VMCStore: A TPM-based Trusted Storage Framework

by

Jonathan M. Rhodes

S.B., C.S. M.I.T., 2006

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2007

Author _____
Department of Electrical Engineering and Computer Science
May 25, 2007

Certified by _____
Srinivas Devadas
Professor of Electrical Engineering and Computer Science
Thesis Co-Supervisor

Certified by _____
Luis Sarmenta
Research Scientist
Thesis Co-Supervisor

Accepted by _____
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

This page left intentionally blank

VMCStore: A TPM-based Trusted Storage Framework
by
Jonathan M. Rhodes

Submitted to the
Department of Electrical Engineering and Computer Science

May 25, 2007

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This thesis introduces VMCStore, a framework for developing trusted storage applications on an untrusted server using a trusted platform module (TPM). The framework allows the server to provide trusted storage to a large number of clients, where each client may own and use several devices that may be offline at different times, and may not be able to communicate with each other, except through the untrusted server (over an untrusted network). The clients only trust the server's TPM; the server's BIOS, CPU, and OS are not assumed to be trusted. VMCStore draws on the ideas of *virtual monotonic counters* and *validity proofs* to provide tamper-evident storage, allowing the user to detect modifications to his data, as well as replay attacks. In particular, VMCStore uses TPM/J, a Java-based API for low-level access to the TPM, to create virtual monotonic counters using the monotonic counters and transport sessions of the TPM 1.2. VMCStore also provides a set of three *log-based* validation algorithms, which have been tested over PlanetLab and analyzed in this thesis. The VMCStore framework has been developed in a modular fashion, allowing the user to develop and test new applications and validation algorithms.

Thesis Co-Supervisor: Srinivas Devadas
Title: Professor of Electrical Engineering and Computer Science

Thesis Co-Supervisor: Luis Sarmenta
Title: Research Scientist

Acknowledgments

I would like to take a moment to thank the myriad of people who have made this thesis possible.

First of all, I would like to express my gratitude to my group: Prof. Srinivas Devadas, for three years of encouragement, support and guidance, and without whom, I would have never ended up with this wonderful group; Luis Sarmenta and Marten van Dijk, for countless hours spent together, pouring over ideas and teaching me the ways of the researcher; Sally Lee, for being an infinite source of administrative wisdom, and a friend in times of trouble; and Charlie O'Donnell, for always being willing to give me a helping hand whenever I was getting confused by Linux or life.

I would also like to thank all my friends here at MIT, who have helped so much to keep me grounded when I thought the stress would be too much. Thanks to all my friends from Next House – Shaun, Min, Minji, and Marj – who have faithfully reached out to me when I seemed to all but disappear from this world. Many thanks to all of the Cross Products, for your friendship, prayers, words of encouragement, and reminders that God gives us hope, even in the bleakest of times. Special thanks to Jon Wetzel, my brother-in-arms in the exploration of life, love, and faith, and Cristi Wilcox, for helping me to relax, and reminding me that God always provides a silver lining behind every gray cloud.

And how can I express enough gratitude to the ones who have given me immeasurable support and love for the past 23 years? The guidance, patience, encouragement, and wisdom of my parents has been central to my development – to my understanding of the world, my faith, and who I am. And without their sacrifice and support, I would have never had these opportunities. Many thanks also to my brother, for always being there when I needed someone to talk to, and my grandparents for faithfully reaching out to me, even when I have been too busy to reach back.

And finally, above all, I would like to thank God, for the strength and hope He has given me for the past five years, His faithfulness when I have been unfaithful, and His promise that He has a plan for me and will sustain me to the end.

