A Collaborative Monitoring Mechanism for Making a Multitenant Platform Accountable

Chen Wang
CSIRO ICT Center
PO Box 76,
NSW 1710, Australia
Email:chen.wang@csiro.au

Ying Zhou
School of Information Technologies
The University of Sydney
NSW 2006, Australia
Email:ying.zhou@sydney.edu.au

Abstract

Multitenancy becomes common as an increasing amount of applications runs in clouds, however, the certainty of running applications in a fully controlled administrative domain is lost in the move. How to ensure that the data and business logic are handled faithfully becomes an issue. We propose to maintain a state machine outside of a multitenant platform to make the platform accountable in this paper. We give a mechanism to support accountability for a multitenant database with a centralized external service. We also describe how to implement a decentralized virtual accountability service via collaborative monitoring. Finally, we discuss the characteristics of the mechanism through experiments in Amazon EC2.

1 Introduction

Technologies such as multitenant database systems and Platform as a Service (PaaS) enable multiple applications to share a platform. This approach can significantly reduce the cost for an organization to maintain dedicated hardware/software resources to run its applications. The success of Force.com [18] shows the power of this technology. The adoption of the technology represents a trend of moving clients' data and business logic to the cloud. To ensure that the data and business logic are handled correctly, a platform service provider often offers service level agreement (SLA) to its clients. However, few means are provided to clients to make a SLA accountable when problems occur. Clients are often required to furnish evidence all by themselves to be eligible to claim credit for a SLA violation [1]. However, the existing application design practice does not take into account of evidence collection functionalities for credit claiming purpose. Run-time logs contain information mainly for bug diagnostics, not for SLA compliance check. Supporting evidence collection certainly adds extra burden for the clients of a multitenant platform. In this paper, we propose to use third party services to do the task. This type of services is referred as *accountability services* in networking services [4] and service oriented architecture [10, 17]. We give a mechanism for clients to authenticate the correctness of the data and the execution of their business logic in a multitenant platform.

A multitenant platform intends to achieve both flexibility for various tenants and efficiency for itself to manage a variety of data and applications. Metadata-driven architecture is often used for this purpose. In a typical multitenant platform, a client's business logic is represented using tenant-specific metadata and executed through applications generated from the metadata. Web-database systems [7] fit into such a multitenant platform well as web pages of a tenant application, including both the layout and the content, can be dynamically generated from the tenant-specific data stored in a shared database system. Metadata-driven architecture greatly eases the construction and management of client applications, however, it also restrains the tasks of client application developers to supplying metadata only. The transformation process between metadata and application as well as how the data is accessed are out of the sight from a client's perspective. It is not a trivial task for a client if she attempts to get assurance that the platform handles her data and business logic correctly.

Accountability is one of the foundations that form real-world trust relationships. The capability of identifying a party that is responsible when things go wrong with undeniable evidences can potentially enhance the trustworthiness of a system. In the business world, one may be reluctant to transact with a total stranger, but a well-known middleman or a group of middlemen can make them contract with each other and resolve possible disputes during their transactions [13]. In a world where data are managed by outsourced data management systems and business logic is implemented through outsourced platforms, there is a strong need for a third party

role to witness and audit the execution of business logic and ensure data correctness.

The responsibilities of an accountability service for a multitenant platform therefore include the following:

1. Collecting and managing evidence based on a given SLA. The SLA defines data states of interests and data state transitions triggered by client operations;

2. Runtime compliance check and problem detection, e.g., the result of a query does not reflect the recent data change under a given consistency protocol.

Even though an accountability service can play certain role in dealing with malicious attacks, the focus of the paper is on the mechanism of collecting evidence that show things go wrong. The rest of the paper discusses how the responsibilities mentioned above are fulfilled and is organized as follows: Section 2 describes the problem; Section 3 discusses how to use a trusted party to make a multitenant platform accountable; Section 4 discusses how client applications can collaboratively monitor the platform; Section 5 gives evaluation results; Section 6 discusses related work and Section 7 concludes the paper.

