding protocol atop standard HTTP.
- We provide new empirical data on the composition of web pages and how their content changes over time.
- We demonstrate that for pages with significant amounts of JavaScript and CSS, Silo's new protocol can reduce load times by 20%–80% while providing finer-grained caching than the standard browser cache.

We also discuss the fundamental challenge of defining "load time" in the context of modern web pages which contain rich interactive content and thousands of lines of JavaScript, Flash, and other code.

The rest of this paper is organized as follows. In Section 2, we provide background information on the HTTP protocol and describe the basic JavaScript features that Silo leverages. In Section 3, we describe how Silo uses these features to layer a custom delta-encoding protocol atop standard HTTP. Section 4 provides our PlanetLab evaluation, wherein we serve real web data under realistic network conditions to explore Silo's benefits. We discuss related work in Section 5 before concluding in Section 6.

2 Background

In this section, we provide a brief overview of the HTTP protocol, describing the particular elements that are relevant to the Silo architecture. We also describe the JavaScript features that we use to implement the client-side portion of the Silo protocol. Finally, we explain how Rabin fingerprints can be used for delta-encoding.

2.1 The HTTP Protocol

A browser uses the HTTP protocol [6] to fetch objects from a web server. A top-level page like `www.cnn.com` is composed of multiple objects. Silo separates these objects into four classes.

- HTML describes a page's content.
- Cascading style sheets (CSS) define how that content is presented.
- JavaScript code allows the page to respond to user inputs and dynamically update itself.
- Multimedia files like images, movies, and sound files provide visual and audio data.

The standard browser cache can store each class of object. However, the first three object types consist of structured, text-based data that changes relatively slowly across object revisions. In contrast, multimedia files have binary data that rarely changes in-place. Thus, only HTML, CSS, and JavaScript are amenable to delta-encoding (§2.3), a technique for describing different versions of an object with respect to a reference version. Importantly, the standard browser cache stores *whole objects* at the *URL-level*—HTTP provides no way to delta-encode arbitrary objects with respect to previously cached versions.

To increase fetch parallelism, a browser tries to open multiple connections to a single web server. HTTP is a TCP-based protocol, and TCP setup and teardown are expensive in terms of RTTs. Thus, HTTP version 1.1 introduced persistent HTTP connections, which allow a single TCP session to be used for multiple HTTP requests and responses. On highly loaded web servers or proxies, maintaining too many persistent connections can exhaust memory, file descriptors, and other computational resources, hurting parallelism if many persistent connections are idle but no more can be created to handle new, active clients. Mindful of this threat, some proxies and web servers use stringent timeouts for persistent connections, or close them after a few objects have been transferred [2, 11].

Figure 1(a) demonstrates how a browser might download a simple web page consisting of a single HTML file and four external objects. First, the browser opens a persistent HTTP connection to the web server and fetches the page's HTML. As the browser parses the HTML, it finds references to the page's external objects. It fetches `a.css` and `c.css` using its preexisting HTTP connection; in parallel, the browser opens a second persistent connection to get `b.css` and `d.js`. The browser constructs the entire page in approximately three HTTP round trips (one RTT to fetch the HTML, and two RTTs to fetch the four objects over two persistent HTTP connections) [1].

In popular browsers like Firefox 3.0, IE 7, and Safari 3, a JavaScript fetch prevents the initiation of new parallel downloads. This is because the fetched JavaScript may change the content of the subsequent HTML (and thus the external objects that need to be fetched). Newer browsers use speculative parsing techniques to contain the side effects of erroneously fetched objects. Developers have also invented various application-level hacks to trick browsers into doing parallel JavaScript fetches [24]. Regardless, a browser can only open a finite number of parallel connections, so the fetching of non-inlined JavaScript generally adds to the load time of a page.

2.2 JavaScript

JavaScript [7] is the most popular language for client-side scripting in web browsers. With respect to Silo, JavaScript has three salient features. First, JavaScript programs can dynamically modify the content of a web page. Second, JavaScript can associate large amounts of persistent local data with each web domain. Third, JavaScript can use AJAX calls [7] to construct new communication protocols atop HTTP.

2.2.1 Manipulating the DOM

JavaScript represents the state of a web page using the Document Object Model (DOM) [28]. The DOM provides a standard, browser-neutral API for querying and manipulating the presentation and content of a page. In the context of Silo, the most important DOM calls are the ones which allow pages to overwrite their own content.

- When a JavaScript program calls `document.open()`, the browser clears any preexisting presentation-layer data associated with the page, i.e., the JavaScript state is preserved, but the page's old HTML is discarded.
- The application writes new HTML to the page using one or more calls to `document.write(html_str)`.
- Once the application has written all of the new HTML, it calls `document.close()`. This method instructs the browser to finish parsing the HTML stream and update the presentation layer.

Using these calls, a web page can completely overwrite its content. Silo leverages this ability to dynamically construct pages from locally cached data chunks and new chunks sent by Silo web servers.

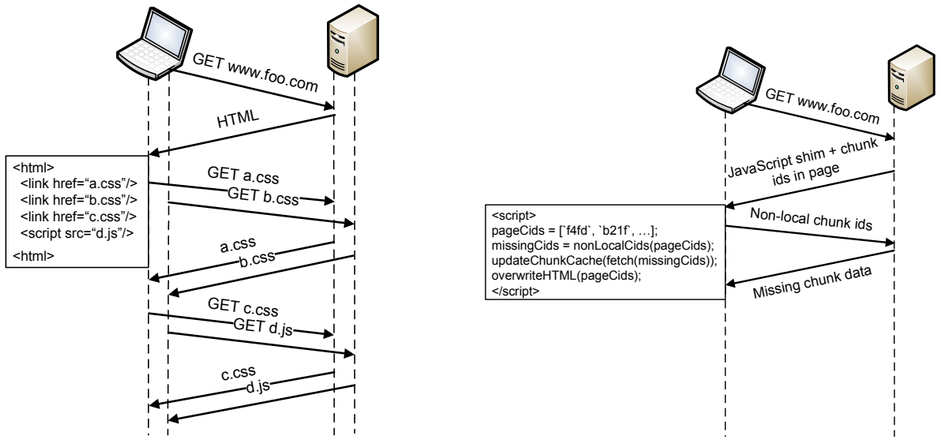
2.2.2 Associating Web Domains With Local Data

JavaScript does not provide an explicit interface to the browser cache. JavaScript-generated requests for standard web objects like images may or may not cause the associated data to lodge in the cache, and JavaScript programs cannot explicitly write data to the cache. Furthermore, there is no way for a JavaScript program to list the contents of the cache.

For many years, the only way for JavaScript to store persistent, programmatically-accessible client-side data was through cookies [10]. A cookie is a small file associated with a particular web domain. When the user visits a page belonging to that domain, the browser sends the cookie in the HTTP request. The domain can then read the cookie and send a modified version in the HTTP response. JavaScript provides a full read/write interface for cookies. However, browsers restrict the size of each cookie to a few kilobytes, making cookies unsuitable for use as a full-fledged data store.

To solve this problem, Google introduced Gears [8], a browser plugin that (among other things) provides a SQL interface to a local data store. Although Gears provides a powerful storage API, it requires users to modify their browsers. Luckily, modern browsers like IE8 and Firefox 3.5 support a new abstraction called DOM storage [26]. DOM storage allows a web domain to store client-side key/value pairs. By default, browsers allocate 5–10 MB of DOM storage to each domain. The DOM storage API has been accepted by the W3C Web Apps Working group [26] and will likely appear in the upcoming HTML5 standard. Given this fact, Google recently announced that it was ramping down active development on Gears, and that it expected developers to

[1] HTTP 1.1 allows clients to pipeline multiple requests over a single connection, but many web servers and proxies do not support this feature or support it buggily. Pipelining is disabled by default in major browsers.



(a) The standard HTTP protocol.

(b) The basic Silo protocol. See Section 3.2 for optimizations.

Figure 1: Fetching a simple web page.

migrate to the standardized HTML5 storage mechanisms once HTML5 gained traction [14].

Web applications most commonly use DOM storage to buffer updates during disconnected operation. Silo uses DOM storage in a much different fashion, namely, as an application-level chunk cache that enables delta-encoding (§2.3) for web pages.

2.2.3 AJAX

A JavaScript program can use the AJAX interface [7] to explicitly fetch new web data. AJAX data is named by URL, and a JavaScript application can inspect and set the headers and the body of the HTTP request and response. Thus, an application can use AJAX to layer arbitrary client-server protocols atop HTTP. Silo uses AJAX to implement a custom delta-encoding HTTP protocol inside unmodified web browsers.

2.3 Delta-encoding

Delta-encoding is a technique for efficiently describing the evolution of a data object. Each version of the object is represented as a set of edits or "deltas" applied to a reference version of the object. Once a client downloads the full reference object, it can cheaply reconstruct a newer version by downloading the deltas instead of the entire new object.

Many distributed systems employ *chunk-based* delta-encoding. Each object is broken into small, contiguous byte ranges; a single edit modifies one or more of these chunks. Hosts transmit their edits by sending the positions of deleted chunks, and the positions and data of new chunks.

If chunk boundaries are determined by fixed byte offsets, an edit which increases or decreases an object's size will invalidate all chunks after the edit point. To avoid this problem, distributed systems typically eschew fixed length chunks and use *content-delimited* chunks instead. In these systems, chunk boundaries are induced by special byte sequences in the data. LBFS [16] popularized a chunking method in which applications push a sliding window across a data object and declare a chunk boundary if the Rabin hash value [18] of the window has N zeroes in the lower-order bits. By varying N, applications control the expected chunk size. With content-based hashing, an edit may create or delete several chunks, but it will not cause an unbounded number of chunk invalidations throughout the object.

Distributed systems typically name each chunk by the SHA1 hash of its content. This allows different hosts to independently pick the same name for the same chunk. Hosts can determine whether they store the same version of an object by exchanging the chunk ids in their copies of the file. By comparing these lists, a peer can determine whether it needs to delete content from its local version or fetch new chunks from the other host. Silo uses a similar protocol to delta-encode the transmission of previously viewed web pages.

3 Silo Architecture

Ideally, the Silo protocol would be implemented as an extension to HTTP, and commodity web servers and browsers would ship with native Silo support. However, to ease deployability, our current Silo architecture leverages JavaScript to execute on *unmodified* browsers. Web servers must still be modified, but this is much less onerous than modifying millions of end-user browsers.

Our Silo architecture consists of three components: a Silo-aware web server, an unmodified client browser, and a JavaScript shim that is generated by the server and which implements the client side of the Silo protocol. In this section, we describe this architecture in more detail. We also describe several optimizations to the basic Silo protocol; some of these optimizations mask performance issues in current JavaScript engines, and others leverage cookies to reduce the number of RTTs needed to construct a page.

Silo's goal is to reduce the time needed to assemble a page's HTML, CSS, and JavaScript. Borrowing Firefox's event terminology, we refer to this time as the page's `DOMContentLoaded` time [15]. Fetching a page's HTML, CSS, and JavaScript is necessary but often insufficient to produce a fully functioning page. For example, pages often contain multimedia files which are not amenable to Silo-style delta encoding. Silo is orthogonal to techniques for improving the load times of these objects. We return to this topic when we describe our evaluation methodology (§4.1).

3.1 The Basic Protocol

Figure 1(a) depicts how a web page is constructed using the standard HTTP 1.1 protocol. The browser first retrieves the HTML for the page. As it parses the file, it issues parallel fetches for the externally referenced objects. In Figure 1(a), we assume that the client cache is empty, and that the browser can issue two fetches in parallel. Thus, the browser must use three HTTP round trips to construct the page (one to send the initial `GET`, and two to download the external objects).

Figure 1(b) depicts how a Silo-enabled page is fetched. The browser issues a standard `GET` for the page, but the web server does not respond with the page's HTML. Instead, the server sends a small piece of JavaScript which acts as the client-side participant in the Silo protocol. The JavaScript shim contains an array called `pageCids`; this array lists the ids of the chunks in the page to construct. The shim inspects the client's DOM storage to determine which of these chunks do not reside locally. The shim uses a synchronous AJAX POST to send the missing chunk ids to the server. The server replies with the raw data for the missing chunks. The client assembles the relevant chunks and overwrites the page's current HTML, reconstructing the original inlined page. In this fashion, the basic Silo protocol uses two HTTP round trips to fetch an arbitrary number of HTML, CSS, and JavaScript files.

```
<html>
   <script>
      /*Code for Silo shim*/
   </script>
   <style type=text/css>
      /*Inlined css for a.css*/
   </style>
   ...
   <script>
      /*Inlined JavaScript for d.js*/
   </script>
</html>
<!--- Chunk manifest
cid0, offset0, len0
cid1, offset1, len1
...
--->
```

Figure 2: When the client chunk cache is empty, the Silo server responds with inlined HTML which is immediately usable by the browser. The client shim asynchronously parses the chunk manifest at the bottom of the HTML and updates the local chunk cache.

3.2 Optimizations

Handling Cold Client Caches: At any given moment, 40%–60% of the users who visit a page will have no cached data for that page [25]. However, as shown in Figure 1(b), a Silo server does not differentiate between clients with warm caches and clients with cold caches—in either case, the server's second message to the client is a string containing raw data for N chunks. If the client has an empty cache, it must *synchronously* perform N substring operations before it can extract the N chunks and recreate the inline page. In current browsers, this parsing overhead may be hundreds of milliseconds if inlined pages contain hundreds of KB of data (and thus hundreds of chunks).

To improve load times in these situations, Silo sends a different second message to clients with empty chunk caches. Instead of sending raw chunk data that the client must parse before it can reconstruct the page, the server sends an inlined version of the page annotated with a special *chunk manifest* at the end (see Figure 2). The chunk manifest resides within an HTML comment, so the client can commit the annotated HTML immediately, i.e., without synchronously performing substring operations. Later, the client asynchronously parses the manifest, which describes the chunks in the inlined HTML using a straightforward offset+length notation. As the client parses the manifest, it extracts the relevant chunks and updates the local chunk cache.

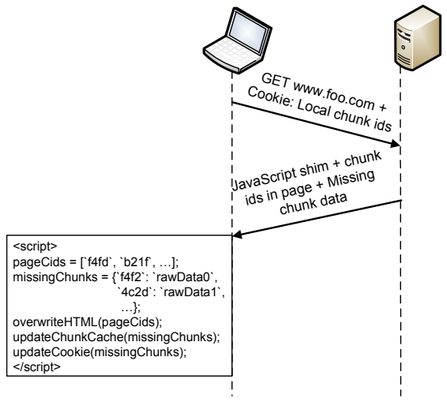


Figure 3: Single RTT Silo protocol, warm client cache.

Leveraging HTTP Cookies: Silo uses cookies (§2.2.2) in two ways to reduce the number of HTTP round trips needed to construct a page. First, suppose that a client has an empty chunk cache. Even if the server uses annotated HTML to eliminate synchronous client-side string operations, the client must expend two HTTP round trips to construct the page. However, the client shim can set a "warm cache" variable in the page's cookie whenever it stores chunks for that page. If the server does not see this cookie variable when the client sends the initial HTTP `GET` operation, the server knows that the client chunk cache is cold, either because the client has never visited the page before (and thus there is no cookie), or the client has visited the page before, but the local cache is empty for some other reason (e.g., because previous writes to DOM storage failed due to a lack of free space). Regardless, the server responds to the initial HTTP `GET` with annotated HTML. This allows a Silo client with a cold cache to fetch all of the HTML, CSS, and JavaScript in a single HTTP round trip.

Clients with warm caches can also use cookies to indicate the contents of their cache. Whenever the client shim updates the chunk cache for a page, it can add the new chunk ids to the page's cookie. The next time that the browser requests the page, the server can inspect the cookie in the initial HTTP `GET` and determine which page chunks already reside on the client. The server can then directly respond with the Silo shim and the missing chunks, eliminating the second HTTP round trip required by the basic Silo protocol.

Browsers typically restrict cookie sizes to 4 KB, the minimum cap allowed by RFC 2109 [10]. SHA1 hashes are 20 bytes, so a cookie could hold a maximum of 204 SHA1 chunk ids. If chunks are 2 KB on average, then a cookie could reference a maximum of 408 KB of local chunk cache data. For many popular web pages, 408 KB is sufficient to delta-encode several versions of the page (§4.3). However, some pages are so big that even a single snapshot of their HTML, CSS, and JavaScript will not fit in 408 KB of chunk-addressable storage. Fortunately, Silo can expand the addressable range by leveraging the fact that Silo servers have complete control over chunk names. Name-by-hash allows different machines to agree on chunk names without a priori communication, but Silo clients never assign chunk ids—servers always determine the mapping between chunk ids and chunk data. Thus, servers can use ids that are much shorter than SHA1 hashes. For example, a server could assign each new, unique chunk a strictly increasing 3 byte id. Since client-side DOM storage is partitioned by domain, these ids only have to be collision-free within a domain, not the entire Internet. With 3 byte ids, a domain could define over 16 million unique chunks before having to "reset" the namespace. A 4 KB cookie could store 1365 of these 3 byte ids. With 2 KB chunks, this would allow clients to name roughly 2.7 MB of local cache data. As we show in Section 4.3, 2.7 MB of storage should be more than sufficient to delta-encode multiple versions of a page's HTML, CSS, and JavaScript.

Finally, we note that Silo is agnostic to the specific method in which data is chunked. Indeed, websites are free to pick the granularity of caching that best suits their needs. For example, if a web site knows that its HTML evolves in a structured way, it can define an HTML-specific chunking function for its pages. Alternatively, a site could decide to chunk at the whole file level, i.e., with each CSS and JavaScript file residing in its own singleton chunk. Silo is not bound to a specific chunking mechanism—it merely leverages chunking to provide inlining without destroying the cacheability of individual objects.

The Full Protocol: Given all of these optimizations, we now describe the full version of the Silo protocol. We separate our description into two cases: when the client has a cold chunk cache, and when the client has a warm cache.

- *Cold cache:* The client generates an HTTP `GET` request for a page, sending a cookie which indicates a cold cache. The server responds with annotated HTML as shown in Figure 2. The browser commits the annotated HTML immediately; asynchronously, the Silo shim parses the chunk manifest, extracts the associated chunks, and writes them to DOM storage. In this scenario, Silo needs one HTTP round trip to assemble all of the page's HTML, CSS, and JavaScript.
- *Warm client cache:* The client generates an HTTP `GET` request for a page, sending a cookie which indicates a warm cache. If the client can fit all of the local chunk ids within the cookie, it can receive the Silo shim and the missing chunk data in a single server response (see Figure 3). Otherwise, it falls back on the basic, two RTT Silo protocol depicted in Figure 1(b).

As we show in the evaluation section, Silo's primary benefit is to reduce load times for clients with cold caches; client with completely warm caches have no fetch latencies to mask. However, for clients with only partially warm caches, Silo reduces the fetch penalty since an arbitrary number of stale objects can be updated using at most two HTTP round trips. Furthermore, if pages use fine-grained chunking, data can be invalidated at a much finer level, reducing the fetch bandwidth in addition to the fetch latency.

3.3 Other Design Decisions

When a client has a partially warm cache, Silo asynchronously writes new chunks to disk. As we show in Section 4.2, Firefox's writes to DOM storage can require more than a hundred milliseconds. Thus, to avoid foreground resource contention during the page load, Silo synchronously extracts in-memory versions of the new chunks needed to assemble the page, but it always defers writes to stable storage for a few seconds (our current prototype waits 5 seconds).

The regular browser cache stores data belonging to `<script>` tags. These tags can store arbitrary data as JavaScript strings. Thus, Silo could use the standard browser cache as a chunk store by writing blobs to scripts whose names were chunk ids. At first glance, this approach seems attractive since it obviates the need for a separate chunk cache, and it would work on unmodified browsers. However, browsers provide no way for JavaScript applications to explicitly insert data into the cache. Instead, applications implicitly warm the browser cache as a side-effect of fetching external data. Even simple web pages will likely contain at least a few tens of chunks; thus, a client which wanted to store these chunks using `<script>` tags would have to issue a large number of HTTP requests. This would obviously lead to huge increases in page load times. Thus, `<script>` tags are a poor substitute for DOM storage.

4 Evaluation

In this section, we evaluate Silo by serving real web content from a Silo deployment on PlanetLab. We demonstrate that Silo can substantially improve load times for pages with large amount of CSS and JavaScript. We also provide an analysis of how content chunks evolve within the same page and across different pages.

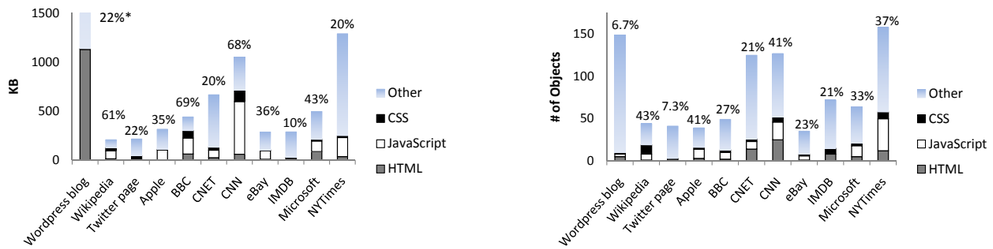
4.1 Methodology

Gathering and Serving Real Web Data: For our Silo-aware web server, we used a modified version of the Mugshot replay proxy [13]. Mugshot is a system for capturing and replaying the behavior of JavaScript-based web applications. A browser-side Mugshot JavaScript library records the bulk of the nondeterminism like GUI activity and calls to the random number generator. A special Mugshot proxy sits between the real web server and the browser; this proxy records the binding between the URLs in client HTTP requests and the data that is returned for those URLs. Later, at replay time, Mugshot uses the replay proxy as a web server, letting it respond to data requests from the replaying client code. This ensures that the data fetched at replay time is the same data that was fetched at logging time.

To test the Silo protocol, we first ran the Mugshot proxy in its standard logging mode, capturing HTTP headers and data from real websites. We then switched the proxy into replay mode and had clients use it as their web proxy. When clients requested a page whose content we previously logged, the proxy served that page's objects directly from its cache. We modified the replay mode to support the Silo chunking protocol described in Section 3.2. Thus, we could simulate Silo's deployment to an arbitrary web site by logging a client's visit to that site, clearing the client-side cache, and then revisiting the page in Silo-enabled replay mode.

Experimental Setup: Our experimental setup consisted of a client machine whose browser fetched content from a Silo-aware web server. The client browser resided on a Lenovo ThinkPad laptop running Windows 7. The laptop had an Intel Core 2 Duo with 2.66 GHz processors and 4GB of RAM. The server ran on a PlanetLab node with a 2.33 GHz Intel Core Duo and 4 GB of RAM. The client communicated with the server over a residential wireless network. Across all experiments, the RTT was stable at roughly 150 ms, and the bandwidth varied between 700–850 Kbps. Thus, network conditions were similar to those experienced by typical cable modems or DSL links. In the results presented below, we used the Firefox 3.5.7 browser, but the results for IE8 were similar. We present Firefox results because IE8 does not yet define the `DOMContentLoaded` event, whose usage we describe shortly. Silo using Rabin chunking with an expected chunk size of 2 KB.

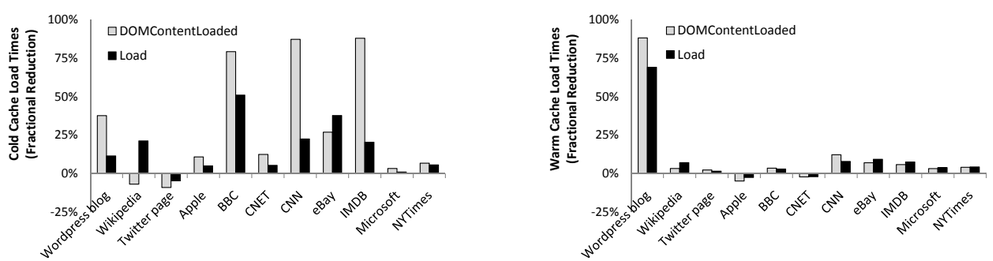
In real life, when a browser loads a page, it opens multiple simultaneous connections to multiple servers and proxies. In our experimental setup, the client browser fetched everything from a single Mugshot proxy. To ensure that we did not unduly constrain fetch parallelism, we configured Firefox to open up to sixteen persistent connections to a proxy instead of the default of eight.



(a) Content type by byte size. ∗Wordpress blog contains 5.1 MB.

(b) Content type by fraction of objects.

Figure 4: Content statistics for several popular web pages. Top-level fractions are the percentage of content that is potentially inlineable (HTML+CSS+JavaScript).



(a) Cold cache.

(b) Warm cache.

Figure 5: Page load times.

In an actual Silo deployment, a web server would always compress the chunk manifests that it sent to clients. However, we found that many web servers do not compress the HTML and the CSS that they transmit [2]. To provide a fairer comparison, the Silo server only `gzip`ed its chunk manifest if the logged HTML page was also compressed. Our Silo server also closed persistent connections whenever the logged persistent connection was closed.

Defining Load Time: One of Silo's benefits is that it reduces page load time. The intuition is straightforward—by decreasing the number of round trips needed to construct a page, the browser spends less time stalling on the network and builds pages faster. Unfortunately, providing a precise definition of load time is difficult.

At the end of the exchanges shown in Figures 1(a) and (b), the page's HTML, JavaScript, and CSS have been fetched and parsed. The browser has calculated the layout and presentation format of the content, but it may not have fetched some of this content. In particular, multimedia objects like images and advertisements may or may not have been downloaded in parallel with the DOM content. Thus, the "full" page load may not coincide with the completion of the DOM load.

To further complicate matters, some sophisticated web pages use JavaScript code to defer certain fetches. For example, upon initial load, some sites defer the fetching of content at the (not yet scrolled-to) bottom of the page. Some pages also split their fetches into a lightweight "top-half" which quickly displays visual elements, and a more heavyweight "bottom-half" which grabs the large resources that are actually represented by the GUI elements. Defining load times in these cases is difficult—the initial page load may seem quick, but if users try to access the rich data too quickly, they may experience a second, more painful load time.

Firefox issues the `DOMContentLoaded` JavaScript event when it has fetched all of a page's HTML, CSS, and JavaScript. It fires the `load` event when *all* of the page's content has been fetched. Silo definitely reduces the time to `DOMContentLoaded`; in the simple example of Figure 1, this time is reduced from three RTTs to two. Silo typically reduces the time to `load` as well. However, for any given page, `load` time is usually heavy-tailed [17, 19, 27]. This is caused by a variety of events, such as random network congestion which slashes throughput for some TCP connections, or heavily loaded servers for widely shared third-party images or advertisements. Techniques for optimizing load times for multimedia files, e.g., through image spriting [23], are complimentary but orthogonal to Silo. However, for the sake of completeness, our results in Section 4.2 describe Silo's impact on both `DOMContentLoaded` time and `load` time.

[2] JavaScript is rarely sent compressed since several browsers have buggy decompression routines for scripts.

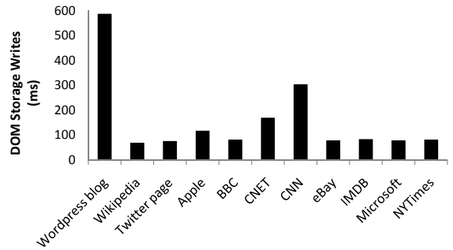


Figure 6: DOM storage write overheads.

4.2 Reducing Page Load Times

Figure 4 shows the content statistics for several popular web pages. It is interesting to note the size of the JavaScript files; as shown in Figure 4(a), many websites have hundreds of KB of JavaScript code, and JavaScript makes up more than 60% of the byte content for the front pages of Wikipedia, BBC, and CNN. This result is perhaps counterintuitive, since modern web applications are popularly characterized as being multimedia-heavy. This conventional wisdom is certainly true, but it does underestimate the pervasiveness and the size of JavaScript code.

Whereas Figure 4(a) breaks down content type by byte size, Figure 4(b) describes content type by object count. Viewed from this perspective, there are fewer opportunities for inlining. For example, in the popular DavisW Wordpress blog, 22% of all bytes are HTML, but only 6.7% of distinct objects are HTML. Similarly, in the BBC front page, 69% of all bytes belong to HTML, JavaScript, and CSS, but only 27% of the total objects are inlineable. The number of distinct objects governs the number of HTTP requests needed to build a page. Thus, the difference between Figure 4(a) and (b) may seem to doom any efforts to reduce load times through inlining. However, web designers typically structure pages such that the most important objects load first. For example, code for ad-generation may load asynchronously, and embedded movie players often do not start to prefetch data until the rest of the page has loaded. Thus, Silo still has many opportunities to reduce load times.

Figure 5 shows how quickly Silo-enabled pages load in comparison to their "standard" versions. Silo's benefit is measured as the fraction of the standard load time that Silo eliminates. The best achievable reduction is 100%, and negative percentages are possible if a Silo page loads slower than its standard counterpart. Figure 5(a) depicts Silo's performance when client caches are empty, i.e., we compare a Silo page load with an empty chunk cache to a load of the regular page when the standard browser cache is empty. In five of the eleven websites, Silo reduces `DOMContentLoaded` times by 25% or more. The other sites have fewer synchronous object fetches on the critical path for page loading, so Silo-enabled versions of these pages load no faster, or even slightly slower due to Silo's computational overheads. However, Silo generally does little harm, and often provides non-trivial benefits.

Unsurprisingly, Figure 5(a) shows that Silo reduces `DOMContentLoaded` times more than it reduces `load` times. However, we emphasize that the `DOMContentLoaded` event represents the earliest point at which a page is ready for human interaction, so minimizing `DOMContentLoaded` time is worthwhile. We also note that for five of the eleven sites, Silo also reduces `load` times by 20% or more.

Figure 5(b) shows load times when clients have warm caches and Silo uses the single RTT protocol described in Figure 3. In all cases, caches were warmed with data from 9 AM on May 11, 2010, and browsers were subsequently directed towards replayed page content from an hour later. Silo generally provides few benefits when caches are warm—few (if any) new objects must be fetched, so there are fewer network round trips for Silo to mask. However, Silo did provide a large benefit to the Wordpress site, since the small change in HTML content necessitated the transfer of the entire 1 MB file in the standard case, but only a few chunks in Silo's case. In this scenario, Silo reduced latency not by hiding a round trip, but by dramatically reducing the amount of data that had to be shipped across the wide area.

Silo reads from DOM storage to gather locally cached chunks, and it writes to DOM storage to cache new blocks sent by the server. In Firefox's current implementation of DOM storage, even small reads and writes are fairly expensive, with writes being two to five times slower than reads. Figure 6 shows Silo's write throughput for committing all of a page's chunks, showing that this operation, undertaken when clients have completely cold caches, generally requires 50–150 ms. The DOM storage API is a new browser feature that is currently unused by most sites, but we expect its throughput to improve as the API becomes more popular and browser implementers have a stronger motivation to make it fast.

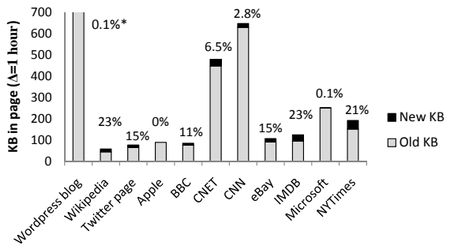


Figure 7: Byte turnover: 1 hour.

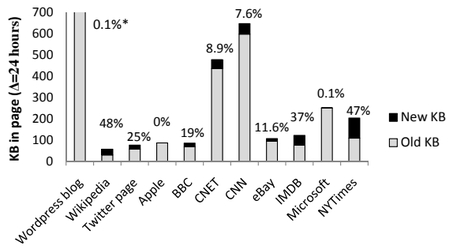


Figure 9: Byte turnover: 24 hours.

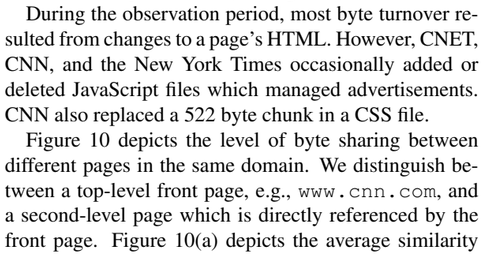


Figure 8: Byte turnover: 12 hours.

