SigFree: A Signature-free Buffer Overflow Attack Blocker

[1]Xinran Wang [2]Chi-Chun Pan [2]Peng Liu [1,2]Sencun Zhu
[1]*Department of Computer Science and Engineering*
[2]*College of Information Sciences and Technology*
The Pennsylvania State University, University Park, PA 16802
{*xinrwang, szhu*}*@cse.psu.edu;* {*cpan, pliu*}*@ist.psu.edu*

Abstract

We propose SigFree, a realtime, signature-free, out-of-the-box, application layer blocker for preventing buffer overflow attacks, one of the most serious cyber security threats. SigFree can filter out code-injection buffer overflow attack messages targeting at various Internet services such as web service. Motivated by the observation that buffer overflow attacks typically contain executables whereas legitimate client requests never contain executables in most Internet services, SigFree blocks attacks by detecting the presence of code. SigFree first blindly dissembles and extracts instruction sequences from a request. It then applies a novel technique called *code abstraction*, which uses data flow anomaly to prune useless instructions in an instruction sequence. Finally it compares the number of useful instructions to a threshold to determine if this instruction sequence contains code. SigFree is signature free, thus it can block new and unknown buffer overflow attacks; SigFree is also immunized from most attack-side code obfuscation methods. Since SigFree is transparent to the servers being protected, it is good for economical Internet wide deployment with very low deployment and maintenance cost. We implemented and tested SigFree; our experimental study showed that SigFree could block all types of code-injection attack packets (above 250) tested in our experiments. Moreover, SigFree causes negligible throughput degradation to normal client requests.

1 Introduction

Throughout the history of cyber security, buffer overflow is one of the most serious vulnerabilities in computer systems. Buffer overflow vulnerability is a root cause for most of the cyber attacks such as server breaking-in, worms, zombies, and botnets. Buffer overflow attacks are the most popular choice in these attacks, as they provide substantial control over a victim host [37].

"A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. This causes the data to overwrite into adjacent memory locations, and, depending on what is stored there, the behavior of the program itself might be affected." [34] (Note that the buffer could be in stack or heap.) Although taking a broader viewpoint, buffer overflow attacks do not always carry code in their attacking requests (or packets)[1], code-injection buffer overflow attacks such as stack smashing count for probably most of the buffer overflow attacks that have happened in the real world.

Although tons of research has been done to tackle buffer overflow attacks, existing defenses are still quite limited in meeting four highly-desired requirements: (R1) *simplicity* in maintenance; (R2) *transparency* to existing (legacy) server OS, application software, and hardware; (R3) *resiliency* to obfuscation; (R4) economical Internet wide deployment. As a result, although several very secure solutions have been proposed, they are not pervasively deployed, and a considerable number of buffer overflow attacks continue to succeed on a daily basis.

To see how existing defenses are limited in meeting these four requirements, let us break down the existing buffer overflow defenses into six classes which we will review shortly in Section 2: (1A) Finding bugs in source code. (1B) Compiler extensions. (1C) OS modifications. (1D) Hardware modifications. (1E) Defense-side obfuscation [11, 28]. (1F) Capturing code running symptoms of buffer overflow attacks [21, 37, 43, 55]. (Note that the above list does not include binary code analysis based defenses which we will address shortly.) We may briefly summarize the limitations of these defenses in terms of the four requirements as follows. (a) Class 1B, 1C, 1D, and 1E defenses may cause substantial changes to existing (legacy) server OSes, application softwares, and hardwares, thus they are not transparent. Moreover, Class 1E defenses generally cause processes to be terminated. As a result, many businesses do not view these changes and the process termination overhead as economical deployment. (b) Class 1F defenses can be very secure, but they either suffer from significant runtime overhead or need special auditing or diagnosis facilities which are not commonly available in commercial services. As a result, Class 1F defenses have limited transparency and potential for economical deployment. (c) Class 1A defenses need source code, but source code is unavailable to many legacy applications.

Besides buffer overflow defenses, worm signatures can be generated and used to block buffer overflow attack packets [29, 42, 47]. Nevertheless, they are also limited in meeting the four requirements, since they either rely on signatures, which introduce maintenance overhead, or are not very resilient to attack-side obfuscation.

To overcome the above limitations, in this paper we propose SigFree, a realtime buffer overflow attack blocker, to protect Internet services. The idea of SigFree is motivated by an important observation that "the nature of communication to and from network services is predominantly or exclusively data and not executable code." [15]. In particular, as summarized in [15], (a) on Windows platforms, most web servers (port 80) accept data only; remote access services (ports 111, 137, 138, 139) accept data only; Microsoft SQL Servers (port 1434) accept data only; workstation services (ports 139 and 445) accept data only. (b) On Linux platforms, most Apache web servers (port 80) accept data only; BIND (port 53) accepts data only; SNMP (port 161) accepts data only; most Mail Transport (port 25) accepts data only; Database servers (Oracle, MySQL, PostgreSQL) at ports 1521, 3306 and 5432 accept data only.

Since remote exploits are typically executable code, this observation indicates that if we can precisely distinguish (service requesting) messages that contain code from those that do not contain any code, we can protect most Internet services (which accept data only) from code-injection buffer overflow attacks by blocking the messages that contain code.

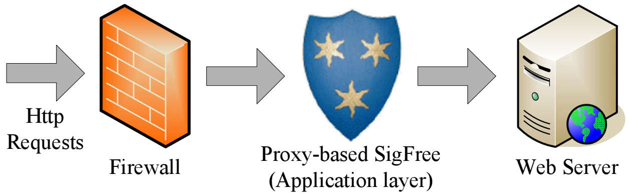


Figure 1: SigFree is an application layer blocker between the web server and the corresponding firewall.

Accordingly, SigFree (Figure 1) works as follows. SigFree is an application layer blocker that typically stays between a service and the corresponding firewall. When a service requesting message arrives at SigFree, SigFree first uses a new $O(N)$ algorithm, where N is the byte length of the message, to disassemble and distill all possible instruction sequences from the message's payload, where every byte in the payload is considered as a possible starting point of the code embedded (if any). However, in this phase some data bytes may be mistakenly decoded as instructions. In phase 2, SigFree uses a novel technique called *code abstraction*. Code abstraction first uses data flow anomaly to prune useless instructions in an instruction sequence, then compares the number of useful instructions to a threshold to determine if this instruction sequence (distilled in phase 1) contains code.

We have implemented a SigFree prototype as a proxy to protect web servers. Our empirical study shows that there exists clean-cut "boundaries" between code-embedded payloads and data payloads when our code-data separation criteria are applied. We have identified the "boundaries" (or thresholds) and been able to detect/block all 50 attack packets generated by Metasploit framework [4], all 200 polymorphic shellcode packets generated by two well-known polymorphic shellcode engine ADMmutate [40] and CLET [23], and worm Slammer, CodeRed and a CodeRed variation, when they are well mixed with various types of data packets. Also, our experiment results show that the throughput degradation caused by SigFree is negligible.

The merits of SigFree are summarized below. They show that SigFree has taken a main step forward in meeting the four requirements aforementioned.

⊙ *SigFree is signature free, thus it can block new and unknown buffer overflow attacks*

⊙ Without relying on string-matching, *SigFree is immunized from most attack-side obfuscation methods.*

⊙ *SigFree uses generic code-data separation criteria instead of limited rules.* This feature separates SigFree from [15], an independent work that tries to detect code-embedded packets.

⊙ *Transparency.* SigFree is a out-of-the-box solution that requires no server side changes.

⊙ *SigFree has negligible throughput degradation.*

⊙ *SigFree is an economical deployment with very low maintenance cost*, which can be well justified by the aforementioned features.

The rest of the paper is organized as follows. In Section 2, we summarize the work related to ours. In Section 3, we give an overview of SigFree. In Sections 4 and 5, we introduce the instruction sequence distiller component and the instruction sequence analyzer component of SigFree, respectively. In Section 6, we show our experimental results. Finally, we discuss some research issues in Section 7 and conclude the paper in Section 8.

2 Related Work

SigFree is mainly related to three bodies of work. [Category 1:] prevention/detection techniques of buffer overflows; [Category 2:] worm detection and signature generation. [Category 3:] machine code analysis for security purposes. In the following, we first briefly review Category 1 and Category 2 which are less close to SigFree. Then we will focus on comparing SigFree with Category 3.

2.1 Prevention/Detection of Buffer Overflows

Existing prevention/detection techniques of buffer overflows can be roughly broken down into six classes:

Class 1A: Finding bugs in source code. Buffer overflows are fundamentally due to programming bugs. Accordingly, various bug-finding tools [14, 24, 51] have been developed. The bug-finding techniques used in these tools, which in general belong to static analysis, include but not limited to model checking and bugs-as-deviant-behavior. Class 1A techniques are designed to handle source code only, and they do not ensure completeness in bug finding. In contrast, SigFree handles machine code embedded in a request (message).