2 Problem Description

We consider a tenant delegates its data management functionality to a multitenant platform provisioned by another party. The tenant has a set of applications (called client application in the following) running on its data stored in the multitenancy environment. These applications provide a set of endpoints for the end-users of the tenant to use its service. We denote the set of endpoints that query or update the data stored in the platform as $\{ep_0, ep_1, ..., ep_{n-1}\}$. We assume that the data can only be accessed through the set of endpoints according to a SLA reached between the tenant and the platform provider. We also assume that $ep_i (0 \leq i < n)$ is well-defined in terms that the data state transition it may trigger is specified in the SLA and deterministic.

There are many ways things can go wrong in this scenario, e.g, a data element is modified without the permission of the data owner (or without going through the specified set of endpoints), or the consistency requirement promised by the platform is broken. A multitenant database often balances the scalability and consistency requirements of client applications and offer different level of consistency [2], but currently there is no practical way for a tenant to be sure that it always has the level of data consistency it pays for. From a client's point of view, it is essential that all the functionalities offered by the platform provider are performed with certainty in runtime and the data contains no surprise to the owner applications.

To make the data and business logic handling accountable in such a multitenancy environment, we introduce a special type of third party provisioned accountability services. The external party offers services to wrap an endpoint $ep_i, 0 \leq i < n$ in an adapter wep_i. The wrapper is capable of capturing the input/output from ep_i and extracting certain information required by the accountability service from the captured data. We denoted the accountability service as W. The scenario is shown in Fig. 1. wep_i is in fact an abstraction of existing tools that are capable of logging user activities, such as HttpWatch (`http://www.httpwatch.com/`), or proxies between end-users and outsourced services. The behaviors of the wrapper provided by an accountability service are verifiable. A tenant trusts the wrapper either through the certification issued by a trusted party or by reviewing its source code. The tenant should also be able to configure the wrapper to anonymize sensitive information to be sent to W.

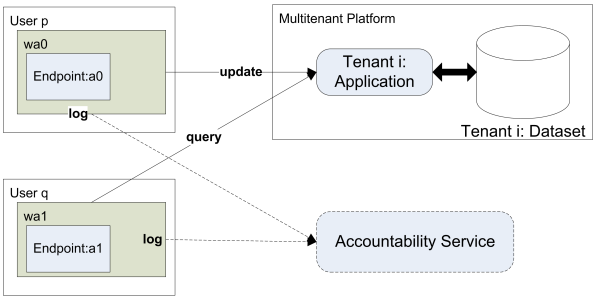

Figure 1: Architecture for Supporting Accountability

The problem is therefore how to make use of the captured information to create a reliable state machine for authenticating the data managed in a multitenant platform.

2.1 Preliminary

Merkle B-tree (MB-tree) [9] is an effective method for a data owner to authenticate the data stored in an outsourced database. It is a combination of Merkle (hash) tree [12] and B^+-tree. Figure 2 illustrates the structure of a MB-tree.

Each MB-tree node is an ordinary B^+-tree node with a hash value associated with each entry. The hash value of data entry in a leaf node is the hash of the data entry itself. The hash value of the data entry in an internal node is the hash of the concatenation of the hashes of its children, denoted as $h = H(h_i|h_{i+1}|...)$. The hash of the root is signed by the data owner for authentication purpose. When answering a range query, e.g, as shown

in Fig. 2, $r_4 < k < r_8$, the database system traverses the MB-tree and locates the left-most data entry, which is r_5, and the right-most data entry, which is r_7, in the leaf nodes. The hash values of the following entries in each internal node on the two traversal paths are inserted into a verification object ($\mathcal{VO}$): the left siblings of k_i, the right siblings of k_j; the left siblings of k_2, the right siblings of k_x. The following items in the leaf nodes are inserted into the $\mathcal{VO}$ as well: the data entries in the query range (r_5, r_6 and r_7), the hash values of data entries to the left of r_5 and those to the right of r_7, and the data entry to the left of r_5, i.e., r_4 and the data entry to the right of r_7, i.e., r_8 (this is for completeness verification).