4.3 Turnover Rates for Inlineable Data

Figures 7, 8, and 9 show how inlineable page content evolved over a 24 hour period starting at 9:30 AM on Friday, January 8, 2010. These results show that byte turnover varies widely across pages. For example, the Apple website showed no variation at all during this particular period. The Wordpress blog also showed little change, since the bulk of its content consisted of a large HTML file with minimal deltas and a set of static CSS. In contrast, the New York Times site had 21% of its chunks replaced in an hour; after a day, almost half of the chunks were new. The Wikipedia front page had a similar level of turnover, since it also rotates top-level stories frequently. Interestingly, despite it high level of visual turnover, CNN only generated 2.8% new chunks in an hour, and 7.6% new chunks over the period of a day. This is because CNN contained large amounts of stable JavaScript and CSS (see Figure 4).

During the observation period, most byte turnover resulted from changes to a page's HTML. However, CNET, CNN, and the New York Times occasionally added or deleted JavaScript files which managed advertisements. CNN also replaced a 522 byte chunk in a CSS file.

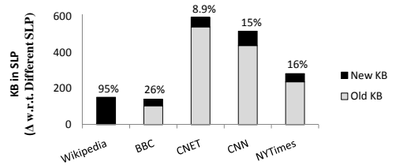
Figure 10 depicts the level of byte sharing between different pages in the same domain. We distinguish between a top-level front page, e.g., `www.cnn.com`, and a second-level page which is directly referenced by the front page. Figure 10(a) depicts the average similarity of five random second-level pages to each other, whereas Figure 10(b) shows the average similarity of these pages to the front page. These results show that second-level pages have more in common with each other than with their front pages. However, the CNN and CNET front pages offer significant opportunities for clients to warm their chunk caches for subsequent second-level page views.

When examining the second-level New York Times pages, Silo reported a large amount of redundant data. Upon checking Silo's log files, we discovered that each page repeated the same `<script>` tag four times. The script was a piece of advertising management code hosted by a third party. Its multiple inclusions were apparently harmless in terms of the page's behavior, but scripts are obviously not required to be idempotent. Thus, Silo's chunking scheme is useful for alerting content providers to potentially unintended duplication of scripts.

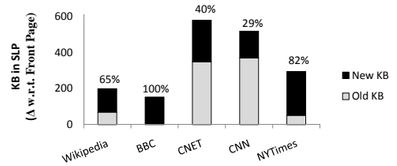
5 Related Work

Silo's most direct inspiration was LBFS [16], a network file system layered atop a distributed chunk store. In LBFS, each file is represented as a set of Rabin-delimited chunks. Clients and servers maintain an index of locally stored chunks. Whenever hosts must exchange files, they transfer chunk manifests indicating the data involved in local file operations; a host only fetches raw chunk data if no local copy exists.

Silo uses the advanced programming environment of modern browsers to implement an LBFS-like protocol atop HTTP. Value-based Web Caching [20] has a similar goal. VBWC introduces two new proxies, one inside the network belonging to an ISP, and another on the client. The browser issues web requests through the local proxy, and the local proxy engages in an LBFS protocol with the ISP proxy, which fetches and chunks web content. Unlike Silo, VBWC requires modification to client machines, hindering deployability. VBWC also does not



(a) Similarity of two second-level pages.



(b) Similarity of front page and second-level page.

Figure 10: Intra-domain page similarity. Top-of-bar fractions are the percentages of new content.

exploit object inlining to reduce the volume of HTTP requests issued by the client.

A variety of other projects have explored delta-encoding for web traffic. For example, Douglis studied the degree of similarity between different pages on the same site [5]. Chan described how objects that already reside in a browser cache can act as reference objects for delta-encoding new files [3]. Savant extended both lines of research, showing that delta-encoding of HTML files from the same site can achieve compression ratios of greater than 80% [21].

Web developers can reduce user-perceived load times by deferring the fetches for components which are not immediately needed. For example, Yahoo ImageLoader [29] provides a JavaScript framework for delaying the load of images that the user will not need within the first few seconds of viewing the page; such images might be positioned beneath the initially visible portion of the page, or they might only be needed if the user performs a certain action. The Doloto [12] tool provides a similar service for JavaScript code. Doloto analyzes a pages JavaScript behavior and identifies two sets of code: code which is invoked immediately, and code which is used infrequently, or only used a few seconds after the initial page load. After collecting this workload data, Doloto rewrites the application code, loading the former code set at page initialization time, and lazily downloading the latter set, only fetching it on demand if its functionality is actually needed. Silo is orthogonal to projects like ImageLoader since its delta-encoding does not apply to multimedia files. Silo is complimentary to projects like Doloto since Silo can reduce the transfer time of any JavaScript that Doloto labels as "immediately necessary."

6 Conclusions

Slow web pages frustrate users and decrease revenues for content providers. Developers have created various ways to defer or hide fetch latencies, but perhaps the most effective technique is the most straightforward: reducing the number of HTTP requests required to build a page. Unfortunately, this strategy presents content providers with a quandary. They can reduce the number of objects in each page, but this can negatively impact the rich content of the page. Alternatively, the content provider can inline the bodies of previously external JavaScript and CSS files. However, this destroys the cacheability of these files, since standard browser caches only store objects named via external URL pointers.

In this paper, we introduce Silo, a new framework for reducing load times while preserving cacheability. Silo exploits JavaScript and DOM storage to implement a delta-encoding protocol atop standard HTTP. Using the Silo protocol, a web server can aggressively inline JavaScript and CSS without fear of losing cacheability. Indeed, since Silo has complete control over its DOM storage cache, it can provide a *finer* granularity of caching than that provided by the browser. Silo's client-side component consists of standard JavaScript, meaning that Silo can be deployed to unmodified browsers. Experiments show that Silo's inlining and chunking protocol can reduce load times by 20%–80% for pages with large amounts of JavaScript and CSS. Additionally, a Silo web server's chunking facilities, in concert with its ability to record HTTP sessions, provide a useful platform for studying the turnover rate of data in web pages.

References

[1] Akamai Technologies. Akamai Reveals 2 Seconds as the New Threshold of Acceptability for eCommerce Web Page Response Times, September 14 2009. `http://www.akamai.com/html/about/press/releases/2009/press_091408.html`.

[2] Chakravorty, R., Banerjee, S., Chesterfield, J., Rodriguez, P., and Pratt, I. Performance Optimizations for Wireless Wide-area Networks: Comparative Study and Experimental Evaluation. In *Proceedings of Mobicom* (Philadelphia, PA, September 2004), pp. 159–173.

[3] Chan, M. C., and Woo, T. Cache-based Compaction: A New Technique for Optimizing Web Transfer. In *Proceedings of INFOCOM* (New York, NY, March 1999), pp. 117–125.

[4] DIXON, P. Shopzilla Site Redesign: We get what we measure. In *Presentation at O'Reilly Conference on Web Performance and Operations* (June 2009).

[5] DOUGLIS, F., AND IYENGAR, A. Application-specific Delta-encoding via Resemblance Detection. In *Proceedings of USENIX Technical* (San Antonio, TX, June 2003), pp. 113–126.

[6] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[7] FLANAGAN, D. *JavaScript: The Definitive Guide*, 5 ed. O'Reilly Media, Inc., 2006.

[8] GOOGLE. Gears: Improving Your Web Browser. `http://gears.google.com/`, 2008.

[9] GOOGLE. AdWords: How does load time affect my landing page quality? `http://adwords.google.com/support/aw/bin/answer.py?answer=87144`, 2009.

[10] KRISTOL, D., AND MONTULLI, L. HTTP State Management Mechanism. RFC 2109 (Draft Standard), February 1997.

[11] LIN, X.-Z., WU, H.-Y., ZHU, J.-J., AND WANG, Y.-X. On the Performance of Persistent Connection in Modern Web Servers. In *Proceedings of the ACM Symposium on Applied Computing* (Fortaleza, Ceara, Brazil, March 2008), pp. 2403–2408.

[12] LIVSHITS, B., AND KICIMAN, E. Doloto: Code Splitting for Network-Bound Web 20 Applications. In *Proceedings of SIGSOFT Symposium on the Foundations of Software Engineering* (2008), pp. 350–360.

[13] MICKENS, J., HOWELL, J., AND ELSON, J. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of NSDI* (San Jose, CA, April 2010).

[14] MILIAN, M. What's powering Web apps: Google waving goodbye to Gears, hello to HTML5. In *Los Angeles Times* (November 30 2009). `http://latimesblogs.latimes.com/technology/2009/11/google-gears.html`.

[15] MOZILLA DEVELOPER CENTER. Gecko-Specific DOM Events. `https://developer.mozilla.org/en/Gecko-Specific_DOM_Events`.

[16] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A Low-bandwidth Network File System. In *Proceedings of SOSP* (Banff, Canada, October 2001), pp. 174–187.

[17] OLSHEFSKI, D., NIEH, J., AND AGRAWAL, D. Inferring Client Response Time at the Web Server. In *Proceedings of SIGMETRICS* (Marina del Rey, CA, June 2002), pp. 160–171.

[18] RABIN, M. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[19] RAJAMONY, R., AND ELNOZAHY, M. Measuring Client-Perceived Response Times on the WWW. In *Proceedings of USITS* (San Francisco, CA, March 2001).

[20] RHEA, S., LIANG, K., AND BREWER, E. Value-Based Web Caching. In *Proceedings of the World Wide Web Conference* (Budapest, Hungary, May 2003), pp. 619–628.

[21] SAVANT, A., AND SUEL, T. Server-Friendly Delta Compression for Efficient Web Access. In *Proceedings of the International Workshop on Web Content Caching and Distribution* (Hawthorne, NY, September 2003).

[22] SINGHAL, A., AND CUTTS, M. Using site speed in web search ranking. `http://googlewebmastercentral.blogspot.com/2010/04/using-site-speed-in-web-search-ranking.html`, April 9 2010.

[23] SOUDERS, S. *High Performance Web Sites: Essential Knowledge for Front-End Engineers*. O'Reilly Media, Cambridge, MA, 2007.

[24] SOUDERS, S. Loading Scripts Without Blocking. In *High Performance Web Sites blog* (April 27 2010). `www.stevesouders.com/blog/2009/04/27/loading-scripts-without-blocking`.

[25] THEURER, T. Performance Research, Part 2: Browser Cache Usage–Exposed! In *Yahoo User Interface Blog* (January 4 2007). `yuiblog.com/blog/2007/01/04/performance-research-part-2`.

[26] W3C WEB APPS WORKING GROUP. Web Storage: W3C Working Draft. `http://www.w3.org/TR/2009/WD-webstorage-20091029`, October 29 2009.

[27] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well–Conditioned, Scalable Internet Services. In *Proceedings of SOSP* (Chateau Lake Louise, Canada, October 2001), pp. 230–243.

[28] WORLD WIDE WEB CONSORTIUM. Document Object Model. `http://www.w3.org/DOM`, 2005.

[29] YAHOO! YUI 2: ImageLoader. `http://developer.yahoo.com/yui/imageloader`, 2010.

Pixaxe: A Declarative, Client-Focused Web Application Framework

Rob King
Principal Researcher, TippingPoint DVLabs

Abstract

This paper provides a brief introduction to and overview of the Pixaxe Web Application Framework ("Pixaxe"). Pixaxe is a framework with several novel features, including a transparent template system that runs entirely within the web browser, an emphasis on developing rich internet applications as simple web pages, and pushing as much logic and rendering overhead to the client as possible. This paper also introduces several underlying technologies of Pixaxe, each of which can be used separately: *Jenner*, a completely client-side template engine; *Esel*, a powerful expression and query language; and *Kouprey*, a parser combinator library for ECMAScript.

1 Introduction

There has been an explosion of frameworks for the building of Rich Internet Applications (RIAs). Frameworks exist using every popular (and unpopular) programming paradigm, language, and server side technology. Frameworks range in complexity from a simple JavaScript libraries that merely ease the handling of normal DOM events to frameworks that completely abstract away HTML and JavaScript. Some frameworks run entirely on the client, while others require considerable server side support.

This paper describes the Pixaxe Web Application Framework ("Pixaxe"). Pixaxe is interesting in that it focuses on creating web pages using a powerful, functional expression language that is a superset of normal XHTML. The compiler and virtual machine for this language is implemented entirely in ECMAScript and runs entirely within a web browser. By evaluating these expressions, web pages are rendered, inputs are validated, and client-server communication is initiated.

Pixaxe's other interesting features include a complete parser combinator library running entirely within a web browser, an extremely server agnostic design (more so than most "server agnostic" frameworks), an extremely easy to use Model-View-Controller (MVC) design, and a very bandwidth-frugal design that transmits a page only once and then transmits only changes to interesting data.

Pixaxe was deisgned to be very useful in developing web interfaces for legacy applications, or in other situations where the web interface is not the primary interface to a set of data. It was also designed to be very efficient in the use of server resources, by limiting required bandwidth and performing as much computation and rendering on the client as possible. In fact, Pixaxe requires nothing more of a server than the ability to serve static files.

In feel, Pixaxe is closest to XForms [9] [1], in that it views web pages as declarative interfaces modifying local models that can then be synchronized with servers without reloading the page.

2 A Short Example

Unlike some other application frameworks, Pixaxe attempts to keep web application development as close to web page authoring as possible. It does not abstract away HTML, CSS, or any other web technology but instead encourages the developer to write directly in HTML using a declarative, template-driven approach. This allows developers to leverage existing technologies to the largest extent possible, and turns XHTML into a powerful interface description language (especially when using the XSLT macros provided by Pixaxe, discussed in section 3.5).

Figure 1 illustrates a simple example of a Pixaxe application. When viewed in a web browser, this example would be rendered as shown in Figure 2.

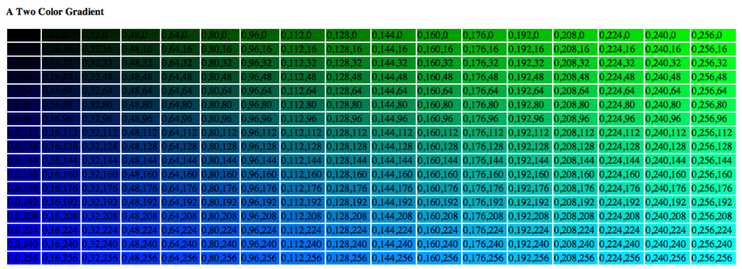
This example shows some interesting features of Pixaxe: there are no explicit calls in any scripting language, it looks like a normal XHTML document, and there are some special template directives freely mixed with the markup. What is unusual is that this page is rendered,

Figure 1 A simple Pixaxe application.

```
<html>
<head>
    <title>Simple Example</title>
    <script type="text/javascript"
            src="pixaxe.js" />
</head>

<body>
    <p><b>A Two Color Gradient</b></p>
    <table style="width: 100%;">
    ${for i from 0 to 256 by 16 return
      <tr>
        ${for j from 0 to 256 by 16 return
          <td style="background:
                     rgb(0, ${j}, ${i});">
              0,${j},${i}
          </td>
        }
      </tr>
    }
  </table>
</body>
</html>
```

Figure 2 Rendered output (in Apple Safari).



along with the special template directives, entirely by the client. This page could be stored in a file on a local filesystem and opened in a web browser, and it would be rendered correctly.

This illustrates one of the core guiding principles of Pixaxe development: all display and rendering should be done by the client. This stems from the assumption that the web interface is but one of many interfaces to the same datasource. This example also provides a sample of Pixaxe's template syntax which is based on a purely functional and side-effect-free expression language known as *Esel* [2].

This example shows how Pixaxe integrates with other technologies and encourages developers to write web pages correctly and portably while still reaping the benefits of Pixaxe.

3 Developing Rich Internet Applications With Pixaxe

Pixaxe attempts to keep development of rich internet applications similar to the development classic web pages. It integrates a classic Model-View-Controller (MVC) development paradigm (as described in [6]) and declarative paradigm, but does not enforce this. Pixaxe itself views a web page as two combined entities: a *template* and a *store*. The template corresponds roughly to the view portion of the MVC model, and the store corresponds roughly to the model portion. The controller portion of the MVC model (handling user input) is handled by a combination of Esel expressions embedded in templates (which validate the input and update the model) and the store (which synchronizes the model with all interested parties and instructs the page to re-render if necessary).

Brief overviews of the template language, store, handling user input, and client-server communications are presented below.

3.1 The Jenner Template Language

The template language of Pixaxe is known as *Jenner*. It is capable of being used independently of Pixaxe as a simple client-side template engine. (Jenner itself is a superset of the *Esel* expression and query language, which itself may be used independently of Jenner).

The Jenner template language is a full-fledged expression and data query language. Jenner expressions are used to retrieve and optionally transform data from a store and insert the results into the rendered page. Jenner expressions can also be used to validate user input and update the store when data changes.

Jenner's syntax and semantics are similar to those of Java's Unified Expression Language (see [1]); these similarities are intentional. While many small expressions are valid in both languages, the two are different enough to be considered generally incompatible.

Jenner has two different types of expressions: *value* expressions and *reference* expressions. The name "value expression" is perhaps misleading, since all Jenner expressions return a value, but it is useful to differentiate between the two types of expressions.

3.1.1 Value Expressions

An Jenner value expression can take two forms: *literal* and *bracketed*. A literal expression is used to declare a single value, though this literal expression may contain other subexpressions (which would be bracketed).

For example, these are all valid literal expressions:

```
42
Hello, World!
true
null
```

Each of the above expressions are, as are all expressions in Jenner, typed. Jenner supports all primitive types defined by ECMAScript, except for the *undefined* value which is treated as equivalent to *null*.

Jenner also supports a special *collection* type, which is roughly equivalent to an ECMAScript array. Jenner provides a powerful list comprehension syntax, discussed below, for expressing collections.

A bracketed expression is one that begins with the special sequence `${` and ends with the character `}`. Inside these delimiters, other Jenner expressions may be embedded, including other bracketed expressions. Within these brackets, the full expression language is available.

Jenner supports most common operators, including basic and modular arithmetic, boolean combination, string concatenation, and a C-style trinary conditional operator. Jenner also includes operators for testing set membership and fast tests for empty strings and collections.

Jenner also supports variable lookups (but not assignment). These can be written in a fashion similar to that of ECMAScript. Variables are exported to the Jenner runtime by the hosting environment; it is not possible to access objects that are not directly referencable from these exported variables.

As an example, this expression returns the sum of the value of the `aNumber` variable and 1, if `aNumber` is defined. If `aNumber` is not defined, *null* is returned:

```
${empty aNumber ? null : aNumber + 1}
```

Bracketed expressions may also contain function calls. Functions cannot be defined in Jenner itself; they must be exported by the hosting application. Functions can be namespaced to avoid name collision. Jenner contains by default a reasonable standard library of functions, but developers are encouraged to write and share new collections of functions.

Perhaps the most interesting form a value expression is Jenner's list comprehension expression, known as a *FLWR* expression. [3] FLWR expressions are used to create Jenner collections and are also used to query collections of data by comprehending a collection of results matching some predicate.

FLWR expressions always evaluate to a collection, even if that collection is empty. FLWR expressions can be viewed as iterative constructs, where a variable is assigned a numeric value from some lower bound to some upper bound. For each iteration of this loop, the value is incremented by an optional step expression or by one. FLWR expressions provide lexical scoping – local variables can be declared that are visible only within the body of a FLWR expression. For each value generated by some generator expression, if it an optional conditional clause evaluates to true, the value is added to the resulting collection in order.

For example, this expression would create a new collection consisting of all members of an array of strings which are longer than three characters, with each result capitalized:

```
for i from 0 to names.length - 1
    var s := names[i]
    where s.length > 3
        return upper(s)
```

FLWR expressions are similar to the *FLWOR expressions* of the XQuery language (see [8]); this similarity is intentional.

3.1.2 Reference Expressions

Jenner also supports expressions that return a reference to an object in the hosting application. References are represented as a tuple consisting of an object reference and a (possibly empty) property name of that object.

Reference expressions are delimited by #{ and }. Within these delimiters, a subset of the full Jenner value syntax is allowed; the result of the embedded expression must be either *null* or an object with an optional property name. Only objects exported from the hosting application are vali.

This example expression would return a reference to the object denoted by `people.addresses` with a property name of `1`:

```
#{people.addresses[1]}
```

Reference expressions cannot be mixed with value expressions and may not be embedded in other expressions; they must be entirely standalone.

Reference expressions do not directly support assignment; this is considered a feature. Instead, they may be used by the hosting application as a target for assignment, but the hosting application is free to use or not use reference expressions in any way.

3.1.3 Node Expressions

Jenner also provides a *node* type. This provides a literal syntax for elements which is identical to the XML syntax for specifying elements. A node literal may be used in an expression anywhere a literal is allowed.

Since nodes may contain other nodes and Jenner expressions, a web page can therefore be considered a single large Jenner expression.

Node literals' names must be specified directly; they cannot be expressions. This applies as well to the names of attributes in node literals. The contents of attributes and nodes, however, can be any valid non-node Jenner expression.

Jenner can leverage any expression. This has some interesting consequences. For example, a conditional expression can be used to only render a node depending on the state of the application. For example, this template could be used to display a list of messages should any be present:

```
<body>
    ${not empty msgs ?
        <ul>
        ${for i from 0 to msgs.length - 1
            return
            <li style="text:
                       ${msgs[i].unread ?
                       'red' : 'black'}">
                ${msgs[i]}
            </li>
        }
        </ul>

    : <p>No messages</p>}
</body>
```

Note the embedded Jenner expression inside the `style` attribute of the `li` element.

Jenner's template language is not particularly new or unique; it bears large similarities to templates used by Sun's Java Server Pages (see [2]) or other server-side

template systems. What makes Jenner interesting is that all template evaluation and rendering is performed entirely by the client. Jenner is believed to be the first completely client-side template system in which markup and template instructions can be freely mixed (other than XSLT).

Jenner has some advantages over other client-side template systems. For example, while most modern web browsers support XSLT templates, there are no standard ways to apply XSLT transformations multiple times; generally the transformations are applied only once, at page load. Jenner may re-render the page at any time. Some other templating systems, such as JavaScriptMVC [4] perform template evaluation and rendering entirely on the client, but do not allow markup and template instructions to be freely mixed.

3.2 Storing Data

As stated above, Pixaxe loosely follows the MVC paradigm for development. A single page has a view component in the form of a Jenner template and a single model in the form of a *store*. The store contains all data that is relevant to the application at any given time.

Pixaxe keeps the store synchronized between all interested parties. When the data in the store changes, Pixaxe calls Jenner to re-render the page. If the user performs an action that results in client-server communication, Pixaxe serializes the store and passes it to the server, and then updates the store with the results of server processing. If the user performs input that updates the store, the page is re-rendered to reflect the new values.

When the store is serialized and sent to the server, the server may make changes and send an updated version of the store back to the client. The client can then re-render the page to reflect the new values. Note that this asynchronous transmission of the serialized store to and from the server is the only client-server communication in Pixaxe; the page itself is downloaded only once. This can result in significant bandwidth savings, and also allows for a very abstract server interface - client-server communication is reduced to synchronizing a simple JSON document.

Pixaxe integrates with Jenner by the simple expedient of setting the global store to be Jenner's default environment. Therefore, all but the simplest Pixaxe applications will contain something like this line somewhere in a script element:

```
com.deadpixi.jenner.defaultEnvironment =
    new com.deadpixi.pixaxe.Model( ...
```

Pixaxe tries to keep the creation of the store as declarative as possible. The `com.deadpixi.pixaxe.Model` constructor takes two arguments: the first is a single object that becomes the store. Developers are encouraged to write this object using ECMAScript object literal syntax, to keep to the declarative programming style as much as possible. An optional second argument to the constructor is a URL whose contents (which must be JSON, see [3]) will be loaded into the store after the page has finished loading.

As an example, this might be the object used as the store of a simple address book, with some potentially useful initial values:

```
{
    "addresses": [
        {"name": "Rob",
         "name": "jking@deadpixi.com"},
        {"name": "Betsy",
         "name": "betsy@example.com"}
    ]
}
```

This object would then be available to Jenner as its default environment, and therefore expressions in the template would have access to a variable called `addresses`.

Alternatively, this object (which is expressed in JSON) could be placed in a separate file, and a URL specifying that file could be passed as the second argument to the `Model` constructor. This would cause this file to be loaded and merged with the store after the page has finished its initial load.

Note that the page is initialized only once, at page load time. The page itself is never again transmitted across the network, and there is no HTML form style "submit-reload" cycle. The store is synchronized between client and server using the *de facto* standard `XMLHttpRequest` method, and the page is re-rendered by re-evaluating the Jenner expression which makes up the page.

3.3 Handling User Input

In keeping with Pixaxe's goal of leveraging existing technology, Pixaxe using XHTML as a rich interface specification language. Standard XHTML form controls are used to create input elements (possibly augmented by XSLT macros, see section 3.5).

Input controls can be placed anywhere in a document. If a control is placed inside of a `form` element, then manipulating the control will result in some type of synchronization of the store with the server (note that this

does not mean a traditional XHTML form submission). Controls outside of a `form` element result only in local changes to the store (which may of course be synchronized with the server later).

Pixaxe supports all HTML control elements, including `input`, `button`, `select`, and `textarea`. Each of these controls can be linked to the page's store by placing a reference expression in the element's `name` attribute. Pixaxe will then ensure that the value in the store and the controls's value are synchronized.

For example, this declaration would create a text input element whose value would be placed in the `name` property of the page's store:

```
<input name="#{name}"
       value="${name}" />
```

In this example, whenever the user activates a submission control, the page's template is re-evaluated, rendering the input control with the current value of the *name* variable in the store. Any change in the control's value by the user would be automatically placed into the store. This synchronization between store, template, and user allows for a very powerful and declarative method of interface specification.

This gives rise to a very simple, two stage process for the handling of input when the user-activated submission control is not part of a form. First, the page's store is updated such that all controls that reference the store have their values placed in the store. The page is then re-rendered.

For example, this page will automatically re-render to display the current value of the name control whenever the user activates the submit control:

```
<p>Hello, ${empty name ? 'Stranger' :
                         name}</p>
<input name="#{name}" />
<input type="submit" value="Ok" />
```

No client-server communication takes place when the user activates the submit control in this example: the store's *name* variable is updated and the page is re-rendered by re-evaluating its template locally.

To integrate form controls with Pixaxe, the semantics of the standard attributes assigned to form controls is overloaded. The meanings assigned to each attribute are described below:

accept The optional `accept` attribute can be used to modify or validate the value of the control before it is copied to the store. The Jenner expression specified in the `accept` attribute is evaluated each time the control's value is evaluated and the value of the accept expression is instead copied to the model.

name If the value of the `name` attribute is an Jenner reference expression, then the control is linked to the model. Whenever the user activates a submit control, the value of the current control is copied to the property pointed to by the reference.

value If the value of this attribute is an Jenner expression, it is evaluated each time the page is rendered and the value of the control is set to the result.

These attributes are all specified as part of the HTML standard. All other attributes may contain Esel expressions; these will be evaluated and set each time the page is rendered (see, for example, the `style` attribute in Figure 1).

Synchronization with the store is bidirectional. If the store is updated through some other means (generally through client-server communication), the control is updated to keep its value synchronized. Thus, controls always accurately reflect the state of the store and vice-versa.

All types of controls can be used, but controls of type *hidden* are treated specially. A hidden control is used to set an initial default value for some part of the store. Thus, the developer can initialize certain values in the store when the page initially loads.

3.4 Client-Server Communications

Client-server communication is done entirely via JSON documents POSTed to URIs asynchronously. Communication is not viewed as a imperative action, but rather as a synchronization of the state of the page's store with the server (by convention this is known as "synchronizing the world").

Client-server communication is initiated when the user activates a submit control that is a child of a `form` element. There are two possible methods of communication in that case: classic form submission, and store synchronization.

If a form element's `enctype` attribute is not set to "text/javascript", then the form is submitted as per the HTML standard and whatever is returned from the form replaces the contents of the page. This is used to interface with legacy code that insists on using normal HTML form submission semantics.

If a form element's `enctype` attribute is set to "text/javascript", however, the form is not submitted using the classic method. Instead, Pixaxe first synchronizes the values of all controls with the page's store. The page's store is then serialized into JSON and POSTed asynchroniously to the URL specified by the form's target. It is expected that the server return a JSON object which is then merged with the page's store. This merge process simply copies all properties from the returned object to the store with the same name, potentially overwriting the values of old properties or adding new ones. Properties that are not named in the returned object are not affected. Note that in this case, the page is not reloaded in any way; it is simply re-rendered locally by Jenner.

For example, assume that the page's store is displayed here, as JSON:

```
{
    "name": "Arthur",
    "id": 42
}
```

If the user activated a submit control contained by a `form` element whose `enctype` attribute is set to "text/javascript", the page's store would be serialized and submitted as described above.

Below is an example response from the server, which would be returned as the body of the response to the POST request.

```
{
    "id": -1,
    "invalid": true
}
```

Pixaxe would merge this response with the page's store, resulting in the new contents of the store.

```
{
    "name": "Arthur",
    "id": -1,
    "invalid": true
}
```

After this synchronization process, the template is re-rendered locally by Jenner.

Forms whose `enctype` is "text/javascript" may also place an Esel expression in their `accept` attribute. This expression must evaluate to *true*, or the server synchronization will not happen. This expression can be used to validate user input before initiating client-server communications. The form will still be re-rendered even if this validation fails, giving the application a chance to inform the user of invalid input.

3.5 XSLT Macros

Pixaxe comes with a collection of XSLT stylesheets. These stylesheets define macros, which consist of special nodes in the original markup that are transformed by XSLT at page load time to a collection of normal HTML markup and Jenner template instructions.

Several useful XSLT macros exist, including macros that create paged tables, modal dialog boxes, lightboxes, tab boxes, and AJAX-style file upload controls. These stylesheets can be applied automatically by most modern web browsers at page load time.

One interesting aspect of these macros is that they are implemented entirely in regular HTML and Jenner instructions. They therefore require no additional server side support. Since the macro expansion is applied only at page loading time, it does not significantly affect page re-render times.

These macro packages can be arbitrarily complex. For example, Figure 3 illustrates the code for a simple tab box on a web page, and Figure 4 illustrates the rendered page. Figure 5 illustrates (partially) the expansion of the `dppx:tab-box` and `dppx:tab` macros.

Macros can be arbitrarily complex, giving developers the ability to abstract away as much XHTML as desired. Additionally, by allowing the free mixing of Jenner markup and HTML, XSLT stylesheets applied to pages have a much richer "target language" than traditional stylesheets.

3.6 Putting It All Together

This example demonstrates a complete, if simple, Pixaxe application that uses a large number of the described features. This example application builds a very simple, shared bulletin board. Users can post short messages which are then visible to all other users.

First, the page store is declared. This should be placed in a separate file from the web page, in this example called "store.js".

```
com.deadpixi.jenner.defaultEnvironment =
    new com.deadpixi.pixaxe.Model({
        msgs: [],
        newMsg: ""
    }, "messages.json");
```

The second argument is a URL that points to a file that is assumed to contain the list of messages currently known to the server as a single JSON object, with a property called "msgs". This URL will be used to load the initial values into the page's store after the page has finished loading.

The rest of the application is defined in a normal XHTML file. For brevity, only the `body` element of this file is shown below. Esel standard function `now` is used to indicate to the user how up to date the page's display is. Below this is a list of messages retrieved from the server.

Figure 3 An example of a page using XSLT macros.

```
<body>

<dppx:tab-box>
    <dppx:tab label="First Tab" selected="true">
        <p>Tab bodies can consist of arbitrary HTML and Jenner markup.</p>
        <p>For example, here is the current value of the "name"
           variable in the Store: ${name}</p>
    </dppx:tab>

    <dppx:tab label="Second Tab">
        <p>Another tab.</p>
    </dppx:tab>

    <dppx:tab label="Third Tab">
        <p>Yet another tab.</p>
    </dppx:tab>

    <dppx:tab label="Fourth Tab">
        <p>Tabs everywhere!</p>
    </dppx:tab>
</dppx:tab-box>

</body>
```

Figure 4 The rendered example of the source in Figure 3.