Class 1B: Compiler extensions. "If the source code is available, a developer can add buffer overflow detection automatically to a program by using a modified compiler." [34] Three such compilers are StackGuard [22], ProPolice [2], and Return Address Defender (RAD) [18]. In addition, Smirnov and Chiueh proposed compiler DIRA [49] can detect control hijacking attacks, identify the malicious input and repair the compromised program. Class 1B techniques require the availability of source code. In contrast, SigFree does not need to know any source code.

Class 1C: OS modifications. Modifying some aspects of the operating system may prevent buffer overflows such as Pax [9], LibSafe [10] and e-NeXsh [48]. Class 1C techniques need to modify the OS. In contrast, SigFree does not need any modification of the OS.

Class 1D: Hardware modifications. A main idea of hardware modification is to store all return addresses on the processor [41]. In this way, no input can change any return address.

Class 1E: Defense-side obfuscation. Address Space Layout Randomization (ASLR) is a main component of PaX [9]. Bhatkar and Sekar [13] proposed a comprehensive address space randomization scheme. Address-space randomization, in its general form [13], can detect exploitation of all memory errors. Instruction set randomization [11, 28] can detect all code injection attacks. Nevertheless, when these approaches detect an attack, the victim process is typically terminated. "Repeated attacks will require repeated and expensive application restarts, effectively rendering the service unavailable." [37]

Class 1F: Capturing code running symptoms of buffer overflow attacks. Fundamentally, buffer overflows are a code running symptom. If such unique symptoms can be precisely captured, all buffer overflows can be detected. Class 1B, Class 1C and Class 1E techniques can capture some - but not all - of the running symptoms of buffer overflows. For example, accessing non-executable stack segments can be captured by OS modifications; compiler modifications can detect return address rewriting; and process crash is a symptom captured by defense-side obfuscation. To achieve 100% coverage in capturing buffer overflow symptoms, dynamic dataflow/taint analysis/program shepherding techniques were proposed in Vigilante [21], TaintCheck [43], and [30]. They can detect buffer overflows during runtime. However, it may cause significant runtime overhead (e.g., 1,000%). To reduce such overhead, another type of Class 1F techniques, namely post-crash symptom diagnosis, has been developed in Covers [37] and [55]. Post-crash symptom diagnosis extracts the 'signature' after a buffer overflow attack is detected. Recently, Liang and Sekar proposed ARBOR [36] which can automatically generate vulnerability-oriented signatures by identifying characteristic features of attacks and using program context. Moreover, ARBOR automatically invokes the recovery actions. Class 1F techniques can block both the attack requests that contain code and the attack requests that do not contain any code, but they need the signatures to be firstly generated. Moreover, they either suffer from significant runtime overhead or need special auditing or diagnosis facilities which are not commonly available in commercial services. In contrast, although SigFree could not block the attack requests that do not contain any code, SigFree is signature-free and does not need any changes to real world services. We will investigate the integration of SigFree with Class 1F techniques in our future work.

2.2 Worm Detection and Signature Generation

Because buffer overflows are a key target of worms when they propagate from one host to another, SigFree is related to worm detection. Based on the nature of worm infection symptoms, worm detection techniques can be broken down into three classes: [Class 2A] techniques use such macro symptoms as Internet background radiation (observed by network telescopes) to raise early warnings of Internet wide worm infection [44]. [Class 2B] techniques use such local traffic symptoms as content invariance, content prevalence and address dispersion to generate worm signatures and/or block worms. Some examples of Class 2B techniques are Earlybird [47], Autograph [29], Polygraph [42], and TRW [27]. [Class 2C] techniques use worm code running symptoms to detect worms. It is not surprising that Class 2C techniques are exactly Class 1F techniques. Some example Class 2C techniques are Shield [52], Vigilante [21], COVERS [37]. [Class 2D] techniques use anomaly detection on packet payload to detect worms and generate signature. Wang and Stolfo [54] first proposed Class 2D techniques called PAYL. PAYL is first trained with normal network flow traffic and then uses some byte-level statistical measures to detect exploit code. Recently, Wang et al. [53] proposed new features of PAYL that based on ingress/egress anomalous payload correlation to detect new worms and automatically generate signatures. FLIPS [39] uses PAYL [54] to detect anomalous inputs. If the anomaly is confirmed by a detector, a content-based signature is generated.

Class 2A techniques are not relevant to SigFree. Class 2C techniques are already discussed. Class 2D techniques could be evaded by statistically mimics normal traffic [31]. Class 2B techniques rely on signatures, while SigFree is signature-free. Class 2B techniques focus on identifying the unique bytes that a worm packet must carry, while SigFree focuses on determining if a packet contains code or not. Exploiting the content invariance property, Class 2B techniques are typically not very resilient to obfuscation. In contrast, SigFree is immunized from most attack-side obfuscation methods.

2.3 Machine Code Analysis for Security Purposes

Although source code analysis has been extensively studied (see Class 1A), in many real world scenarios source code is not available and the ability to analyze binaries is desired. Machine code analysis has three main security purposes: (P1) malware detection, (P2) to analyze obfuscated binaries, and (P3) to identify and analyze the code contained in buffer overflow attack packets. Along purpose P1, Chritodorescu and Jha [16] proposed static analysis techniques to detect malicious patterns in executables, and Chritodorescu et al. [17] exploited semantic heuristics to detect obfuscated malware. Along purpose P2, Lakhotia and Eric [35] used static analysis techniques to detect obfuscated calls in binaries, and Kruegel et al. [33] investigated disassembly of obfuscated binaries.

SigFree differs from P1 and P2 techniques in design goals. The purpose of SigFree is to see if a message contains code or not, instead of determining if a piece of code has malicious intent or not. (Note that SigFree does not check if the code contained in a message has malicious intent.) Due to this reason, SigFree is immunized from most attack-side obfuscation methods. Nevertheless, both the techniques in [33] and SigFree disassemble binary code, although their disassembly procedures are different. As will be seen, disassembly is not the kernel contribution of SigFree.

Fnord [2], the preprocessor of Snort IDS, identifies exploit code by detecting NOP sled. Toth and Kruegel [50] also aim at detecting NOP sled. They employed binary disassembly to find the sequence of execution instructions as an evidence of a NOP sled. However, Some attacks such as worm CodeRed do not include NOP sled and, as mentioned in [15], mere binary disassembly is not adequate. Moreover, polymorphic shellcode [23, 40] can bypass the detection for NOP instructions by using fake NOP zone. SigFree does not rely on the detection of NOP sled.

Finally, being generally a P3 technique, SigFree is most relevant to two P3 works [15, 32]. Kruegel et al. [32] innovatively exploited control flow structures to detect polymorphic worms. Unlike string-based signature matching, their techniques identify structural similarities between different worm mutations and use these similarities to detect more polymorphic worms. The implementation of their approach is resilient to a number of code transformation techniques. Although their techniques also handle binary code, they perform offline analysis. In contrast, SigFree is an online attack blocker. As such, their techniques and SigFree are complementary to each other with different purposes. Moreover, unlike SigFree, their techniques [32] may not be suitable to block the code contained in *every* attack packet, because some buffer overflow code is so simple that very little control flow information can be exploited.

Independent of our work, Chinchani and Berg [15] proposed a rule-based scheme to achieve the same goal of SigFree, that is, to detect exploit code in network flows. However, there is a fundamental difference between SigFree and their scheme [15]. Their scheme is rule-based, whereas SigFree is a *generic* approach which does not require any pre-known patterns. More specifically, their scheme [15] first tries to find certain pre-known instructions, instruction patterns or control flow structures in a packet. Then, it uses the found patterns and a data flow analysis technique called program slicing to analyze the packet's payload to see if the packet really contains code. Four rules (or cases) are discussed in their paper: Case 1 not only assumes the occurrence of the call/jmp instructions, but also expects the push instruction appears before the branch; Case 2 relies on the *interrupt* instruction; Case 3 relies on instruction *ret*; Case 4 exploits hidden branch instructions. Besides, they used a special rule to detect polymorphic exploit code which contains a loop. Although they mentioned that the above rules are initial sets and may require updating with time, it is always possible for attackers to bypass those pre-known rules. Moreover, more rules mean more overhead and longer latency in filtering packets. In contrast, SigFree exploits a different data flow analysis technique, which is much harder for exploit code to evade.

3 SigFree Overview

3.1 Basic Definitions and Notations

This section provides the definitions that will be used in the rest of the paper.