Commit to the LORD whatever you do, and your plans will succeed. – Proverbs 16:3

Contents

1 Introduction **8**

2 Related Work **10**

3 Abstract System Model and Protocols **13**
3.1 Virtual Counter Manager . . . 13
3.2 Global Clock Operations Using TPM 1.2 . . . 14
3.3 Log-Based Scheme Protocols Overview . . . 15
3.4 Protocol Details . . . 16
3.5 Improvements . . . 21

4 Implementation **23**
4.1 System Overview . . . 23
4.2 CounterHost . . . 24
4.2.1 Interfaces . . . 24
4.2.2 Log-based Implementation . . . 26
4.3 CounterGateway . . . 26
4.3.1 Interfaces . . . 26
4.3.2 Log-based Implementation . . . 27
4.4 Communication . . . 28
4.4.1 Interfaces . . . 28
4.4.2 Sending Data Structures Over the Network: ByteArrayable . . . 28
4.4.3 Implementations . . . 29
4.5 Client . . . 30
4.5.1 Interfaces . . . 30
4.5.2 Log-based Implementation . . . 32
4.6 Simulation . . . 32
4.6.1 Operational Overview . . . 33
4.6.2 Interfaces . . . 33
4.6.3 Data Collection Structures . . . 35

5 Theoretical Data Structures in VMCStore **37**
5.1 Counters and the Virtual Counter Manager . . . 37
5.2 Validation Proofs and Confirmation Certificates . . . 37
5.3 Operations and Operation Certificates . . . 38
5.4 Algorithms, Nonces, and Schedules . . . 38

6 Results **41**
6.1 The Experiment . . . 41
6.2 Experimental Parameters . . . 41
6.3 Experimental Results . . . 42
6.3.1 Operational Efficiency . . . 43
6.3.2 *Simple* . . . 43
6.3.3 *Shared* . . . 49
6.3.4 *Multiplexed* . . . 55
6.4 Conclusion . . . 60

7 Future Work and Conclusions **62**

List of Figures

1 Diagram of the increment-without-validation and read-without-validation protocols. 17
2 Diagram of the validation process. 19
3 Modular diagram of the VMCStore system. 25
4 Detailed diagram of the Simulation module. 34
5 Operational efficiency of the *Simple* algorithm. 44
6 Total operations performed by the *Simple* algorithm. 45
7 Average increment certificate size for the *Simple* algorithm. 45
8 Average read certificate size for the *Simple* algorithm. 46
9 Average server increment time for the *Simple* algorithm. 47
10 Average server read time for the *Simple* algorithm. 48
11 Operational efficiency of the *Shared* algorithm. 50
12 Total operations performed by the *Shared* algorithm. 50
13 Average increment certificate size for the *Shared* algorithm. 51
14 Average read certificate size for the *Shared* algorithm. 52
15 Average server increment time for the *Shared* algorithm. 53
16 Average server read time for the *Shared* algorithm. 54
17 Comparison of the operational efficiency of the *Multiplexed* and *Shared* algorithms. 56
18 Comparison of the total operations performed by the *Multiplexed* and *Shared* algorithms. . . 57
19 Comparison of the average increment certificate size per client for the *Multiplex* and *Shared* algorithms. 57
20 Comparison of the average read certificate size per client for the *Multiplexed* and *Shared* algorithms. 58
21 Comparison of the average server increment time for the *Multiplexed* and *Shared* algorithms. 59
22 Comparison of the average server read time for the *Multiplexed* and *Shared* algorithms. . . . 59

1 Introduction

This thesis presents VMCStore, a framework for developing trusted storage applications on an untrusted server using a trusted platform module (TPM). In particular, VMCStore provides the user with a framework for ***using an untrusted server with a trusted platform module (TPM) to provide trusted storage for a large number of clients, where each client may own and use several different devices that may be offline at different times and may not be able to communicate with each other except through the untrusted server (over an untrusted network)***[45][1]. This idea was first presented in [35], and expanded upon in [45]; this thesis presents an implementation of these ideas, and analyzes the limitations of this approach in real-life scenarios.