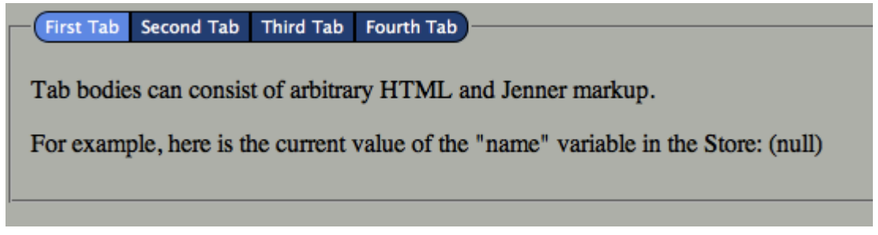


Figure 5 A partial expansion of the `dppx:tab-box` and `dppx:tab` macros.

```
<fieldset><input type="hidden" value="id4127134"
                 name="#{controller.dppx_tabselid4127132}"/>
        <legend><input type="submit"
                       name="#{controller.dppx_tabselid4127132}"
                       class="dppx-tab dppx-tab-left
                       dppx-tab-${controller.dppx_tabselid4127132
                       != 'id4127134' ?
                       'un' : ''}selected"
                       accept="id4127134" value="First Tab" />
                <input type="submit"
                       name="#{controller.dppx_tabselid4127132}"
                       class="dppx-tab dppx-tab-${
                       controller.dppx_tabselid4127132
                       != 'id4127149' ?
                       'un' : ''}selected"
                       accept="id4127149" value="Second Tab" />
...
    <div class="dppx-tab-body
               dppx-tab-body-${controller.dppx_tabselid4127132
               != 'id4127134' ? 'un' : ''}selected">
        <p>Tab bodies can consist of arbitrary HTML and Jenner markup.</p>
        <p>For example, here is the current value of the "name"
           variable in the Store: ${name}</p>
    </div>
```

```
<body>
    <p>${msgs.length}
        message${msgs.length != 1 ?
                 's' : ''}</p>
    <p>Updated at ${now()}.</p>

    <ul>
     ${for i from 0 to msgs.length - 1
        var m := msgs[i] return
        <li>${m}</li>
     }
    </ul>
```

After this, the user is given a text input area to post a new message. This input control is linked to the page’s store by placing a reference expression in the control’s `name` attribute.

```
    <hr />
    <input name="#{newMessage}" />
```

A submit control is placed inside a `form` element whose action points to the message posting script on the server.

```
    <form enctype="text/javascript"
          action="/cgi-bin/post.cgi">
        <input type="submit"
               value="Add Message" />
    </form>
```

This would create a complete and fully functional application, except for the server components. The list of messages could be a simple JSON document. The message posting script would need only to deserialize the contents of the POSTed store from JSON, extract the ”newMsg” property from this deserialized object, append its value to the messages document, and return that document to the client.

4 Comparison to Other Frameworks

Pixaxe was designed to be very light in terms of server resources. This makes it well suited for situations where it is not the primary interface to a service, or where the service is a legacy application.

This section compares Pixaxe to some other frameworks, and helps illustrate the situations for which Pixaxe is best suited.

4.1 Pixaxe and the Google Web Toolkit

The Google Web Toolkit (`http://code.google.com/webtoolkit/`) is a mixed client-server toolkit

from Google. The toolkit (commonly referred to as "GWT") relies heavily on the Java language for development. GWT in fact compiles Java sources to JavaScript code which is then executed on the browser.

GWT is in many ways the opposite of Pixaxe. Developing with GWT follows much the same process as developing any large Java application - there is a compile/debug/edit cycle, and the use of Java-centric tools is encouraged. GWT encourages a mixed object-oriented and procedural development paradigm by building interfaces programatically in Java (though there has been more support lately for declarative interface specification).

While GWT can be used in a server agnostic manner, much of the code is written assuming a Java Servlet Container [5] on the server. GWT also abstracts away large amounts of HTML.

GWT has many advantages: an extremely large user base, and the support of one of the largest technology companies in the world, as well as numerous mature tools (such as Eclipse [6]) that can make the development of large applications considerably easier.

By the same token, GWT is often overkill for small projects or in situations where a Java Servlet Container is not available on the server. For smaller, rapidly-developed applications, Pixaxe's lack of a compile/debug/edit cycle can be a major advantage. In situations where server resources are limited or Java technologies cannot be used, Pixaxe's server agnosticism makes it an attractive choice. Pixaxe would likely not scale well to applications the size of Google Mail [7], but has easily handled smaller applications.

4.2 Pixaxe and SproutCore

SproutCore (`http://www.sproutcore.com`) is a popular client-centric toolkit used by many popular websites. SproutCore is close in feel to Pixaxe. SproutCore is client-focused, running its code solely within the browser. It also favors a declarative interface specification, and development follows the Model-View-Controller pattern. Its use in large, successful applications such as Apple's MobileMe [8] portal illustrates that it can be successfully used for large projects and is a mature choice.

However, SproutCore differs from Pixaxe in a few interesting ways. SproutCore still follows a shortened compile/debug/edit cycle. SproutCore requires Ruby for application development, though not to run or view applications. It is server agnostic for the most part, though much of its documentation makes assumptions that Ruby is used on the server side.

SproutCore development tends to use much more JavaScript than Pixaxe. SproutCore could be called a "JavaScript framework", in that much of the business logic of code is specified in JavaScript. Indeed, interfaces are often built (semi-declaratively) using JavaScript.

Pixaxe enables an arguably simpler development path, allowing purely declarative interfaces in XHTML and a simple expression language. New "widgets" in Pixaxe are easily created through simple XHTML, and optionally made reusable through simple XSLT macros.

Pixaxe and SproutCore are well suited to many of the same tasks. SproutCore's programming interface is more powerful than that of Pixaxe, but is concomitantly more complicated. For simple, radpily developed applications, especially single-page applications, Pixaxe may be the better choice.

4.3 Pixaxe and XForms

XForms (`http://www.w3.org/MarkUp/Forms/`) was the direct inspiration for Pixaxe. XForms is an application of XML for the specification of data processing models for XML, and user interfaces to those models. It does not require, but is often used "on top of" XHTML, using the latter as part of its presentation layer.

The original applications for which Pixaxe was used were originally to be written in XForms. Unfortunately, XForms is not natively supported in any mainstream web browser. Therefore, Pixaxe was developed to provide a similar development experience, but usable on any modern web browser.

XForms has the benefit of being extremely well specified, and builds on the extensive base of XML technologies specified by the W3 Consortium. XForms allows for the free mixing of presentational markup and logic specification in a single page, and lends itself well to forms-based development cycles.

Due to this strong inspiration, Pixaxe could easily be used in any situation where XForms could be used, but with the benefit of wider support. XForms was designed to help create applications that were still web pages for the most part, and this is a design goal shared by Pixaxe.

5 The Implementation of Pixaxe

This section of the paper discusses the implementation details of Pixaxe, including all of the technologies upon which it is built. The bulk of Pixaxe's code lies in the compiler and virtual machine for the Esel and Jenner languages. Also detailed here is the Kouprey parser combinator library, which is used to create the parser for Esel.

5.1 The Kouprey Parser Combinator Library

Pixaxe uses a parser combinator library known as *Kouprey* [9]. Kouprey eases the development of developing parsers by allowing developers the ability to express grammars using simple ECMAScript statements in something resembling Extended Backaus-Naur Form (EBNF, see [5]). The generated parsers are based on the Parsing Expresson Grammar (PEG) formalism (see [4]).

Kouprey is available to be used separately from Pixaxe. It has no dependencies other than a standard ECMAScript runtime. It is sufficiently powerful that a complete parser for the Component Pascal [10] programming language has been written entirely using Kouprey.

A full discussion of Kouprey is beyond the scope of this document, but interested readers are encouraged to consult the Kouprey home page at `http://www.deadpixi.com/kouprey`.

5.2 Esel, Jenner, and Their Virtual Machine

The Jenner template language is built on top of a small expression language known as Esel. Jenner's syntax is a pure superset of that of Esel, adding only the literal node type syntax.

Esel uses Kouprey to generate its parser. Esel expressions are compiled into abstract syntax trees, which are then passed to a code generator. This code generator creates programs for a virtual machine designed to run Esel expressions.

The Esel compiler, code generator, and virtual machine are written entirely in ECMAScript and are available for use independently from Pixaxe. Esel's virtual machine is a simple stack-based virtual machine with 32 instructions. The virtual machine itself is Turing complete, and provides support for such advanced features as lexically closed environments and a foreign function interface with ECMAScript.

5.3 The Jenner Template Engine

Jenner, Pixaxe's template engine, is remarkably simple in its implementation. Upon page load, Jenner is passed a DOM element object to treat as the root of the template; by default this is the page's `body` element.

The DOM of the page is then traversed. All text nodes and all comment nodes of a special syntax (by default, any comment node whose first two characters are "##") are appended to a single Jenner expression. Any time an element node is encountered, it is assigned a unique ID and a call to the special `jenner:nodeset` function is appended to the expression. This is done recursively until the entire page has been converted to a single large Jenner expression. To render the page, the root of this expression is simply evaluated.

The `jenner:nodeset` function is used to insert nodes dynamically into the page using standard DOM manipulation. The original page is copied to serve as a template for each render. Jenner also performs extensive caching of rendered and compiled results for speed.

Jenner also allows developers to override default handling of elements and attributes by name. Pixaxe uses this functionality to assign special meaning to various attributes, mostly for form and input handling.

Jenner is available for use independently of Pixaxe. Jenner could be very useful as a display technology for other frameworks.

5.4 Pixaxe, Input Processing and Data Management

Pixaxe itself builds data management capabilities on top of Kouprey, Esel, and Jenner. It is essentially a very thin layer on top of these technologies.

The vast majority of Pixaxe's code is used in processing user input, primarily performing form control value processing and input validation. Page store management is relatively simple, consisting mostly of copying values from controls into the model and instructing Jenner to re-render the page.

5.5 Client-Server Communications

Client-server communications in Pixaxe are simple, using the de facto standard `XMLHttpRequest` (see [7]) support in modern web browsers to POST serialized versions of the page's store and merge the returned information into the store.

Pixaxe uses native JSON processing functions if possible for speed, but will fall back to using an JSON library written in ECMAScript if native functions are not available.

One interesting feature provided by Pixaxe's client-server communications subsystem is the application of callbacks. All communication takes place asynchronously. The page store can include specially named functions that will be invoked when the store is serialized, when it is merged with the server, and on various error conditions. Pixaxe provides a standard set of callback functions that will render an "input shield" over the page, preventing any user interaction until the communications cycle is complete. Their use is, of course, optional.

5.6 Cross-Browser Support

Kouprey and Esel are written in pure ECMAScript and should work without modification in all conformant environments. Jenner and Pixaxe, however, are intimately involved in the way the browser represents pages and events and therefore must be written with cross-browser support in mind.

Jenner and Pixaxe officially support Apple Safari (versions 4 and later), Microsoft Internet Explorer (versions 7 and later), and Mozilla Firefox (version 3 and later). Other versions of these browsers and other vendors' browsers may work but they are not extensively tested.

Different browsers require different syntax in certain situations for Jenner templates. Most notably, some browsers require expressions to be inside comment nodes when they are inside `table` elements, while other browsers require them to be bare. Similar situations arise when dealing with `ol` and `ul` elements. Jenner and Pixaxe provide XSLT stylesheets that can be applied to the page as it is loaded that automatically translate pages to use the appropriate format, transparently to the developer.

6 The Future

Kouprey, Esel, Jenner, and Pixaxe are all under active development and several interesting features are planned for a future release.

Kouprey's next version is expected to be considerably faster and have better error handling and reporting. It will also be rewritten to make grammar definitions more natural when using the ECMAScript Compact Profile.

Esel's virtual machine is being rewritten to be faster and smaller. There are also some proposed language extensions, including destructuring assignment and n-way case statements.

Jenner and Pixaxe will be much more tightly integrated in a future release (though it is planned that Jenner will still be usable without Pixaxe). A faster rendering algorithm is also being worked on that involves static analysis of Esel expressions to determine which portions of the page would be affected by certain changes to the page's store. Additionally, a real-time synchronization mechanism is under development that would not require users to manually indicate that a form is ready for processing.

7 Availability

All of the technologies discussed in this paper are available under a a free software license. All of these technologies are currently available and in active use.

Download links and detailed documentation for all technologies are available at `http://wwww.deadpixi.com`.

References

[1] CHUNG, K.-M., DELISLE, P., AND ROTH, M. Expression language specification. Part of the Java Community Process, see `http://www.jcp.org/en/jsr/detail?id=245`.

[2] CHUNG, K.-M., DELISLE, P., AND ROTH, M. Java serverpages 2.1. Part of the Java Community Process, see `http://www.jcp.org/en/jsr/detail?id=245`.

[3] ECMA INTERNATIONAL. Ecmascript language specification. copy available at `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`.

[4] FORD, B. Parsing expression grammars: A recognition-based syntactic foundation. copy available at `http://www.brynosaurus.com/pub/lang/peg-slides.pdf`.

[5] PATTIS, R. E. Ebnf: A notation to describe syntax. copy available at `http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf`.

[6] REENSKAUG, T. Thing-model-view-editor. archived copy available at `http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf`.

[7] W3 CONSORTIUM. Xmlhttprequest. a working draft, copy available at `http://www.w3.org/TR/XMLHttpRequest/`.

[8] W3 CONSORTIUM. Xquery 1.0: An xml query language. copy available at `http://www.w3.org/TR/xquery/`.

[9] W3 CONTORTIUM. Xforms 1.1. copy available at `http://www.w3.org/TR/xforms/`.

Notes

[1] In fact, Pixaxe's original name was "JSONForms".

[2] The name "Esel" was inspired by "ECMAScript Expression Language".

[3] "FLWR" from "for", "let", "where", and "return", the four basic operations of the expression.

[4] See `http://www.javascriptmvc.org`.

[5] `http://java.sun.com/products/servlet/`

[6] http://www.eclipse.org

[7] http://mail.google.com

[8] `http://www.me.com`

[9] Kouprey was named after the Cambodian ox, by analogy to other parser generators such as `yacc` and `bison`.

[10] See `http://www.oberon.ch/`.

Featherweight Firefox

Formalizing the Core of a Web Browser

Aaron Bohannon
University of Pennsylvania

Benjamin C. Pierce
University of Pennsylvania

Abstract

We offer a formal specification of the core functionality of a web browser in the form of a small-step operational semantics. The specification accurately models the asynchronous nature of web browsers and covers the basic aspects of windows, DOM trees, cookies, HTTP requests and responses, user input, and a minimal scripting language with first-class functions, dynamic evaluation, and AJAX requests. No security enforcement mechanisms are included—instead, the model is intended to serve as a basis for formalizing and experimenting with different security policies and mechanisms. We survey the most interesting design choices and discuss how our model relates to real web browsers.

1 Introduction

Web browsers are complex: they coordinate network communication via many different protocols; they parse dozens of languages and file formats with flexible error recovery mechanisms; they render documents graphically, both on screen and in print, using an intricate system of rules and constraints; they interpret JavaScript programs; they manage users' browsing history, bookmarks, passwords, and RSS feeds; they execute cryptographic algorithms and protocols—and these are just the obvious tasks! This complexity makes it very challenging to design effective security mechanisms for web browsers: there are too many features to consider at once, and it is easy for the some of the fundamental security concerns to be obscured by more superficial problems.

"Web browser security" is actually an ambiguous term that can refer to a diverse range of issues. First, the browser codebase needs to be free from bugs such as buffer overflows that could lead to a complete compromise of a running browser. Next, the browser must correctly implement the protocols related to cryptography and the public key infrastructure for securing HTTPS communication and verifying digital signatures. To many users, "browser security" may be about how the interface helps them avoid phishing attacks and accidental installation of malware. And finally, a particularly interesting aspect of browser security is the restrictions that must be placed on web page scripts for the purpose of ensuring information security when managing documents and scripts from different sources. This last security problem is the one we are interested in addressing; we will refer to it as *web script security* to distinguish it from these other aspects of web browser security.

In current practice, web script security revolves around the idea of a *same-origin policy*. An origin is based on the domain name of a web page, and restrictions are placed on the interactions that can take place between pages with different origins. However, it is actually somewhat difficult to characterize the "policy" that these restrictions are intended to enforce; indeed, it is not even very clear which particular restrictions are meant by the phrase. Browsers implement origin-based restrictions on accessing the state of other windows, on navigating other windows to new locations, on accessing windows by their string name, on making AJAX requests, and on accessing and mutating cookies, but the idea of an "origin" is defined and used differently in each of these cases. For instance, whereas the ability to access other windows' state depends on the value of the `document.domain` property, the ability to make an AJAX request does not, and whereas some of these restrictions check that that the effective origins are identical, the ability for one window to navigate another depends on a complex set of rules about the windows' relationship and does not actually require that their origins be the same. Nonetheless, we will continue to use "same-origin policy" to refer to all of these restrictions.

Beyond these inconsistencies, the same-origin policy has other deficiencies. Whereas direct cross-domain interaction via window data structures and AJAX messages is disallowed, indirect cross-domain interaction by in-

sertion of new document nodes that trigger HTTP requests is unrestricted. Furthermore, a script's origin is based solely on the URL of the page where it is placed rather than the URL from which the script was retrieved, and the origin of the data handled by scripts is never taken into account. Another problem is that the mechanisms that allow interaction between related domains (involving setting the `document.domain` property) are inflexible and potentially dangerous [8]. Moreover, many corner cases (especially those involving function closures) are missing in written specifications and are implemented differently in different browsers (see the work on Subspace [4] for an example). Finally, the same-origin policy offers no help in addressing the pervasive security problem of script-injection attacks, which are known as XSS ("cross-site scripting") attacks.

In comparison, what would an ideal proposal for web script security look like? There should be a clearly articulated policy that applies consistently to the web browser's operation as a whole. It should be flexible enough to guarantee rigid security boundaries similar to the same-origin policy when desired and also to permit cross-domain communication for mashup-like behavior if necessary. It should offer some account of XSS and CSRF[1] attacks. It should be written at a level of detail that makes it feasible to implement. Finally, it should be specified in such a way that claims about it can be verified through rigorous proof or model checking.

This last goal makes the task especially challenging: a formal specification of a security mechanism necessitates a formal specification of web browser behavior at some level of detail, and browsers are large systems with many interdependent components. However, the complexity of web browsers is exactly the reason it is important to write a rigorous specification! There has been confusion about the implementation of the relatively simple same-origin policy; if one is interested in security mechanisms that are even more sophisticated, it will be essential to pin down their behavior in a precise and formal manner.

How does one develop a formal model of a web browser? In theory, one could take a particular browser implementation to *be* the model. But this is too much detail: browser implementations are far too large to reason about formally. If we build a model with the right level of abstraction, we can study something of a more tractable size, and more importantly, we can focus on the fundamental aspects of web script security, putting aside the more superficial concerns until after the deeper logical issues have been ironed out.

Using the right level of abstraction cuts down on the size of a model, but there are still too many potentially security-relevant browser features for us to consider them all at once. We must begin with a core set of features. Once polices and mechanisms for these features are well understood, we can hope to extend the security mechanisms to cover additional features. What we have put in our initial browser model has been guided by our interest in the following aspects of the browser setting:

- The system works with data and code from many different principals, and security boundaries may need to take into account the author of the data, the author of the code, and the principal that caused the code to be run.
- Scripts can be downloaded at any time and are always run in an environment that is shared with other scripts. Therefore, dynamic evaluation of code is fundamental to the web browser scripting model.
- There are multiple top-level environments in which scripts can be run, and these environments change each time the user visits a new page.
- Scripts contain first-class functions, which are, among other things, used for event handling.
- The system is event-driven and events can be interleaved in complex and possibly unexpected ways.
- Network messages can be generated in a variety of ways by scripts and by the user; any message sent could be relevant to information security.

The primary contribution of the work presented in this paper is a formal specification of the core data structures and operations of a web browser. We use a small-step, reactive semantics that faithfully models the event-driven nature of web browsers. In cases where the appropriate browser behavior was not obvious, we referred to the HTML5 specification [3] and ran tests on browser implementations. At the end of Section 5, we discuss some of the differences between the HTML5 specification, browser implementations, and our model. Concretely our model is presented in the form of an OCaml program[2] (about 2,500 LOC, including many comments). The program is written in a form that corresponds very closely to logical rules of inference, which are commonly used in giving formal semantics to programming languages, and we can view it as a document to be read by researchers interested in formal browser semantics. Using OCaml as a concrete notation yields many benefits. First, the type system of the language gives a sanity check on the definitions. Second, the specification is executable, which facilitates testing and experimentation. (To this end, we have also written parsers for the relevant subsets of HTML, JavaScript, and HTTP.) The presentation here is structured around the main type declarations.

[1] A CSRF ("cross-site request forgery") attack occurs when a script on a page from one site generates an HTTP request resulting in an unwanted server-side transaction on another site.

[2] It can be accessed here: `http://www.cis.upenn.edu/~bohannon/browser-model/`

2 Key Concepts

We follow a top-down approach in describing our model. In this section, we survey the browser features that we included and mention those we omitted. The remainder of the paper describes the specification itself, beginning from the outermost layer—the parts of a browser that are directly observable via its user interface and network connections—then moving on to internals.

Browsers display information in *windows*, many of which may be open at one time. A window can be *navigated* to a URL, which is the address of a document. The default URL for a newly opened window is "`about:blank`", which refers to a blank page. URLs that begin with "`http:`" refer to remote resources. When a window is navigated to such a URL, the browser sends an HTTP network request. The browser runs asynchronously and can handle other events while it is waiting for a network response.

For each URL that the user visits, browsers keep a mapping of key-value pairs known as *cookies*. These can be set each time an HTTP response is received, and the ones corresponding to a particular URL will be sent with each HTTP request for that URL. Including cookies allows us to model CSRF attacks; moreover, the unauthorized acquisition of session cookies is one of the most critical information security threats posed by XSS.

Browsers are designed to display HTML documents. For our purposes, it suffices to consider a subset of HTML with just a few basic features. It should allow tags to be nested in one another in some fashion. It should have some kind of text elements and link elements. It should also have text input elements and button elements that can be equipped with handlers, which are triggered by user interaction. Finally, it should have tags for including scripts in documents, both as source code that is written in-line in the document and as references to remote script files.

A document is transformed into a mutable document *node tree* and placed in a window. We use the term *page* to refer to a document node tree plus its related metadata, such the URL from which it was retrieved.[3] A window will display a sequence of pages over time as it is navigated to new locations.

Obviously, to study web script security, one must also have a model of the scripting language. The JavaScript language, which is used in current browsers, has many peculiarities and many of its peculiarities do have security implications, given the particular manner in which browser structures are represented in the JavaScript interpreter. However, instead of addressing problems that are specific to JavaScript at this time, we are more interested in understanding the security problems that are fundamental to *any* browser scripting language that can manipulate the browser in the ways that JavaScript can. This is a prerequisite step for understanding how to improve the security of a specific language such as JavaScript. Two of JavaScript's features are of interest to us because they seem especially useful in a web scripting environment and pose special challenges in the context of security: first-class functions and dynamic evaluation of code.

Although we want to restrict ourselves to a very simple core scripting language, we do want to capture a complete interface for scripting the browser components in our model. Scripts should be able to refer to their own window and refer to other windows by their name (which is just a string). They should be able to open, close, and navigate windows, as well as read and update their name. The name of a window is important to have in our model; it is relevant to security, both as a means of acquiring access to windows and as a means to transfer information. Scripts should be able to access a page's node tree and a page's global scripting environment through a reference to the page's window. We do not model the standard DOM in all of its details, but the scripting language should have enough power to construct and manipulate document node trees in arbitrary ways. Scripts should also be able to read and write the browser's cookies. Finally, scripts should be able to initiate AJAX-style HTTP requests and handle the responses.

Although our model encompasses many of the core features of a web browser, and certainly enough to make the issue of web script security challenging and interesting, there are many more features that we must leave out of our model for now, with the hope of considering them in the future. We do not consider relative URLs and fragment identifiers, although these would be fairly trivial. We do not consider virtual URLs schemes such as "`javascript:`" and "`data:`". We do not consider object-oriented features in the scripting language, nor do we consider object-style access to windows via the keyword `this` (instead we include the expression `self`, a synonym for `window`). We do not consider timer events, page load events, or low-level input events related to keystrokes and mouse movements. We do not consider any sort of frames, which offer a slightly different relationship between pages than having only separate top-level windows. We do not consider HTML forms nor browser history. We do not consider accessing files on the local machine. We do not consider any HTTP return codes other than "200 OK"; in particular, we do not consider HTTP redirects. We make no distinction between `http` and `https` URLs. We do not consider the interaction of web pages and scripts with plug-ins such as Flash or Java. Finally, we do not consider the password manager mechanism, nor any other browser extensions that

[3] We reserve the term *document* to refer to a static tree-like data structure, such as an HTML document.

might interact with web pages. We view all of these features as interesting and relevant to security, but we leave them out for now in order to minimize the complexity of our model.

Since our goal is to provide a foundation for researching new security policies and mechanisms, we designed our execution model as a "blank slate," without any security restrictions built in to it by default.

3 Reactive Systems

To begin with, we need to consider what the high-level "shape" of a browser's specification should be. A browser does not simply take an input, run for a while, produce an output, and then halt. Rather, it is an event-driven system: it consumes an input when one is available, which triggers some computation that may produce one or more outputs; when (and if) it is finishes running in response to one input, it will wait for the next input event. From the perspective of the scripts it runs, a browser should appear as if it has just a single event loop [3]. This sort of event-driven behavior can be captured by a fairly simple sort of state machine:

3.1 Definition: A *reactive system* is a tuple

$$(\mathit{ConsumerState}, \mathit{ProducerState}, \mathit{Input}, \mathit{Output}, \rightarrow)$$

where $\rightarrow$ is a labeled transition system whose states are $\mathit{State} = \mathit{ConsumerState} \cup \mathit{ProducerState}$ and whose labels are $\mathit{Act} = \mathit{Input} \cup \mathit{Output}$, subject to the following constraints:

- for all $C \in \mathit{ConsumerState}$, if $C \xrightarrow{a} Q$, then $a \in \mathit{Input}$ and $Q \in \mathit{ProducerState}$,
- for all $P \in \mathit{ProducerState}$, if $P \xrightarrow{a} Q$, then $a \in \mathit{Output}$,
- for all $C \in \mathit{ConsumerState}$ and $i \in \mathit{Input}$, there exists a $P \in \mathit{ProducerState}$ such that $C \xrightarrow{i} P$, and
- for all $P \in \mathit{ProducerState}$, there exists an $o \in \mathit{Output}$ and $Q \in \mathit{State}$ such that $P \xrightarrow{o} Q$.

Reactive systems never get "stuck," although they may get into a loop within the set of producer states and never accept another input. When a reactive system is in a consumer state, it must have some manner to handle whatever input comes along next, although it could choose to drop it and not do anything interesting in response. Reactive systems have a natural interpretation as a function from (possibly infinite) streams of inputs to (possibly infinite) streams of outputs. Bohannon, et al. [1] discuss these systems in more detail.

Given this template for a reactive system, what remains is to instantiate the system parameters—$\mathit{ConsumerState}$, $\mathit{ProducerState}$, Input, and Output—with the particular data structures that are relevant for web browsers. This will be done over the next two sections. What the system *does* is described by the transition relation of the reactive system, which we only have space to informally summarize in this paper as we describe the data structures; its full definition is given in the accompanying OCaml code.

4 Browser Inputs and Outputs

In this section, we present the structure of all of the data that goes into and comes out of a browser in our model. We describe these data structures using abstract syntax; for the purposes of an information security analysis, even user input and GUI updates can be modeled syntactically, as we will discuss in this section. We begin by looking at the abstract syntax for URLs, which is shown in Figure 1. We consider two kinds of URLs: a URL for a blank page and a basic `http` URL (without any port number or fragment identifier). The $\mathit{request\_uri}$ contains the path and query string of the URL.

The abstract syntax that we use for documents (see Figure 2) corresponds to an extremely simplified version of HTML. For comparison, the literal, concrete HTML syntax that would correspond to the abstract syntax is given on the right. (In order to literally translate our document constructs into well-formed HTML fragments, some must be mapped to HTML expressions with more than one tag, as shown.) Each construct in the abstract syntax has an optional string $\mathit{elt\_id}$, which should be thought of as the value of the `id` attribute of an HTML tag, if present. Unlike HTML, no further tags are allowed in the text of paragraph, link, or button elements. We append the suffix $\mathit{\_list}$ to syntactic categories as a way to indicate a sequence of zero of more items, such as the use of $\mathit{doc\_list}$ in the `div` construct.

The syntax of scripts is given in Figure 3. As with the browser as a whole, the goal of designing the scripting language is to capture the fundamental issues that make web script security interesting. We settled upon a very simple JavaScript-like core language—a dynamically typed language with mutable variables, a few basic data types, first-class functions, and dynamic evaluation—along with a set of constructs for manipulating the data structures of our browser model that is fairly complete in comparison with the standard "browser object model" (BOM) used in JavaScript. We didn't attempt to capture all of the idiosyncrasies of the BOM interface (the method for accessing cookies in JavaScript, for instance, is pointlessly absurd); however, we did aim to make the correspondence with the standard BOM very straightforward. We also did not try to aggressively eliminate redundant constructs since some constructs will likely need to be added, removed, and altered during the investigation of any particular security enforcement mechanism.

$$\begin{array}{rcl} \mathit{url} & ::= & \texttt{blank\_url} \\ & \mid & \texttt{http\_url}(\mathit{domain}, \mathit{request\_uri}) \end{array}$$

Figure 1: URL syntax.

$$\mathit{elt\_id} \quad ::= \quad \cdot \mid \mathit{string}$$

doc	$::=$	$\texttt{para}(\mathit{elt\_id}, \mathit{string})$	e.g., *string*
	$\mid$	$\texttt{link}(\mathit{elt\_id}, \mathit{url}, \mathit{string})$	e.g., *string*
	$\mid$	$\texttt{text}(\mathit{elt\_id}, \mathit{string})$	e.g., <input type="text" value="*string*"></input>
	$\mid$	$\texttt{button}(\mathit{elt\_id}, \mathit{string})$	e.g., <button>*string*</button>
	$\mid$	$\texttt{inline\_script}(\mathit{elt\_id}, \mathit{expr})$	e.g., <script>*expr*</script>
	$\mid$	$\texttt{remote\_script}(\mathit{elt\_id}, \mathit{url})$	e.g., <script src="*url*"></script>
	$\mid$	$\texttt{div}(\mathit{elt\_id}, \mathit{doc\_list})$	e.g., *doc_list*

Figure 2: Document syntax.

As in JavaScript, there are a variety of basic types (we do not consider classes, objects, or other user-defined types). The types `Null`, `Bool`, `Int`, `String`, and their corresponding literal expressions are straightforward. There is a type for URLs in the language, with corresponding literal URL expressions. In JavaScript, URLs are handled purely as strings, being parsed as needed; however, by having a special type for URLs in this language, we can avoid putting a string parsing algorithm in the main part of our semantics (such parsing could be done by a library function with a semantics specified completely separately). A value of type `Type` is a concrete representation of another value's type.

For compactness and uniformity, our language has no distinction between expressions and statements. Expressions can be sequenced with a semicolon (;), and the combined expression yields the result of the second expression when it is finished executing. If the first expression in a sequence results in an error, the second expression is not run. If the second expression results in an error, the effects of the first expression are still registered in the browser state. There is no need for a `return` construct when there is no distinction between statements and expressions. Conditionals and loops are standard. A `while` loop always evaluates to `null` when it terminates, as do other constructs with no sensible result. The $\mathit{primitive\_functions}$ include any pure functions whose semantics is independent of the web browser environment, such as functions to check the type of a value, to perform arithmetic operations, or to convert data types to and from strings. The specification of these operations is completely orthogonal to the problem of a browser specification and is therefore not included in our work.