Definition 1 *(instruction sequence) An instruction sequence is a sequence of CPU instructions which has one and only one entry instruction and there exist at least one execution path from the entry instruction to any other instruction in this sequence.*

An instruction sequence is denoted as s_i, where i is the entry address of the instruction sequence. A fragment of a program in machine language is an instruction sequence, but an instruction sequence is not necessarily a fragment of a program. In fact, we may distill instruction sequences from any binary strings. This poses the fundamental challenge to our research goal. Figure 2 shows four instruction sequences distilled from a substring of a GIF file. These four instruction sequences are not fragments of a real program, although they may also be executed in a specific CPU. Below we call them *random instruction sequences*, whereas use the term *binary executable code* to refer to a fragment of a real program in machine language.

Definition 2 *(instruction flow graph) An instruction flow graph (IFG) is a directed graph $G = (V, E)$ where each node $v \in V$ corresponds to an instruction and each edge $e = (v_i, v_j) \in E$ corresponds to a possible transfer of control from instruction v_i to instruction v_j.*

Unlike traditional control flow graph (CFG), a node of an IFG corresponds to a single instruction rather than a basic block of instructions. To completely model the control flow of an instruction sequence, we further extend the above definition.

Definition 3 *(extended instruction flow graph) An extended instruction flow graph (EIFG) is a directed graph $G = (V, E)$ which satisfies the following properties: each node $v \in V$ corresponds to an instruction, an illegal instruction, or an external address; each edge $e = (v_i, v_j) \in E$ corresponds to a possible transfer of control from instruction v_i to instruction v_j, to illegal instruction v_j, or to an external address v_j.*

Accordingly, we name the types of nodes in an EIFG *instruction node*, *illegal instruction node*, and *external address node*.

The reason that we define IFG and EIFG is to model two special cases which CFG cannot model (the difference will be very evident in the following sections). First, in an instruction sequence, control may be transferred from an instruction node to an illegal instruction node. For example, in instruction sequence s_{08} in Figure 2, the transfer of control is from instruction “lods [ds:esi]” to an illegal instruction at address $0F$. Second, control may be transferred from an instruction node to an external address node. For example, instruction sequence s_{00} in Figure 2 has an instruction “jmp ADAAC3C2”, which jumps to external address ADAAC3C2.

3.2 Attack Model

An attacker exploits a buffer overflow vulnerability of a web server by sending a crafted request, which contains a malicious payload. Figure 3 shows the format of a HTTP request. There are several HTTP request methods among which GET and POST are most often used by attackers. Although HTTP 1.1 does not allow GET to have a request body, some web servers such as Microsoft IIS still dutifully read the request-body according to the request-header's instructions (the CodeRed worm exploited this very problem).

The position of a malicious payload is determined by the exploited vulnerability. A malicious payload may be embedded in the Request-URI field as a query parameter. However, as the maximum length of Request-URI is limited, the size of a malicious payload, hence the behavior of such a buffer overflow attack, is constrained. It is more common that a buffer overflow attack payload is embedded in Request-Body of a POST method request. Technically, a malicious payload may also be embedded in Request-Header, although this kind of attacks have not been observed yet. In this work, we assume an attacker can use any request method and embed the malicious code in any field.

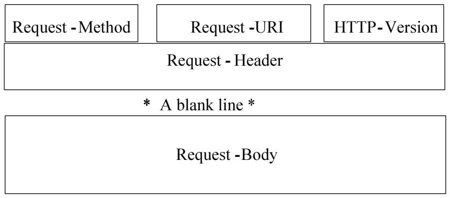


Figure 3: A HTTP Request. A malicious payload is normally embedded in Request-URI or Request-Body

3.3 Assumptions

In this paper, we focus on buffer overflow attacks whose payloads contain executable code in machine language, and we assume normal requests do not contain

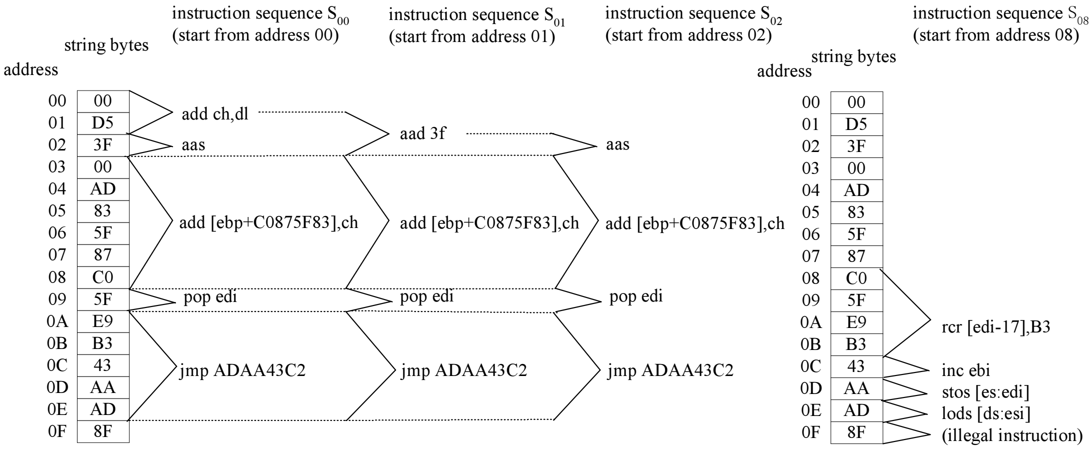


Figure 2: Instruction sequences distilled from a substring of a GIF file. We assign an address to every byte of the string. Instruction sequences s_{00}, s_{01}, s_{02} and s_{08} are distilled by disassembling the string from addresses 00, 01, 02 and 08, respectively.

executable machine code. A normal request may contain any data, parameters, or even a SQL statement. Note that although SQL statements are executable in the application level, they cannot be executed directly by a CPU. As such, SQL statements are not viewed as executable in our model. Application level attacks such as data manipulation and SQL injection are out of the scope.

Though SigFree is a generic technique which can be applied to any instruction set, for concreteness we assume the web server runs the Intel IA32 instruction set, the most popular instruction set running inside a web server today.

3.4 Architecture

Figure 4 depicts the architecture of SigFree and it is comprised of the following modules:

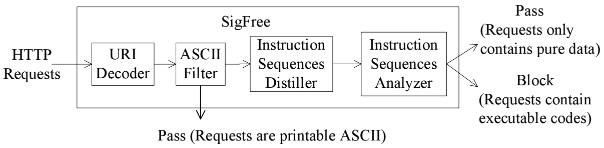

SigFree
HTTP Requests
URI Decoder
ASCII Filter
Instruction Sequences Distiller
Instruction Sequences Analyzer
Pass (Requests only contains pure data)
Block (Requests contain executable codes)
Pass (Requests are printable ASCII)

Figure 4: The architecture of SigFree

URI decoder. The specification for URLs [12] limits the allowed characters in a Request-URI to only a subset of the ASCII character set. This means that the query parameters of a request-URI beyond this subset should be encoded [12]. Because a malicious payload may be embedded in the request-URI as a request parameter, the first step of SigFree is to decode the request-URI.

ASCII Filter. Malicious executable code are normally binary strings. In order to guarantee the throughput and response time of the protected web system, if the query parameters of the request-URI and request-body of a request are both printable ASCII ranging from 20-7E in hex, SigFree allows the request to pass (In Section 7.2, we will discuss a special type of executable codes called alphanumeric shellcodes [45] that actually use printable ASCII) .

Instruction sequences distiller (ISD). This module distills all possible instruction sequences from the query parameters of Request-URI and Request-Body (if the request has one).

Instruction sequences analyzer (ISA). Using all the instruction sequences distilled from the instruction sequences distiller as the inputs, this module analyzes these instruction sequences to determine whether one of them is (a fragment of) a program.

4 Instruction Sequence Distiller

This section first describes an effective algorithm to distill instruction sequences from http requests, followed by several excluding techniques to reduce the processing overhead of instruction sequences analyzer.

4.1 Distilling Instruction Sequences

To distill an instruction sequence, we first assign an address to every byte of a request. Then, we disassemble the request from a certain address until the end of the request is reached or an illegal instruction opcode is encountered. There are two traditional disassembly algorithms: *linear sweep* and *recursive traversal* [38, 46]. The linear sweep algorithm begins disassembly at a certain address, and proceeds by decoding each encountered instruction. The recursive traversal algorithm also begins disassembly at a certain address, but it follows the control flow of instructions.

In this paper, we employ the recursive traversal algorithm, because it can obtain the control flow information during the disassembly process. Intuitively, to get all possible instruction sequences from a N-byte request, we simply execute the disassembly algorithm N times and each time we start from a different address in the request. This gives us a set of instruction sequences. The running time of this algorithm is $O(N^2)$.