The research presented in this thesis is driven by two primary motivations:

a *Promoting the TPM*

The TPM is still a relatively new technology. Recently, a number of applications have been developed, providing a range of TPM-based services, from file and folder encryption [15], to file system tamper-detection [33], to full-volume encryption [30]. However, the TPM still requires a more solid application base to gain greater recognition and evolve into a critical component of modern computer systems. Therefore, one goal of VMCStore is to demonstrate trusted storage as another possible application of the current version of the TPM (1.2).

b *Developing New Trusted Storage Applications*

Since the TPM is not as susceptible to external software attacks as traditional software-based cryptographic service providers [44], it is possible that a TPM-based trusted storage system would make certain applications more feasible. Another goal of VMCStore, then, is to encourage the pursuit of new trusted storage applications, which could help the TPM find a pervasive role in society. By manipulating the roles of the client and the server, many new applications can be imagined, ranging from peer-to-peer systems, to count-limited certificates and DRM [45, 35].

With these two motivations in mind, the VMCStore system has been developed as a modular framework, allowing the user to manipulate and extend the system to develop and test new algorithms and applications. Four log-based validation algorithms are included as examples of how to use the VMCStore framework.

[1] For more a more detailed discussion of the problem statement, please refer to the Problem Statement section of [45].

The rest of this thesis is organized as follows. Section 2 first presents an overview of work and systems related to VMCStore. The abstract system model and protocols behind VMCStore, adapted from [45], are then presented in Section 3. Section 4 provides an overview of the VMCStore system. This is followed by a description of how the theoretical data structures are implemented in VMCStore, in Section 5, followed by an analysis of the four log-based validation algorithms in Section 6. Future directions of VMCStore are considered along with the conclusion of the thesis in Section 7.

2 Related Work

Protecting and validating the integrity of data storage has been a well studied topic due mainly to its high importance to a wide range of applications. For this, cryptographic one-way hash functions [11] are often used by a client to create a small local checksum of large remote data. Merkle proposed hash trees (authentication trees) as a means to update and validate data hashes efficiently by maintaining a tree of hash values over the objects [28]. Recent systems [10, 13, 9, 22, 26] make a more distinct separation between untrusted storage and a trusted compute base (TCB), which can be a trusted machine or a trusted coprocessor. These systems run a trusted program on the TCB (usually a trusted machine or machine with a trusted coprocessor) that uses hash trees to maintain the integrity of data stored on an untrusted storage. The untrusted storage is typically some arbitrarily large, easily accessible, bulk store in which the program regularly stores and loads data which does not fit in a cache in the TCB.

The work on certificate authentication trees in [17] has led to the introduction of authenticated dictionaries [32] and authenticated search trees [4, 3]. In the model of authenticated dictionaries, a trusted source maintains all the data which is replicated over multiple untrusted directories. Whenever the trusted source performs an update, it transmits the update to the untrusted directories. The data is maintained in an authenticated tree structure whose root is signed together with a timestamp by the trusted source. If a user/client queries an untrusted directory, then it uses the tree structure and the signed root to verify the result. A persistent authenticated dictionary [1, 14, 23, 24] maintains multiple versions of its contents as it is modified. Timeline entanglement [25] creates a tamper-evident historic record of different persistent authenticated dictionaries maintained by mutually distrusting sources.

Byzantine-fault-tolerant file systems [7, 8] consist of storage state machines that are replicated across different nodes in a distributed system. The system provides tamper-evidence and recovery from tampering, but ***both*** properties rely on the assumption that at least two-thirds of the replicas will be honest. The expectation is that replicas are weakly protected, and possibly hostile, so the difficulty of an adversary taking over k hosts increases significantly with k. Byzantine-fault-tolerant file systems distribute trust but assume a threshold fraction of honest servers.

SUNDR [27, 20] is another general-purpose, multi-user network file system that uses untrusted storage servers. SUNDR protects against ***forking attacks***, which is a form of attack where a server uses a replay attack to give different users a different view of the current state of the system. SUNDR does not prevent

forking attacks, but guarantees ***fork consistency***, which essentially ensures that the system server either behaves correctly, or that its failure or malicious behavior will be detected later ***when users are able to communicate with each other***. This is achieved by basing the authority to write a file on the public keys of each user. Plutus [16] is another efficient storage system for untrusted servers that cannot handle these forking attacks.