Like JavaScript, this language has first-class, anonymous functions with local variables. However, in this language, functions always have exactly one parameter, and their local variables must be declared at the beginning of the function. Unary function application is denoted by $\mathit{expr}(\mathit{expr})$. Values of type `Code` represent a syntax tree for an expression. Any expression can be treated as a syntax tree by enclosing it with the `code` construct. `Code` values can be evaluated with the `eval` construct. (The expression will be evaluated in the environment that lexically encloses the `eval` expression.) As with URLs, having a special type for syntax trees differs from JavaScript (which passes strings to `eval`) but allows us to define a rigorous semantics for dynamic evaluation while putting aside the complex but uninteresting process of turning strings into expressions. Variables are dereferenced in the nearest (lexically) enclosing scope in which they are defined. If not defined elsewhere, a variable is dereferenced in the global scope of the script, which is the environment associated with the page in which the script was loaded. It is a runtime error to dereference a variable that is not defined in some enclosing scope. Variable assignment updates the variable (or function parameter) in the nearest enclosing lexical scope in which it is defined; if it is not defined in any enclosing scope, then the assignment will create a binding in the script's global scope.

The scripting operations that are specifically relevant to the web browsing environment are shown in Figure 4. The construct $\texttt{get\_cookie}(u, k)$ evaluates to the cookie value associated with the string key k for the URL u or evaluates to `null` if no such cookie is defined. The construct $\texttt{set\_cookie}(u, k, v)$ sets the cookie with key

$$
\begin{array}{rcl}
\mathit{type} & ::= & \texttt{Null} \mid \texttt{Bool} \mid \texttt{Int} \mid \texttt{String} \mid \texttt{Url} \mid \texttt{Type} \\
& \mid & \texttt{Function} \mid \texttt{Code} \mid \texttt{Window} \mid \texttt{Node} \\
\mathit{expr} & ::= & \texttt{null} \mid \mathit{bool} \mid \mathit{int} \mid \mathit{string} \mid \mathit{url} \mid \mathit{type} \\
& \mid & \mathit{expr};\ \mathit{expr} \\
& \mid & \texttt{if}(\mathit{expr})\ \{\mathit{expr}\}\ \texttt{else}\ \{\mathit{expr}\} \\
& \mid & \texttt{while}(\mathit{expr})\ \{\mathit{expr}\} \\
& \mid & \mathit{primitive\_functions} \\
& \mid & \mathit{function}(x)\ \{\texttt{var}\ x_1;\ \ldots;\ \texttt{var}\ x_n;\ \mathit{expr}\} \\
& \mid & \mathit{expr}(\mathit{expr}) \\
& \mid & \texttt{code}(\mathit{expr}) \\
& \mid & \texttt{eval}(\mathit{expr}) \\
& \mid & x \\
& \mid & x = \mathit{expr} \\
& \mid & \mathit{browser\_operations}
\end{array}
$$

Figure 3: Script syntax.

$$
\begin{array}{rcl}
\mathit{browser\_operations} & ::= & \texttt{get\_cookie}(\mathit{expr}, \mathit{expr}) \\
& \mid & \texttt{set\_cookie}(\mathit{expr}, \mathit{expr}, \mathit{expr}) \\
& \mid & \texttt{xhr}(\mathit{expr}, \mathit{expr}, \mathit{expr}) \\
& \mid & \mathit{window\_operations} \\
& \mid & \mathit{node\_operations}
\end{array}
$$

Figure 4: Browser operation syntax.

k for the URL u to the string value v. The construct $\texttt{xhr}(u, m, h)$ initiates an AJAX request to the URL u, sending the string m as the message body. The network response will be expected to contain a script, which will be passed in the form of a `Code` value as an argument to the handler function h (which may then run it with `eval` or dissect it in some other way using some of the primitive operations that we leave unspecified).

The scripting operations relating to windows and pages are shown in Figure 5. In JavaScript, there is a distinction between the `Window` object and `Document` object. As implemented in JavaScript, this distinction does not add any real expressive power, so we do not attempt to emulate it in our language. A window can hold only one page at a time; so a reference to a window is implicitly a reference to a page, although which page that is may change over time (if, say, the user navigates the window elsewhere). The important thing for one to understand is which window-accessible data can vary when the window's page changes. These include the window's location URL, the root node of the document tree in the window, and the global environment of the window.

The keyword `self` refers to the window that holds (or held) the page in which the script was loaded; the construct $\texttt{opener}(w)$ refers to the window from which w was opened. The construct $\texttt{named\_win}(s)$ evaluates to the window whose name is the string s or evaluates to `null` if there is no such window. A new window can be opened to a particular URL u using $\texttt{open}(u)$. A window with a name n can be opened to a URL u using $\texttt{open\_named}(u, n)$; however, if a window with that name is already open, then that window will be navigated to the URL u, and no new window will be opened. Both constructs for opening windows evaluate to the new (or navigated) window. A window can be closed with $\texttt{close}(w)$, and $\texttt{closed}(w)$ yields a `Bool` indicating whether or not w is a valid reference to an open window.

The URL of the document currently in a window w can be read using $\texttt{get\_location}(w)$, and a window can be navigated to a new URL u using $\texttt{navigate}(w, u)$. (In JavaScript, this is done by assigning a value to the `location` property of a `Window` object.) A window name can be read or updated using `get_name` or `set_name`. The root node of the document node tree in a window w can be read using $\texttt{get\_root\_node}(w)$ and can be set to a new node node using $\texttt{set\_root\_node}(w, \mathit{node})$. Every page has an associated environment that serves as the global environment for any scripts loaded into the page. For a page in a window w, the variables in the global environment can be read or updated using the constructs $w.x$ and $w.x = \mathit{expr}$.

$$
\begin{array}{rcl}
\mathit{window\_operations} & ::= & \texttt{self} \mid \texttt{opener}(\mathit{expr}) \mid \texttt{named\_win}(\mathit{expr}) \\
& \mid & \texttt{open}(\mathit{expr}) \mid \texttt{open\_named}(\mathit{expr}, \mathit{expr}) \\
& \mid & \texttt{close}(\mathit{expr}) \mid \texttt{closed}(\mathit{expr}) \\
& \mid & \texttt{get\_location}(\mathit{expr}) \mid \texttt{navigate}(\mathit{expr}, \mathit{expr}) \\
& \mid & \texttt{get\_name}(\mathit{expr}) \mid \texttt{set\_name}(\mathit{expr}, \mathit{expr}) \\
& \mid & \texttt{get\_root\_node}(\mathit{expr}) \mid \texttt{set\_root\_node}(\mathit{expr}, \mathit{expr}) \\
& \mid & \mathit{expr}.x \mid \mathit{expr}.x = \mathit{expr}
\end{array}
$$

Figure 5: Window operation syntax.

$$
\begin{array}{rcl}
\mathit{node\_operations} & ::= & \texttt{remove}(\mathit{expr}) \\
& \mid & \texttt{insert}(\mathit{expr}, \mathit{expr}, \mathit{expr}) \\
& \mid & \texttt{remove\_handlers}(\mathit{expr}) \\
& \mid & \texttt{add\_handler}(\mathit{expr}, \mathit{expr}) \\
& \mid & \ldots
\end{array}
$$

Figure 6: Window operation syntax.

The most interesting among the constructs related to document nodes are shown in Figure 6. Nodes have a graph structure derived from the fact that some nodes (`div` nodes in particular) can have children; all manipulation of nodes must maintain the invariant that the node graph is a forest. The construct $\texttt{remove}(\mathit{node})$ updates the node store so that node is removed from being the child of any `div` node or is removed from being the root node of any page, if either is applicable; the relationship of node and its children are unaffected by this, which means that neither node nor any of its descendants will be visible in any page after the operation. The construct $\texttt{insert}(\mathit{parent}, \mathit{child}, n)$ first removes the node child (just as if $\texttt{remove}(\mathit{child})$ had been evaluated) and then inserts child as the nth child of parent. It is an error (and nothing is mutated) if parent is not a `div` node, if parent has fewer than n children, or if parent is a descendant of child in a node tree. The construct $\texttt{remove\_handlers}(\mathit{node})$ removes all handlers from a text input or button node, and the construct $\texttt{add\_handler}(\mathit{node}, h)$ adds the function h as a handler for a text input or button node. When the value of a text input box is changed or when a button is pushed, each of the node's handler functions will be applied to the argument node and run.

We represent user interactions using the syntactic messages shown in Figure 7. We assume that a user refers to a window using a natural number representing it's relative age among the open windows, the oldest window being 0. In this way, we need not model an actual two-dimensional graphical display. There are several basic actions a user can take. The operations `load_in_new_window` and `load_in_window` represent cases where the user directs the browser to some URL—perhaps by typing in a URL, by selecting a bookmark, or by clicking on a link in a page.[4] The constructs `link_to_new_window` and `link_to_named_window` are used to represent cases where the user follows links that open in different windows due to the `target` attribute of the link. They both must include the window where the link was found as their first piece of data, since that window will be the deemed the "opener" of the new window. The user can also close windows, of course. The constructs `input_text` and `click_button` both take a window and a natural number, representing the position of the input box or button in the page. When these input events are received, they will trigger the handlers of the appropriate element.

There are four basic outputs that are visible to the user in this model. First, a new window can be opened. There is no data associated with the `window_opened` event because new windows are always created with an empty page having the URL `about:blank`. When pages are loaded or updated, there may be visible changes to the document rendered on the screen. There is a data type $\mathit{rendered\_doc}$ (not defined here) that captures the structure of the document node tree that is visible from the user interface. In the `page_loaded` and `page_updated` events, the entire document in the window is sent to the user, regardless of how much of it was actually changed. User output events play a rather non-obvious role in the model. For the purposes of an information security analysis, we assume that all browser inputs, including those from the network, are visible to the user; moreover, the

[4] If we included the HTTP `Referer` header in our model, we would need to distinguish between clicking a link and typing in a URL.

$$
\begin{array}{rcl}
\mathit{user\_window} & ::= & \texttt{window}(\mathit{nat}) \\
\mathit{user\_input} & ::= & \texttt{load\_in\_new\_window}(\mathit{url}) \\
& | & \texttt{load\_in\_window}(\mathit{user\_window}, \mathit{url}) \\
& | & \texttt{link\_to\_new\_window}(\mathit{user\_window}, \mathit{url}) \\
& | & \texttt{link\_to\_named\_window}(\mathit{user\_window}, \mathit{string}, \mathit{url}) \\
& | & \texttt{close\_window}(\mathit{user\_window}) \\
& | & \texttt{input\_text}(\mathit{user\_window}, \mathit{nat}, \mathit{string}) \\
& | & \texttt{click\_button}(\mathit{user\_window}, \mathit{nat}) \\
\mathit{user\_output} & ::= & \texttt{window\_opened} \\
& | & \texttt{window\_closed}(\mathit{user\_window}) \\
& | & \texttt{page\_loaded}(\mathit{user\_window}, \mathit{url}, \mathit{rendered\_doc}) \\
& | & \texttt{page\_updated}(\mathit{user\_window}, \mathit{rendered\_doc})
\end{array}
$$

Figure 7: User I/O syntax.

browser operates deterministically, modulo the ordering of the inputs. So in a theoretical sense, the user always knows the complete browser state regardless of what user interface outputs are generated in the model. We also assume that no other principal can see these outputs, so they do not have much significance in developing confidentiality policies. However, they do become significant if we wish to develop and reason about integrity policies.

The network-related input and output events are shown in Figure 8. The `request` construct is a simplified version of an HTTP request. Our model does not distinguish between GET and POST requests since the difference has little impact on web script security. The *cookies* are key-value mappings, and an extra string can be sent as the body of the request, as would be done in a POST request. An output on the network consists of a *domain* and a *request*. We abstract away from the DNS name resolution process. We model a network connection with the domain name to which the connection was made and a natural number to distinguish between multiple connections to the same domain. We take 0 to be the oldest connection for which a response has not yet been received, 1 to be the next-oldest, and so on. As stated earlier, we do not model any HTTP responses other than "200 OK". We assume that the body of a response consists of either a well-formed document or a well-formed script.

5 Internal Browser Structures

We have seen the parts of the specification that describe how a browser interacts with its environment. Now we need to consider what internal bookkeeping is needed for a browser to operate. There are choices to make here. For example, one could choose to have document nodes maintain references just to their parents, just to their children, or to both. Our goal was to find a clear and succinct way to describe how a browser operates; usually (but not always) this seemed easiet to achieve by avoiding maintaining redundant information in the state. So, in the example of document nodes, we chose to have document nodes maintain references only to their children.

A browser's basic state is a tuple of six components as shown in Figure 9: it has stores for windows, pages, document nodes, activation records, and cookies, and a list of the open network connections. The syntactic elements with the suffix *_ref* are unique atomic identifiers that are generated freshly when new items are put into a store during the browser's execution; then they are used to refer to the associated data in the various stores. A *cookie_id* consists of a domain name, a path, and a string value representing the key.

The basic browser data structures are defined in Figure 10. The data for a window includes its name (an optional string) and a reference to its page. There is a `simple_window` structure for windows that were opened directly by the user, and an `opened_window` structure for windows that have a reference to the window from which they were opened. A window is considered to be open (and therefore visible to the user) iff a reference to the window appears in the window store of the browser. Pages contain their location, a reference to their root document node, a reference to the activation record containing their global environment, and a queue of scripts and of markers for not-yet-received scripts that will need to be executed in the future. If the browser is waiting idly for its next input, the script queue of every page in the browser will either be empty or will contain a marker for a not-yet-received script at its head; any scripts at the front of a queue that are in hand will be executed before the browser halts.

The structure of document nodes mirrors the structure of documents, except for a couple of details. First, there is an extra piece of data for `text` and `button` nodes, a

$$
\begin{array}{rcl}
\mathit{request} & ::= & \texttt{request}(\mathit{request\_uri}, \mathit{cookies}, \mathit{string}) \\
\mathit{network\_output} & ::= & \texttt{send}(\mathit{domain}, \mathit{request}) \\
\mathit{response} & ::= & \texttt{doc\_response}(\mathit{cookie\_updates}, \mathit{doc}) \\
& | & \texttt{script\_response}(\mathit{cookie\_updates}, \mathit{expr}) \\
\mathit{network\_connection} & ::= & \texttt{connection}(\mathit{domain}, \mathit{nat}) \\
\mathit{network\_input} & ::= & \texttt{receive}(\mathit{network\_connection}, \mathit{response})
\end{array}
$$

Figure 8: Network I/O syntax.

$$
\begin{array}{rcl}
\mathit{window\_store} & ::= & [(\mathit{window\_ref}_1, \mathit{window}_1), \ldots, (\mathit{window\_ref}_n, \mathit{window}_n)] \\
\mathit{page\_store} & ::= & [(\mathit{page\_ref}_1, \mathit{page}_1), \ldots, (\mathit{page\_ref}_n, \mathit{page}_n)] \\
\mathit{node\_store} & ::= & [(\mathit{node\_ref}_1, \mathit{node}_1), \ldots, (\mathit{node\_ref}_n, \mathit{node}_n)] \\
\mathit{act\_rcd\_store} & ::= & [(\mathit{act\_rcd\_ref}_1, \mathit{act\_rcd}_1), \ldots, (\mathit{act\_rcd\_ref}_n, \mathit{act\_rcd}_n)] \\
\mathit{cookie\_store} & ::= & [(\mathit{cookie\_id}_1, \mathit{string}_1), \ldots, (\mathit{cookie\_id}_n, \mathit{string}_n)] \\
\mathit{browser} & ::= & \texttt{browser}(\mathit{window\_store}, \mathit{page\_store}, \mathit{node\_store}, \mathit{act\_rcd\_store}, \mathit{cookie\_store}, \\
& & \quad \mathit{open\_connection\_list})
\end{array}
$$

Figure 9: Browser state.

value_list, which is their set of handlers. Second, both kinds of script nodes need a flag to know whether they have already been queued for execution on some page. A script node must only be queued for execution once, even if it is later moved. Finally, the `div` node keeps a list of *references* to child nodes instead of a list of the literal child data elements as is done in the *doc* data structure.

The type *act_rcd* is used for activation records, an important part of the script evaluation. They contain a *bindings* data structure, which is a mapping of variable names to fully evaluated expressions. Moreover, activation records for local scopes must keep a reference to their parent record for proper variable access and update.

The last component of a browser's data is its list of open network connections. A record must be kept of the domain to which each request was made, the resource that was requested, and enough data to properly handle the resource when it arrives. A `doc_connection` is expecting a response with a document, which will be used to build a new page in a window; so, a reference to that window must be recorded. A `script_connection` is expecting a response that pertains to a script node in some document node tree; in order to find the appropriate queue item for the script on the appropriate page, a reference to that script node must be kept. Given our other implementation choices, the *page_ref* data here is actually redundant information, but tracking it here simplifies our operational specification a bit. An `xhr_connection` is expecting a response that should be given to a specific handler function. The handler function, a *value*, is kept as part of the connection data structure, but the handler must run with some definition for the window `self`. Here we have an choice of recording this information using a *window_ref* or a *page_ref*. Either one would work, as long as we can ensure that an AJAX response is never run on a page other than the one for which it was intended. This is slightly easier to do by keeping track of the *page_ref*.

A few more data structures are needed to manage the small-step evaluation of script expressions in browsers (see Figure 11). The internal language of expressions (*iexpr*) is a slight extension of the external scripting language. It has a set of values that includes closures, a term that represents an error, and a term that represents an expression in a particular scope, which will arise when closures are applied during evaluation. Closures and scoped expressions refer to a static context that includes both an activation record and a window reference that will determine the evaluation of the keyword `self`.

A browser in a running state requires a queue of *tasks* in addition to the basic browser state. The task queue keeps track of the script expressions that the browser must evaluate before it can accept another input. A task comprises an internal expression paired with the window with respect to which it should be evaluated. A task could equivalently be associated with a page instead of a window. The association between windows and pages cannot change between the time a task is enqueued and when it is executed (this association can only change immediately after receiving a network response). Since this information is needed primarily for evaluating the `self` keyword in the expression, we choose the window instead of the page.

$$
\begin{array}{rcl}
\mathit{window} & ::= & \texttt{simple\_window}(\mathit{window\_name}, \mathit{page\_ref}) \\
& | & \texttt{opened\_window}(\mathit{window\_name}, \mathit{window\_ref}, \mathit{page\_ref}) \\
\mathit{queued\_expr} & ::= & \texttt{known\_expr}(\mathit{expr}) \\
& | & \texttt{unknown\_expr}(\mathit{node\_ref}) \\
\mathit{page} & ::= & \texttt{page}(\mathit{url}, \mathit{node\_ref}, \mathit{act\_rcd\_ref}, \mathit{queued\_expr\_list}) \\
\mathit{node} & ::= & \texttt{para}(\mathit{elt\_id}, \mathit{string}) \\
& | & \texttt{link}(\mathit{elt\_id}, \mathit{url}, \mathit{string}) \\
& | & \texttt{text}(\mathit{elt\_id}, \mathit{string}, \mathit{value\_list}) \\
& | & \texttt{button}(\mathit{elt\_id}, \mathit{string}, \mathit{value\_list}) \\
& | & \texttt{inline\_script}(\mathit{elt\_id}, \mathit{expr}, \mathit{bool}) \\
& | & \texttt{remote\_script}(\mathit{elt\_id}, \mathit{url}, \mathit{bool}) \\
& | & \texttt{div}(\mathit{elt\_id}, \mathit{node\_ref\_list}) \\
\mathit{act\_rcd} & ::= & \texttt{global}(\mathit{bindings}) \\
& | & \texttt{local}(\mathit{bindings}, \mathit{act\_rcd\_ref}) \\
\mathit{open\_connection} & ::= & \texttt{doc\_connection}(\mathit{domain}, \mathit{req\_uri}, \mathit{window\_ref}) \\
& | & \texttt{script\_connection}(\mathit{domain}, \mathit{req\_uri}, \mathit{page\_ref}, \mathit{node\_ref}) \\
& | & \texttt{xhr\_connection}(\mathit{domain}, \mathit{req\_uri}, \mathit{page\_ref}, \mathit{value})
\end{array}
$$

Figure 10: Browser data structures.

$$
\begin{array}{rcl}
\mathit{iexpr\_context} & ::= & \texttt{context}(\mathit{window\_ref}, \mathit{act\_rcd\_ref}) \\
\mathit{value} & ::= & \texttt{closure}(\mathit{iexpr\_context}, x, x_1, \ldots, x_n, \mathit{iexpr}) \\
& | & \texttt{win}(\mathit{win\_ref}) \\
& | & \texttt{node}(\mathit{node\_ref}) \\
& | & \ldots \\
\mathit{iexpr} & ::= & \mathit{value} \\
& | & \texttt{error} \\
& | & \texttt{scoped}(\mathit{iexpr\_context}, \mathit{iexpr}) \\
& | & \ldots \\
\mathit{task} & ::= & \texttt{task}(\mathit{window\_ref}, \mathit{iexpr}) \\
\mathit{running\_browser} & ::= & \texttt{running}(\mathit{task\_list}, \mathit{browser})
\end{array}
$$

Figure 11: Internal expressions and running browser states.

Each kind of input will initialize the task queue in a different manner. When the user interacts with a button or text box, the task queue will be initialized with one task for each of the input control's handlers. When a network response to an AJAX request is received, the queue will be initialized with the corresponding `xhr_connection` handler. When a document is received, a page will be created along with its script queue, and the browser's task queue will be initialized with the items from the front of the page script queue that are ready for execution. Similarly, when a network response is matched with a `script_connection`, it will cause the appropriate page script queue to be updated, and then the ready items from the front of that queue will be transferred to the browser's task queue. If script nodes are inserted into a page as the browser executes tasks at the head of the task queue, additional tasks corresponding to those scripts will be enqueued at the back of the browser task queue, provided they are not blocked by an `unknown_expr` in their page's script queue.

Given the definitions thus far, we can now state explicitly how a browser forms a reactive system. The instantiations of *ConsumerState*, *ProducerState*, *Input*, and *Output* are given in Figure 12. There is nothing very interesting here, except for a couple of technicalities. According to the definition of a reactive system, there must always be exactly one output when stepping from a *ProducerState*; however, given our internal data structures, in some cases a single small-step of execution in our model may produce multiple outputs. Such a system can be trivially reduced to a formal reactive system by adding an output buffer to the producer state structure: if more than one output happens to be produced in a step of the original machine, the derived machine

$$
\begin{array}{rcl}
\mathit{ConsumerState} & ::= & \mathit{browser} \\
\mathit{ProducerState} & ::= & \mathit{running\_browser}' \\
\mathit{Input} & ::= & \mathit{user\_input} \mid \mathit{network\_input} \\
\mathit{Output} & ::= & \mathit{user\_output} \mid \mathit{network\_output} \mid \bullet
\end{array}
$$

Figure 12: Internal expressions and running browser states.

will remain in a producer state and release one output at a time over multiple steps. Thus $\mathit{running\_browser}'$ would be identical to $\mathit{running\_browser}$ but with an output buffer. On the other hand, since *ProducerState* is technically required to produce an output on every step, we use the symbol $\bullet$ to represent a trivial, "silent" output, when there would otherwise be no output.[5]

6 Formalization Challenges

One particularly tricky issue is deciding exactly when scripts will get executed. The HTML5 specification recommends that there be three different queues of scripts that will get executed at different times, depending on whether script tags have a `defer` attribute, an `async` attribute, or neither. (This is motivated in part by the existence of JavaScript's `document.write()` method; we intentionally omitted such functionality from our specification because it introduces a huge complexity overhead while being a technique that should be avoided in modern web programming.) Moreover, HTML5 prescribes that remotely retrieved scripts should be executed after parsing finishes, whereas inline scripts should be executed "immediately." On the other hand, some browsers such as Firefox appear to nonetheless execute all scripts on a page in the order they appear, regardless of whether the scripts are inline or remote. We wanted to avoid the additional complexity associated with executing scripts midway through parsing; so we chose a behavior that was close to how Firefox behaves when all script tags have the `async` attribute set. This choice could be tweaked in the future, but the precise order of script execution seems unlikely to affect the design of security mechanisms.

Another tricky case is what to do when a user navigates away from a page but that page had some code that is still runnable, say, as a button handler in another page. When such a closure runs, it may attempt to access its global environment or to use the `self` construct. In the browsers we tested, if the window's new page is from the same origin as the old page, there are no difficulties in running such a closure. However, if the new page is from a different origin, some browsers raise errors that are presumably security-related. For instance, in recent versions of Firefox, simply evaluating the expression `self` in such a closure generates an error, even if no access of its properties or methods is attempted. This is an interesting corner case of the same-origin policy for which the correct restrictions are unclear and browsers vary widely in their behavior. However, since we are not modeling security restrictions, our specification raises no errors in such cases. A related situation occurs when a closure from a page remains executable after the page's window has been closed. It is not clear that security plays any role in this case; nonetheless, there are a variety of behaviors that can be observed in browser implementations. Some browsers will raise an error if the closure tries to evaluate the expression `self` and others will not; some browsers will purge the variables in the closure's global environment and others will not. We have chosen to allow `self` to be evaluated to a reference to the closed window and to leave the closure's environment intact. Our model does perform one sort of garbage collection, though: a page will be removed from the page store when the page's window is closed or the page is replaced by a navigation operation. Primarily, this is done to ensure that scripts received in network responses will not be executed on a page that is no longer visible. However, the page's associated nodes and activation records are left in their respective stores after the page is removed since these may be referenced elsewhere in the browser.

When a user navigates away from a page, there may also be outstanding AJAX requests that have not received a response. In this case, many browsers will call the state change handlers for the requests before leaving the page, as if the responses had come back with a dummy HTTP response code such as 0. Similarly, when a user closes a window, there may be outstanding AJAX requests for the page in the window. Some browsers call the handlers before closing the window, but others do not. Since we are not modeling HTTP error responses, we simply chose not to trigger the handlers in any of these situations.

7 Related Work

Our work was motivated by an investigation of the paper "Information-Flow-Based Access Control for Web Browsers" by Yoshihama, et al. [7]. Their work offers a reasonable outline of a browser formalization; our work

[5] Reactive systems that force each step to have an output are simpler to reason about and no less useful for studying information security [1].

is an attempt to fill in the concrete details of a more realistic browser model. Our model uses heap structures with references, which are important for understanding how information flows through a browser. Furthermore, our model adds first-class functions, activation records, and multiple global environments that are associated with page structures. In addition, we have broken the browser's behavior down into a small-step, event-driven semantics with a complete characterization of the possible inputs and outputs of the system.

There have been other efforts to formalize subsystems of a browser. Maffeis, et al. [5] have written a formal specification for the JavaScript language. Their work is in the same spirit as ours but nearly orthogonal in terms of its content, in that they consider all of the details of the JavaScript language itself without formalizing the language's integration with the browser. Gardner, et al. [2] have developed a formal specification for a literal subset of DOM Level 1 [6]. Their work is again in the same spirit as ours but strives for greater accuracy in a narrower domain. We did not attempt to implement a literal subset of the DOM specification (as they did) but instead specialized our node operations for the particular types of nodes in our document model. Moreover, they developed a full-blown compositional Hoare-logic semantics, whereas for our purposes a simpler operational specification is sufficient.

The Browser Security Handbook [8], published on Google's web site, provides thorough documentation of many different security-related behaviors that can be observed in different browsers. Our methodology has been first to try to understand how browsers would work *without* any security restrictions, thus offering a platform on which many different experiments with security enforcement mechanisms can be performed. Moreover, for our baseline, restriction-free semantics, we are not especially interested in documenting every observable browser behavior, but rather in having a single semantics that embodies a reasonable compromise between the different existing behaviors and specifications. The key point for us is to ensure that our formalization is close enough to real-world browsers so that experimenting with new security mechanisms on top of it will offer meaningful insight into how these designs would fare in reality.

The HTML5 [3] specification goes into a great deal of detail about browser behavior. In fact, it seems to be the only written account of many aspects of browser behavior. It covers many more features than we can put in our formal model at this time; however, as a specification written in English, it is necessarily somewhat imprecise, and it does not cover all of the corner cases involving integration with a scripting language, such as cases involving function closures. In contrast, our work is intended to be a platform for carrying out rigorous proofs.

8 Future Work

Our next goal is to use our formal model to experiment with concrete confidentiality and integrity policies. To start, this means designing a system of security levels and associating them with the input and output events of the model [1]. In concept, this is a straightforward process; however, there are different ways to do it, giving rise to different policies when combined with the requirement of reactive noninterference. In comparison with the constraints of the same-origin policy, noninterference-based polices will be more strict about cross-domain interactions over the network but more lax about cross-domain interactions that are confined within the browser. Policy design, which should include an account of declassification and server-guided policy customization, is an interesting topic, but even a basic policy has a wide range of enforcement techniques that we may wish to study—anything from globally removing operations from the scripting language to implementing a fine-grained tracking of information flow. Our preliminary investigations suggest that proving enforcement mechanisms sound with respect to policies will be challenging, given the size of our browser model, but that it is nonetheless feasible.

References

[1] BOHANNON, A., PIERCE, B. C., SJÖBERG, V., WEIRICH, S., AND ZDANCEWIC, S. Reactive noninterference. In *Proceedings of the ACM Conference on Computer and Communications Security* (2009), ACM Press.

[2] GARDNER, P. A., SMITH, G. D., WHEELHOUSE, M. J., AND ZARFATY, U. D. Local Hoare reasoning about DOM. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (2008), ACM Press.

[3] HYATT, D., AND HICKSON, I. HTML 5. Tech. rep., W3C, 2010. `http://www.w3.org/TR/html5/`.

[4] JACKSON, C., AND WANG, H. J. Subspace: Secure cross-domain communication for web mashups. In *Proceedings of the International Conference on World Wide Web* (2007).

[5] MAFFEIS, S., MITCHELL, J., AND TALY, A. An operational semantics for JavaScript. In *Proceedings of the Asian Symposium on Programming Languages and Systems* (2008). See also: Dep. of Computing, Imperial College London, Technical Report DTR08-13, 2008.

[6] NICOL, G., WOOD, L., SUTOR, R., APPARAO, V., ISAACS, S., HORS, A. L., WILSON, C., CHAMPION, M., ROBIE, J., BYRNE, S., AND JACOBS, I. Document object model (DOM) level 1 specification (second edition). W3C working draft, W3C, Sept. 2000. `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/`.

[7] YOSHIHAMA, S., TATEISHI, T., TABUCHI, N., AND MATSUMOTO, T. Information-flow based access control for web browsers. *IEICE Transactions on Information and Systems E92.D*, 5 (2009), 836–850.

[8] ZALEWSKI, M. Browser security handbook, Dec. 2009. `http://code.google.com/p/browsersec/wiki/Main`.

DBTaint: Cross-Application Information Flow Tracking via Databases *

Benjamin Davis
University of California, Davis

Hao Chen
University of California, Davis

Abstract

Information flow tracking has been an effective approach for identifying malicious input and detecting software vulnerabilities. However, most current schemes can only track data within a single application. This single-application approach means that the program must consider data from other programs as either all tainted or all untainted, inevitably causing false positives or false negatives. These schemes are insufficient for most Web services because these services include multiple applications, such as a Web application and a database application. Although system-wide information flow tracking is available, these approaches are expensive and overkill for tracking data between Web applications and databases because they fail to take advantage of database semantics.

We have designed DBTaint, which provides information flow tracking in databases to enable cross-application information flow tracking. In DBTaint, we extend database datatypes to maintain and propagate taint bits on each value. We integrate Web application and database taint tracking engines by modifying the database interface, providing cross-application information flow tracking transparently to the Web application. We present two prototype implementations for Perl and Java Web services, and evaluate their effectiveness on two real-world Web applications, an enterprise-grade application written in Perl and a robust forum application written in Java. By taking advantage of the semantics of database operations, DBTaint has low overhead: our unoptimized prototype incurs less than 10-15% overhead in our benchmarks.

*This research is partially supported by NSF CNS award 0644450 and by an AFOSR MURI award.

1 Introduction

Information flow tracking has been very successful in protecting software from malicious input. The program identifies the sources of untrusted input, tracks the flow of such input, and prevents this input from being used in security sensitive contexts, such as the return addresses of function calls or the parameters of risky system calls [16, 19]. Currently there are two types of information flow tracking mechanisms: application-wide and system-wide. The former tracks information flow within the same application [16, 19], while the latter tracks information flow in the entire operating system [17].