One drawback of the above algorithm is that the same instructions are decoded many times. For example, instruction “pop edi” in Figure 2 is decoded many times by this algorithm. To reduce the running time, we design a memorization algorithm [20] by using a data structure, which is an EIFG defined earlier, to represent the instruction sequences. To distill all possible instruction sequences from a request is simply to create the EIFG for the request. An EIFG is used to represent all possible transfers of control among these instructions. In addition, we use an instruction array to represent all possible instructions in a request. To traverse an instruction sequence, we simply traverse the EIFG from the entry instruction of the instruction sequence and fetch the corresponding instructions from the instruction array. Figure 5 shows the data structure for the request shown in Figure 2. The details of the algorithm for creating the data structure are described in Algorithm 1. Clearly, the running time of this algorithm is $O(N)$, which is optimal as each address is traversed only once.

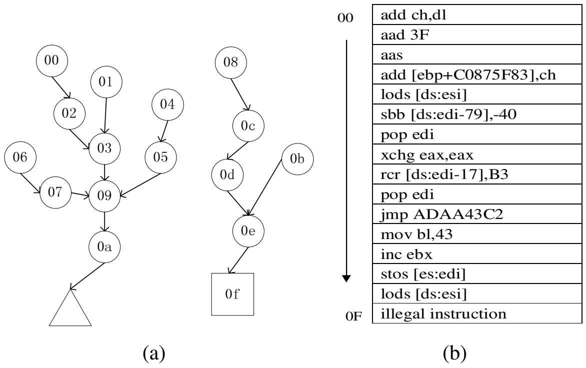


Figure 5: Data structure for the instruction sequences distilled from the request in Figure 2. (a) Extended instruction flow graph. Circles represent instruction nodes; triangles represent external addresses; rectangles represent illegal instructions. (b) The array of all possible instructions in the request.

4.2 Excluding Instruction Sequences

The previous step may output many instruction sequences at different entry points. Next we exclude some of them based on several heuristics. Here *excluding an instruction sequence means that the entry of this sequence is not considered as the real entry for the embedded code (if any).*

Algorithm 1 Distill all instruction sequences from a request

```
initialize EISG $G$ and instruction array $A$ to empty
for each address $i$ of the request do
    add instruction node $i$ to $G$
$i \leftarrow$ the start address of the request
while $i <=$ the end address of the request do
    $inst \leftarrow$ decode an instruction at $i$
    if $inst$ is illegal then
        $A[i] \leftarrow$ illegal instruction $inst$
        set type of node $i$ “illegal node” in $G$
    else
        $A[i] \leftarrow$ instruction $inst$
        if $inst$ is a control transfer instruction then
            for each possible target t of $inst$ do
                if target $t$ is an external address then
                    add external address node $t$ to $G$
                add edge e(node $i$, node $t$) to G
        else
            add edge e(node $i$, node $i + inst.length$) to G
    $i \leftarrow i + 1$
```

The fundamental rule in excluding instruction sequences is not to affect the decision whether a request contains code or not. This rule can be translated into the following technical requirements: if a request contains a fragment of a program, the fragment must be one of the remaining instruction sequences or a subsequence of a remaining instruction sequence, or it differs from a remaining sequence only by few instructions.

Step 1 If instruction sequence s_a is a subsequence of instruction sequence s_b, we exclude s_a. The rationale for excluding s_a is that if s_a satisfies some characteristics of programs, s_b also satisfies these characteristics with a high probability.

This step helps exclude lots of instruction sequences since many distilled instruction sequences are subsequences of the other distilled instruction sequences. For example, in Figure 5(a), instruction sequence s_{02}, which is a subsequence of instruction sequence s_{00}, can be excluded. Note that here we only exclude instruction sequence s_{02} rather than remove node v_{02}. Similarly, instruction sequences s_{03},s_{05}, s_{07}, s_{09},s_{0a},s_{0c},s_{0d} and s_{0e} can be excluded.

Step 2 If instruction sequence s_a merges to instruction sequence s_b after a few instructions (e.g., 4 in our experiments) and s_a is no longer than s_b, we exclude s_a. It is reasonable to expect that s_b will preserve s_a’s characteristics.

Many distilled instruction sequences are observed to merge to other instructions sequences after a few instructions. This property is called self-repairing [38] in Intel IA-32 architecture. For example, in Figure 5(a) instruction sequence s_{01} merges to instruction sequence s_{00}

only after one instruction. Therefore, s_{01} is excluded. Similarly, instruction sequences s_{04}, s_{06} and s_{0b} can be excluded.

Step 3 For some instruction sequences, if we execute them, whatever execution path being taken, an illegal instruction is *inevitably reachable*. We say an instruction is inevitably reachable if two conditions holds. One is that there are no cycles (loops) in the EIFG of the instruction sequence; the other is that there are no external address nodes in the EIFG of the instruction sequence.

We exclude the instruction sequences in which illegal instructions are inevitably reachable, because causing the server to execute an illegal instruction is not the purpose of an buffer overflow attack (this assumption was also made by others [15, 32], implicitly or explicitly). Note that however the existence of illegal instruction nodes cannot always be used as a criteria to exclude an instruction sequence unless they are inevitably reachable; otherwise attackers may obfuscate their program by adding *non-reachable* illegal instructions.

Based on this heuristic, we can exclude instruction sequence s_{08} in Figure 5(a), since it will eventually execute an illegal instruction v_{0f}.

After these three steps, in Figure 5(a) only instruction sequence s_{00} is left for consideration in the next stage.

5 Instruction Sequences Analyzer

A distilled instruction sequence may be a sequence of random instructions or a fragment of a program in machine language. In this section, we propose two schemes to differentiate these two cases. Scheme 1 exploits the operating system characteristics of a program; Scheme 2 exploits the data flow characteristics of a program. Scheme 1 is slightly faster than Scheme 2, whereas Scheme 2 is much more robust to obfuscation.

5.1 Scheme 1

A program in machine language is dedicated to a specific operating system; hence, a program has certain characteristics implying the operating system on which it is running, for example calls to operating system or kernel library. A random instruction sequence does not carry this kind of characteristics. By identifying the call pattern in an instruction sequence, we can effectively differentiate a real program from a random instruction sequence.

More specifically, instructions such as "call" and "int 0x2eh" in Windows and "int 0x80h" in Linux may indicate system calls or function calls. However, since the op-codes of these call instructions are only one byte, even normal requests may contain plenty of these byte values. Therefore, using the number of these instructions as a criteria will cause a high false positive rate. To address this issue, we use a pattern composed of several instructions rather than a single instruction. It is observed that before these call instructions there are normally one or several instructions used to transfer parameters. For example, a "push" instruction is used to transfer parameters for a "call" instruction; some instructions that set values to registers al, ah, ax, or eax are used to transfer parameters for "int" instructions. These call patterns are very common in a fragment of a real program. Our experiments in Section 6 show that by selecting the appropriate parameters we can rather accurately tell whether an instruction sequence is an executable code or not.

Scheme 1 is fast since it does not need to fully disassemble a request. For most instructions, we only need to know their types. This saves lots of time in decoding operands of instructions.

Note that although Scheme 1 is good at detecting most of the known buffer overflow attacks, it is vulnerable to obfuscation. One possible obfuscation is that attackers may use other instructions to replace the "call" and "push" instructions. Figure 5.1 shows an example of obfuscation, where "call eax" instruction is substituted by "push J4" and "jmp eax". Although we cannot fully solve this problem, by recording this kind of instruction replacement patterns, we may still be able to detect this type of obfuscation to some extent.

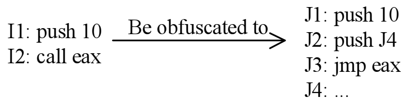


Figure 6: An obfuscation example. Instruction "call eax" is substituted by "push J4" and "jmp eax".

Another possible obfuscation is one which first encrypts the attack code and then decrypts it using a decryption routine during execution time [40]. This decryption routine does not include any calls, thus evading the detection of Scheme 1.

5.2 Scheme 2

Next we propose Scheme 2 to detect the aforementioned obfuscated buffer overflow attacks. Scheme 2 exploits the data flow characteristics of a program. Normally, a random instruction sequence is full of data flow anomalies, whereas a real program has few or no data flow anomalies. However, the number of data flow anomalies cannot be directly used to distinguish a program from a random instruction sequence because an attacker may obfuscate his program easily by introducing enough data flow anomalies.