In our system, we place a small TCB in the form of a TPM at an untrusted third party. This TCB is used to maintain virtual counters for timestamping, which allows client devices to ***immediately*** (without the need to communicate with any of the other client's devices) detect server misbehavior whenever a critical operation needs to be performed. In our system, server misbehavior also includes replay attacks. Since replay attacks and other server misbehavior can be detected immediately, forking attacks are prevented, and data freshness, integrity, and consistency is guaranteed.

We reduce the trusted computing base to only a single TPM 1.2 [43, 31], which is a standard component on machines today. This differs from many other systems that require complex secure processors [38, 2, 21, 46, 40, 41, 39].

We use the TPM as a secure counter which can be used to timestamp events according to their causal relations. An order in logical time can be extracted if we know which events logically cause other events. Lamport clocks [18, 19] are conceptual devices for reasoning about event ordering. Our scenario with a centralized untrusted third party which implements a virtual counter manager with access to the TPM does not need Lamport clocks to reason about logical time. Our difficulty is how to reason about malicious behavior.

For completeness we mention that in accountable time-stamping systems [5, 6] all forgeries can be explicitly proven and all false accusations explicitly disproven; it is intractable for anybody to create a pair of contradictory attestations. Buldas et al. [4] introduced a primitive called ***undeniable attester***. Informally, the attester is such that it is intractable to generate both a positive and a negative attestation based on an element x and a set S such that the attester concludes $x \in S$ for the positive attestation and $x \notin S$ for the negative attestation. Their intention is to prove the existence or non-existence of objects in a database. An ***authenticated search tree*** [17, 32, 4, 3] can be used for the construction of an undeniable attester.

One technique also worth noting is that described by Schneier and Kelsey for securing audit logs on untrusted machines [36, 37]. Each log entry contains an element in a linear hash chain that serves to check

the integrity of the values of all previous log entries. It is this element that is actually kept in trusted storage, which makes it possible to verify all previous log entries by trusting a single hash value. The technique is suitable for securing append-only data that is read sequentially by a verifying trusted computer.

Another concept worth noting is that of a certificate revocation list (CRL), which is a list signed by a certification authority (CA) together with a timestamp. By obtaining the most recent CRL, one can easily verify whether a certificate is still valid. Based on CRLs, less communication intensive solutions are proposed in [29, 17, 32]. In [29] the idea is proposed to sign a message for every certificate stating whether it is revoked or not, and to use an off-line/on-line signature scheme [12] to improve the efficiency. The idea to use a log of certificates in which each of the certificates attests to a positive or a negative action is a general principle which we also use in our approach (we use a log of increment certificates in which each of the increment certificates attests to whether a certain virtual monotonic counter has been incremented or not).

3 Abstract System Model and Protocols

The VMCStore system is a proof-of-concept implementation based on ideas proposed by Luis Sarmenta and Marten van Dijk in the Computation Structures Group of CSAIL at MIT. The following section is adapted from [45], and provides an overview of the theory behind *virtual monotonic counters* and *validity proofs*, and how they can be used to build a trusted storage system. The relationship between the abstract structures and protocols and implemented classes are explained in Section 5. Implementation details of the system model are explained in Section 4.

To provide some context for the following section, note that the *virtual monotonic counters* will be built on top of the monotonic counters included in the TPM 1.2. According to the TPM 1.2 specification, while the TPM may contain multiple monotonic counters, only one may be used during a particular boot-cycle [43]. Because of this, we will need a module to act as a manager to virtualize this resource. This module is called the *virtual counter manager*.

3.1 Virtual Counter Manager

In order to implement a virtual counter manager based on TPM 1.2 we use one of the TPM's built-in physical monotonic counters as a "***global clock***". Essentially, we use the TPM with its global clock as a timestamping device to timestamp each virtual counter. Whenever we wish to increment a virtual counter, we increment the global clock and timestamp the virtual counter's ID with the incremented value of the global clock. We then define the value of a specific virtual counter as the value of its most recent timestamp. In other words, ***the value of a particular virtual counter is defined as the value of the global clock at the last time that the virtual counter's increment protocol was invoked***.