As computation moves to the Web, Web services have become highly attractive targets to attackers. In fact, attacks involving malicious input to Web applications, such as Cross-site Scripting (XSS) attacks, are among top software vulnerabilities [4]. Information flow tracking is a logical approach for preventing these attacks by tracking malicious input [10, 14]. However, the application and effectiveness of information flow tracking is limited by the only two types of current mechanisms: single-application and system-wide tracking.

A typical Web service consists of multiple applications, such as a Web application, which implements business logic and generates Web pages, and a database, which stores user and application data. In multi-application settings like Web services, single-application information flow tracking is inadequate, as it would force Web applications to decide between treating all the results of database queries as tainted or treating them as untainted. This would inevitably result in false positive or false negative when the database contains both tainted and untainted data.

To enable cross-application information flow tracking, one might resort to system-wide information flow tracking systems. However, there are several problems with these systems. They track more information than needed for protecting Web services, which comes at an unnec-

essary performance cost. Protecting against XSS attacks requires tracking information flow only in the Web application, database, and the information flow between them, rather than in every operation in the entire system. Also, these system-wide mechanisms fail to take advantage of application semantics. Without the high-level semantics, these systems cannot properly perform taint propagation throughout complex database operations.

We introduce DBTaint, a system that provides information flow tracking across databases for enabling cross-application information flow tracking. DBTaint extends the database to associate each piece of data with a taint tag, propagates these tags during database operations, and integrates this system with existing single-application taint tracking systems. By providing this integration via SQL rewriting at the database client-server interface, DBTaint is completely transparent to Web applications. This allows developers to use DBTaint with existing, real-world legacy applications without modifying any Web application code.

DBTaint can provide more accurate information flow tracking than single-application taint tracking systems. Services using DBTaint will have fewer false positives than systems that consider all values from external applications (like databases) as tainted. Similarly, services using DBTaint will have fewer false negatives than systems that consider all values from the database as untainted. Furthermore, since DBTaint tracks taint propagation inside the database, it takes advantage of database semantics to track taint propagation accurately. Besides providing a more accurate taint tracking system, DBTaint can also be used to detect potential vulnerabilities in applications. If user input is untainted during sanitization in a Web application, inspecting the taint values of database columns may reveal subtle security vulnerabilities. Our insight is that if a column in the database contains both tainted and untainted data, it may signal incomplete sanitization in the database client, e.g., when user input is santized only on a subset of program paths. Such observation should alert the programmers to audit the sanitization functions in the program carefully.

We make the following contributions:

- We design and implement a system that tracks information flow across databases. This allows cross-application information flow tracking to protect Web applications from malicious user input.

- We improve on single-application taint tracking systems by reducing false positives/false negatives, and improve on system-wide taint tracking systems by tracking only the taint propagation that matters to the target application and by taking advantage of database semantics to improve tracking accuracy.

- Our system is also useful for analyzing certain behavior of database client applications, such as identifying potential incomplete sanitization in Web applications.

- We design a flexible system to integrate single-application taint tracking systems with the PostgreSQL database. This system allows legacy applications to take advantage of our system transparently.

- We implemented two prototypes of DBTaint that work with real-world Web applications written in Perl and Java. These prototypes, although unoptimized, have low performance overhead.

2 Design

DBTaint is a system that allows developers to track information flow throughout an entire Web service consisting of Web applications and database servers. DBTaint provides information flow tracking in databases and integration with single-application information flow mechanisms. The system is completely transparent to the Web applications, which do not need to be modified in any way to take advantage of DBTaint.

We assume that the developer is benign, and has the ability to replace the database interface and database datatypes used in the Web service with the modified (DBTaint) versions. We also assume that the single-application taint tracking engine(s) used for each individual Web application appropriately taints input from unsafe sources (e.g. user input).

DBTaint propagates the taint information throughout the multi-application system, but does not attempt to actively prevent the program from operating unsafely. Rather, by propagating and maintaining taint values for each piece of data in the system, we provide developers with the information needed to perform the sink checking and handling appropriate for their setting. Although DBTaint was motivated by Web services, our prototypes provide cross-application information flow tracking to any multi-application setting where applications communicate via databases. For brevity, we will refer to these applications as Web applications onward.

2.1 Taint Model

2.1.1 Soundness

DBTaint helps improve the security of Web services by tracking the trustworthiness of all the data values used by the service. DBTaint marks each value as either *untainted* or *tainted*. DBTaint marks a value as *untainted* only if it can determine that the value is trusted. Therefore, when DBTaint marks a value as *tainted*, it could be because DBTaint has determined that the value is indeed untrusted or because DBTaint cannot determine whether the value is trusted.

We say two values are in the same *context* if they belong to the same column or their respective columns are compared in a `JOIN` operation. With DBTaint, the database marks an output value as untainted only if there was an occasion when the same value in the same context was marked as untainted when it entered the database, or if the output value is derived from untainted values only.

The above property implies that:

- If a context contains two identical values but one is tainted and the other is untainted, DBTaint may return this value either as tainted or untainted. Our prototype chooses to always return this value as untainted to improve the accuracy of taint tracking in the Web application.

- DBTaint will never return a value as untainted if this value has never entered the context as untainted and is not derived from only untainted values.

2.1.2 Scope of Taint

DBTaint can work with any taint tracking mechanism inside the Web application. In the simplest, and most common, case, the taint value of a data when it exits the database is the same as its taint value when it enters the database. Consider an SQL query for the `MAX` of two values in the database, where one value is 3 and tainted, and the other value is 5 and untainted. DBTaint returns the value 5 to the database client (the Web application) untainted, because the value 5 was untainted when it entered the database. Similarly, data in the result of a `JOIN` query carry their taint values in the database, regardless of the taint values of other data (e.g., data in the common columns during `JOIN`) that may have affected the selection of the data in the result.

2.1.3 Backward Compatibility

We adopt the principle of "backwards compatibility", similar to the one described by Chin and Wagner [7], and design DBTaint such that a DBTaint-unaware application should behave exactly the same regardless of whether it is retrofitted with DBTaint. Under this principle, when DBTaint compares two data, it ignores their taint values. Besides ensuring backward compatibility, this principle also allows DBTaint to set the taint value of certain output data more accurately. For example, consider computing the `MAX` of a tainted value 2 and an untainted value 2. Either value is an acceptable result for this query, but we choose to return the untainted value. Similarly, in a `SELECT DISTINCT` query, we again prefer to return untainted versions of equal values when available.

2.2 Information Flow Tracking in the Database Server

Because current mainstream database systems do not natively provide a mechanism for storing taint information associated with each piece of data, DBTaint provides a mechanism for storing this information without losing precision of the original data. Furthermore, DBTaint propagates the taint information for database values during database operations.

2.2.1 Storing Taint Data

DBTaint provides information flow tracking capabilities in databases at the SQL level, requiring no changes to the underlying database server implementation. Capabilities added to the database at the SQL-level are likely simpler and more portable than those made by modifying the underlying implementation of a particular database server. Furthermore, by utilizing SQL to maintain and operate on the taint information, we avoid the need to provide new mechanisms to insert, retrieve and operate on taint information in the database server.

Many databases support *composite* data types, where each data cell may store a tuple of data. We used this feature to store taint information alongside associated data values, allowing DBTaint to use the well-understood SQL API for interacting with these taint values. The additional functionality (like auxiliary functions) DBTaint needs to add to the database can be done at the SQL-level as well (e.g. via `CREATE FUNCTION`).

Compared to alternative implementation approaches (e.g. storing taint bits in mirrored tables or additional columns), we hypothesized that composite types would be the simplest. It allowed our SQL-rewriting operations to rewrite each original query into exactly one new query, avoiding the need for extra queries to maintain mirrored tables. Also, using composite types allowed us to build taint propagation logic into the database type system rather than into each rewritten query.

2.2.2 Operating on Taint Data

In addition to creating the database composite types, DBTaint provides some database functions that make operating on these types simpler. We provide the database functions `getval()` and `gettaint()` to extract the data and taint values from a DBTaint tuple, respectively. These functions are used in the SQL rewriting phase, described in Section 2.4.1. DBTaint also provides neces-

sary database functions for these composite types (e.g. equality and comparison operators). Finally, DBTaint provides functions to propagate taint values in aggregate functions (like `MIN` and `MAX`) and arithmetic operations (when one or more operand is tainted, the result is tainted).

2.3 Information Flow Tracking in the Database Client

DBTaint leverages existing single-application information flow tracking systems to manage taint information in the client, and integrates the single-application taint tracking system with the new database server functionality at the interface between the two applications. DBTaint works with any mechanism for taint tracking in the database client (the Web application). For instance, we have implemented a version of DBTaint for Perl that uses a modified version of Perl's taint mode. We also developed a prototype version of DBTaint that uses an efficient character-level taint tracking system for Java [7]. While the single-application taint engines propagate taint throughout the single application, DBTaint handles the propagation of this taint information across application boundaries when this data is used in a SQL query.

Many other single-application taint tracking systems exist, and DBTaint can be easily extended to integrate with these engines as well. For example, there also exist preliminary implementations of PHP with support for tainted variables.[1]

2.4 Database Client-Server Integration

Once we can track the information flow within a single application and within a database, DBTaint must provide a way to propagate the taint information between database client applications and the augmented database server. While we could perform this by modifying the Web application directly, this approach does not scale well, as the user would need to modify every new Web application individually. Furthermore, the amount of work required to make the changes would scale with the size and complexity of each Web application.

Instead, DBTaint integrates the information flow systems of the database client and database server at the interface between these two systems. For example, Perl programs generally use the DBI (DataBase Interface) module to access database servers, and Java applications often use JDBC (Java DataBase Connectivity) API. By adding our DBTaint functionality at these interfaces, we can integrate the taint tracking systems of multiple applications completely transparently to the Web application.

DBTaint requires three changes to the database interface:

- Rewrite all queries to add additional placeholders for taint values associated with the data values, and to add appropriate taint values where appropriate.
- When the application supplies the parameter values, determine and pass the corresponding taint values.
- When retrieving the composite tuples from the database, collapse them into appropriately tainted data values then return them to the Web application.

2.4.1 Rewriting SQL Queries

In DBTaint, the database server tables are composed of composite values that contain both the data and the taint value associated with that piece of data. However, since the Web applications that use these databases are not modified in any way, their data values and corresponding SQL queries do not include the necessary information to maintain the data taint values in the database. A key component of the DBTaint system is the way the SQL queries from the Web application are dynamically rewritten to propagate taint information between the Web application and the database server transparently to the database client.

DBTaint performs two main types of transformations on portions of SQL queries: *tupling* and *flattening*. These operations performed on the appropriate parts of a SQL query before passing it through to the database server.

Tupling is the process of taking a data value and converting it into a tuple that contains the original value and the associated taint value. For example, when a Web application sends an `INSERT` query that includes a data value to the database interface, DBTaint rewrites that portion of the query into a tuple containing the data value and the taint value of that data. If the Web application passes a parameterized query (with `?` placeholders for data values to be supplied later), DBTaint rewrites the query to include additional placeholders for the corresponding taint values.

Assume we specify a composite type using the PostgreSQL syntax: `ROW(x,y)` where `x` is the data value, and `y` is the corresponding taint value. If the Web application passes the following query to the database:

```
INSERT INTO posts (id, msg) VALUES
(1, ?)
```

then DBTaint rewrites this query to include the taint value of the `1` substring (e.g. 0 if untainted), and adds a place for the taint value of the message data to be supplied later.

```
INSERT INTO posts (id, msg) VALUES
(ROW(1,0), ROW(?,?))
```


Flattening is the process of taking a tuple value in the database and removing the associated taint value when it is unneeded. We have designed DBTaint such that using the system does not change the behavior of the Web application. So, sometimes it is necessary for DBTaint to extract only the data value for certain SQL operations in order to perform the appropriate operations. For example, if the Web application wishes to select rows where a specific column is equal to a hardcoded value, then we disregard the taint value during the selection process.

For example, when the Web application issues the request:

```
SELECT username FROM users WHERE
user_id = 0
```

Since in this case the taint value of the `user_id` field is unimportant, DBTaint extracts only the data value and the query becomes:

```
SELECT username FROM users WHERE
getval(user_id) = 0
```

2.4.2 Rebinding Parameterized Query Values

Applications often use parameterized queries for defense against SQL injection attacks, improved performance, and increased maintainability. Parameterized queries use placeholders for parameters that the Web application passes later. Often DBTaint must augment these queries by adding additional placeholders for the corresponding taint values. Unfortunately, this means that the index-based bindings the Web application uses may no longer be valid (e.g. binding a value to placeholder three may no longer be the third parameter in the rewritten query). Furthermore, because the Web application does not know about these new taint parameters, DBTaint must provide them to the database.

When a Web application attempts to bind a parameter to a particular position in a SQL query, DBTaint intercepts this request and computes the new, proper index for that data value. Then, DBTaint not only binds that data value, but the corresponding taint value, if appropriate. This allows the Web application to use parameterized queries with no knowledge of the underlying implementation of the composite types used by DBTaint.

2.4.3 Retrieving Database Values

The results of database queries are tuples of data and taint values. DBTaint extracts the data values from these tuples, then taints them as appropriate in the single-application taint tracking engine used by the Web application. This completes the propagation of taint values back into the Web application.

3 Implementation

We have developed two different prototype implementations of our DBTaint system (one for Perl and one for Java) to demonstrate the effectiveness of our approach. These prototypes are fully capable of working with real-world Web services that use the PostgreSQL database engine.

3.1 Database

Both DBTaint prototypes assume the use of the PostgreSQL database server. PostgreSQL is a popular, full-featured, enterprise-class object-relational database system. Users can create composite types from base types, add custom functions, and overload operators. We leverage these features to manage the taint information stored in our modified database tables.

3.1.1 Composite Types

DBTaint uses *composite types* to store data and taint information in PostgreSQL database tables. A composite type is a type with the structure of a user-defined record, and can be used in place of simple types in the database. Each composite type we create has two elements: the data value, and the associated taint value. We can maintain taint values at whatever granularity we like (e.g. per character) but to simplify our examples here we use a single taint bit. PostgreSQL uses the `ROW()` syntax to specify composite type values, so we express a tainted `INT4` as the `INT4t` composite value `ROW(37, 1)`.

3.1.2 Auxiliary Functions

During initialization, DBTaint determines all the native database types used by the Web application by inspecting the original database tables' schemas. DBTaint uses the `CREATE TYPE` command to create a new PostgreSQL composite type for each of these native types. Before these composite types can be used to create new composite versions of the original database tables, DBTaint creates a number of auxiliary functions to support these new types. These auxiliary functions are used to preserve the behavior expected by the database clients, and to simplify the SQL query processing DBTaint performs at the boundary of the database and other applications.

DBTaint generates the standard comparison operators to allow the database to sort and compare composite type values, and uses PostgreSQL's operator overloading capabilities to add taint-aware capabilities to the common operators (e.g. `>`, `<=`, `+`, `-`). DBTaint creates aggregate functions (using `CREATE AGGREGATE`) in the PostgreSQL database to create taint-aware versions of common aggregate functions, like `MIN` and

MAX. Additionally, DBTaint generates length() functions for composite types with string values, and arithmetic operators for numeric composite types. DBTaint overloads arithmetic operators to provide interoperability with base types, while propagating the taint information to the resulting composite values. For example, the operation "adding the integer 2 to the tuple ROW(5, 1)::INT4t" returns ROW(7, 1)::INT4t, which retains the taint bit from the original tuple. DBTaint also creates getval() and gettaint() functions for the composite types, which extracts just the value or taint bit for a particular piece of data. DBTaint sets the values and taint bits using normal SQL statements, and therefore requires no additional PostgreSQL functions to manipulate this data on the server.

3.1.3 Table Creation

After creating the necessary composite types and auxiliary functions in the database, DBTaint automatically replaces all of the simple types in the database tables with their associated composite type versions. Note that unless a Web application creates new tables during operation, this table creation phase only occurs during the initial installation and configuration stage. DBTaint adapts default values, column constraints, and other table properties as needed to match the new composite types. Default values are considered "untainted" in this process. For example, DBTaint converts a column with type INT4 and default value of 0 into a composite type column of type INT4t with default value ROW(0,0).

3.2 Perl Implementation

We developed an implementation of DBTaint for Web applications written in Perl that use the popular DBI module for accessing PostgreSQL databases. We use a modified version of Perl's "Taint Mode" to perform information flow tracking within the Web application.

3.2.1 Perl Taint Tracking

Our Perl implementation of DBTaint leverages the Perl taint mode to track information flow through the Web application. Perl's taint mode is an active mechanism that prevents some Perl operations from using untrusted user input unsafely. Perl taints user input, and halts when tainted values are used in certain unsafe situations (like as a parameter tosystem). We only needed a passive taint tracking engine for DBTaint, so we provide a modified Perl engine that does not halt in these situations, allowing us to use DBTaint with applications not normally compatible with Perl's taint mode.

3.2.2 Perl DBI

In our Perl implementation we add our DBTaint database interface functionality to the DBI (DataBase Interface) module. The Perl taint mode engine we use in our implementation has a limitation: it only tracks the taint bit of the entire variable as a whole. This means that, for example, a string is either completely tainted, or completely untainted. If a Web application assembles a query string by concatenating tainted and untainted data, by the time this string reaches the database interface it is impossible to determine what parts of the original query was tainted, and what was untainted. Note that this is not a problem if we use a more sophisticated taint tracking engine, such as the one used in our Java implementation below.

But, the Perl taint mode engine is still completely sufficient for DBTaint if the application uses prepared statements for its database queries. Prepared statements are SQL statements with placeholders for parameters to be supplied later. Prepared statements are used for performance reasons, to separate SQL logic from the data supplied, and to help prevent of SQL injection, and are quite common in modern Web applications. When these parameters are supplied later, the DBTaint system can inspect the taintedness of these data values at the database interface. In this way, we can properly propagate the taint information across the boundary of the Web application and the database application.

3.2.3 Other Modifications

The Web application we chose to use with DBTaint used prepared statements for all of its SQL queries, which made the DBI rewriting relatively simple. We slightly modified the Apache-Session Perl module to use prepared statements in a way that matched the rest of the Perl application to simplify our DBI rewriting logic. We also needed to modify the Encode Perl module to avoid user values being inadvertently untainted during conversion from UTF-8 encoding.

3.3 Java Implementation

We developed an implementation of DBTaint for Web applications written in Java that use the popular JDBC API to access PostgreSQL databases. We use a character-level taint tracking system for Java, which allows us to properly rewrite both prepared and non-prepared statements without losing any taint information from the Web application.

3.3.1 Java Taint Tracking

We use a character-level taint tracking engine for Java. [7] This taint engine marks all elements of incoming HTTP requests as tainted (e.g. form parameters, cookies, etc.), and propagates the taint bit throughout the Web application. When these values are passed to the database interface, DBTaint rewrites the queries appropriately to propagate the taint bits between the applications. We were able to use this taint tracking engine without any special configuration or modifications.

3.3.2 JDBC

In our Java implementation we add our DBTaint database functionality to the JDBC (Java DataBase Connectivity) classes. The Java information flow tracking engine we use tracks taint bits on each character of each String object. With this more precise information, we are no longer limited to only prepared statements, as we no longer depend on the parameters being separate from the query to determine if they are tainted or not. When the database interface receives a query with literals embedded in the query string, DBTaint inspects the taint values for the characters of that literal, and then adds the appropriate taint information when tupling the value.

For example, if the DBTaint system receives the following query in the JDBC interface:

```
INSERT INTO messages (msg) VALUES
('first post')
```

DBTaint will inspect the taint values of the substring consisting of first post. DBTaint will then rewrite the query with the appropriate taint values based on the taintedness of the substrings. For example, if the first post value was tainted, the query would be rewritten to:

```
INSERT INTO messages (msg) VALUES
(ROW('first post', 1))
```

We use Zql [5], a Java SQL parser, to parse the queries so they can be rewritten in DBTaint. Rewriting parameterized queries is performed using the same approach described above in the Perl implementation.

4 Evaluation

To demonstrate the effectiveness of DBTaint in real-world systems, we evaluate the performance of our taint-aware database operations, and run two popular Web services with DBTaint. We executed all benchmarks on a virtual machine running Cent OS 5 on a 2.6 Ghz Intel Core 2 Quad host with 4 GB of RAM. Our DBTaint implementations are based on PostgreSQL version 8.3.7, Perl version 5.10.0, and Java version 6 (1.6).

4.1 Database Operations

We first attempt to evaluate the overhead of the changes we make on the database server. By replacing all primitive data types in the database tables with composite types, the database server now has more information to manage, and is using custom composite types that have not been optimized as thoroughly as the native types. Table 1 contains the average run time of each of the following tasks. Between each run, the database was restarted and cached results were cleared to avoid measuring the effectiveness of the database caches.

Operation	native	DBTaint	overhead
INSERT row	0.5ms	0.6ms	20%
SELECT ALL	23ms	26ms	13%
SELECT WHERE	23ms	26ms	13%
EQUALS op	0.2ms	5ms	2400%
LESS THAN op	0.2ms	2.3ms	1050%
ADDITION op	0.2ms	2.4ms	1100%

Table 1: Database operations incur high overhead (later shown to not dramatically impact overall performance)

We note that the composite versions of many of these operations are a great deal slower than their native counterparts. We hypothesize that these discrepancies are due to the fact that our DBTaint database operations were defined to be simple and portable. The impact of these slower operations in a benchmark of actual Web application performance (Sections 4.5 and 4.6) indicates that the performance penalties paid for more portable implementations of DBTaint may be of little concern in many environments. Furthermore, it may be possible to greatly improve these results by implementing the datatypes and associated functions more efficiently (e.g. in C rather than SQL). We analyze the source of these results in more detail in Section 5.2.

4.2 Web Application: RT

We selected the enterprise-grade ticket tracking Web application named Request Tracker (RT) [1] to evaluate the effectiveness of DBTaint in a realistic environment. RT is not designed to be used with Perl taint mode, and was not created with DBTaint or any other information flow tracking system in mind. It has over 60,000 lines of code. It uses 21 different database tables to store information about tickets entered into the system, users of the system, transaction history of system modifications, access control, and more. Other than installing our composite datatypes and removing the inadvertent untainting in a Unicode conversion function, we ran RT with DBTaint without making any further changes to the Web application. We successfully tracked the flow of user input throughout the entire Web service: from the Web application, into the database, and back.

To demonstrate that DBTaint does not alter the behav-

ior of the Web application, we recorded a series of interactions with the Web application installed in an unmodified environment. We saved the database contents resulting from using the application in the unmodified environment for later reference. Then, after deploying the Web application and running it in the DBTaint system, we replayed these recorded Web actions in this environment. When we compared the values of the database tables in the DBTaint system with the values from the unmodified run, the only differences we observed were expected variances in values like timestamps. We observed that DBTaint allows the Web application to behave exactly as it would in a normal environment, transparently providing the information flow tracking capabilities to the entire Web service.

The RT application was not designed to function in taint mode, and halts immediately if taint mode is enabled in a normal Perl environment. We modified the Perl engine (see Section 3.2.1) to allow RT to function in the taint mode. While we did not use Perl taint mode to prevent active attacks, we analyzed the taint information in the database to learn about information flow through the Web application. Note that if the Web application were designed to function in taint mode, it would not need our modified Perl engine to work with DBTaint.

4.3 Analyzing Database Taint Values

After running RT in DBTaint, we could infer knowledge about the application by simply inspecting the taint values of the data in the database. By glancing through the taint values of the database records, we see that nearly all of the user input stored in the database is marked tainted. This implies that the Web application performs little input filtering, and relies on output filtering to escape characters to prevent XSS attacks. Upon inspection of the application code, we found that the application stores user input directly into the database, and escapes and replaces dangerous characters before displaying them in Web pages.

Columns with only untainted data Many of the database columns contained only untainted values. We observed that the values in these untainted columns were either provided or generated by the Web application, rather than originating from user input. For example, the "type" field of the "tickets" table was always untainted, because these values ranged across only a few hard-coded choices in the RT application. Another column contains a timestamp for internal logging of actions within the database. Because these timestamps were generated by the Web application and not specified by user input, they also appeared untainted in the DBTaint database tables.

Columns with only tainted data There were also columns composed entirely of tainted elements, such as the "subject" column of the "tickets" table. Columns with this property corresponded to mandatory form fields that the user completes while using the application. Because this Web application uses output filtering rather than input filtering, it passed user data directly from the Web application to the database without sanitization. Each element in the column contains untrusted data from user input, and we can immediately tell that the application is not performing input filtering on these values before storing them.

Columns with mixed tainted and untainted data While most table columns contained uniformly tainted or untainted data, there were several columns containing both tainted and untainted data. Upon further investigation, we observe that most of these are the columns for optional form fields. The Web application provides a default (untainted) value, but if the user provides a value of their own, it will show up as tainted in the database. For example, the "finalpriority" column of the "tickets" table has a default value of 0, which is untainted in the database if the user does not specify any value. However, if the user does provide a value it will show up as tainted in the database.

We investigated whether the application might not have sanitized any of these user-supplied values. We discovered that the application always sends data from these columns to the Web framework, which sanitizes the data before outputing them. Even though we did not find any sanitization bugs in RT, DBTaint helped us gain confidence in the completeness of sanitization in RT.

4.4 Enhancing Functionality

While DBTaint can be used to gain insight into the way that data flows throughout the Web application, it can also be used to enhance the functionality of the application without incurring additional security risk. RT escapes angle braces and other potentially dangerous characters from database values before using them to create a HTML page. While this can certainly help prevent cross-site scripting attacks, it also prevents the application from using these dangerous characters in its default values. For example, when a database column contains mixed tainted (user input) and untainted (application default) data, without DBTaint the application must sanitize all of them, unnecessarily restricting default values even though they are safe.

With the cross-application information flow provided by DBTaint, we were able to expand the functionality of the Web application without losing security. Since we can reliably track the flow of tainted data through the Web application and the database, we can avoid concerns of false positives and false negatives that come with single-application taint tracking schemes. Instead of escaping all data values before returning them to a Web visitor, we modified the application to only escape tainted values. Since user data remains tainted through the entire Web service, dangerous characters will be escaped in malicious input. On the other hand, trusted values (such as application defaults) will be untainted and can be safely included in HTML pages without undergoing this same escaping procedure.

Web Application	Overhead
Request Tracker (RT)	12.77%
JForum	8.49%

Table 2: Overall Web service overhead

4.5 Performance

We tested the impact of DBTaint on the performance of the RT Web application by timing the round trip time of making a request to the Web application, processing the request, and receiving the response. We performed 10 sets of 1,500 requests for the original (unmodified) version of RT, and of RT running with our DBTaint prototype. To simulate and environment of a Web application under load rather than just starting up, we recorded the time for the last 1,250 requests of each set. Recall that our Perl implementation is completely unoptimized, and each SQL query is reprocessed every time the Web application makes a database request. As Table 2 shows, we note that even with no attempt at optimization, we achieve less than 13% overhead in our prototype, which we believe provides a high upper bound of the performance impact of our approach.

4.6 Web Application: JForum

To evaluate the effectiveness of our Java implementation of DBTaint, we selected JForum version 2.1.8, which is (according to the documentation) a "powerful and robust discussion board system." [2] JForum includes more than 30,000 lines of code in 350 Java classes, and uses 59 database tables to maintain subforums, posts, messages, access control, and more. We deployed JForum to a Tomcat server with the character-based taint tracking engine described in Section 3.3.1.

We evaluated our Java implementation of DBTaint in a similar way to our Perl evaluation in Section 4.2. We recorded a series of Web events including logging in, posting to the forum, and viewing existing posts. We determined the performance overhead of our Java implementation on JForum to be less than 9% (Table 2).

Because our Java implementation uses a character-based taint tracking engine, the query rewriting phase is more sophisticated and complex than our Perl implementation. This is because it handles both parameterized and non-parameterized queries, and checks the taint values of each character in data strings. With this approach, we originally observed an overhead of close to 30%. However, in our Java implementation, we added very simple memoization to the parsing and rewriting of parameterized queries, which dropped the performance overhead to less than that of the Perl implementation, despite the increased complexity. In situations where Web services serve far more requests than there are distinct parameterized queries (which we believe is the common case), caching the results from the query rewriting phase is a simple way to improve performance in implementations with sophisticated character-level query rewriting analysis.

5 Discussion

DBTaint is an effective system for providing cross-application information flow tracking through databases. In this section we outline some of the benefits of DBTaint over other systems, reflections on our prototype implementations, and applications of DBTaint to interesting security problems.

5.1 Benefits

DBTaint has the following major benefits:

- End-to-end taint tracking throughout all applications in a Web service.
- Full support of the semantics of database operations. DBTaint tracks taint flow at the high database operational semantics level, rather than at the low instruction level.
- Efficiency. DBTaint only tracks the information flow within the database and between the database and its client applications, avoiding the overhead of the extra tracking that system-wide solutions perform. Our unoptimized prototypes add only a minor performance penalty to Web services.
- Only SQL-level changes to the database server, and no changes to the Web application. Our major implementation work is in modifying the database interface. We don't need to make any changes to the database client because DBTaint intercepts and automatically rewrites all SQL queries from the client as needed for our information flow tracking.

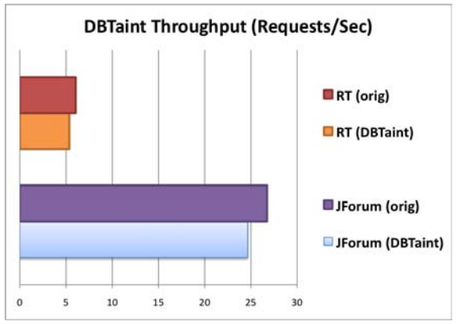


Figure 1: DBTaint Serial Throughput

5.2 Database Performance

The performance impact of the composite forms of database operations (summarized in Table 1) may appear to be surprising at first. We note that the composite versions of `INSERT` statements execute somewhat more slowly than the original queries. This is not particularly surprising, as the database client is inserting twice as many values in the composite version (a taint value for each data value). Similarly, basic `SELECT` statements are slightly slower than the original queries, likely for similar reasons – there is simply more data to work with in the composite version.

However, we note that the other operations (the equality operator, the less than operator, and the addition operator) are *much* slower than their native counterparts. We suspect that this is because native types have been highly optimized in the underlying database engine. In contrast, we added the composite version of these operators using high-level SQL functions. We hypothesize that while most Web application performance is not bound by mathematical operators in the database, substantial performance improvements could be made by implementing these composite types and their associated functionality in C rather than SQL. These optimized datatypes could be dropped in as replacements for our prototype SQL implementation if the extra performance was necessary. However, in testing our implementations with real-world Web applications (Sections 4.5 and 4.6) we observed that despite the large performance overhead of these operations, the overall performance of the Web application was not dramatically impacted. As shown in Figure 1, DBTaint has relatively little impact on the throughput of the Web application serving requests despite the overhead of database operations.

5.3 Applications of DBTaint

Persistence of taint information DBTaint allows the taint information stored in the database to remain persistent through multiple runs of the applications. This allows an application that uses the database to run many times without losing the taint information from the previous executions.

Comparison of different versions of application DBTaint can be used to compare two different versions of an application that use a database for storage. After refactoring some user input sanitization code, for example, programmers can run the old and new versions of a Web application under DBTaint and compare the resulting database tables. Variations in the taint patterns of the database columns may indicate a change in input sanitization policies.

Identification of incomplete input sanitization Incomplete input sanitization contribute to many security vulnerabilities. Common solutions for detecting incomplete input sanitization are static analysis and runtime testing. Static analysis techniques are often expensive and are prone to false positives and negatives. For runtime testing, the testers must understand the sanitization functions of the program to design malicious input to test the completeness of the sanitization functions. DBTaint provides an alternative mechanism to detect incomplete input sanitization at runtime and requires no understanding of the sanitization functions in the program.

Web applications typically sanitize untrusted input at two moments: (1) input filtering, which sanitizes an input as it enters the program; and (2) output filtering, which sanitizes untrusted data just before the program embeds the data into a generated Web page.