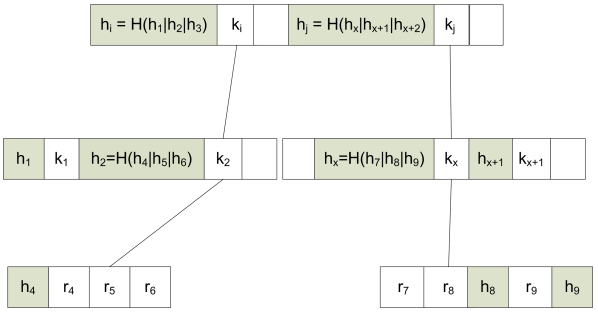

Figure 2: Example of Merkle B-tree

A client can then compute the hashes of the sub-tree corresponding to the query result set. By combining the items in the $\mathcal{VO}$, the client can iteratively recalculate the root of the MB-tree. The comparison of the root and the recalculated value will reveal the correctness and completeness of the query result.

3 Supporting Accountability with a Dedicated Service

In this section, we discuss how the MB-tree can be used to support accountability by a dedicated service. When using a dedicated accountability service, a tenant needs to have an agreement with the service provider on how a call to an application endpoint may affect the data state in the data management service. The data state change can be insertion, updating or deletion of a few tuples. We generalize the operations triggering data state change as *update*. Similarly, we refer operations that do not trigger data state change as *query* operations. A query can be composed in a form of point query or range query. We assume an update and a query operation can be uniquely identified by the operation type and a set of sortable attributes of the affected tuples in the database.

When endpoint ep_i is called for updating a data tuple, wep_i extracts the summary of the operation and sends it to accountability service W. When endpoint ep_j is called for data query, wep_j extracts the summary of the result and authenticates the result against the record in W. W maintains a MB-tree to serve these requests. The process is described using pseudocode in Algorithm 1.

Algorithm 1 Evidence Collection and Processing

The logging process inside wep_i:

Input: Endpoint ep_i associated with the specification of the data state transition as a result of the call.

1: **if** ep_i triggers state change **then**
2: extract the operation signature op:
$op = \{(type, k, h(k|v), t)\}$,
in which $type$ is the operation type; k identifies the affected data; $h(k|v)$ is the hash of the concatenated identifier and the new value of the data; and t is the time when ep_i is called.
3: invoke ep_i
4: **if** successful **then**
5: mark op as committed and sent it W
6: **end if**
7: **else**
8: invoke ep_i
9: extract query result set rs:
$rs = \{(type, k, h(k|v), t)\}$,
in which t is the time when the result is received.
10: send rs to W for authentication
11: **end if**

The request processing in W side:

Input: Request r (either op or rs) from a client.

1: insert r into a pending request queue
2: sort the requests in the queue based on their timestamps
3: **for all** request r in the queue **do**
4: **if** $r.t \leq current\ time - sliding\ window\ size$ **then**
5: **if** $r.type$ == UPDATE **then**
6: update the MB-tree with op
7: **else**
8: authenticate rs
9: **end if**
10: **end if**
11: **end for**

We assume a reliable communication channel between wep_i and W, i.e., a message will eventually arrive in W once sent out. All wep_i synchronize their clocks with W periodically. When W receives an operation signature from a client, it updates the MB-tree. Upon receiving an authentication request, W traverses the MB-tree for the left and right boundary of the result set. The returned $\mathcal{VO}$ is used together with the hashes in the result set to recompute the root in order to check the correctness and

completeness of the result set. An alert will be raised if the recomputed root does not match and existing root.