In this paper, we use the detection of data flow anomaly in a different way called *code abstraction*. We

```
...
I1: mov eax,2
...
I2: mov eax,3
...
```

(a)

```
...
(ecx is undefined at
this point)
K1: mov eax,ecx
...
```

(b)

```
...
J1: mov eax,2
...
(ebx is undefined at
this point)
J2: mov eax,ebx
...
```

(c)

Figure 7: Data flow anomaly in execution paths. (a) define-define anomaly. Register eax is defined at I1 and then defined again at I2. (b) undefine-reference anomaly. Register ecx is undefined before K1 and referenced at K1 (c) define-undefine anomaly. Register eax is defined at J1 and then undefined at J2.

observe that when there are data flow anomalies in an execution path of an instruction sequence, some instructions are useless, whereas in a real program at least one execution path have a certain number of useful instructions. Therefore, if the number of useful instructions in an execution path exceeds a threshold, we conclude the instruction sequence is a segment of a program.

Data Flow Anomaly The term data flow anomaly was originally used to analyze programs written in higher level languages in the software reliability and testing field [25, 26]. In this paper, we borrow this term and several other terms to analyze instruction sequences.

During a program execution, an instruction may impact a variable (register, memory location or stack) on three different ways: *define*, *reference*, and *undefine*. A variable is defined when it is set a value; it is referenced when its value is referred to; it is undefined when its value is not set or set by another undefined variable. Note that here the definition of undefined is different from that in a high level language. For example, in a C program, a local variable of a block becomes undefined when control leaves the block.

A data flow anomaly is caused by an improper sequence of actions performed on a variable. There are three data flow anomalies: *define-define*, *define-undefine*, and *undefine-reference* [26]. The define-define anomaly means that a variable was defined and is defined again, but it has never been referenced between these two actions. The undefine-reference anomaly indicates that a variable that was undefined receives a reference action. The define-undefine anomaly means that a variable was defined, and before it is used it is undefined. Figure 7 shows an example.

Detection of Data Flow Anomalies There are static [25] or dynamic [26] methods to detect data flow anomalies in the software reliability and testing field. Static methods are not suitable in our case due to its slow speed; dynamic methods are not suitable either due to the need for real execution of a program with some inputs. As such, we propose a new method called code abstraction, which does not require real execution of code. As a result of the code abstraction of an instruction, a variable could be in one of the six possible states. The six possible states are state U: undefined; state D: defined but not referenced; state R: defined and referenced; state DD : abnormal state define-define; state UR: abnormal state undefine-reference; and state DU: abnormal state define-undefine. Figure 8 depicts the state diagram of these states. Each edge in this state diagram is associated with $d, r,$ or u, which represents “define”, “reference”, and “undefine”, respectively.

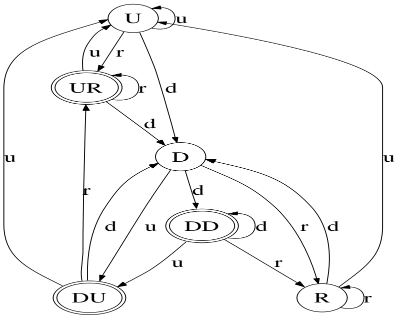


Figure 8: State diagram of a variable. State U: undefined, state D: defined but not referenced, state R: defined and referenced, state DD : abnormal state define-define, state UR: abnormal state undefine-reference and state DU: abnormal state define-undefine.

We assume that a variable is in “undefined” state at the beginning of an execution path. Now we start to traverse this execution path. If the entry instruction of the execution path defines this variable, it will enter the state “defined”. Then, it will enter another state according to the next instruction, as shown in Figure 8. Once the variable enters an abnormal state, a data flow anomaly is detected. We continue this traversal to the end of the execution path. This process enables us to find all the data flow anomalies in this execution path.

Pruning Useless Instructions Next we leverage the detected data flow anomalies to remove useless instructions. A *useless* instruction of an execution path is an instruction which does not affect the results of the execution path; otherwise, it is called *useful* instructions. We may find a useless instruction from a data flow anomaly. When there is an undefine-reference anomaly in an execution path, the instruction which causes the “reference” is a useless instruction. For instance, the instruction $K1$ in Figure 7, which causes undefine-reference anomaly, is a useless instruction. When there is a define-define or define-undefine anomaly, the instruction that caused the former “define” is also considered as a useless instruction. For instance, the instructions $I1$ and $J1$ in Figure 7 are useless instructions because they caused the former “define” in either the define-define or the define-undefine anomaly.

After pruning the useless instructions from an execution path, we will get a set of useful instructions. If the

Algorithm 2 check if the number of useful instructions in an execution path exceeds a threshold

```
Input: $entry$ instruction of an instruction sequence, EISG $G$
  $total \leftarrow 0$; $useless \leftarrow 0$ ; $stack \leftarrow$ empty
  initialize the $states$ of all variables to "undefined"
  push the entry $instruction$,$states$,$total$ and $useless$ to $stack$
  while $stack$ is not empty do
    pop the top item of $stack$ to $i$,$states$,$total$ and $useless$
    if $total - useless$ greater than a threshold then
      return true
    if $i$ is visited then
      continues
    mark $i$ visited
    $total \leftarrow total + 1$
    Abstractly execute instruction $i$ (change the $states$ of variables according to instruction $i$)
    if there is a define-define or define-undefine anomaly then
      $useless \leftarrow useless + 1$
    if there is a undefine-reference anomaly then
      $useless \leftarrow useless + 1$
    for each instruction $j$ directly following $i$ in the $G$ do
      push $j$, $states$ ,$total$ and $useless$ to $stack$
  return false
```

number of useful instructions in an execution path exceeds a threshold, we will conclude the instruction sequence is a segment of a program.

Algorithm 2 shows our algorithm to check if the number of useful instructions in an execution path exceeds a threshold. The algorithm involves a search over an EISG in which the nodes are visited in a specific order derived from a depth first search. The algorithm assumes that an EISG G and the entry instruction of the instruction sequence are given, and a push down stack is available for storage. During the search process, the visited node (instruction) is abstractly executed to update the states of variables, find data flow anomaly, and prune useless instructions in an execution path.

Handling Special Cases Next we discuss several special cases in the implementation of Scheme 2.

General purpose instruction The instructions in the IA32 instruction set can be roughly divided into four groups: general purpose instructions, floating point unit instructions, extension instructions, and system instructions. General purpose instructions perform basic data movement, arithmetic, logic, program flow, and string operation, which are commonly used by programmers to write applications and system software that run on IA-32 processors [3]. General purpose instructions are also the most often used instructions in malicious code. We believe that malicious codes must contain a certain number of general purpose instructions to achieve the attacking goals. Other types of instructions may be leveraged by an attacker to obfuscate his real-purpose code, e.g., used as garbage in garbage insertion. As such, we prune other groups of instructions as well.

Initial state of registers It is hard for attackers to know the run-time values of registers before malicious code is executed. That is, their values are unpredictable to attackers. Therefore, it is reasonable to assume that the initial states of all variables are "undefined" at the beginning of an execution path. The register "esp", however, is an exception since it is used to hold the stack pointer. Thus, we set register esp "defined" at the beginning of an execution path.

Indirect address An indirect address is an address that serves as a reference point instead of an address to the direct memory location. For example, in the instruction "move eax,[ebx+01e8]", register "ebx" may contain the actual address of the operand. However, it is difficult to know the run-time value of register "ebx". Thus, we always treat a memory location to which an indirect address points as state "defined" and hence no data flow anomaly will be generated. Indeed, this treatment successfully prevents an attacker from obfuscating his code using indirect addresses.

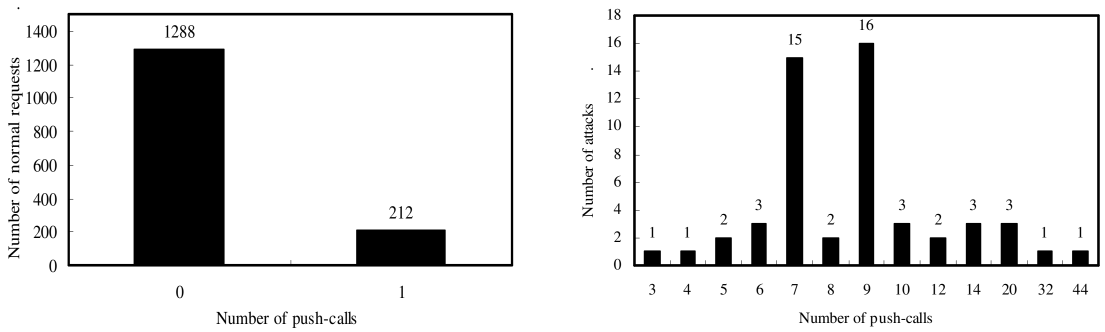
We will defer the discussion on the capability of Scheme 2 in defending against obfuscation until Section 7.