Note that this results in ***non-deterministic*** monotonic virtual counters [35]. The unpredictability of the virtual counter's increments relative to the global clock means that a client needs to check every increment that is performed on the global clock.[2] That is, in order to verify whether a retrieved virtual counter value is fresh and valid, the client's device needs to examine a ***log*** of timestamps that resulted from increments on the global clock. Checking each timestamp determines whether the client's particular virtual counter has been incremented (i.e., timestamped by the global clock) more recently. This is because these applications only need to be able to tell if the value of a monotonic counter has changed from its previous value or not.

[2] Except when using time-multiplexing, which is described in Section 3.5.

It does not matter what the new value is, as long as it is different from any other value in the past [35].

In the following section, we explain the details of our ***log-based scheme*** based on this idea. We first explain how to use a TPM 1.2 chip to implement two global clock primitives: *IncAndSignClock* – which increments and signs the resulting global clock value together with an input nonce, and *ReadAndSignClock* – which reads and signs the current value of the global clock together with an input nonce. We then show how these primitives can be used to implement read and increment protocols for individual virtual counters.

3.2 Global Clock Operations Using TPM 1.2

A TPM 1.2 chip has three features which are useful for our purposes. First, the TPM has the ability to hold an ***attestation identity key (AIK)***, which is a unique signing keypair whose private key is never revealed outside the TPM, and whose public key is certified by a trusted third party (and can be verified through this certificate without contacting the trusted third party). Second, the TPM has at least one built-in (or "physical") ***monotonic counter*** whose value is non-volatile (i.e., it persists through reboots), and monotonic (i.e., it can be increased by 1, but it can never be reverted to an older value, even if one has complete physical access to the entire machine hosting and invoking the TPM). Third, the TPM supports ***exclusive and logged transport sessions***, which allow the TPM to prove to an external party that it has executed certain operations (atomically) by signing (with the AIK) a log of the operations performed on the TPM together with their inputs and outputs and an anti-replay nonce. These features allow us to implement our schemes by using the TPM's built-in monotonic counter as the global clock, and using the AIK and transport sessions to produce trusted signatures, or timestamps, using this global clock.

Specifically, we implement the *IncAndSignClock*$(nonce)$ primitive by using the TPM's built-in TPM_Increment_Counter command (which increments the TPM's built-in monotonic counter) inside an exclusive and logged transport session using the AIK as the signing key. This produces a signature over a data structure that includes the anti-replay nonce and a hash of the transport session log, which consists of the inputs, commands, and outputs encountered during the entire transport session. This signature can then be used together with the input nonce $nonce$ and the transport session log, to construct an ***increment certificate***. Note that by making this transport session exclusive, we ensure that the TPM will not allow other exclusive transport sessions to successfully execute at the same time. This ensures the ***atomicity*** of the increment operation.

The verification algorithm for such an increment certificate is as follows: First, it checks that the nonce in the certificate is the same as the input nonce. If they are the same, the input nonce $nonce$ together with the transport log, the signed output, and the certified public key of the TPM's AIK is used to verify the certificate. Finally, if the certificate verifies as valid, the algorithm retrieves the global clock's value, which is included in the transport session log of inputs and outputs as part of the certificate.

The *ReadAndSignClock*($nonce$) primitive is implemented like the *IncAndSignClock*($nonce$) primitive where the TPM_Increment_Counter command in the transport session is replaced by the TPM's built-in TPM_Read_Counter command. In this case, instead of an increment certificate, we produce a ***current global clock certificate*** certifying the current value of the global clock (i.e., TPM's built-in monotonic counter).

3.3 Log-Based Scheme Protocols Overview

We begin by assuming that each client has her own unique public-private key pair stored in each of her devices. A client's devices use the client's private key to sign increment requests and create ***confirmation certificates***. Confirmation certificates are produced and given to the virtual counter manager whenever a client's device successfully performs a read or increment with validation, as defined below.