Note, we do not assume that requests arrive in W in the order of their creating time Algorithm 1. Such an assumption is difficult to satisfy in some cases due to the links between clients and W are likely to be heterogeneous and the processing speed of each wep may vary. Instead, we introduce a queue for request processing in W. The requests in the queue are sorted by the request timestamps. The queue keeps requests arriving in a sliding time window with a pre-defined window size. Requests with timestamps outside of the window are removed from the queue and written into the persistent B-tree.The time window size is determined by the maximum delay of passing a log entry from a client to W. An authentication request with timestamp earlier than the starting time of the sliding window can be answered immediately by recomputing the root of the MB-tree. The processing of an authentication request with timestamp fall inside the sliding time window will be postponed till the starting time of the window passes the timestamp. Failed authentication requests are temporarily stored.

Even though the size of the sliding window is determined by the maximum delay of transferring a log entry from a client to W, exceptions can happen where an update event arrives outside of the window. An alert will be raised in this case. Some manual procedure may be used to resolve the exceptions. The failed authentication requests arriving after this delayed update will be reviewed to check if the update is indeed valid. The frequency of such type of events may trigger the adjustment of the sliding window size.

It is not difficult to see that the accountability service supports eventual consistency. When there is no new update arrives, the answer to an authentication request will reflect the data state caused by the last update.

4 A Collaborative Monitoring Mechanism

The approach described above requires a centralized accountability service to respond to authentication requests and maintain the authentication data structure. It requires that the service is highly available, trustworthy and scalable when the demand increases. High availability and scalability may be achieved through adding computing and storage resources to the centralized service, but the trustworthiness is better achieved through separating the responsibility to multiple services. We hereby give a distributed approach to reduce the cost of maintaining such a service as well as enhancing the trustworthiness of the accountability service.

In a distributed setting, the data state of a multitenant database is maintained by a set of data state services. Each service maintains a view of the data state. The set of data state services form a virtual accountability service. They can be co-hosted with the client applications if the host is accessible by other client applications. The scenario is shown in Fig. 3.

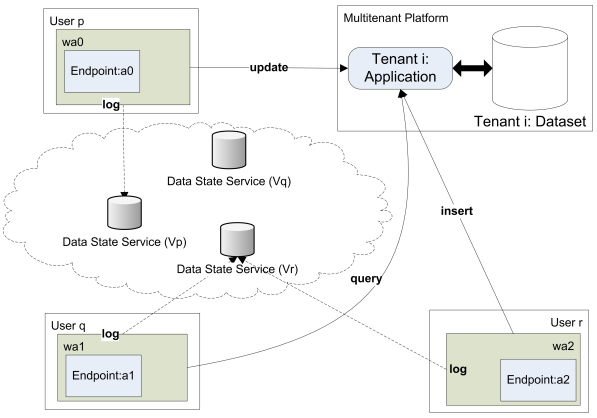

Figure 3: Distributed Architecture for Supporting Accountability

There are two extremes of design choices for supporting a client application to log its update operations and submit authentication requests:

1. An update log is sent to any of the data state services the client application knows. The service that receives the log propagates the log to other data state services in a synchronous manner, or to enhance trustworthiness, the propagation can be done through the use of a Byzantine fault tolerance protocol, e.g., [5], which ensure the data state is correctly maintained even when up to 1/3 data state services become faulty. This approach ensures strong consistency among data state services. As a result, an authentication request can be answered by any data state service. However, the drawback of this approach is that the logging performance can be dragged down, particularly when the number of data services increases.

2. An update log is sent to any of the data state services and the service receiving the log propagates the log to other nodes asynchronously. The log will be eventually reflected in all data state services, however it is not guaranteed that the answer to a subsequent authentication request will reflect the data state change carried in the log sent from the data state service where the change is initially recorded. In another words, the data state services maintains a weak consistency. This approach gives client applications better logging performance, but also causes uncertainty in answering an authentication request, especially when the inconsistency window is hard to determine or when faults occur.