6 Experiments

6.1 Parameter Tuning

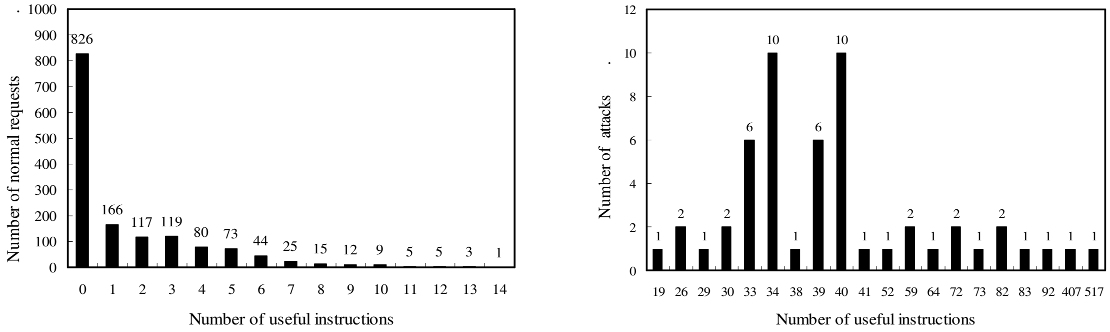
Both Scheme 1 and Scheme 2 use a threshold value to determine if a request contains code or not. Clearly, it is critical to set the threshold values appropriately so as to minimize both detection false positive rate and false negative rate. To find out the appropriate thresholds, we tested both schemes of SigFree against 50 unencrypted attack requests generated by Metasploit framework, worm Slammer, CodeRed (CodeRed.a) and a CodeRed variation (CodeRed.c), and 1500 binary HTTP replies (52 encrypted data, 23 audio, 195 jpeg, 32 png, 1153 gif and 45 flash) intercepted on the network of College of Information Science and Technology. Note that we tested on HTTP replies rather than requests as normal data for parameter tuning, because HTTP replies include more diverse binaries (test over real traces of web requests is reported in Section 6.3). Also note that although worm Slammer attacks Microsoft SQL servers rather than web servers, it also exploits buffer overflow vulnerabilities.

Threshold of Push-calls for Scheme 1 Figure 9(a) shows that all instruction sequences distilled from a normal request contain at most one push-call code pattern.



(a) (b)

Figure 9: The number of push-calls in a request. (a) Normal requests. It shows that any instruction sequences of a normal request contain at most one push-call code pattern. (b) Attack requests. It shows that an attack request contains more than two external push-calls in one of its instruction sequences.



(a) (b)

Figure 10: The number of useful instructions in a request. (a) Normal requests. It shows that no normal requests contain an instruction sequence which has over 14 useful instructions. (b) Attack requests. It shows that there exists an instruction sequence of an attack request which contain more than 18 useful instructions.

Figure 9(b) shows that for all the 53 buffer overflow attacks we tested, every attack request contains more than two push-calls in one of its instruction sequences. Therefore, by setting the threshold number of push-calls to 2, Scheme 1 can detect all the attacks used in our experiment.

Threshold of Useful Instructions for Scheme 2 Figure 10(a) shows that no normal requests contain an instruction sequence that has more than 14 useful instructions. Figure 10(b) shows that an attack request contains over 18 useful instructions in one of its instruction sequences. Therefore, by setting the threshold to a number between 15 and 17, Scheme 2 can detect all the attacks used in our test. The three attacks, which have the largest numbers of instructions (92, 407 and 517), are worm Slammer, CodeRed.a and CodeRed.c, respectively. This motivates us to investigate in our future work whether an exceptional large number of useful instructions indicates the occurrence of a worm.

6.2 Detection of Polymorphic Shellcode

We also tested SigFree on two well-known polymorphic engine, ADMmutate v0.84 [40] and CLET v1.0 [23]. Basically, ADMmutate obfuscates the shellcode of buffer overflow attacks in two steps. First, it encrypts the shellcode. Second, it obfuscates the decryption routine by substituting instructions and inserting junk instructions. In addition, ADMmutate replaces the No OPerations(NOP) instructions with other one-byte junk instructions to evade the detection of an IDS. This is because most buffer overflow attacks contain many NOP instructions to help locate shellcode, making them suspicious to an IDS.

CLET is a more powerful polymorphic engine compared with ADMmutate. It disguises its NOPs zone with 2,3 bytes instructions (not implemented yet in CLET v1.0), referred to as fake-NOPs, and generates a decipher routine with different operations at each time, which makes classical IDS pattern matching ineffective. Moreover, It uses spectrum analysis to defeat data mining methods.

Because there is no push-call pattern in the code, Scheme 1 cannot detect this type of attacks. However, Scheme 2 is still very robust to these obfuscation techniques. This is because although the original shellcode contains more useful instructions than the decryption routine has and it is also encrypted, Scheme 2 may still find enough number of useful instructions in the decryp-

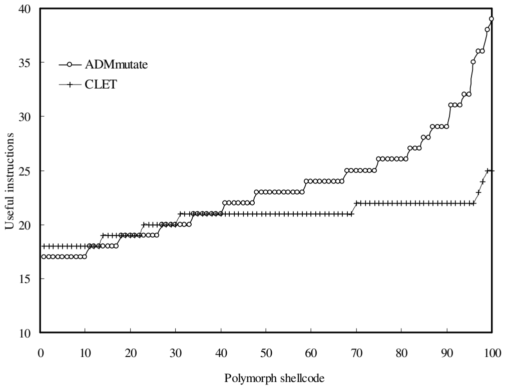


Figure 11: The number of useful instructions in all 200 polymorphic shellcodes. It shows that the least number of useful instructions in ADMmutate and CLET polymorphic shellcodes is 17.

tion routines.

We used each of ADMmutate and CLET to generate 100 polymorphic shellcodes, respectively. Then, we used Scheme 2 to detect the useful instructions in the code. Figure 11 shows the (sorted) numbers of useful instructions in 200 polymorphic shellcodes. We observed that the least number of useful instructions in these ADMmutate polymorphic shellcodes is 17, whereas the maximum number is 39; the least number of useful instructions in the CLET polymorphic shellcodes is 18, whereas the maximum number is 25. Therefore, using the same threshold value as before (i.e., between 15 and 17), we can detect all the 200 polymorphic shellcodes generated by ADMmutate and CLET.

6.3 Testing on Real Traces

We also tested SigFree over real traces. Due to privacy concerns, we were unable to deploy SigFree in a public web server to examine realtime web requests. To make our test as realistic as possible, we deployed a client-side proxy underneath a web browser. The proxy recorded a normal user's http requests during his/her daily Internet surfing. During a one-week period, more than ten of our lab members installed the proxy and helped collect totally 18,569 HTTP requests. The requests include manually typed urls, clicks through various web sites, searchings from search engines such as Google and Yahoo, secure logins to email servers and bank servers, and HTTPs requests. In this way, we believe our data set is diverse enough, not worse than that we might have got if we install SigFree in a single web server that provides only limited Internet services.

Our test based on the above real traces did not yield an alarm. This output is of no surprise because our normal web requests do not contain code.

6.4 Performance Evaluation

To evaluate the performance of SigFree, we implemented a proxy-based SigFree prototype using the C programming language in Win32 environment. SigFree was compiled with Borland C++ version 5.5.1 at optimization level O2. The prototype implementation was hosted in a Windows 2003 server with Intel Pentium 4, 3.2GHz CPU and 1G MB memory.

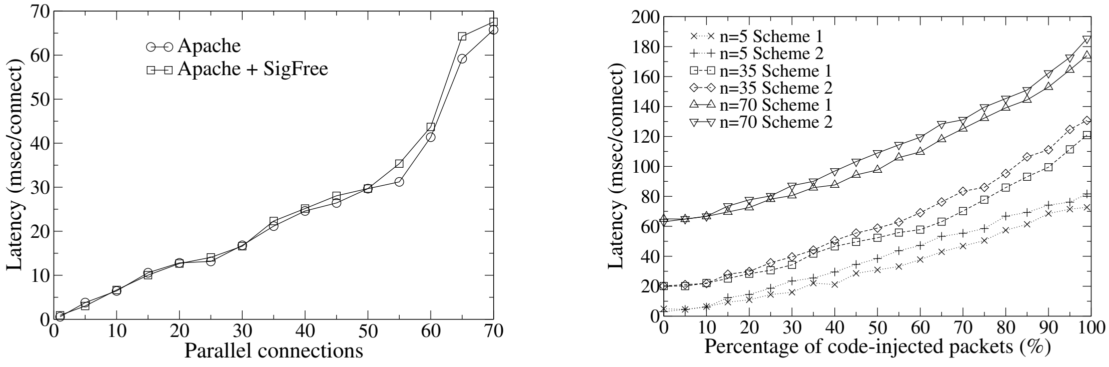
The proxy-based SigFree prototype accepts and analyzes all incoming requests from clients. The client testing traffics were generated by Jef Poskanzer's http_load program [3] from a Linux desktop PC with Intel Pentium 4 2.5GHz CPU connected to the Windows server via a 100 Mbps LAN switch. We modified the original http_load program so that clients can send code-injected data requests.