On the manager-side, we assume that the ***virtual counter manager*** has a certified trusted TPM 1.2 chip, and some software, memory, and persistent (e.g., disk) storage, which can all be untrusted. The software on the virtual counter manager keeps track of:

1 an array of the ***most recent*** confirmation certificates for each virtual counter, and

2 an array of each of the increment certificates which were generated since the generation of the ***oldest*** most recent confirmation certificate.

Using these, together with the TPM and the global clock operations described earlier, the virtual counter manager implements four protocols for operating on individual virtual counters:

1 ***Increment-without-validation (aka Fast-Increment)***, in which a client's device requests to increment one of the client's virtual counters, and which results in an increment certificate,

2 ***Read-without-validation (aka Fast-Read)***, in which the virtual counter manager returns the current value of a virtual counter,

3 ***Read-with-validation (aka Full-Read)***, in which not only the current value of a virtual counter is returned, but also a proof of the validity of this virtual counter, and

4 ***Increment-with-validation (aka Full-Increment)***, which combines the increment-without-validation and read-with-validation protocols into a single protocol.

The read and increment protocols with validation produce a ***validity proof***, which is composed of:

1 the most recent confirmation certificate of the corresponding virtual counter, together with

2 a list (or log) of each of the increment certificates which were generated since the creation of this most recent confirmation certificate, and

3 a current global clock certificate or a new increment certificate.

Given these, a client can reconstruct the global clock values at which the virtual counter was incremented since the creation of the most recent confirmation certificate. This reconstruction detects any malicious behavior in the past. Specifically, the reconstruction of past increments is used to determine whether the virtual counter values on which these increments were based are valid. That is, as described in (5) below, for each of the past increments by any of the client's devices, the client checks whether the increment was based on a retrieved counter value (received from the virtual counter manager during one of its protocols) that is equal to the current counter value just prior to the increment. This check is made possible by having an increment certificate also certify the value on which the corresponding increment is based (hence, the consistency of the reconstructed list of increments can be verified).

We note that in order to verify the freshness of data it is sufficient to verify the validity of the counter values on which past increments were based, since only newly incremented values are used for timestamping data in our virtual storage application.

3.4 Protocol Details

We proceed with the details of the different protocols:

Increment-without-Validation protocol: Figure 1.a shows the interaction between a client device, the virtual counter manager, and its TPM during an increment-without-validation protocol. If a client's device

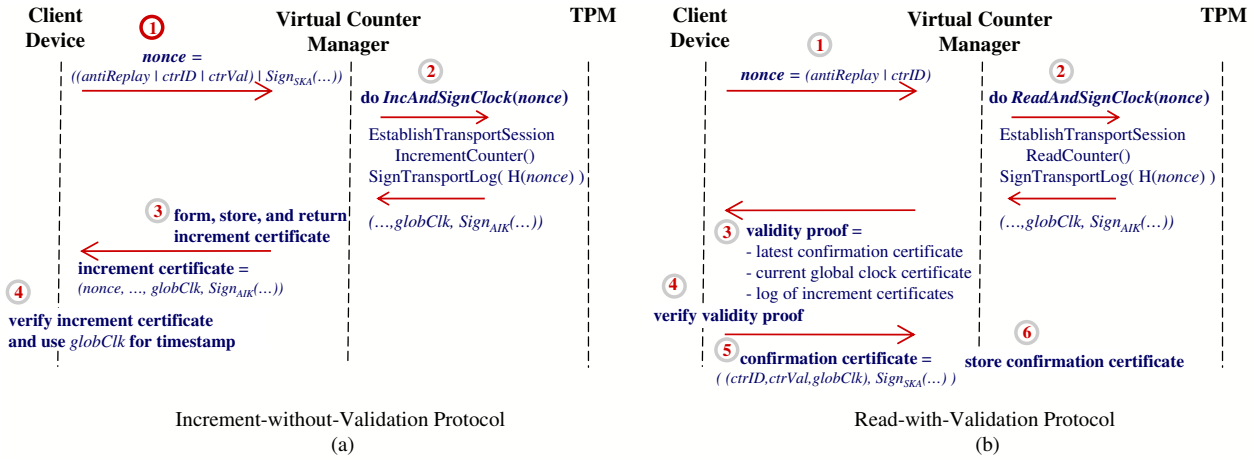

Figure 1: Protocols. (a) Increment-without-Validation. (b) Read-with-Validation. (Note: Increment-with-Validation is similar to Read-with-Validation, except that an increment operation is performed and the increment certificate is used in place of the current global clock certificate.)