Our design is between the two extremes. We partition the whole range of the indexed attribute into a few non-overlapped regions. Each region is mapped to one or more (greater than one in general) data state services. An update to a key falling into certain region will be logged to one of the services that is responsible for the region. The corresponding wep directs the log to the correct service (through a shared encoding and mapping method). Once receiving the log, the data state service synchronizes the new data state with other services that are responsible for the same region, meanwhile it propagates the update to other data services in an asynchronous manner. This approach ensures strong consistency among data state services that are mapped to the same region and enhances the availability of data state services. Faults in a single service is unlikely to make the whole data state uncertain.

Similarly, an authentication request will be directed to a data state service whose assigned region overlaps most with the data range associated with the request. If the data range in the result set of the request totally falls inside the region of the service, the service can answer the request with certainty; otherwise, it will have to wait for an allowable delay window for the update logs from other involved region to arrive before answering the request, in a similar way as described in Algorithm 1.

5 Evaluation

We evaluate our mechanism through experiments in Amazon EC2. The testing environment consists of a data management service, an accountability service built on top of the authenticated indexed structure library provided by [9] and a few clients that use the database servers. The data management service consists of a Web service that eventually maps business logic to *insert*, *point query* and *range query* operations, and a database server running MySQL version 14.12 that is capable of supporting multiple such services. The Web services are implemented using gSOAP [16]. Each party runs on an EC2 small instance created from the same Ubuntu image (Linux version 2.6.21.7-2.fc8xen).

The data used by client applications and managed by the database server is the "Census Income" dataset from the UCI Machine Learning Repository [3]. The indexed column is "fnlwgt" in our experiment.

5.1 The Overhead

We measure the overhead introduced by a centralized accountability service by comparing the average response time for insert operations, point and range queries from the client side. The calls to the data management service and accountability service are both synchronous ones for measurement purpose. Fig. 4 shows the overhead results of insert operations and point queries. The data points used in testing point queries are randomly selected from the whole dataset.

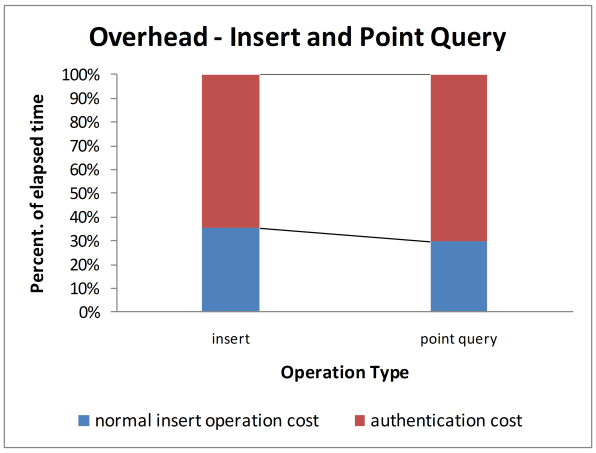

Figure 4: Breakdown of the Average Response Time of Insert Operations and Point Queries

The average cost of logging operation summary to the accountability service for an insert operation is about 65% of the elapsed time of the transaction that includes calling both data management service and accountability service. For point query, the cost of result authentication is about 70% of the elapsed time. Authenticating the query result incurs higher cost is due to that the additional step of recalculting of $\mathcal{VO}$ based on the client-supplied result set and the hash values retrieved from the MB-tree. Inserting into a MB-tree does not require such a step and the update to the MB-tree is not as expensive when the dataset is not very big.

Fig. 5 shows the overhead of processing range queries changing with the result set size. In our experiment, the range starting points are randomly selected from the whole dataset while the end points are calculated based on the corresponding starting points and the given range sizes.

It is apparent that the overhead increases along with the result set size from around 55% to above 90%. This can be attributed to the rising cost of transferring the hash values of the result set to the accountability service as well as the increasing cost of calculating the $\mathcal{VO}$. One method for reducing the overall overhead is to reduce the frequency of sending authentication requests of range query results to the accountability service, which requires a mechanism to determine the frequency so that the size of possible window of attacks can be minized while the performance is maintained at a satisfactory level for clients.