For the requests which SigFree identifies as normal, SigFree forwards them to the web server, Apache HTTP Server 2.0.54, hosted in a Linux server with dual Intel Xeon 1.8G CPUs. Clients send requests from a predefined URL list. The documents referred in the URL list are stored in the web server. In addition, the prototype implementation uses a time-to-live based cache to reduce redundant HTTP connections and data transfers.

Rather than testing the absolute performance overhead of SigFree, we consider it more meaningful measuring the impact of SigFree on the normal web services. Hence, we measured the *average response latency* (which is also an indication of *throughput* although we did not directly measure throughput) of the connections by running http_load for 1000 fetches. Figure 12(a) shows that when there are no buffer overflow attacks, the average response time in the system with SigFree is only slightly higher than the system without SigFree. This indicates that, despite the connection and ASCII checking overheads, the proxy-based implementation does not affect the overall latency significantly.

Figure 12(b) shows the average latency of connections as a function of the percentage of attacking traffics. We used CodeRed as the attacking data. Only successful connections were used to calculate the average latency; that is, the latencies of attacking connections were not counted. This is because what we care is the impact of attack requests on normal requests. We observe that the average latency increases slightly worse than linear when the percentage of malicious attacks increases. Generally, Scheme 1 is about 20% faster than Scheme 2.

Overall, our experimental results from the prototype implementation show that SigFree has reasonably low performance overhead. Especially when the fraction of attack messages is small (say $< 10\%$), the additional latency caused by SigFree is almost negligible.



(a) (b)

Figure 12: Performance impact of SigFree on Apache HTTP Server

7 Discussions

7.1 Robustness to Obfuscation

Most malware detection schemes include two-stage analysis. The first stage is disassembling binary code and the second stage is analyzing the disassembly results. There are obfuscation techniques to attack each stage [19, 38] and attackers may use them to evade detection. Table 1 shows that SigFree is robust to most of these obfuscation techniques.

Obfuscation in The First Stage *Junk byte insertion* is one of the simplest obfuscation against disassembly. Here junk bytes are inserted at locations that are not reachable at run-time. This insertion however can mislead a linear sweep algorithm, but can not mislead a recursive traversal algorithm [33], which our algorithm bases on.

Opaque predicates are used to transform unconditional jumps into conditional branches. Opaque predicates are predicates that are always evaluated to either true or false regardless of the inputs. This allows an obfuscator to insert junk bytes either at the jump target or in the place of the fall-through instruction. We note that opaque predicates may make SigFree mistakenly interpret junk byte as executable codes. However, this mistake will not cause SigFree to miss any real malicious instructions. Therefore, SigFree is also immune to obfuscation based on opaque predicates.

Obfuscation in The Second Stage Most of the second-stage obfuscation techniques obfuscate the behaviors of a program; however, the obfuscated programs still bear characteristics of programs. Since the purpose of SigFree is to differentiate executable codes and random binaries rather than benign and malicious executable codes, most of these obfuscation techniques are ineffective to SigFree. Obfuscation techniques such as instruction reordering, register renaming, garbage insertion and reordered memory accesses do not affect the number of calls or useful instructions which our schemes are based on. By exploiting instruction replacement and equivalent functionality, attacks may evade the detection of Scheme 1, but cannot evade the detection of Scheme 2.

Disassembly stage	Obfuscation	SigFree	
	Junk byte insertion	Yes	
	Opaque predict	Yes	
	Branch function	partial	
Analysis stage	Obfuscation	Scheme 1	Scheme 2
	Instruction reordering	Yes	Yes
	Register renaming	Yes	Yes
	Garbage insertion	Yes	Yes
	Instruction replacement	No	Yes
	Equivalent funcationality	No	Yes
	Reordered memory accesses	Yes	Yes

Table 1: SigFree is robust to most obfuscation

7.2 Limitations

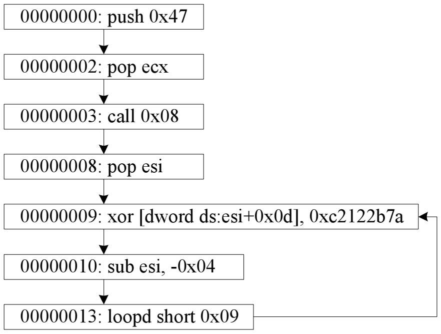
SigFree also has several limitations. First, SigFree cannot fully handle the branch-function based obfuscation, as indicated in Table 1. Branch function is a function $f(x)$ that, whenever called from x, causes control to be transferred to the corresponding location $f(x)$. By replacing unconditional branches in a program with calls to the branch function, attackers can obscure the flow of control in the program. We note that there are no general solutions for handling branch function at the present state of the art.

With respect to SigFree, due to the obscurity of the flow of control, branch function may cause SigFree to break the executable codes into multiple instruction sequences. Nevertheless, it is still possible for SigFree to find this type of buffer overflow attacks as long as SigFree can still find enough push-calls or useful instructions in one of the distilled instruction sequences.

Second, the executable shellcodes could be written in alphanumeric form [45]. Such shellcodes will be treated as printable ASCII data and thus bypass our analyzer.

By turning off the ASCII filter, Scheme 2 can successfully detect alphanumeric shellcodes; however, it will increase unnecessary computational overhead. It therefore requires a slight tradeoff between tight security and system performance.

Finally, the current implementation of SigFree cannot detect malicious code which consists of fewer useful instructions than current threshold 15. Figure 13 shows a possible evasion which has only 7 useful instructions for a decryption routine. One solution to catch this evasion is to use a comprehensive score rather than the absolute number of useful instructions as the threshold. For example, we may give larger weights to instructions that are within a loop because most decryption routines contain loops. This approach, however, may introduce some false positives, which we will report in our future work.



```
00000000: push 0x47
00000002: pop ecx
00000003: call 0x08
00000008: pop esi
00000009: xor [dword ds:esi+0x0d], 0xc2122b7a
00000010: sub esi, -0x04
00000013: loopd short 0x09
```

Figure 13: A decryption routine with 7 useful instructions. The first two instructions are used to set the initial value for loop counter ecx. The next two instructions are used to acquire the value of EIP (instruction pointer register). The last three instructions form the decryption loop.

7.3 Application-Specific Encryption Handling

The proxy-based SigFree could not handle encrypted or encoded data directly. A particular example is SSL-enabled web server. Enhancing security between web clients and web servers by encrypting HTTP messages, SSL also causes the difficulty for out-of-box malicious code detectors.

To support SSL functionality, an SSL proxy such as Stunnel [6] (Figure 14) may be deployed to securely tunnel the traffic between clients and web servers. In this case, we may simply install SigFree in the machine where the SSL proxy is located. It handles the web requests in cleartext that have been decrypted by the SSL proxy. On the other hand, in some web server applications, SSL is implemented as a server module (e.g., mod_ssl in Apache). In this case, SigFree will need to be implemented as a server module (though not shown in Figure 14), located between the SSL module and the WWW server. We notice that most popular web servers allow us to write a server module to process requests and specify the order of server modules. Detailed study will be reported in our future work.

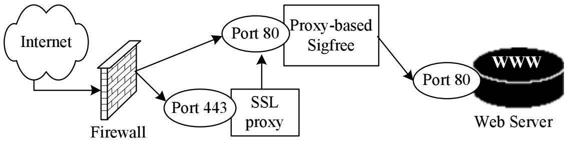


Figure 14: SigFree with a SSL proxy

7.4 Applicability

So far we only discussed using SigFree to protect web servers. It is worth mentioning that our tool is also widely applicable to many programs that are vulnerable to buffer overflow attacks. For example, the proxy-based SigFree may be used to protect all internet services which do not permit executable binaries to be carried in requests, e.g., database servers, email servers, name services, and so on. We will investigate the deployment issue in our future work.

In addition to protecting severs, SigFree can also provide file system real-time protection. Buffer overflow vulnerabilities have been found in some famous applications such as Adobe Acrobat and Adobe Reader [5], Microsoft JPEG Processing (GDI+) [1], and WinAmp [8]. This means that attackers may embed their malicious code in PDF, JPEG, or mp3-list files to launch buffer overflow attacks. In fact, a virus called Hesive [7] was disguised as a Microsoft Access file to exploit buffer overflow vulnerability of Microsoft's Jet Database Engine. Once opened in Access, infected .mdb files take advantage of the buffer overflow vulnerability to seize control of vulnerable machines. If mass-mailing worms exploit these kinds of vulnerabilities, they will become more fraudulent than before, because they may appear as pure data-file attachments. SigFree can be used alleviate these problems by checking those files and email attachments which should not include any code.