wants to increment one of the client's virtual counters, then it selects a random anti-replay nonce $antiReplay$ and concatenates the anti-replay nonce, the counter identity $ctrID$ of the virtual counter which needs to be incremented, and the current value $ctrVal$ of this counter according to the knowledge of the client's device. Let SK be the client's secret key. The device computes

$$nonce = (conc \| Sign_{SK}(conc)) \text{ where } conc = (antiReplay \| ctrID \| ctrVal). \quad (1)$$

The nonce is forwarded to the virtual counter manager with the request to use nonce as the input $nonce$ of the *IncAndSignClock* primitive (step 1 in Figure 1.a). Besides verifying the nonce's signature, the virtual counter manager checks whether the current value of the counter with identity $ctrID$ is equal to $ctrVal$. If not, then the virtual counter manager notifies the client's device about its out-of-date knowledge. If $ctrVal$ does match the current counter value, then the virtual counter manager uses the TPM to execute *IncAndSignClock*($nonce$) (step 2 in Figure 1.a) and the resulting increment certificate is sent back to the client's (step 3 in Figure 1.a) device, which verifies the certificate. In this scheme we do not protect against denial of service; if the increment certificate does not arrive within a certain time interval, then the client's device should retransmit its request with the same nonce. As soon as an increment certificate is accepted (that is, its verification passed in step 4 in Figure 1.a), the client's device may use the new counter value to timestamp data. If the client's device accepts the increment certificate, then we call the increment ***successful***.

Since the anti-replay nonce is chosen at random, replay attacks of previously generated increment certifi-

cates (by, for example, a malicious virtual counter manager or a man-in-the-middle) will be detected by the client's device. We will show that the validity of a client's virtual counter can be verified in the read-with-validation protocol (even in the presence of a malicious virtual counter manager, but with a trusted TPM) because the client's device's knowledge of the current counter value $ctrVal$ is included in the input nonce of the *IncAndSignClock* primitive. The role of the counter ID in the input nonce is to distinguish which increment certificates correspond to which virtual counters. The nonce's signature proves the authenticity of the request to the virtual counter manager. The signature is also used in the read-with-validation protocol to prove that each increment certificate originated from an authentic request and not a fake increment.

Read-without-Validation protocol: In the read-without-validation protocol, a client's device asks for the most recent value of a specific virtual counter. The virtual counter manager simply signs and returns the most recent counter value without making use of the TPM.

Read-with-Validation protocol: Figure 1.b shows the interaction between a client device, the virtual counter manager, and its TPM during a read-with-validation protocol. If a client's device wants to read and obtain a validity proof of the value of one of the client's virtual counters, it first selects a random anti-replay nonce $antiReplay$. Let $ctrID$ be the counter ID of the virtual counter which current value needs to be returned by the protocol. The concatenation,

$$nonce = (antiReplay \| ctrID),$$

is forwarded to the virtual counter manager (step 1 in Figure 1.b) with the request to use it as the input nonce of the *ReadAndSignClock* primitive. The virtual counter manager uses the TPM to execute *ReadAndSignClock*($nonce$) (step 2 in Figure 1.b). The resulting current global clock certificate is transmitted to the client's device together with the most recent confirmation certificate of the virtual counter, $ctrID$, and a log of each of the increment certificates generated since the creation of this most recent confirmation certificate (step 3 in Figure 1.b). The sequence of certificates in this transmission form a validity proof as depicted in Figure 2.