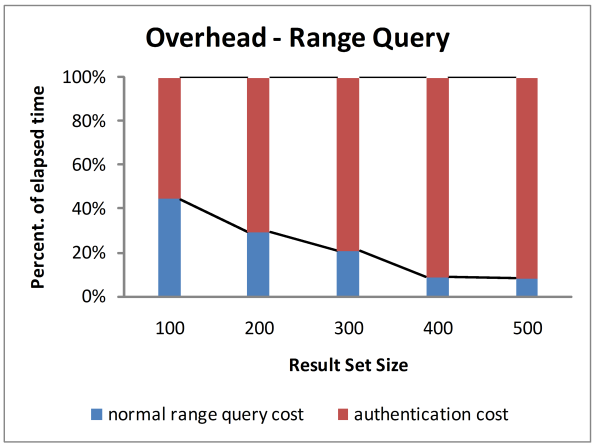

Figure 5: Breakdown of the Average Response Time of Range Queries

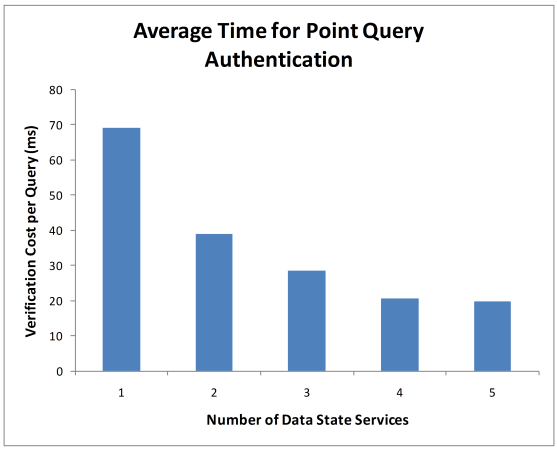

Figure 6: The Effect of Data State Service Number on the Average Authentication Time of Point Queries

5.2 Performance Improvement with Multiple Data State Services

From the above, we can see that a single accountability service faces the scalability problem when the number of client applications of a multitenant platform increases, as the processing of a data state change or an authentication request by an accountability service is slower than the processing of an insert or query operation by a normal data management service. Our collaborative monitoring mechanism addresses the issue effectively. Fig. 6 shows how the average response time for an authentication request of a point query changes along with the number of available data state services.

In this experiment, we setup 3 client applications running on different EC2 instances to continuously send 100 point queries to the data management service and authenticate each result against a randomly selected data state service. The data points in the queries are randomly selected from the dataset. As shown in Fig. 6, when the number of available data state services increases, the average response time for authentication a result is significantly reduced compared with the centralized approach, e..g., the reduction is 59% when the number of data state services goes up to 3.

6 Related Work

Our work is related to research on query authentication. P. T. Devanbu et. al. [6, 11] gives a general Merkle tree based data structure to verify the query results of on-line databases. F. Li et.al. [9] utilizes the data structure to handle the situations where data in an on-line database is periodically updated. H. Pang et. al. [14] uses different hash method to reduce the size of $\mathcal{VO}$. K. Pavlou et.al. [15] gives a method for further detecting *when* and *what* kind of tampering occurs. Most work on query authentication adopt an architecture where the data owner maintains a master database and authenticates the query results of its clients. In most database as a service scenarios, building such a mater database is too expensive to benefit from the cost-effectiveness of outsourcing data management functionalities. Particularly, when the authentication is required to be done on multiple sortable attributes, a separate tree has to be maintained for each attribute. Our approach separates the MB-tree from the data manager and allows distributed parties to share the burden of maintaining the trees.