If the buffer being overflowed is inside a JPEG or GIF system, ASN.1 or base64 encoder, SigFree cannot be directly applied. Although SigFree can decode the protected file according to the protocols or applications it protects, more details need to be studied in the future.

8 Conclusion

We proposed SigFree, a realtime, signature free, out-of-the-box blocker that can filter code-injection buffer overflow attack messages, one of the most serious cyber security threats, to various Internet services. SigFree does not require any signatures, thus it can block new, unknown attacks. SigFree is immunized from most attack-side code obfuscation methods, good for economical Internet wide deployment with little maintenance cost and negligible throughput degradation, and can also handle encrypted SSL messages.

Acknowledgments We would like to thank our shepherd Marc Dacier and the anonymous reviewers for their valuable comments and suggestions. We are grateful to Yoon-Chan Jhi for helpful suggestions. We also thank the members of Penn State Cyber Security Lab for collecting real traces. The work of Xinran Wang and Sencun Zhu was supported in part by Army Research Office (W911NF-05-1-0270) and the National Science Foundation (CNS-0524156); the work of Chi-Chun Pan and Peng Liu was supported in part by NSF CT-3352241.

References

[1] Buffer overrun in jpeg processing (gdi+) could allow code execution (833987). http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx

[2] Fnord snort preprocessor. http://www.cansecwest.com/spp_fnord.c.

[3] Intel ia-32 architecture software developer's manual volume 1: Basic architecture.

[4] Metasploit project. http://www.metasploit.com.

[5] Security advisory: Acrobat and adobe reader plug-in buffer overflow. http://www.adobe.com/support/techdocs/321644.html.

[6] Stunnel – universal ssl wrapper. http://www.stunnel.org.

[7] Symantec security response: backdoor.hesive. http://securityresponse.symantec.com/avcenter/venc/data/backdoor.hesive.html

[8] Winamp3 buffer overflow. http://www.securityspace.com/ smysecure/catid.html?id=11530.

[9] Pax documentation. http://pax.grsecurity.net/docs/pax.txt, November 2003.

[10] BARATLOO, A., SINGH, N., AND TSAI, T. Transparent run-time defense against stack smashing attacks. In *Proc. 2000 USENIX Technical Conference* (June 2000).

[11] BARRANTES, E., ACKLEY, D., PALMER, T., STEFANOVIC, D., AND ZOVI, D. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communications security* (October 2003).

[12] BERNERS-LEE, T., MASINTER, L., AND MCCAHILL, M. Uniform Resource Locators (URL). RFC 1738 (Proposed Standard). Updated by RFCs 1808, 2368, 2396, 3986.

[13] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security* (2005).

[14] CHEN, H., DEAN, D., AND WAGNER, D. Model checking one million lines of c code. In *NDSS* (2004).

[15] CHINCHANI, R., AND BERG, E. V. D. A fast static analysis approach to detect exploit code inside network flows. In *RAID* (2005).

[16] CHRISTODORESCU, M., AND JHA, S. Static analysis of executables to detect malicious patterns. In *Proceedings of 12th USENIX Security Symposium* (August 2003).

[17] CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy, Oakland* (May 2005).

[18] CKER CHIUEH, T., AND HSU, F.-H. Rad: A compile-time solution to buffer overflow attacks. In *ICDCS* (2001).

[19] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. Tech. Rep. 148, Department of Computer Science,University of Auckland, July 1997.

[20] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[21] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-end containment of internet worms. In *SOSP* (2005).

[22] COWAN, C., PU, C., MAIER, D., HINTON, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of 7th USENIX Security Conference* (January 1998).

[23] DETRISTAN, T., ULENSPIEGEL, T., MALCOM, Y., AND UNDERDUK, M. S. V. Polymorphic shellcode engine using spectrum analysis. http://www.phrack.org/show.php?p=61&a=9.

[24] EVANS, D., AND LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE Software 19*, 1 (2002).

[25] FOSDICK, L. D., AND OSTERWEIL, L. Data flow analysis in software reliability. *ACM Computing Surveys 8* (September 1976).

[26] HUANG, J. Detection of data flow anomaly through program instrumentation. *IEEE Transactions on Software Engineering 5*, 3 (May 1979).

[27] JUNG, J., PAXSON, V., BERGER, A., AND BALAKRISHNAN, H. Fast portscan detection using sequential hypothesis testing. In *Proc. IEEE Symposium on Security and Privacy* (2004).

[28] KC, G., KEROMYTIS, A., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security* (October 2003).

[29] KIM, H.-A., AND KARP, B. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th Usenix Security Symposium* (August 2004).

[30] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. Secure execution via program shepherding. In *Proceedings of USENIX Security Symposium* (2002).

[31] KOLESNIKOV, O., AND LEE, W. Advanced polymorphic worms: Evading ids by blending in with normal traffic.

[32] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Polymorphic worm detection using structural information of executables. In *RAID* (2005).

[33] KRUEGEL, C., ROBERTSON, W., VALEUR, F., AND VIGNA, G. Static disassembly of obfuscated binaries. In *Proceedings of USENIX Security 2004* (August 2004).

[34] KUPERMAN, B. A., BRODLEY, C. E., OZDOGANOGLU, H., VIJAYKUMAR, T. N., AND JALOTE, A. Detecting and prevention of stack buffer overflow attacks. *Communications of the ACM 48*, 11 (2005).

[35] LAKHOTIA, A., AND ERIC, U. Stack shape analysis to detect obfuscated calls in binaries. In *Proceedings of Fourth IEEE International Workshop on Source Code Analysis and Manipulation* (September 2004).

[36] LIANG, Z., AND SEKAR, R. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In *Proceedings of the Annual Computer Security Applications Conference(ACSAC)* (2005).

[37] LIANG, Z., AND SEKAR, R. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proc. 12th ACM Conference on Computer and Communications Security* (2005).

[38] LINN, C., AND DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. In *10th ACM Conference on Computer and Communications Security (CCS)* (October 2003).

[39] LOCASTO, M. E., WANG, K., KEROMYTIS, A. D., AND STOLFO, S. J. Flips: Hybrid adaptive intrusion prevention. In *RAID* (2005).

[40] MACAULAY, S. Admmutate: Polymorphic shellcode engine. http://www.ktwo.ca/security.html.

[41] MCGREGOR, J., KARIG, D., SHI, Z., AND LEE, R. A processor architecture defense against buffer overflow attacks. In *Proceedings of International Conference on Information Technology: Research and Education (ITRE)* (2003), pp. 243 – 250.

[42] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatic signature generation for polymorphic worms. In *IEEE Security and Privacy Symposium* (May 2005).

[43] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS* (2005).

[44] PANG, R., YEGNESWARAN, V., BARFORD, P., PAXSON, V., AND PETERSON, L. Characteristics of internet background radiation. In *Proc. ACM IMC* (2004).

[45] RIX. Writing ia32 alphanumeric shellcodes. http://www.phrack.org/show.php?p=57&a=15, 2001.

[46] SCHWARZ, B., DEBRAY, S. K., AND ANDREWS, G. R. Disassembly of executable code revisited. In *Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE)* (October 2002).

[47] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. The earlybird system for real-time detection of unknown worms. Tech. rep., University of California at San Diego, 2003.

[48] S.KC, G., AND KEROMYTIS, A. D. e-nexsh: Achieving an effectively non-executable stack and heap via system-call policing. In *Proceedings of the Annual Computer Security Applications Conference(ACSAC)* (2005).

[49] SMIRNOV, A., AND CKER CHIUEH, T. Dira: Automatic detection, identification, and repair of control-hijacking attacks. In *NDSS* (2005).

[50] TOTH, T., AND KRUEGEL, C. Accurate buffer overflow detection via abstract payload execution. In *RAID* (2002).

[51] WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium* (February 2000).

[52] WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the ACM SIGCOMM Conference* (August 2004).

[53] WANG, K., , CRETU, G., AND STOLFO, S. J. Anomalous payload-based worm detection and signature generation. In *RAID* (2005).

[54] WANG, K., AND STOLFO, S. J. Anomalous payload-based network instrusion detection. In *RAID* (2004).

[55] XU, J., NING, P., KIL, C., ZHAI, Y., AND BOOKHOLT, C. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proc. 12th ACM Conference on Computer and Communications Security* (2005).

Notes

[1] An attack may direct execution control to existing system code or change the values of certain function arguments.

[2] http://www.research.ibm.com/trl/projects/security/ssp/

[3] http://www.acme.com/software/http_load/