DBTaint has the ability to provide immediate insight into the taint properties of the data in an application without requiring the user to understand the application code. For Web applications that perform no input filtering,

- Columns that contain only tainted data are likely for storing mandatory user data.
- Columns that contain only untainted data are likely for storing data hardcoded from or generated by the applications themselves, rather than user input.
- Columns that contain mixed taint and untainted values are used for multiple purposes (e.g. data coming from different applications or code paths), or for optional data fields whose default value (set by the program) is untainted but user value is tainted.

In applications that perform input filtering, such column analysis can be even more useful.

- Columns where data are completely untainted indicate that all of the values are either sanitized user input or values produced by the application.
- Columns where data are completely tainted suggest user data that has not been properly sanitized, which may indicate a security vulnerability.
- Columns containing both tainted and untainted data may indicate that the input sanitization is incomplete, i.e., the program sanitizes input data on some paths but not on the other paths.

In the last two cases, DBTaint helps the auditor to reduce the search space for potential sanitization bugs.

5.4 Inadvertent Untainting

When using a taint engine, one must be careful to never inadvertently untaint tainted data. DBTaint does not untaint any tainted values (manually or automatically), but unfortunately the two single-application taint tracking engines we used for our Perl and Java prototypes did perform some inadvertent untainting. This is not the fault of DBTaint, but the problems in these other engines did make our evaluation more difficult.

Perl's taint mode is designed to automatically remove the taint bit on data when it is matched against a regular expression. Perl assumes that a programmer using a regular expression on a variable is validating the contents of the variable, so Perl automatically untaints the value. However, some Perl application and library code uses regular expression for simple string processing, rather than validation or sanitization, leading to inappropriate untainting. We discovered that an encoding/decoding UTF-8 conversion function was untainting all user input in the RT application before the data reached the DBTaint database interface. We addressed this problem by manually retainting the results of the function when the original string was tainted before the decoding.

We encountered similar difficulties using our character-based taint tracking engine for Java. The Java engine we used provides efficient taint tracking by extending String and other String-based classes to maintain taint data. However, because the primitive data types are not similarly extended, the engine cannot track taint bits for a String converted to a character array and back, for example. All taint bits are lost, inadvertently and incorrectly untainting the resulting String. Due to this limitation, some of the tainted values from the JForum Web application received via POST submissions became untainted before they reached the DBTaint database interface. As some user input values were inadvertently untainted, we were unable to perform a meaningful analysis of the taint values of each database column.

6 Related Work

The ability to access a Web service from anywhere means that it must be able to handle input from any source. Unchecked malicious input can lead to some of the top reported software vulnerabilities in Web applications [3]. The information flow tracking provided by DBTaint is like a coarse-grained version of where-provenance [6], allowing developers to identify unchecked user input though multiple applications in the Web service without requiring a whole-system solution.

Application-wide information flow tracking Splint [9] supports source code annotations that help a programmer identify the flow of tainted information within a program. TaintCheck [16] identifies vulnerabilities automatically by performing this analysis on a binary running within its own emulation environment. Xu et al. [19] leverage the source code of a program produce a version of that program that can efficiently track information flow and identify attacks. WebSSARI [12] targets web applications written in PHP specifically with static analysis to identify the information flow of unvalidated input and adds runtime sanitization routines to potentially vulnerable code using that input. Lam et al. [13] also targets web vulnerabilities with the automatic generation of static and dynamic analyses of a program from a description of an information flow pattern. Because most modern Web services include multiple applications, single-application information flow tracking systems result in false positives and/or negatives because they must assume database values are either tainted or untainted without complete runtime information. DBTaint avoids any need for manual annotation and automatically provides information flow propagation across Web service applications.

System-wide information flow tracking With architectural support, information flow tracking systems can trace untrusted I/O throughout the system [17, 8] at a fine memory address level granularity. These systems however, require substantial changes to the underlying hardware or must emulate the entire system with a performance penalty. Ho et al [11], provide the same system-wide tracking with the Xen virtual machine monitor and switch to a hardware emulator only when needed to mitigate the performance penalty. However, these approaches pay an unnecessary performance cost by tracking much more than necessary for most Web services. Other system-wide information flow tracking systems like HiStar [20] and Asbestos [18] are too coarsely grained to track taint values of individual values throughout the Web application and the database. These system-wide approaches also fail to take advantage of the se-

mantics of information flow during database operations. WASC [15] targets web applications and provides a dynamic checking compiler to identify flows and automatically instrument programs with checks. They add support for inter-process flow tracking through databases by maintaining external logs of all SQL transactions, operands and associated tags. This approach lacks DBTaint's ability to use taint values during internal database operations (e.g. preferring untainted values in equality operations for `SELECT DISTINCT` queries).

7 Conclusion

We have designed and implemented DBTaint, which provides information flow tracking in databases to enable cross-application information flow tracking. When database clients, such as Web applications, write into the database, DBTaint stores data together with their taint information. When the database clients retrieve data, DBTaint tags the data with proper taint information. Our implementation requires no modification to Web applications, and only SQL-level additions to the database. By interposing on the database interfaces between Web applications and databases, DBTaint is transparent to Web applications. We demonstrated how two Web applications, an enterprise-grade application written in Perl (RT) and a robust forum application written in Java (JForum), easily work with DBTaint. DBTaint not only can enable cross-application taint tracking but may also identify potential security vulnerabilities due to incomplete sanitization without the need to understand sanitization functions in the Web application. Because DBTaint takes advantage of the semantics of database operations, its overhead is low, and our unoptimized prototype implementations add only 10-15% overhead to the entire system.

8 Acknowledgment

We wish to thank David Wagner and Erika Chin for helpful discussions and for providing Java character-level taint tracking.

References

[1] Best Practical: Request Tracker. `http://bestpractical.com/rt/`.

[2] JForum. `http://jforum.net/`.

[3] OWASP top 10 2007. `http://www.owasp.org/index.php/Top_10_2007`.

[4] SANS: Top 20 internet security problems, threats and risks. `http://www.sans.org/top20/`.

[5] Zql: Java SQL Parser. `http://www.gibello.com/code/zql/`.

[6] BUNEMAN, P., KHANNA, S., AND TAN, W. C. Why and where: A characterization of data provenance. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory* (London, UK, 2001), Springer-Verlag, pp. 316–330.

[7] CHIN, E., AND WAGNER, D. Efficient character-level taint tracking for java. In *SWS '09: Proceedings of the 2009 ACM workshop on Secure web services* (New York, NY, USA, 2009), ACM, pp. 3–12.

[8] CRANDALL, J. R., WU, S. F., AND CHONG, F. T. Minos: Architectural support for protecting control data. *Transactions on Architecture and Code Optimization 3* (2006), 359–389.

[9] EVANS, D., AND LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE Software* (2002).

[10] GUNDY, M. V., AND CHEN, H. Noncespaces: using randomization to enforce information ow tracking and thwart cross-site scripting attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2009), pp. 1–18.

[11] HO, A., FETTERMAN, M., CLARK, C., WAR, A., AND HAND, S. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (2006), pp. 29–41.

[12] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D. T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web* (2004), pp. 40–51.

[13] LAM, M. S., LIVSHITS, B., AND WHALEY, J. Securing web applications with static and dynamic information flow tracking. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial Evaluation and Semantics-based Program Manipulation* (2008).

[14] NADJI, Y., SAXENA, P., AND SONG, D. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium* (2009).

[15] NANDA, S., LAM, L.-C., AND CHIUEH, T.-C. Dynamic multiprocess information flow tracking for web application security. *ACM/IFIP/USENIX 8th International Middleware Conference (Middleware'07)* (2007), 1–20.

[16] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2005).

[17] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2004), ACM, pp. 85–96.

[18] VANDEBOGART, S., EFSTATHOPOULOS, P., KOHLER, E., KROHN, M., FREY, C., ZIEGLER, D., KAASHOEK, F., MORRIS, R., AND MAZIÈRES, D. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst. 25*, 4 (2007), 11.

[19] XU, W., BHATKAR, S., AND SEKAR, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. *Usenix Security 2006* (2006).

[20] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 263–278.

Notes

[1] We have not yet overcome the bugs in the original PHP taint implementation, which crash the PHP interpreter.

xJS: Practical XSS Prevention for Web Application Development

Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, Evangelos P. Markatos
Institute of Computer Science,
Foundation for Research and Technology - Hellas
email: {elathan, vpappas, krithin, ligouras, markatos}@ics.forth.gr
Thomas Karagiannis
Microsoft Research,
Cambridge, United Kingdom
email: {thomas.karagiannis}@microsoft.com

Abstract

We present xJS, a practical framework for preventing code-injections in the web environment and thus assisting for the development of XSS-free web applications. xJS aims on being fast, developer-friendly and providing backwards compatibility.

We implement and evaluate our solution in three leading web browsers and in the Apache web server. We show that our framework can successfully prevent all 1,380 real-world attacks that were collected from a well-known XSS attack repository. Furthermore, our framework imposes negligible computational overhead in both the server and the client side, and has no negative side-effects in the overall user's browsing experience.

1 Introduction

Code-injection attacks through Cross-Site Scripting (XSS) in the web browser have observed a significant increase over the previous years. According to a September-2009 report published by the SANS Institute [34], *attacks against web applications constitute more than 60% of the total attack attempts observed on the Internet. Web application vulnerabilities such as SQL injection and Cross-Site Scripting flaws in open-source as well as custom-built applications account for more than 80% of the vulnerabilities being discovered.* XSS threats are not only targeted towards relatively simple, small-business web sites, but also towards infrastructures that are managed and operated by leading IT vendors [2]. Moreover, recently widely adopted technologies, such as AJAX [15], exacerbate potential XSS vulnerabilities by promoting richer and more complex client-side interfaces. This added complexity in the web browser environment provides additional opportunities for further exploitation of XSS vulnerabilities.

Several studies have proposed mechanisms and architectures based on policies, communicated from the web server to the web browser, to mitigate XSS attacks. The current state of the art includes XSS mitigation schemes proposing whitelisting of legitimate scripts [17], utilizing randomized XML namespaces for applying trust classes in the DOM [16], or detecting code injections by examining modifications to a web document's original DOM structure [26]. While we believe that the aforementioned techniques are promising and in the right direction, they have weaknesses and they fail in a number of cases. As we show in this paper, whitelisting fails to protect from attacks that are based on already whitelisted scripts, while DOM-based solutions fail to protect from attacks where the DOM tree is absent [7].

To account for these weaknesses, in this paper, we propose xJS, which is a practical and simple framework that isolates legitimate client-side code from any possible code injection. Our contributions are thus twofold: i) we describe, implement and evaluate xJS and ii) we outline limitations of previous methodologies and a number of attacks that defeat existing approaches.

Our framework could be seen as a *fast randomization* technique. Instruction Set Randomization (ISR) [20] has been proposed for defending against code injections in native code or in other environments, such as code executed by databases [9]. However, we believe that adapting ISR to deal with XSS attacks is not trivial. This is because *web client-side code* is produced by the server and is executed in the client; the server lacks all needed functionality to manipulate the produced code. For example, randomizing the JavaScript instruction set in the web server requires at least one full JavaScript parser running at the server. Thus, instead of blindly implementing ISR for JavaScript, our design introduces *Isolation Operators*, which transpose all produced code in a new isolated domain. In our case, this is the domain defined by the XOR operator.

We design xJS with two main properties in mind:

- **Backwards Compatibility.** We aim for a practical, developer-friendly solution for constructing secure web applications and we ensure that the scheme

```
1:<div>                          1:<div>
2:<img onload=''render();''>     2:<img onload=''AlCtV...''>
3:<script>                       3:<script>
4:alert(''Hello World'');        4:  vpSUlJTV2NHGwJyW/NHY...
5:<script>                       5:</script>
6:</div>                         6:</div>
```

Figure 1: Example of a web page that is generated by our framework.

provides backwards compatibility. xJS allows web servers to communicate to web browsers when the scheme is enabled or not. A web browser not supporting the framework may still render web applications, albeit without providing any of the security guarantees of xJS.

- **Low Computation Overhead**. Our design avoids the additional overhead of applying ISR in both web server and client, which would significantly increase the computational overheads. This is because the web code would be parsed twice (one in the server during serving and one in the client during execution). Instead, the isolation operator introduced in xJS applies the XOR function to the whole source corpus of all legitimate client-side code. Thus, the randomization process is fast, since XOR exists as a CPU instruction in all modern hardware platforms, and does not depend on any particular instruction set.

We implement and evaluate our solution in three leading web browsers namely FireFox, WebKit[1] and Chromium, and in the Apache web server.

Our evaluation shows that xJS can successfully prevent *all* 1,380 attacks of a well-known repository [14], imposes at the same time negligible computational overhead in the server and in the client side. Finally, our modifications appear to have no negative side-effects in the user web browsing experience. To examine user-perceived performance, we examine the behavior of xJS-enabled browsers through a leading JavaScript benchmark suite [3], which produces the same performance results in both the xJS-enabled and the original web browsers.

2 The xJS Framework

The fundamental concepts of our framework are *Isolation Operators* and *Action Based Policies* in the browser environment. We review each of these concepts in this section and, finally, we provide information about our implementation prototypes.

[1]WebKit is not a web browser itself, it is more like an application framework that provides a foundation upon which to build a web browser. We evaluate our modifications on WebKit using the Safari web browser.

xJS is a framework that can address XSS attacks carried out through JavaScript. However, our basic concept can be also applied to other client-side technologies, such as Adobe Flash. The basic properties of the proposed framework can be summarized in the following points.

- xJS prevents JavaScript code injections that are based on third party code or on code that is already used by the trusted web site.
- xJS prevents execution of trusted code during an event that is not scheduled for execution. Our framework guarantees that *only* the web site's code will be executed and *only* as the site's logic defines it.
- xJS allows for multiple trust-levels depending on desired policies. Thus, through xJS, parts of a web page may require elevated trust levels or further user authentication to be executed.
- xJS in principle prevents attacks that are based on injected data and misuse of the JavaScript `eval()` function. We discuss `eval()` semantics in detail in Sections 4 and 5.

Isolation Operators

xJS is based on Instruction Set Randomization (ISR), which has been applied to native code [20] and to SQL [9]. The basic concept behind ISR is to randomize the instruction set in such a way so that a code injection is not able to *speak the language of the environment* [21] and thus is not able to execute. In xJS, inspired by ISR, we introduce the concept of Isolation Operators (IO). An IO essentially transposes a source corpus to a new isolated domain. In order to de-isolate the source from the isolated domain a unique key is needed. This way, the whole source corpus, and not just the instruction set, is randomized.

Based on the above discussion, the basic operation of xJS is the following. We apply an IO such as the XOR function to effectively randomize and thus isolate all JavaScript source of a web page. The isolation is achieved since all code has been transposed to a new domain: the XOR domain. The IO is applied by the web server and all documents are served in their isolated form. To render the page, the web browser has to *de-isolate* the source by applying again the IO and then execute it.

Note that, in xJS, we follow the approach of randomizing the whole source corpus and not just the instruction set as in the basic ISR concept. We proceed with this choice since the web code is produced in the web server and it is executed in the web browser. In addition, the server lacks all needed functionality to manipulate the produced code. For example, randomizing the JavaScript instruction set needs at least one full JavaScript parser running at the server. This can significantly increase the computational overhead and user-perceived latency, since the code would be parsed twice (one in the server during serving and one in the client during execution). However, the isolation can break web applications that explicitly evaluate dynamic JavaScript code using `eval()`. In that case, the web developer must use a new API, `xeval()`, since xJS alters the semantics of `eval()`. We further discuss this in Section 5. Finally, we select XOR as the IO because it is in general considered a fast process; all modern hardware platforms include a native implementation of the XOR function. However, our framework may be applied with any other IO.

Figure 1 depicts an xJS example. On the left, we show the source code as it exists in the web server and on the right, we provide the same source as it is fetched by the web browser. The JavaScript source has been XORed and a Base64 [18] encoding has been applied in order to transpose all non-printable characters to the printable ASCII range.

Action Based Policies

xJS allows for multiple trust-levels for the same web site depending on the desired operation. In general, our framework suggests that policies should be expressed as actions. Essentially, all trusted code should be treated using the policy "*de-isolate and execute*". For different trust levels, multiple IOs can be used or the same IO can be applied with a different key. For example, portions of client-side code can be marked with different trust levels. Each portion will be isolated using the XOR function, but with a different key. The keys are transmitted in HTTP headers (see the use of `X-IO-Key`, later in this section) every time the server sends the page to the browser.

Expressing the policies in terms of actions has the following benefit. The injected code cannot bypass the policy, unless it manages to produce the needed result after the action is applied to it. The latter is considered practically very hard, even for trivial actions such as the XOR operation. One possible direction for escaping the policy is using a brute force attack. However, if the key is large enough the probability to succeed is low.

Defining the desired policy set is out of the scope of this paper. For the purpose of our evaluation (see Section 4) we use one policy, which is expressed as "*de-isolate (apply XOR) and execute*". Other example policies can be expressed as "de-isolate and execute under user confirmation", "de-isolate with the X key and execute", etc.

2.1 Implementation

Browser Modifications. All three modified web browsers operate in the following way. A custom HTTP header field, `X-IO-Key`, is identified in each HTTP response. If the key is present, this is an indication that the web server supports the framework, and the field's value denotes the key for the de-isolation process. This is also a practical way for incremental deployment of the framework in a backwards compatible fashion. At the moment, we do not support multiple keys, but extending the browser with such a feature is considered trivial. On the other hand, the web browser communicates to the web server that it supports the framework using an `Accept`[2] header field for every HTTP request.

As far as WebKit and Chromium are concerned, we had to modify two separate functions. First, the function that handles all events (such as `onload`, `onclick`, etc.), and second, the function that evaluates a JavaScript code block. We modified these functions to (i) decode all source using Base64 and (ii) apply the XOR operation with the de-isolation key (the one transmitted in `X-IO-Key`) to each byte. FireFox has a different design. It also uses two functions, one for compiling a JavaScript function and one for compiling a script. However, these functions operate recursively. We further discuss this issue in Section 4.

Server Modifications. For the server part of xJS we are taking advantage of the modular architecture of the Apache web server. During Apache's start-up phase all configuration files are parsed and modules that are concerned with processing an HTTP request are loaded. The main processing unit of the apache web server is the content generator module. A module can register content generators by defining a handler that is configurable by using the `SetHandler` or `AddHandler` directives. These can be found in Apache's configuration file (httpd.conf).

Various request phases that precede the content generator exist. They are used to examine and possibly manipulate some request headers, or to determine how the request will be handled. For example the request URL will be matched against the configuration, because a certain content generator must be used. In addition the request URL may be mapped to a static file, a CGI script or a dynamic document according to the content generator's operation. Finally after the content generator has sent a reply to the browser, Apache logs the request.

Apache (from version 2 and above) also supports filters. Consider the filter chain as a data axis, orthogonal to the request processing axis. The request data may be processed by input filters before reaching the content generator. After the generator has finished generating the response various output filters may process it before being sent to the browser.We have created an

[2]For the definition of the `Accept` field in HTTP requests, see: `http://www.w3.org/Protocols/HTTP/HTRQ_Headers.html#z3`

Apache module which operates as a content generator. For every request, that corresponds to an HTML file in the disk, the file is fetched and processed by our module. The file is loaded in memory and stored in a buffer. The buffer is transfered to an HTML parser (based on the `HTMLParser` module from libxml2 [39]). This is an HTML 4.0 non-verifying parser with API compatible with the XML parser ones. When the parsing is done our module traverses the parser's XML nodes in memory and searches for all nodes that contain JavaScript (`<script>` nodes and events). If there is a match the XOR operation is applied using the isolation key to each byte of the JavaScript source. Finally all source is encoded in Base64.

After encoding all possible JavaScript source in the web page, the buffer is sent to the next operating module in the chain; this might be an output filter or the web browser. Implementing `xJS` as a content generator module has the benefit of isolating by encryption all JavaScript source before any dynamic content, which might include XSS attacks, is inserted. Our framework can cooperate with other server-side technologies, such as PHP, in two ways: (a) by using two Apache's servers (one running `xJS` and the other one the PHP module) and (b) by configuring PHP to run as a filter. All evaluation results presented in Section 4 are collected using the second setup.

Secret Key. The secret key that is used for the `XOR` operation is a string of random alphanumeric characters. The length of the string can be arbitrary. For all experiments presented in this paper a two-character string is used. Assuming that S_l is the JavaScript source of length l and K_L is the secret key of length L, the encoding works as follows: $Enc(S_i) = S_i \oplus K_{(i\,\%\,L)}, 0 < i < l$. It is implied that the ASCII values of the characters are used. The secret key is refreshed per request. We do not consider Man-in-the-Middle (MiM) attacks, since during a MiM an attacker can alter the whole JavaScript source without the need of an injection through XSS.

3 Attacks Covered

In this section we present a new form of XSS attack, which we refer to as *return-to-JavaScript* attack, in analogy with the *return-to-libc* attack in native code. This kind of XSS attack can escape script whitelisting, used by existing XSS mitigation schemes. We further highlight some important issues for DOM-based XSS mitigation schemes. All the attacks listed in this section can be successfully prevented by `xJS`.

3.1 *return-to-JavaScript* Attacks

A practical mitigation scheme for XSS attacks is script whitelisting, proposed in BEEP[17]. BEEP works as follows. The web application includes a list of cryptographic hashes of valid (trusted) client-side scripts. The browser, using a *hook*, checks upon execution of a script if there is a cryptographic hash in the whitelist. If the hash is found, the script is considered trusted and executed by the browser. If not, the script is considered non-trusted and the policy defines whether the script may be rendered or not. *Script whitelisting is not sufficient.* Despite its novelty, we argue here that simple whitelisting may not prove to be a sufficient countermeasure against XSS attacks. To this end, consider the following.

Location of trusted scripts. As a first example, note that BEEP does not examine the script's location inside the web document. Consider the simple case where an attacker injects a trusted script, initially configured to run upon a user's click (using the `onclick` action), to be rendered upon document loading (using the `onload`[3] action). In this case the script will be executed, since it is already whitelisted, but not as intended by the original design of the site; the script will be executed upon site loading and not following a user's click. If, for example, the script deletes data, then the data will be erased when the user's browser loads the web document and not when the user clicks on the associated hyperlink.

Exploiting legitimate whitelisted code. Attacks may be further carried out through legitimate white-listed code. XSS attacks are typically associated with injecting arbitrary client-side code in a web document, which is assumed to be foreign, i.e., not generated by the web server. However, it is possible to perform an XSS attack by placing code that *is* generated by the web server in different regions of the web page. This attack resembles the classic *return-to-libc* attack [11] in native code applications and thus we refer to as *return-to-JavaScript*. Return oriented programming suggests that an exploit may simply transfer execution to a place in `libc`[4], which may cause again execution of arbitrary code on behalf of the attacker. The difference with the traditional buffer overflow attack [29] is that the attacker has not injected any *foreign* code in the program. Instead, she transfers execution to a point that already hosts code that can assist her goal. A similar approach can be used by an attacker to escape whitelisting in the web environment. Instead of injecting her own code, she can take advantage of existing *whitelisted* code available in the web site. Note that, typically, a large fraction of client-side code is not executed upon document loading, but is triggered during user events, such as mouse clicks. Below we enumerate some possible scenarios for XSS attacks based on whitelisted code, which can produce (i) annoyance, (ii) data loss and (iii) complete takeover of a web site.

Annoyance. Assume the blog site shown in Figure 2. The blog contains a JavaScript function `logout()`, which is executed when the user clicks the corresponding hyperlink, *Logout* (line 4 in Fig. 2). An attacker could perform an XSS attack by placing a script that calls `logout()` when a blog entry is rendered (see line 7 in Fig. 2). Hence, a user reading the blog story will be forced to logout. In a similar fashion, a web site that uses JavaScript code to perform redirection (for example using `window.location.href = new-site`) can be also attacked by placing this whitelisted code in an `onload` event (see line 8 in Fig. 2).

Data Loss. A web site hosting user content that can be deleted using client-side code can be attacked by injecting the whitelisted deletion code in an `onload` event (see line 9 in Fig. 2). AJAX [15] interfaces are popular in social networks such as Facebook.com and MySpace.com. This attack can be considered similar to a SQL injection attack [5], since the attacker is implicitly granted access to the web site's database.

Complete Takeover. Theoretically, a web site that has a full featured AJAX interface can be completely taken over, since the attacker has all the functionality she needs a-priori whitelisted by the web server. For example, an e-banking site that uses a JavaScript `transact()` function for all the user transactions is vulnerable to XSS attacks that perform arbitrary transactions.

A workaround to mitigate the attacks presented above is to include the event type during the whitelisting process. Upon execution of script `S1`, which is triggered by an `onclick` event, the browser should check the whitelist for finding a hash key for `S1` *associated with an* `onclick` *event*. However, this can mitigate attacks which are based on using existing code with a different event type than the one initially intended to by the web programmer. Attacks may still happen. Consider the *Data Loss* scenario described above, where an attacker places the deletion code in `onclick` events associated with new web document's regions. The attacker achieves to execute legitimate code upon an event which is not initially scheduled. Although the attacker has not injected her own code, she manages to escape the web site's logic and associate legitimate code with other user actions. Attacks against whitelisting, based on injecting malicious data in whitelisted scripts, have been described in [26].

3.2 DOM-based Attacks

There is a number of proposals [16, 26, 13] against XSS attacks, which are based on information and features provided by DOM [24]. Every web document is rendered according to DOM, which represents essentially its esoteric structure. This structure can be utilized in order to detect or prevent XSS attacks. One of the most prominent and early published DOM-based techniques is DOM sandboxing, introduced originally in BEEP.

DOM sandboxing works as follows. The web server places all scripts inside `div` or `span` HTML elements that are attributed as *trusted*. The web browser, upon rendering, parses the DOM tree and executes client-side scripts only when they are contained in *trusted* DOM elements. All other scripts are marked as non-trusted and they are treated according to the policies defined by the web server. We discuss here in detail three major weaknesses of DOM sanbdoxing as an XSS mitigation scheme: (i) element annotation and (ii) DOM presence.

Element annotation. Enforcing selective execution in certain areas of a web page requires identification of those DOM elements that may host untrusted code or parts of the web application's code that inject unsafe content. This identification process is far from trivial, since the complexity of modern web pages is high, and web applications are nowadays composed of thousands lines of code. To support this, in Table 1 we highlight the number of `script`, `div` and `span` elements of a few representative web page samples. Such elements can be in the order of thousands in modern web pages. While there is active research to automate the process of marking untrusted data [35, 23] or to discover taint-style vulnerabilities [19, 25], we believe that, currently, the overhead of element annotation is prohibitive, and requires, at least partially, human intervention. On the contrary, `xJS` does not require taint-tracking or program analysis to identify trusted or untrusted parts of a web document or a web application.

	Facebook.com	MySpace.com	Digg.com
`script`	23	93	82
`div`	2708	264	302
`span`	982	91	156

Table 1: Element counts of popular home pages indicating their complexity.

DOM presence. All DOM-based solutions require the presence of a DOM tree. However, XSS attacks do not always require a DOM tree to take place. For example, consider an XSS attack which bypasses the content-sniffing algorithm of a browser and is *carried within* a PostScript file [7]. The attack will be launched when the file is previewed, and there is high probability that upon previewing there will be no DOM tree to surround the injected code. As browsers have been transformed to a generic preview tool, we believe that variants of this attack will manifest in the near future.

[3] One can argue that the `onload` action is limited and usually associated with the `<body>` tag. The latter is considered hard to be altered through a code-injection attack. However, note, that the `onload` event is also available for other elements (e.g. images, using the `<img>` tag) included in the web document.

[4] This can also happen with other libraries as well, but `libc` seems ideal since (a) it is linked to every program and (b) it supports operations like `system(), exec(), adduser()`, etc., which can be (ab)used accordingly. More interestingly, the attack can happen with no function calls but using available combinations of existing code [36].

```
1: <html>
2:  <head> <title> Blog! </title> <head>
3:  <body>
4:   <a onclick="logout();">Logout</a>
5:   <div class="blog_entry" id="123"> {...} <input type="button" onclick="delete(123)"></div>
6:   <div class="blog_comments"> <ul>
7:    <li> <img onload="logout();" src="logo.gif">
8:    <li> <img onload="window.location.href='http://www.google.com';" src="logo.gif">
9:    <li> <img onload="delete(123);">
10:  </div>
11:  <a onclick="window.location.href='http://www.google.com';">Google</a>
12: </body>
13:</html>
```

Figure 2: A minimal Blog site demonstrating the whitelisting attacks.

Another example is the unofficially termed *DOM-Based XSS* or *XSS of the Third Kind* attacks [22]. This XSS type alters the DOM tree of an already rendered page. The malicious XSS code does not interact with the sever in any way. In such an attack, the malicious code is embedded inside a URI after the fragment identifier. [5] This means that the malicious code (a) is not part of the initial DOM tree and (b) is never transmitted to the server. Unavoidably, DOM-based solutions [16, 26] that define trust classes in the DOM tree at server side will fail. The exploit will never reach the server and, thus, never be associated with or contained in a trust class.

3.3 Attacks Not Addressed

xJS aims on protecting against XSS attacks that are based on JavaScript injections. The framework is not designed for providing defenses against `iframe` injections and drive-by downloads [30], injections that are non-JavaScript based (for example, through arguments passed in Flash objects) and Phishing [12]. However, some fundamental concepts of xJS can be possibly applied to non-JavaScript injections.

4 Evaluation

In this section we evaluate the xJS prototype. Our evaluation seeks to answer four questions: (a) how many real XSS attacks can be prevented, (b) what the overhead on the server is, (c) what the overhead on the web browser is and, finally, (d) if the framework imposes any side-effects in the user's browsing experience.

4.1 Attack Coverage

We first evaluate the effectiveness of the xJS framework to prevent real-world XSS attacks. xJS aims on preventing traditional XSS attacks, as well as the XSS attacks described in Section 3.

Real-world exploits. To verify that xJS can cope with real-world XSS exploits, we use the repository hosted by XXSed.com [14] which includes a few thousands of XSS vulnerable web pages. This repository has been also used for evaluation in other papers [26]. The evaluation of the attack coverage through the repository is not a straightforward process. First, XSSed.com mirrors all vulnerable web pages with the XSS code embedded in their body. Some of them have been fixed after the publication of the vulnerability. These updated pages cannot be of use, since xJS prevents the code injection before it takes place and there is no way for us to have a copy of the original vulnerable web page (without the XSS code in its body). Second, we have no access to the vulnerable web server and, thus, we cannot use our server-side filter for the evaluation.

To address the aforementioned limitations, we conduct the evaluation as follows. First, we resolve all web sites that are still vulnerable. To this end, we download all 10,154 web pages listed in XSSed.com, along with their attack vectors. As the attack vector we define the URL along with the parameters that trigger the vulnerability.[6] Since XSS attacks that are based on a redirection without using any JavaScript cannot be addressed by xJS, we remove all such cases. Thus, we exclude 384 URLs that have an `iframe` as attack vector, 416 URLs that have a redirection to XSSed.com as attack vector and 60 URLs that have both an `iframe` and a redirection to XSSed.com as attack vector.

After this first pre-processing stage, the URL set contains all web pages that were vulnerable at some period in time and their vulnerability can be triggered using JavaScript; for example, the attack vector contains a call to the `alert()` function. We then exclude from the set all web-pages for which their vulnerability has been fixed after it became public in XSSed.com. To achieve this, we request each potentially vulnerable page through a custom proxy server we built using BeautifulSoup [31]. The task of the proxy is to attach some JavaScript code that overrides the `alert()` function with a URL request to a web server located in our local network. Since all attack vectors are based on the `alert()` function the web server recorded all successful attacks in its access logs. Using this methodology we manage to identify 1,381 web pages which are still vulnerable as of early September 2009. Our methodology suggests that about 1 in 9 web pages have not been fixed even after the vulnerability was published.