PeerReview [8] gives a set of protocols for distributed nodes to detect faults and misbehaviors. These nodes collaborate without relying on a centralized coordinator. Our collaborative monitoring mechanism has similarity to PeerReview in terms of decentralization, however, our method has a focus on features such as query result completeness, continuous result authentication and addressing the consistency issues among the distributed state machines.

7 Conclusion

In this paper, we described a mechanism for making a multitenant platform accountable. We gave a centralized and a decentralized model to achieve accountability. The mechanism was developed based on the principle that the involvement of a third party often smooths business transactions between strangers. It enables the state machine of the outsourced data to be maintained outside of the data management service so that the clients know what the data state should be at a certain time. We argue it is an increasingly important type of services to be of-

fered in a multitenancy environment. We also discussed technical challenges for maintaining such a state machine and gave some preliminary experimental results. Further work will be done on issues such as modeling the sliding window size and dealing with logging exceptions.

References

[1] AMAZON INC. Amazon EC2 service level agreement. `http://aws.amazon.com/ec2-sla/`, 2008.

[2] AMAZON INC. SimpleDB consistency enhancements. `http://developer.amazonwebservices.com/connect/entry.jspa?externalID=3572`, February 2010.

[3] ASUNCION, A., AND NEWMAN, D. UCI machine learning repository, 2007.

[4] BENDER, A., SPRING, N., LEVIN, D., AND BHATTACHARJEE, B. Accountability as a service. In *SRUTI'07: Proceedings of the 3rd USENIX workshop on Steps to reducing unwanted traffic on the internet* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–6.

[5] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*, 4 (2002), 398–461.

[6] DEVANBU, P. T., GERTZ, M., MARTEL, C. U., AND STUBBLEBINE, S. G. Authentic third-party data publication. In *DBSec* (2000), pp. 101–112.

[7] GUIRGUIS, S., SHARAF, M. A., CHRYSANTHIS, P. K., LABRINIDIS, A., AND PRUHS, K. Adaptive scheduling of web transactions. In *ICDE* (2009), pp. 357–368.

[8] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. Peerreview: practical accountability for distributed systems. In *SOSP* (2007), pp. 175–188.

[9] LI, F., HADJIELEFTHERIOU, M., KOLLIOS, G., AND REYZIN, L. Dynamic authenticated index structures for outsourced databases. In *SIGMOD Conference* (2006), pp. 121–132.

[10] LIN, K.-J., PANAHI, M., ZHANG, Y., ZHANG, J., AND CHANG, S.-H. Building accountability middleware to support dependable SOA. *IEEE Internet Computing 13*, 2 (2009), 16–25.

[11] MARTEL, C. U., NUCKOLLS, G., DEVANBU, P. T., GERTZ, M., KWONG, A., AND STUBBLEBINE, S. G. A general model for authenticated data structures. *Algorithmica 39*, 1 (2004), 21–41.

[12] MERKLE, R. C. A certified digital signature. In *CRYPTO* (1989), pp. 218–238.

[13] O'HARA, E. A. Trustworthiness and contract. In *Moral Markets: The critical role of values in the economy*, P. J. Zak, Ed. Princeton University Press, 2008, pp. 173 – 203.

[14] PANG, H., AND TAN, K.-L. Authenticating query results in edge computing. In *ICDE* (2004), pp. 560–571.

[15] PAVLOU, K. E., AND SNODGRASS, R. T. Forensic analysis of database tampering. *ACM Trans. Database Syst. 33*, 4 (2008).

[16] VAN ENGELEN, R., AND GALLIVAN, K. The gSOAP toolkit for web services and peer-to-peer computing networks. In *CCGRID* (2002), pp. 128–135.

[17] WANG, C., CHEN, S., AND ZIC, J. A contract-based accountability service model. In *ICWS* (2009), pp. 639–646.

[18] WEISSMAN, C. D., AND BOBROWSKI, S. The design of the force.com multitenant internet application development platform. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data* (New York, NY, USA, 2009), ACM, pp. 889–896.