We use the remaining 1,381 pages as our final testing set. Since we cannot install our modified Apache in each of the vulnerable web sites, we use our proxy for simulating the server-side portion of xJS. More precisely, for each vulnerable page, we request the vulnerable document through our proxy with a slightly altered vector. For example, for the following attack vector,

```
http://site.com/page?
id=<script>alert("XSS");</script>
```

the proxy instead requests the URL,

```
http://site.com/page?
id=<xscript>alert("XSS");</xscript>.
```

Notice that the `script` tag has been modified to `xscript`. Using this methodology, we manage to build all vulnerable web pages with *the attack vector embedded but not in effect*. However, the JavaScript code contained in the web document is not isolated. Thus, the next step is to force the proxy to parse all web documents and apply the XOR function to the JavaScript code. At this point, all vulnerable web pages have the JavaScript code isolated and the attack vector defunct. Hence, the last step is to re-enable the attack vector by replacing the `xscript` with `script` and return the web page to the browser. All web pages also include some JavaScript code responsible for the `alert()` overloading. This code modifies all `alert()` calls to perform a web request to a web server hosted in our local network. If our web server records requests, the `alert()` function is called or, in other words, the XSS exploit run.

To summarize the above process, our experiment to evaluate the efficacy of the xJS framework is the following. We request each web page from the collected set which includes 1,381 still vulnerable web pages through a custom proxy that performs all actions described above. All web pages are requested using a modified Firefox. We select the modified Firefox in Linux, because it is easier to instrument through a script. We manually tested a random sample of attacks with modified versions of WebKit and Chromium and recorded identical behavior.

After Firefox has requested all 1,381 vulnerable pages through our custom proxy, we inspect our web server's logs to see if any of the XSS attacks succeeded. Our web server recorded just one attack. We carefully examined manually this particular attack and found out that it is a web page that has the XSS exploit stored inside its body and not in its attack vector [4]. The particular attack succeeded just as a side-effect of our evaluation methodology. If xJS were deployed in the vulnerable web server, this particular attack would also have been prevented. Hence, *all* 1,380 real-world XSS attacks were prevented successfully by our framework.

Attacks presented in Section 3. For the attacks presented in Section 3, since to our knowledge they have not been observed in the wild yet, we performed various custom attack scenarios using a popular web framework, Ruby on Rails [37]. We created a vulnerable blog and then installed the vulnerable blog service to a modified Apache server and browsed the blog using all three modified web browsers. As expected, in all cases, xJS succeeded in preventing the attacks.

We now look at specific attacks such as the ones based on a code injection in data and the careless use of `eval()`. The injected code is in plain text (non-isolated), but unfortunately it is attached to the isolated code after the de-isolation process. The injected code will be executed as if it is trusted. However, there is a way to prevent this. In fact, the internal design of Firefox gives us this feature with no extra cost. Firefox uses a `js_CompileScript()` function in order to compile JavaScript code. The design of this function is recursive and it is essentially the implementation of the actual `eval()` function of JavaScript. When Firefox identifies the script `eval($_GET('id'));`, de-isolates it, calls the `eval()` function, which in principle calls itself in order to execute the `$_GET('id')` part. At the second call, the `eval()` again de-isolates the `$_GET('id')` code, which is in plain text. The second de-isolation process fails and thus the code does not execute.

Our Firefox implementation can address this type of attack. WebKit and Chromium must be further modified to support this functionality. We have successfully implemented this process in Chromium after a small amount of code changes. However, this modification affects the semantics of `eval()`. For a more detailed discussion, please see Section 5.

4.2 Server Overhead

We now measure the overhead imposed on the server by xJS. To this end, we request a set of web pages that embed a significant amount of JavaScript. We choose to use the SunSpider suite [3] for this purpose. The SunSpider suite is a collection of JavaScript benchmarks that ship with WebKit and measure the performance of JavaScript engines. It is composed of nine different groups of programs that perform various complex operations. We manually select three JavaScript tests from the SunSpider suite. The *heavy* test involves string operations with many lines of JavaScript. This is probably the most

[5] For more details about the fragment identifier, we refer the reader to `http://www.w3.org/DesignIssues/Fragment.html`.

[6] For example, consider the attack vector: `http://www.needforspeed.com/undercover/home.action?lang=\"⟩⟨script⟩alert(document.cookie);⟨/script⟩&region=us`

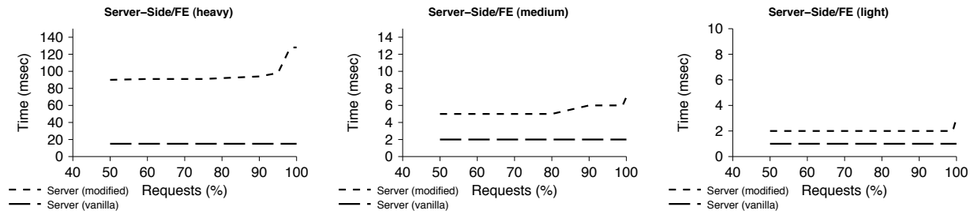


Figure 3: Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a Fast Ethernet link. In the worst case (heavy) the server imposes delay of a factor of five greater, while in the normal case the delay is only a few milliseconds.

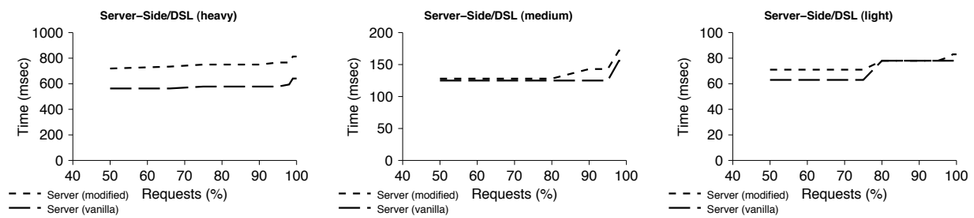


Figure 4: Server side evaluation when the Apache benchmark tool (`ab`) is requesting each web page through a DSL link. In the worst case (heavy) the server imposes a fixed delay of a few tens of milliseconds, like in the case of the Fast Ethernet setup (see Figure 3). However, this delay does not dominate the overall delivery time.

processing-intensive test in the whole suite, composed of many lines of code. The *normal* test includes a typical amount of source code like most other tests that are part of the suite. Finally, the *light* test includes only a few lines of JavaScript involving bit operations.

We conduct two sets of experiments. For the first set we use `ab` [1], which is considered the de-facto tool for benchmarking an Apache web server, over a Fast Ethernet (FE) network. We configure `ab` to issue 100 requests for the heavy, normal and light web pages, while the `xJS`module is enabled. Then, we perform the same experiments using the tests and with the `xJS` Apache module removed. We repeat all the above with the `ab` client running in a typical downstream DSL line (8 Mbps).

Figure 3 summarizes the results for the case of the `ab` tool connecting to the web server through a FE connection. The modified Apache imposes an overhead that ranges from a few (less than 6 ms and less than 2 ms for the normal and light test, respectively) to tens of milliseconds (about 60 ms) in the worst case (the heavy web page). While the results are quite promising for the majority of the tests, the processing time for the heavy page, which is over a factor of five greater, could be considered significant. In Figure 4 we present the same experiments over the DSL link. The overhead is still the same and it is negligible (less than a roundtrip in today's Internet) since now the delivery overhead dominates. This drives us to conclude that the Apache module imposes a fixed overhead of a few milliseconds per page, which is not the dominating overhead.

4.3 Client Overhead

Having examined the server-side overhead, we now measure the overhead imposed on the browser by `xJS`. We use the SunSpider test suite with 100 iterations, with every test executed 100 times. We use the `gettimeofday()` function to measure the execution time of the modified functions in each browser. Each implementation has two functions altered. The one that is responsible for handling code associated with events, such as `onclick`, `onload`, etc., and the one that is responsible for evaluating JavaScript code blocks. The modifications of WebKit and Chromium are quite similar (Chromium is based partially on WebKit). The modifications of Firefox are substantially different. In Firefox we have modified internally the JavaScript `eval()` function which is recursive. These differences affect the experimental results in the following way. In WebKit and Chromium we record fewer long calls in contrast with Firefox, in which we record many short calls.

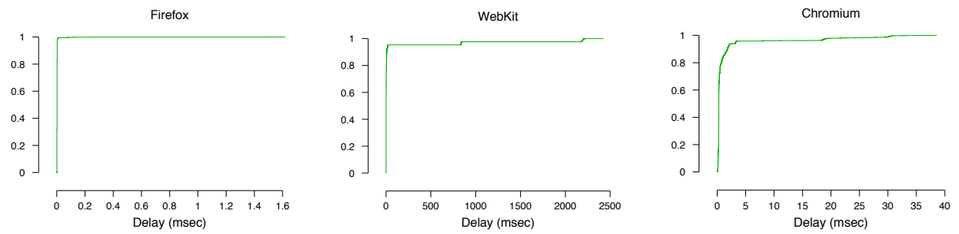


Figure 5: Cumulative distribution for the delay imposed by all modified function calls in the Firefox, WebKit and Chromium implementation, respectively. As delay we assume the time needed for the modified function to complete minus the time needed for the unmodified one to complete. Notice that the majority of function calls imposes a delay of a few milliseconds.

In Figure 5 we present the cumulative distribution of the delays imposed by all modified recorded function calls for Firefox, WebKit and Chromium, during a run of the SunSpider suite for 100 iterations. As delay we define the time needed for the modified function to complete minus the time needed for the unmodified one to complete. Observe that the Firefox implementation seems to be the faster one. All delays are less than 1 millisecond. However, recall that Firefox is using a lot of short calls, compared to the other two browsers. Firefox needs about 500,000 calls for the 100 iterations of the complete test suite. In Figure 5 we plot the first 5,000 calls for Firefox (these calls correspond to one iteration only) of the complete set of about 500,000 calls, for visualization purposes and to facilitate comparison, and all 4,800 calls needed for WebKit and Chromium to complete the test suite, respectively. Chromium imposes an overhead of a few milliseconds per call, while WebKit seems to impose larger overheads. Still, the majority of WebKit's calls impose an overhead of a few tens of milliseconds.

4.4 User Browsing Experience

We now identify whether user's browsing experience changes due to `xJS`. As user browsing experience we define the performance of the browser's JavaScript engine (i.e., running time), which would reflect the user-perceived rendering time (as far as the JavaScript content is concerned) for the page. We run the SunSpider suite *as-is* for 100 iterations with all three modified web browsers and with the equivalent unmodified ones and record the output of the benchmark. In Figure 6 we plot the results for all different categories of tests. Each category includes a few individual benchmark tests. As expected there is no difference between a modified and a non modified web browser for all three platforms, Firefox, WebKit and Chromium. This result is reasonable,

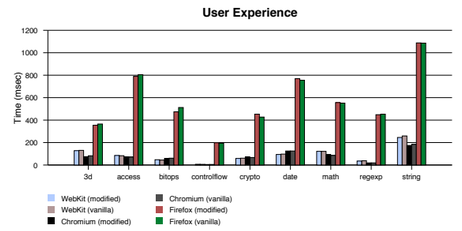


Figure 6: Results from the SunSpider test suite. Notice that for each modified browser the results are comparable with the results of its unmodified equivalent. That is, all de-isolated JavaScript executes as expected in both modified and unmodified browser.

since after the de-isolation process the whole JavaScript source executes normally as it is in the case with a non compatible with the `xJS` framework web browser. Moreover, this experiment shows that `xJS` is not restrictive with legitimate web sites, since all the SunSpider suite (some thousands of JavaScript LoCs) run without any problem or side-effect.

5 Discussion

We now discuss potential limitations of our approach and offer possible workarounds.

JavaScript Obfuscation. Web pages served by `xJS` have all JavaScript encoded in Base64. Depending on the context this may be considered as a feature or not. For example, there are plenty of custom tools that obfuscate JavaScript on purpose. Such tools are used by certain web sites for protecting their JavaScript code and prevent visitors from copying the code. We should make clear that emitting all JavaScript encoded does not harden the

development process, since all JavaScript manipulation takes place during serving time. While debugging, web developers may safely switch off xJS. Blueprint [38] also emits parts of a web page in Base64.

eval() Semantics and Dynamic Code. As previously discussed (see Section 4), in order for xJS to cope with XSS attacks that are based on malicious injected data, the semantics of eval() must change. More precisely, our Firefox modifications alter the eval() function in the following way. Instead of simply evaluating a JavaScript content, the modified eval() function performs de-isolation before evaluation. This behavior can break web applications that are based on the generation of dynamic JavaScript code, which is executed using eval() at serving time. While this type of programming might be considered inefficient and error-prone, we suggest the following workaround. The JavaScript engine can be enhanced with an xeval() variant which does not perform any de-isolation before evaluation. The web programmer must explicitly call xeval() if this is the desired behavior. Still, there is no possibility for the attacker to evaluate her code (using xeval()), since the original call to xeval() must be already isolated.

Code Templates and Persistent XSS. Web developers frequently use templates in order to produce the final web pages. These templates are stored usually in a database and sometimes they include JavaScript. The database may also contain data produced by user inputs. In such cases, the code injection may take place *within* the database (Persistent XSS). This may occur if trusted code and a user input containing malicious code are merged together before included in the final web page. This case is especially hard to track, since it involves the programmer's logic to a great extent. The challenge lies in that client-side code is hosted in another environment (the database) which is also vulnerable to code injections. xJS assumes that all trusted JavaScript is stored in files and not in a database. If the web developer wishes to store legitimate JavaScript in a database then she can place it in read-only tables. With these assumptions, xJS can cope with persistent XSS. Recall from Section 2 that xJS module is the first to run in the Apache module chain and, thus, all JavaScript isolation will take place before any content is fetched from databases or other external sources.

6 Related Work

The closest studies to xJS are BEEP [17], Noncespaces [16] and DSI [26]. Throughout the paper, we have highlighted certain cases where the aforementioned methodologies fail (e.g., see Section 3). We have presented attacks that escape whitelisting (proposed in [17]) and cases where DOM-based solutions [16, 26] are not efficient. Our framework, xJS, can cope with XSS attacks that escape whitelisting [6], and does not require any information related to DOM; xJS can also prevent attacks that leverage the content-sniffing algorithms of web browsers [7].

Our technique is based on Isolation Operators and it is inspired by Instruction Set Randomization (ISR) [20]. Solutions based on ISR have been applied to native code and to SQL injections [9]. Keromytis discusses ISR as a generic methodology for countering code injections in [21] and he mentions that the technique can be potentially applied in XSS mitigation. However, to the best of our knowledge there has been no systematic effort towards this approach before.

In [40] the authors propose to use dynamic tainting analysis to prevent XSS attacks. Taint-tracking has been partially or fully used in other similar approaches [26, 35, 28, 27]. Although xJS does not rely at all on tainting, a source-code based tainting technique [43] can certainly assist in separating all server-produced JavaScript. The server side of xJS will be able to efficiently mark all legitimate client-side code and also identify malicious data. However, the performance might degrade.

Blueprint [38] is a server-only approach which guarantees that untrusted content is not executed. The application server pre-renders the page and serves each web document in a form in which all dynamic content is correctly escaped to avoid possible code injections. However, Blueprint requires the web programmer to inject possible unsafe content (for example comments of a blog story) using a specific Blueprint API in PHP. Spotting all unsafe code fragments of a web application is not trivial. Blueprint imposes further a significant overhead compared to solutions based on natively browser modifications, like xJS.

Enforcing separation between structure and content is another prevention scheme for code injections [32]. This proposed framework can deal with XSS attacks as well as SQL injections. As far as XSS is concerned, the basic idea is that each web document has a well defined structure in contrast to a stream of bytes, as it is served nowadays by web servers. This allows the authors to enforce a separation between the authentic document's structure and the untrusted dynamic content from user input, which is attached to it. However, in contrast to xJS, this technique cannot deal with attacks that are based on the content-sniffing algorithms of browsers [7] as well as attacks that modify the DOM structure using purely client-side code [22].

Script accenting [10] is based also on XOR for isolating scripts in the web browser. Script accenting aims on providing an efficient mechanism for implementing the *same origin policy* in a web browser, but it is not explicitly related with XSS. Leaks that can take place due to the DOM separation from the JavaScript engine, inside a web browser, and can lead to browser compromising have been studied in [8]. These attacks can be considered more severe than XSS and cannot be captured by xJS. MashupOS [41] and subsequent work Gazelle [42] view browsers as a multi-principal OS where a principal is labeled by a web site's origin following the *same-origin policy* [33]. MashupOS analyzed and proposed protection and communications abstractions that a browser should expose for web application developers. In particular, <sandbox> is proposed to embed untrusted content and can be used by developers to prevent XSS attacks as long as they can correctly differentiate trusted content from untrusted ones. In comparison, our work does not require explicit inclusion of untrusted content from developers.

7 Conclusion

In this paper we present xJS, a practical and developer-friendly framework against the increasing threat of XSS attacks. The motivation for developing xJS is twofold. First, we want an XSS prevention scheme that can cope with the new *return-to-JavaScript* attacks presented in this paper and second, we want the solution to be easily adopted by web developers.

We implement and evaluate our solution in three leading web browsers and in the Apache web server. Our evaluation shows that (a) every examined real-world XSS attack can be successfully prevented, (b) negligible computational overhead is imposed on the server and browser side, and (c) the user's browsing experience and perceived performance is not affected by our modifications.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback, Professor Angelos D. Keromytis for early discussions and our shepherd Helen Wang. Her assistance for improving the paper was invaluable. Elias Athanasopoulos, Antonis Krithinakis, Spyros Lygouras and Evangelos P. Markatos are also with the University of Crete. Elias Athanasopoulos is funded by the Microsoft Research PhD Scholarship project, which is provided by Microsoft Research Cambridge.

References

[1] ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] McAfee: Enabling Malware Distribution and Fraud. http://www.readwriteweb.com/archives/mcafee_enabling_malware_distribution_and_fraud.php.

[3] SunSpider JavaScript benchmark. http://www2.webkit.org/perf/sunspider-0.9/sunspider.html.

[4] XXSed.com vulnerability 35059. http://www.xssed.com/mirror/35059/.

[5] C. Anley. Advanced SQL injection in SQL server applications. *White paper, Next Generation Security Software Ltd*, 2002.

[6] E. Athanasopoulos, V. Pappas, and E. Markatos. Code-Injection Attacks in Browsers Supporting Policies. In *Proceedings of the 2nd Workshop on Web 2.0 Security & Privacy (W2SP)*, Oakland, CA, May 2009.

[7] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers or How to Stop Papers from Reviewing Themselves. In *Proceedings of the 30th IEEE Symposium on Security & Privacy*, Oakland, CA, May 2009.

[8] A. Barth, J. Weinberger, and D. Song. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Quebec, August 2009.

[9] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, 2004.

[10] S. Chen, D. Ross, and Y.-M. Wang. An Analysis of Browser Domain-Isolation Bugs and a Light-Weight Transparent Defense Mechanism. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, pages 2–11, New York, NY, USA, 2007. ACM.

[11] S. Designer. Return-to-libc attack. *Bugtraq, Aug*, 1997.

[12] R. Dhamija, J. Tygar, and M. Hearst. Why Phishing Works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 581–590. ACM New York, NY, USA, 2006.

[13] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to Strangers Without Taking their Candy: Isolating Proxied Content. In *SocialNets '08: Proceedings of the 1st Workshop on Social Network Systems*, pages 25–30, New York, NY, USA, 2008. ACM.

[14] K. Fernandez and D. Pagkalos. XSSed.com. XSS (Cross-Site Scripting) information and vulnerable websites archive. http://www.xssed.com.

[15] J. Garrett et al. Ajax: A New Approach to Web Applications. *Adaptive path*, 18, 2005.

[16] M. V. Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 8-11, 2009.

[17] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[18] S. Josefsson. RFC 4648: The Base16, Base32, and Base64 Data Encodings, 2006. `http://tools.ietf.org/html/rfc4648`.

[19] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[20] G. Kc, A. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction-Set Randomization. In *Proceedings of the 10th ACM conference on Computer and Communications Security*, pages 272–280. ACM New York, NY, USA, 2003.

[21] A. D. Keromytis. Randomized Instruction Sets and Runtime Environments Past Research and Future Directions. Number 1, pages 18–25, Piscataway, NJ, USA, 2009. IEEE Educational Activities Department.

[22] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. Web Application Security Consortium, Articles, 4.7. 2005.

[23] L. C. Lam and T.-c. Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[24] A. Le Hors, P. Le Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification. *World Wide Web Consortium, Recommendation REC-DOM-Level-3-Core-20040407*, 2004.

[25] M. Martin and M. S. Lam. Automatic Generation of XSS and SQL Injection Attacks with Goal-directed Model Checking. In *Proceedings of the 17th USENIX Security symposium*, pages 31–43, Berkeley, CA, USA, 2008. USENIX Association.

[26] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 8-11, 2009.

[27] S. Nanda, L. Lam, and T. Chiueh. Dynamic Multi-Process Information Flow Tracking for Web Application Security. In *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*. ACM New York, NY, USA, 2007.

[28] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[29] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 49(7), 1996.

[30] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All your iFRAMES point to us. In *Proceedings of the 17th conference on Security symposium*, pages 1–15. USENIX Association, 2008.

[31] L. Richardson. Beautiful Soup-HTML/XML parser for Python, 2008.

[32] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Quebec, August 2009.

[33] J. Ruderman. The same-origin policy, 2001. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[34] SANS Insitute. The Top Cyber Security Risks. September 2009. `http://www.sans.org/top-cyber-security-risks/`.

[35] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 8-11, 2009.

[36] H. Shacham. The Geometry of Innocent Flesh on the Bone: return-into-libc without Function Calls (on the x86). In *CCS '07: Proceedings of the 14th ACM conference on Computer and Communications Security*, pages 552–561, New York, NY, USA, 2007. ACM.

[37] B. Tate and C. Hibbs. *Ruby on Rails: Up and Running*. O'Reilly Media, Inc., 2006.

[38] M. Ter Louw and V. Venkatakrishnan. Blueprint: Precise Browser-neutral Prevention of Cross-site Scripting Attacks. In *Proceedings of the 30th IEEE Symposium on Security & Privacy*, Oakland, CA, May 2009.

[39] D. Veillard. Libxml2 project web page. *http://xmlsoft. org*, 2004.

[40] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[41] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In T. C. Bressoud and M. F. Kaashoek, editors, *SOSP*, pages 1–16. ACM, 2007.

[42] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[43] W. Xu, E. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium*, pages 121–136, 2006.

SeerSuite: Developing a scalable and reliable application framework for building digital libraries by crawling the web

Pradeep B. Teregowda
Pennsylvania State University

Isaac G. Councill
Google

Juan Pablo Fernández R.
Pennsylvania State University

Madian Khabsa
Pennsylvania State University

Shuyi Zheng
Pennsylvania State University

C. Lee Giles
Pennsylvania State University

Abstract

SeerSuite is a framework for scientific and academic digital libraries and search engines built by crawling scientific and academic documents from the web with a focus on providing reliable, robust services. In addition to full text indexing, SeerSuite supports autonomous citation indexing and automatically links references in research articles to facilitate navigation, analysis and evaluation. SeerSuite enables access to extensive document, citation, and author metadata by automatically extracting, storing and indexing metadata. SeerSuite also supports MyCiteSeer, a personal portal that allows users to monitor documents, store user queries, build document portfolios, and interact with the document metadata. We describe the design of SeerSuite and the deployment and usage of CiteSeer^x as an instance of SeerSuite.

1 Introduction

Efficient and reliable access to the vast scientific and scholarly publications on the web requires advanced citation index retrieval systems [21] such as CiteSeer [22, 4], CiteSeer^x [5], Google Scholar [9], ACM Portal [1], etc. The SeerSuite application framework provides an unique advanced and automatic citation index system that is usable and comprehensive, and provides efficient access to scientific publications. To realize these goals our design focuses on reliable, scalable and robust services.

A previous implementation, CiteSeer (maintained as of this date), was designed to support such services. However, CiteSeer was a research prototype and, as such, suffered serious limitations. SeerSuite was designed to provide a framework that would replace CiteSeer, to provide most of its functionality, but designed to be extensible. SeerSuite improves on several aspects of the original CiteSeer with features such as reliability, robustness and scalability. It does this by adopting a multi-tier architecture with a service orientation and a loose coupling of modules.

CiteSeer^x, an instance of SeerSuite is one of the top ranked resources on the web and indexes nearly one and half million documents. The collection spans computer and information science (CIS) and related areas such as mathematics, physics and statistics. CiteSeer^x acquires its documents primarily by automatically crawling authors web sites for academic and research documents. CiteSeer^x daily receives approximately two million hits and has more than two hundred thousand documents downloaded from its cache. The MyCiteSeer personal portal has over ten thousand registered users.

While the SeerSuite application framework has most of the functionality of CiteSeer, SeerSuite represents a complete redesign of CiteSeer. SeerSuite takes advantage of and includes state of the art machine learning methods and uses open source applications and modules. The structure and architecture of SeerSuite is designed to enhance the ease of maintenance and to reduce the cost of operations. SeerSuite is designed to run on off the shelf hardware infrastructure.

With CiteSeer^x, SeerSuite focuses primarily on CIS. However, there are requests for similar focused services in other fields such as chemistry [33] and archaeology [37]. SeerSuite can be adopted to providing services similar to those provided by CiteSeer^x in these areas. SeerSuite is a part of the project Chem$_X$Seer [3], a digital library search engine and collaboration service for chemistry. Though designed as a framework for CiteSeer^x-like digital libraries and search engines, the modular and extensible framework allows for applications that use SeerSuite components as stand alone services or as a part of other systems.

2 Background and Related Work

Domain specific repositories and digital library systems have been very popular over the last decades with several examples such as arXiv [23] for physics and RePEc [15]

for economics. The Greenstone digital library [10] was developed with a similar goal. These repository systems, unlike CiteSeer^x, depend on manual submission of document metadata, a tedious and resource expensive process. As such CiteSeer^x which crawls authors web page for documents is in many ways a unique system, closest in design to Google Scholar.

The popularity of services provided by the original CiteSeer and limitations in its design and architecture were the motivation behind the design of SeerSuite [18, 30]. The design of SeerSuite was an incremental process involving users and researchers. User input was received in the form of feedback and feature requests. Exchange of ideas between researchers and users across the world through collaborations, reviews and discussions have made a significant contribution to the design. In addition to ensuring reliable scalable services, portability of the overall system and components was identified as an essential feature that would encourage adoption of SeerSuite elsewhere. During the process of designing the architecture of SeerSuite, other academic repository content management system (CMS) architectures such as Fedora [28] and DSpace [8] were studied. The Blackboard architecture pattern [34] had a strong influence on the design of the metadata extraction system. The main obstacles in adopting existing repository and CMS systems were the levels of customization and the effort required to meet SeerSuite design requirements. More specifically, implementation of the citation graph structure with a focus on automatic metadata extraction and workflow requirements for maintaining and updating citation graph structures made these approaches cumbersome to use.

In addition to repository, search engine and digital library architectures, advances in metadata extraction methods [24, 25, 19, 27, 32] and the availability of open source systems have influenced SeerSuite design. We begin our discussion of SeerSuite by describing the architecture.

3 Architecture

In the context of SeerSuite *reliability* refers to the ability of the framework instances to provide around the clock service with minimal downtime, *scalability* to the ability to support increasing number of user requests and documents in the collection, and *robustness* to the ability of the instance to continue providing services while some of the underlying services are unavailable or resource constrained.

An outline of SeerSuite architecture is shown in figure 1. By adopting a loosely coupled approach for modules and subsystem, we ensure that instances can be scaled and can provide robust service. We describe the overall architecture in the web application, data storage, metadata extraction and ingestion, crawling and maintenance sections.

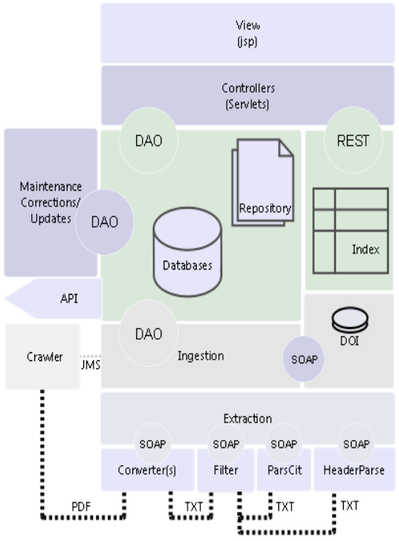


Figure 1: SeerSuite Architecture

3.1 Web application

The web application makes use of the model view controller architecture implemented with the Spring framework. The application presentation uses a mix of java server pages and javascript to generate the user interface. Design of the user interface allows the look and feel to be modified by switching Cascading Style Sheets (CSS). The use of JavaServer Pages Standard Tag Library (JSTL) supports template based construction. The web pages allow further interaction through the use of javascript frameworks. While the development environment is mostly Linux, components are developed with portability as a focus. Adoption of the spring framework allows development to concentrate on application design. The framework supports the application by handling interactions between the user and database in a transparent manner.

Servlets use Data Access Objects (DAO) with the support of the framework to interact with databases and the repository. The index, database and repository enable the data objects crawled from the web to be stored and accessed through the web, using efficient programmatic interfaces. The web application is deployed through a web application archive file.

3.2 Data storage

The databases store metadata and relationships in the form of tables providing transaction support, where transactions include adding, updating or deleting objects. The database design partitions the data storage across three main databases. This allows for growth in any one component to be handled by further partitioning the database horizontally.

The main database contains objects and is focused on transactions and version tracking of objects central to SeerSuite and digital libraries implementations. Objects stored in the main database include document metadata, author, citation, keywords, tags, and hub (URL). The tables in the main database are linked together by the document they appear in and are identified by the document object id.

One of the most unique aspects of SeerSuite is the citation graph. The nodes of this graph correspond to documents or citations and the edges to the relationships among them. This graph is stored in its own database. The citation relationship is stored in a graph table in the form of 'cited by' and 'cites' fields. In addition the database stores a canonical representation of citation or document metadata which is basically the most frequent and correct of the citations, as determined by a Bayesian network. These records serve as a grouping point for metadata collected about a particular document or citation. This database thus provides support for autonomous citation indexing and related features. The citation relationships are established by generating and matching keys for the document metadata records during ingestion or updates. The use of triggers allows data in the graph database to be updated when transactions such as insertions, deletions and corrections occur in the main database. In addition to triggers, application logic and maintenance functions maintain the link between the graph and the main database. MyCiteSeer personalization portal stores user information, queries and document portfolios in a separate database. The link between the user and the main database is through references in the tags in the MyCiteSeer database and the main database version tracking tables. A conventional RDBMS with support for triggers is used to host these databases.

The repository system provides SeerSuite with the ability, to provide cached copies of the documents crawled. In addition to the cached copy, the repository stores xml files containing extracted document metadata. These files serve as backup copies of the metadata stored in the databases. This is similar to the Fedora Object XML document representation [28]. The repository is organized into a directory tree. The top of the tree consists of a root folder containing sub directories mapped according to the segments in the document identifier. The final level contains metadata, text and cached files. The document identifier structure ensures that the there are a nominal number of sub folders under any folder in the tree structure.

An index provides a fast efficient method of storing and accessing text and metadata through the use of an inverted file. SeerSuite uses the Apache Solr an Index application [16] supported by Apache Tomcat to provide full text and metadata indexing operations. The metadata items are obtained from the database and the full text from the repository. The interaction of the application in the form of controllers is through the REST (Representational state transfer) [20] interface. This allows any indexing application supporting REST API's to be adopted by SeerSuite. In addition, this enables introduction of newer feature sets in the index or new versions of Solr without disruptions.

3.3 Metadata Extraction and Ingestion

Metadata extraction methods are built using Perl and C++. To enable interaction with the extraction services, a service oriented architecture is utilized using a Business Process Execution Language (BPEL) or, for convenience, scripts. Each component of the extraction system individually contributes to the final document, in this case an xml file, which is then ingested. The feedback to individual systems is manual adjusting of parameters or replacing components. The system can be rewired to include or exclude any extraction modules or applications.

The user can either batch process the incoming data or process each item individually. Addition of metadata into the system is controlled by the ingestion system, which interacts with main database using DAOs. The ingestion system ensures that essential metadata is correctly and uniquely labeled, with the help of an object identifier and checksum based de-duplication. In addition, the ingestion system makes use of listeners to share notification data such as alerts that inform users and programs about objects of interest.

The ingestion process can itself be distributed across machines, taking advantage of the Document Object Identifier (DOI), database and shared repository services. The DOI is issued by on of our web services with its own database and tracks document identifiers and the time of their issue.

3.4 Crawler

The suite also includes a Heritrix [11] based crawler for harvesting documents from the web. The interface between the ingestion system and the crawler is based on the Java Messaging Service over ActiveMQ [2]. Due to the modularity of the design, other crawlers can be used.

The ingestion system sends URL submissions messages containing a job identifier and url through the ingestion channel, and the crawler responds with 'new content' messages pointing to the acquired resource and metadata to the ingestion system. The crawler can use other optional channels to provide status messages to listeners. The crawler uses the Heritrix job submission system which consumes messages in the submissions channel and processes these submissions.

3.5 Maintenance

Maintenance services support and enable associated functionality for the ingestion and presentation systems. Maintenance systems are responsible for updating the index, identifying metadata by inference, generating citation and document statistics, charts generation, and external data linking in the system. For convenience, common maintenance processes are available through a command line interface.

An evidence based inference system [19] utilizes a Bayesian inference framework to determine canonical metadata for documents in the collection. The system builds an ideal citation record for a document inferring from the information provided by citations and the document metadata.

Citation graphs which accompany the document view are generated from the graph database, by examining the citation relationships and the distribution across years. In addition to enabling access to extensive document metadata, SeerSuite allows documents in the collection to be linked to copies or bibliographic records in other collections. SeerSuite provides components to map and link to other services such as DBLP [7] and CiteULike [6].

The configuration of an SeerSuite instance or application is controlled through properties and context files, through which information about the file system for the repository, database, index and system parameters can be specified. The web application and the maintenance and indexing functions use similar configuration files. In addition the SeerSuite distribution provides configuration files or examples of configurations for applications used such as the Solr index and Tomcat.

4 Workflow

The outline of the process of adding documents to SeerSuite instances is shown in Figure 2. Documents are harvested from the web using focused crawlers (step 1). These documents are first converted from PDF or PostScript format to text with application tools such as PDFBox and TET or GhostScript for PostScript documents in the step labelled 2.

In step 3, to prevent processing of documents such as resumes and non-academic documents part of the harvested collection, SeerSuite uses a regular expression based filter on the converted text file. The converted, filtered text is processed using state of the art automatic metadata extraction systems. These include the Support Vector Machine based header parser [24], which extracts metadata such as titles, publication information, authors and their affiliation, and abstract from the document. The ParsCit [27] citation extraction system extracts citation and context information from the document.

The ingestion system identifies unique documents and requests a document identifier for the document. If the document is found to be a checksum duplicate based on content, URL mappings are updated to include alternate URL(s) in step 4. In the same step, the repository is updated with a complete set of files including the document in the original format. The converted text files, citations, crawler and document metadata are placed in the relevant document directory under the repository tree. In addition to the individual metadata files, a file copy of the complete document metadata is stored in the form of an xml file. With updates and corrections, the xml files are updated and stored with a version tag. In the main database papers, authors citation and url mapping tables are updated, triggering updates to the graph database. Updates to the graph database ensure that the citation relationship of the incoming metadata to the data already existing in the collection is accurately maintained.

Documents in the database have a time stamp indicating time of update, helping the maintenance scripts perform incremental updates of the index. Incremental updates are crucial in reducing the time required for maintenance. With step 5, new or updated metadata are indexed. The maintenance script scans the database for updated and new documents and creates an in-memory indexable document, including the document metadata fields and citation information gathered across the main and citation graph databases. This document is then indexed by the main Solr index over the REST interface with an update command. The maintenance system optimizes the index after each update, using an built in feature in Solr.

Statistics provide users with a perspective of the collection from a citation, document and author ranking using aggregated citation information. Statistics are generated from the graph database, in the form of text files, which are then presented to the user through the statistics servlet interface. Maintenance scripts are manually scheduled or run by the administrator.

5 An Instance of SeerSuite: CiteSeer^x

CiteSeer^x serves as a flagship deployment of SeerSuite. It utilizes Apache Tomcat as the supporting platform

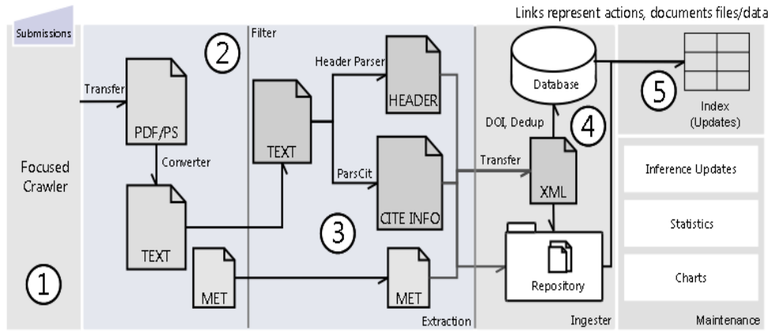


Figure 2: SeerSuite Workflow: Links represent actions and documents data

with MySQL as the RDBMS. The deployment spans multiple machines with varied configurations. Components are distributed based on functionality on these machines. A pair of level 4 load balancing servers direct traffic to a pair of webservers. The load balancers use a connection based metric to determine to which server a particular request will be directed. To ensure availability, the load balancers are configured as a high availability asymmetric cluster that uses open source linux high availability software.

The web servers host the application on the Apache Tomcat platform with the Tomcat instances as part of an Apache Tomcat TCP cluster with session information shared across the cluster. The application processes these requests and processes them with the help of the index, database or the repository.

In case of an search query, the application translates the query to a suitable format and dispatches the query to the Solr indices. The results are processed and presented to the user. CiteSeer^x uses indices for document and citation objects, table objects and disambiguated author objects. Requests for metadata are handled by the MySQL database system containing the main graph and the user databases. The repository is responsible for storing cached documents and the text and metadata files. Requests for cached files are handled by the application with support from the repository stored on a storage server. The repository system is shared with the ingestion system and web servers, using the Global File System within the cluster.

The processing system is maintained separately from the web application and data storage infrastructure. The document processing systems are responsible for converting and extracting the metadata from converted text files. This operation is distributed across machines by dividing the incoming data into distinct sets, each set being processed by individual machines across the cluster.

The deployment is supported by a staging and development system, where new features are introduced and reviewed before being introduced in the production system. Major components in the system are backed up either using component level replication services and or by file level backups. In addition to backup on site, off site backups are utilized to ensure redundancy.

CiteSeer^x depends on operating system based security and application security provided by firewalls plus intrusion detection systems and the underlying framework. Application logs and Tomcat error and access logging provide audit trails for bug fixes and troubleshooting.

Infrastructure adopted by CiteSeer^x has led to several issues. Frequent freezes occur due to deadlocks involving the shared file system. Hardware failures have led to loss of data across the repository database. In such cases back ups have helped restore services. The ability to recover from these unfortunate losses showcase CiteSeer^x robustness.

5.1 Focused Crawling

For the initial crawling seeds, CiteSeer^x assimilated the complete collection of documents and their URLs from CiteSeer with some exceptions. These documents were ingested into the system by utilizing the already existing information in the CiteSeer databases.

Though equipped with Heritrix, the new CiteSeer^x uses a customized focused crawler, which runs incrementally. The crawler is run on a daily basis and can fetch several thousand new documents every day. The system maintains a list of parent URLs where documents were previously found. The parent URL's include academic homepages containing lists of online publications.

CiteSeer^x has nearly two hundred thousand unique URLs containing links to publications. The crawl process begins with the crawl scheduler, which selects a

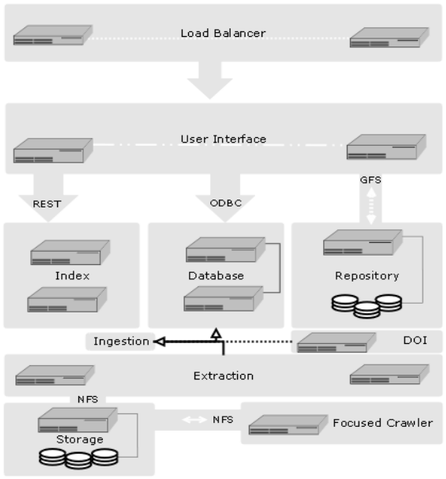


Figure 3: CiteSeer^x Deployment

thousand parent URLs based on an estimated likelihood of these pages having new documents. The selected URLs are fed to a seed crawler. The seed crawler revisits these thousand URLs and also follow links up to two hops from each scheduled URL. By parsing fetched pages, the seed crawler will retrieve all document links eligible for processing. These represent documents which will be later downloaded. If a document link is found on a new URL, this URL will be added to the parent URL list. This enables the parent URL list to expand. The crawling system also maintains a list of document URLs and a list of document checksum hash values for all previously crawled documents. These two lists help avoid downloading duplicate documents. When the seed crawler outputs a list of discovered document URLs, it first compares them to the maintained document URL list and filters out duplicate URLs. Therefore, documents with same URLs will not be downloaded again. Only new document URLs will be fed to the document crawler.

The document crawler then simply fetches all documents in PDF or PostScript from the input list. After the documents are crawled, their checksum values are calculated based on their content and compared to the maintained checksum list by the ingestion system. Thus, content based duplicates are removed. Finally, only new documents are ingested into CiteSeer^x. User submissions to CiteSeer^x are directly submitted to the crawler seed listing. A user interface allows tracking of the documents obtained from a submitted URL. This allows the user to determine, whether a document has been ingested into CiteSeer^x.

6 User Interface

A user interacts with SeerSuite through a number of web pages. The most common of these include results of searches, document details, and citation graphs. We describe a set of pages, whose coverage spans most of the functionality provided by SeerSuite. The user interfaces are built using jsp, backed by the controller servlets. A navigation panel allows the user quick access to the main pages. The access control to pages across the application is based on the login system provided as part of MyCiteSeer component.

6.1 Search Interface

Figure 4: Document Search Results

The search interface in SeerSuite allows users to make queries across document, author, and table metadata either by using the default query interface or an advanced search interface. Search results are generated as a result of a user query. The results can be sorted according to the relevance, citations, and recency of the document ingestion. Since SeerSuite indexes document metadata across entities such as title, authors and their affiliation, year of publication, publication venue, keywords and abstract, the user can make queries which span one or more document or citation entities through the advanced query interface.

Each search result includes the title, author names, publication information and a short snippet of the text containing the query terms. A javascript interface enables users to view the abstract of a document from a mouse over the down arrow associated with each search result. The process and content of the search results are

unique and can be ranked based on citations, relevance and recency.

SeerSuite applications use a Solr instance configured to index metadata fields defined by SeerSuite across documents in the collection. SeerSuite uses a simple ranking algorithm based on the default ranking algorithm provided by Solr for ranking results. These results are boosted based on the location of where the results were found and citations, e.g. a result containing the query term in the title or author names have higher rank. New ranking methods are readily implemented in Solr.

Relevance based searches are used by default in SeerSuite. The ability to rank documents based on the number of citations is an optional default.

SeerSuite uses author normalization for the metadata extracted from the documents to provide a comprehensive set of variations of the author names. The normalization allows authors to be searched for through name variants.

In addition to viewing results on the search results page, the user can subscribe to search feeds, which update the user when new results are available for a particular query.

6.2 Document Summary

The document view in figure 5, provides the user a view of metadata information aggregated for each document. This view contains extensive metadata about a individual document including - title, author names, venue of publication, year of publication and the citations contained in the document. The tabbed interface allows the user to browse citation relationships and the metadata version information. Link to the source of the document along with the option to download a cached copy are provided. In case the document is found at multiple URLs, the page lists all the alternate document URLs. The citation linking information provided along with the document includes links to documents cited by this document. The documents cited by the document are ranked by how well they are cited in the collection. A graph illustrating the citations to the document across years which can be useful for identifying trends and impact is also displayed.

The document summary page provides several MyCiteSeer interaction points. Add to Collection, Correct Errors and Monitor changes links allow the user to insert documents to his collection, correct metadata and monitor the document.

Corrections to the metadata involve several changes to the document metadata and citation relationships. SeerSuite examines these relationships, updating the citation graphs as necessary. It either creates or updates the citation cluster established for the current document after the corrections are submitted. A versioning system enables the administrator to track changes to document metadata. A user can view the modifications to the document through the versions tab, which displays all versions of the document metadata and the attribution for each update or correction.

The document view page contains data spanning multiple databases and the generation of this page is resource intensive. In addition to providing metadata, the docu-

Figure 5: Document Summary

ment also allows the user to download the bibtex of the article or collect the bibtex of the article in a meta cart for download later. The page also provides several bookmarking links and the ability to copy document data provided on the page using browser plugins.

6.3 Citation Relationships

The citation graph generated and stored as part of the ingestion process and updated as part of the maintenance process allows SeerSuite to provide users with tools for citation analysis. Citation based relationships such as active bibliography and co-citations are available for each document through the related documents tab in the document summary page. This relationship provides valuable information to users, allowing users to track documents of their interest. Such analysis is helpful in exploring topics and literature surveys. Active Bibliography provides links between documents citing the same set of documents and is one way grouping of documents. Another method is by identifying Co-citations which are links between documents which cite the same documents to a particular document. Figure 6 shows the Active Bibliography of a document in CiteSeer^x.

The citation graph is dynamic, changing as a result of corrections and other metadata updates. SeerSuite indexes the citation relationships with the document. Therefore, in addition to the database, rendering citation relationships such as active bibliography and co-citations requires queries to the database followed by queries to the index. The listing of each citation in the relationship is based on a similarity document measure implemented at the index. The links utilize the cluster ID, which is mapped to the document in case the document is available in the collection. In case where the document is not available, it points to the index listing of documents citing the citation entity. The citation relationships and ranking of authors, documents and citations are also summarized in a detailed year by year list in the statistics pages.

Figure 6: Active Bibliography

7 MyCiteSeer

SeerSuite aims to provide the user with services which improve the efficiency of the user in accessing information. MyCiteSeer plays a crucial role in providing and supporting these services. MyCiteSeer allows users to store queries, document portfolios, tag documents, and monitor and track documents of interest. The portal space is available after user registration and login. We briefly describe the user interface of MyCiteSeer.

Figure 7 shows the index page for MyCiteSeer. The index page serves as the landing page, with the menu providing links to other pages including the profile, collections tags and monitoring pages.

The profile page presents the user with interfaces to update information stored as part of his profile on MyCiteSeer including the password for the account. In the case where API support has been enabled, the profile page also allows the user to request an API key.

The collections page allows the user to view collections of documents stored within the account. These collections are user defined sets of documents, aggregated under their profile for ease of access. The user can use a collection to download bibliographic data for all documents in a collection.

Tags provide the user with a listing of tags defined by the user and link to the documents tagged with that tag. The tag portal page allows the user to view and delete tags defined by the user and the documents these tags are linked to.

The monitoring page allows users to track changes to a document in the SeerSuite collection. Any updates to document metadata, including the citation graph linked with the document for documents in the monitored collection are sent to the user through e-mail registered with the system.

In addition to providing the user with a portal, MyCiteSeer enables SeerSuite to utilize crowd sourcing or distributed error correction [29] for corrections to document metadata. By assessing weights based on prior corrections, the evidence based system can detect malicious changes.

The application program interface component utilizes MyCiteSeer user data for generating the access key and controlling access to services provided. An user marked as an administrator has additional functionality available to him through MyCiteSeer. A subset of configuration and administrative interfaces are made available through an admin console.

The portal framework shares the structure and storage with the main application. While the servlets for MyCiteSeer are developed with SeerSuite in mind, the interaction with SeerSuite applications can easily be extended to other projects and services. The MyCiteSeer component interacts with SeerSuite ingestion and maintenance modules, using listeners. The maintenance and ingestion service provide notifications on objects being updated or processed through these listeners.

8 Other Interfaces

8.1 OAI

The Open Archives Initiative (OAI) provides an efficient protocol metadata dissemination framework for data sources such as a SeerSuite. A low barrier mechanism, OAI is particularly suitable for SeerSuite applications, enabling instances that provide metadata sharing, publishing and archiving. SeerSuite supports interfaces

Figure 7: MyCiteSeer^x Index Page

compliant to the OAI-Protocol for metadata harvesting (PMH) [14] previously established with CiteSeer [36].

In the earlier CiteSeer system, modified CGI scripts were utilized for handling queries and generating compliant content for the OAI. In contrast, requests made to the SeerSuite OAI interface are handled by servlets, which translate requests into data access calls. The servlets assemble results in an OAI compliant manner, which is presented to the client. SeerSuite supports the full complement of OAI-PMH version 2 verb requests and provides content in the Dublin Core format. Thus, all document metadata is accessible to a client through the OAI interface.

8.2 API

Application Programmable Interfaces (APIs) are central for programmatic access to the repository. Through REST [20] web services, SeerSuite supports the needs of both programmers aiming to access metadata from the digital library and software agents looking to exchange information between digital libraries. SeerSuite API is a revamped version of the previously developed CiteSeer API [35] which was SOAP/WSDL-based.

The goal of the SeerSuite API is to share metadata with developers and software agents. Moving to a REST-based web service from a SOAP/WSDL-based one reduces the size of requests and the responses exchanged between the client and the server. Hence, developers can retrieve information faster, and total network traffic is reduced. The version of the API deployed in CiteSeer^x provides access to the papers, authors, citations, keywords, and citation contexts. The API caller may provide a SQL-like query to be executed as a filter on the matching set. The resource URI formats are shown in table 1. Objects are identified by document IDs (*docid*), author IDs (*aid*) and citation IDs (*cid*). SeerSuite API can output the results in both XML and JSON formats depending on the callers preference.

Type	URI Format
Paper	`http://host/papers/[docid]`
Author	`http://host/authors/[aid]`
Citation	`http://host/authors/[cid]`

Table 1: CiteSeer^x API Resource URI Formats

SeerSuite adopts Jersey [12] as a library to build the RESTful web service, which in turn implements the JAX-RS [13] reference. SeerSuite requires users looking to use the API to have a valid MyCiteSeer account. Account information in MyCiteSeer is used to generate an Application ID (*appid*) which has to be passed in every HTTP request as a mechanism of authentication. Daily limits for users are monitored and can be managed by administrators for performance.

9 Federation of Services

CiteSeer^x includes several unique services, which are not part of the SeerSuite application framework. Provisioning for these services is an unique aspect of the SeerSuite framework. Many of these services have evolved as a result of research and are still being developed. The developer builds and operates these services independently, sharing hosting infrastructure with the main application. Separate tables and databases and index operations maybe provisioned for each service. In the following sections, we briefly discuss Table search and author disambiguation search.

9.1 Table Search

Tables in documents often contain important data not present elsewhere. Table search services are based on TableSeer developed as part of the project [31]. Table search automatically extracts tables metadata, and indexes and ranks tables present in a SeerSuite collection. While table search shares components of the web application and shares the repository with SeerSuite, the index and extraction components are independent of SeerSuite.

The SeerSuite interface utilizes the main application framework for interaction with the table index. The queries results from the index are again processed and presented by the main application framework. Independent operation of the index from the main index allows for more efficient query processing and ranking of table

search results. The results utilize the SeerSuite file system infrastructure view the result of particular pages of the tables in cached documents. The ingestion, maintenance and updates services for Table search are independent of SeerSuite, allowing for flexibility in research and development. Some aspects of the table search ingestion system require access to the document metadata such as title, author not extracted as part of Table extraction, which are acquired from the main SeerSuite instance metadata.

Table search has served as a template for the development of similar services such as algorithm and figure search, which are in development.

9.2 Author Disambiguation

Author disambiguation enables users to identify whether records of publications in a SeerSuite collection refer to the same person. The author disambiguation service provided by SeerSuite is based on an efficient integrative framework for solving the name disambiguation problem. A blocking method retrieves candidate classes of authors with similar names and a clustering method, DBSCAN, clusters papers by author. The distance metric between papers used in DBSCAN is calculated by an online active selection Support Vector Machine algorithm(LASVM) [26]. This system has been utilized in CiteSeer^x. The disambiguation application identifies distinct authors based on header information which includes author affiliation and co-authorship.

The implementation makes use of already existing author object data in the main database, and generates cluster IDs for disambiguated authors that are stored in a separate table and index. Results for disambiguated author queries are handled by main application framework by interacting with the main database and the index. A profile page exists for each disambiguated author, with author affiliation, impact, and publications garnered from the SeerSuite instance. The profile page also provides a link, if available, to the author homepage obtained through a system HomePageSeer. An incremental algorithm replacing the offline batch algorithm currently used is in development.

10 Usage

CiteSeer^x receives nearly two million requests from across the globe. A significant portion of this traffic is as a result of document views, downloads and searches. An analysis of access logs is presented in this section.

The graph in figure 8 shows average hits per month for CiteSeer^x during the year 2009. The search grouping includes requests for document and author search.

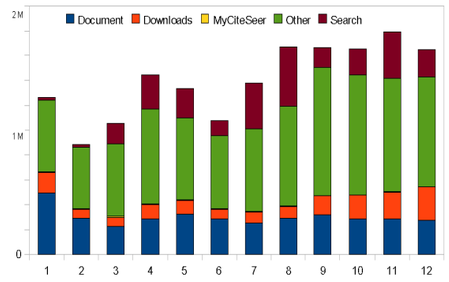


Figure 8: CiteSeer^x Traffic in 2009

The 'other' grouping includes queries for citing documents, legacy mappings (redirects from CiteSeer), OAI, and requests to author profile and static pages. Document related requests include downloads and summary views. The graph indicates a growth in the number of hits, driven by downloads, views of citation relationships, and search. The number of document views have grown by a lesser margin. During this time, the collection of documents in CiteSeer^x grew by 200,000 documents with updates to document metadata through corrections.

The majority of the referrals to CiteSeer^x are through pages hosted on CiteSeer^x (67 %). A number of users (29 %) arrive without a referrer,(i.e., users landing directly on CiteSeer^x or requests by crawlers). Redirection from CiteSeer contribute (1 %); references from Google, Google Scholar (2 %,1 %), Yahoo, and Bing (all <1 %).

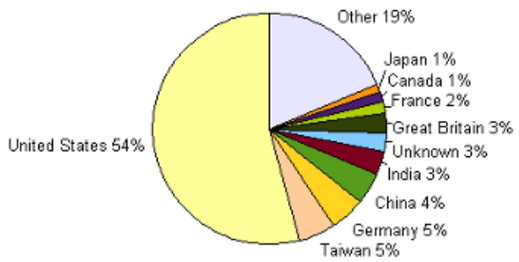


Figure 9: Country Profile

Figure 9 shows a division of traffic from different countries in 2009 identified using GeoIP. Among users of CiteSeer^x, nearly half of users are from the United States. Taiwan, Germany, China, India, UK, France, and Canada are other major sources of traffic. Traffic from two hundred and twenty countries are grouped under 'other'.

Along with valid accesses, CiteSeer^x experiences a variety of attacks every day. These attacks involve access to forbidden areas, portscans, SQL injection attacks, double decoding, buffer overflow, and cross scripting attempts.

11 Collaboration and Distribution

SeerSuite has been developed in collaboration with several groups across the world. This collaboration includes exchange of ideas, development of modules and hosting of services. Independent copies of CiteSeer^x are maintained by these research groups at University of Arkansas, National University of Singapore and King Saud University.

Data collected as part of crawls, document metadata, anonymized user log data are available on request. Several research groups have already taken advantage of these datasets. The source code for SeerSuite has been released under the Apache software license version 2 on sourceforge.

To improve adoption of SeerSuite based systems and provide the user an efficient way to explore and deploy instances. A virtual appliance with SeerSuite has been made available for such a purpose. The appliance uses open source components and contains a complete deployment of SeerSuite. This allows users to explore and operate instances of SeerSuite, in the pre-installed VM. The use of the virtual appliance also benefits developers and testers.

User feedback and comments have helped improve both user the experience and the troubleshooting bugs. SeerSuite enables other innovative and valuable tools to be built using metadata extracted and published. Such efforts include projects such as PaperCube [17] and JabRef.

12 Lessons Learned and Future Work

By adopting a multi-tier architecture, open source applications and the use of software frameworks, SeerSuite has improved upon the client server architecture initially adopted by CiteSeer. In addition this is an approach that has provided reliable scalable services in CiteSeer^x.

One of the lessons learned is in the provisioning of infrastructure. To allow SeerSuite instances to grow to large sizes without constraints on storage and computation, virtualization and distributed computing need to be utilized and managed.

The number of requests for metadata and data from the CiteSeer^x collection emphasizes the need for developing automated methods data sharing such as API and OAI services for SeerSuite instances. There have also been requests for a content based similarity and duplicate detection service, which is under development.

With the new design adding features in SeerSuite is more straight forward. However, further work on separating components is required to support systems which make use of a smaller subset of services. These systems may not include the citation graph service. Among other services, MyCiteSeer should be developed as a stand alone service. This will easily enable other services such as HomePageSeer that take advantage of login based access. Development of MyCiteSeer as an independent service, shared across many Seer instances is under consideration. This would allow users to share information across projects and instances.

One of the continuing major challenges is new and high quality metadata extraction. Another is that modules will need to be refactored to support massive crawls using parallelization at the module level.

A number of optimizations have been implemented in SeerSuite, prominent among these is the citation relationships stored in the index. This allows citation relationship queries which form document summary views to be more readily handled. Such an optimization has the drawback in that updates to these relationships are only available to the user once there has been updates to the index.

A federation of services model adopted for newer services benefits both users and developers. This model gives users a peek into new features and researchers an easy a way to include new services. From a development standpoint, this is also useful since services can be tested and users can provide feedback.

Improvements to the User Interface will be required to support upcoming features. Ranking of results and improving relevance in search are active topics of research. Performance analysis tests for SeerSuite to identify possible improvements and issues are being designed.

13 Summary

We have described the design, architecture, and deployment of SeerSuite. We believe SeerSuite overcomes many of the issues in an earlier system, CiteSeer, which, due to its design, limited growth and extensions to other services. SeerSuite is designed to take advantage of open source applications, frameworks and state of the art components. It also allow users to readily build mashups and related applications. The use of loose coupling of modules and federated services enables SeerSuite to easily offer new features and components.

The user interface allows search, document summary views, and citations clustering. We identify workflows for generating these pages. Various features such as tagging, building collections and correcting documents were identified for the MyCiteSeer portal. In addition to providing services through the web user interface, SeerSuite also provides services through the OAI/PMH and API interfaces.

The statistics and usage pattern for CiteSeer^x, a SeerSuite instance, provide information about growth in traffic and the profile of users based on the country of origin. We show that SeerSuite is a collaborative venture with open source code and virtual appliances that encourage adoption and research. We believe that SeerSuite will continue to improve and support a wide variety of services and user needs while remaining scalable, reliable and robust.

14 Acknowledgments

We gratefully acknowledge partial support from the NSF and useful comments by Yves Petinot.

References

[1] ACM Portal. http://portal.acm.org/portal.cfm.

[2] ActiveMQ. http://activemq.apache.org/.

[3] Chem$_x$Seer. http://chemxseer.ist.psu.edu.

[4] CiteSeer. http://citeseer.ist.psu.edu.

[5] CiteSeer^x. http://citeseerx.ist.psu.edu.

[6] CiteULike. http://www.citeulike.org/.

[7] DBLP. http://www.informatik.uni-trier.de/~ley/db/.

[8] Dspace. http://www.dspace.org/.

[9] Google Scholar. http://scholar.google.com/.

[10] Greenstone. http://www.greenstone.org/.

[11] Heritrix. http://crawler.archive.org/.

[12] Jersey API. https://jersey.dev.java.net/.

[13] JSR 311. https://jsr311.dev.java.net/nonav/releases/1.1/index.html.

[14] Open archives initiative - protocol for metadata harvesting v.2.0. http://www.openarchives.org/OAI/openarchivesprotocol.html.

[15] RePEc. http://repec.org/.

[16] Solr. http://lucene.apache.org/solr/.

[17] BERGSTROM, P. Papercube. http://papercube.peterbergstrom.com.

[18] COUNCILL, I. G., GILES, C. L., IORIO, E. D., GORI, M., MAGGINI, M., AND PUCCI, A. Towards next generation citeseer: A flexible architecture for digital library deployment. In *Research and Advanced Technology for Digital Libraries, ECDL 2006* (2006), pp. 111–122.

[19] COUNCILL, I. G., LI, H., ZHUANG, Z., DEBNATH, S., BOLELLI, L., LEE, W. C., SIVASUBRAMANIAM, A., AND GILES, C. L. Learning metadata from the evidence in an on-line citation matching scheme. In *JCDL* (2006), pp. 276–285.

[20] FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, Irvine, California, 2000.

[21] GARFIELD, E. "Science Citation Index"a new dimension in indexing. *Science 144*, 3619 (1984), 649 – 654.

[22] GILES, C. L., BOLLACKER, K. D., AND LAWRENCE, S. Citeseer: An automatic citation indexing system. In *Digital Libraries* (1998), pp. 89–98.

[23] GINSPARG, P. Can Peer Review be better Focussed. http://people.ccmr.cornell.edu/~ginsparg/blurb/pg02pr.html.

[24] HAN, H., GILES, C. L., MANAVOGLU, E., ZHA, H., ZHANG, Z., AND FOX, E. A. Automatic document metadata extraction using support vector machines. In *JCDL '03: Proceedings of the 3rd ACM/IEEE-CS joint conference on Digital libraries* (2003), pp. 37–48.

[25] HAN, H., MANAVOGLU, E., ZHA, H., TSIOUTSIOULIKLIS, K., GILES, C. L., AND ZHANG, X. Rule-based word clustering for document metadata extraction. In *SAC* (2005), pp. 1049–1053.

[26] HUANG, J., ERTEKIN, S., AND GILES, C. L. Efficient name disambiguation for large scale databases. In *The 10th European Conference on Principles and Practice of Knowledge Discovery in Databases* (2006), pp. 536–544.

[27] ISAAC COUNCILL, C. L. G., AND KAN, M.-Y. Parscit: an open-source crf reference string parsing package. In *Proceedings of the Sixth International Language Resources and Evaluation (LREC'08)* (Marrakech, 2008), European Language Resources Association.

[28] KAHN, R., AND WILENSKY, R. A framework for distributed digital object services. *International Journal on Digital Libraries 6*, 2 (2006), 115–123.

[29] LAWRENCE, S., BOLLACKER, K., AND GILES, C. L. Distributed error correction. In *DL '99: Proceedings of the fourth ACM conference on Digital libraries* (1999), p. 232.

[30] LI, H., COUNCILL, I., LEE, W.-C., AND GILES, C. L. Citeseerx: an architecture and web service design for an academic document search engine. *Poster Session 15th International World Wide Web Conference* (2006).

[31] LIU, Y., BAI, K., MITRA, P., AND GILES, C. L. Tableseer: automatic table metadata extraction and searching in digital libraries. In *JCDL* (2007), pp. 91–100.

[32] MCCALLUM, A., FREITAG, D., AND PEREIRA, F. C. N. Maximum entropy markov models for information extraction and segmentation. In *ICML* (2000), pp. 591–598.

[33] MITRA, P., GILES, C. L., SUN, B., AND LIU, Y. Chemxseer: a digital library and data repository for chemical kinetics. In *CIMS '07: Proceedings of the ACM first workshop on CyberInfrastructure: Information Management in eScience* (2007), pp. 7–10.

[34] NII, H. P. Blackboard systems, part one: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine 7*, 2 (1986), 38–53.

[35] PETINOT, Y., GILES, C. L., BHATNAGAR, V., TEREGOWDA, P. B., HAN, H., AND COUNCILL, I. Citeseer-api: towards seamless resource location and interlinking for digital libraries. In *CIKM* (2004), pp. 553–561.

[36] PETINOT, Y., TEREGOWDA, P. B., HAN, H., GILES, C. L., LAWRENCE, S., RANGASWAMY, A., AND PAL, N. ebizsearch: an oai-compliant digital library for ebusiness. In *JCDL* (2003), pp. 199–209.

[37] TAN, Q., MITRA, P., AND GILES, C. Metadata extraction and indexing for map search in web documents. In *Proceeding of the 17th ACM CIKM* (2008), pp. 1367–1368.