The most recent confirmation certificate is a certificate generated by one the client's devices during a previous read-with-validation or increment-with-validation protocol. It is a certificate of the concatenation of $ctrID$, the value $globClk'$ of the global clock at the time when either protocol was executed, and the

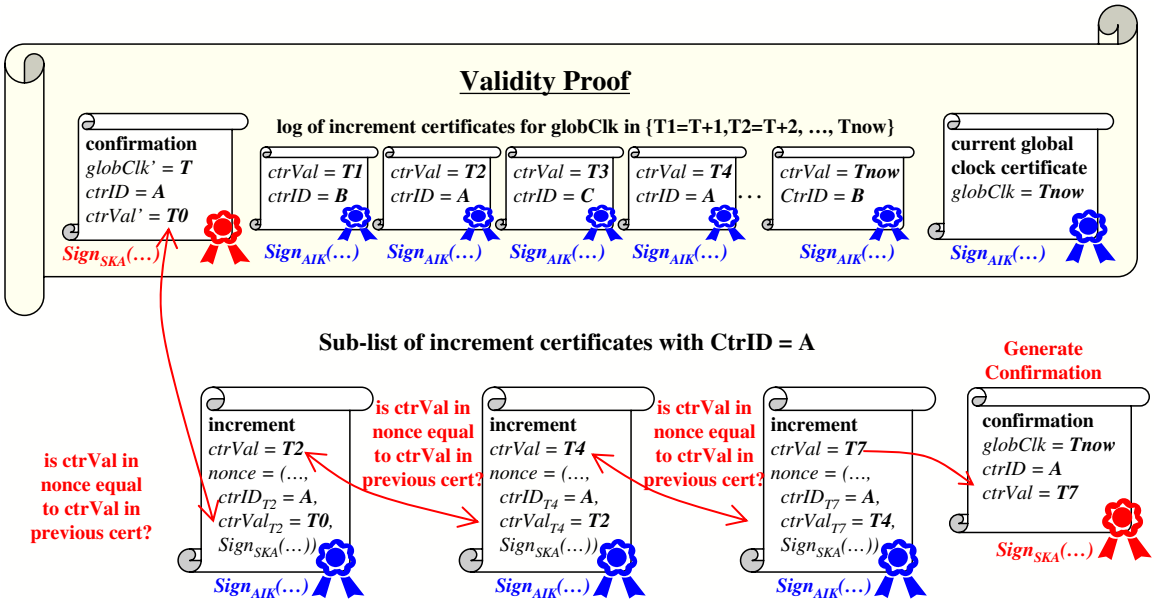

Figure 2: The Validation Process. After receiving the validity proof, the client checks the AIK signatures on all the increment certificates, extracts the certificates with increments for its counter ID, and verifies that each increment was done with the correct knowledge of the previous value of the virtual counter. If this validation succeeds, the client produces a new confirmation certificate.

value $ctrVal'$ of the counter with identity $ctrID$ at the time when either protocol was executed. That is, the confirmation certificate is a pair

$$(confirm, Sign_{SK}(confirm)) \text{ where } confirm = (ctrID \| ctrVal' \| globClk') \tag{2}$$

and SK is the client's secret key.

Let $globClk$ be the global clock value as certified by the current global clock certificate. Since the anti-replay nonce is chosen at random, replay attacks of previously generated current global clock certificates will be detected by the client's device. Therefore, we may assume that $globClk$ represents the current global clock value.

The client's device also verifies each of the signatures in the log of increment certificates (which were generated since the creation of this most recent confirmation certificate), and uses the client's public key to verify the confirmation certificate. Then, for each value t in the range

$$globClk' < t < globClk, \tag{3}$$

there should exist an increment certificate for which its verification algorithm retrieves the value t (see

the log of increment certificates in Figure 2). See (1), let

$$conc_t = (antiReplay_t \| ctrID_t \| ctrVal_t) \tag{4}$$

be part of the input nonce of the increment certificate for t. The list of these input nonces contains a sublist of input nonces with $ctrID_t = ctrID$ (see the sublist of increment certificates in Figure 2). The sublist of these input nonces corresponds to all the increments of the counter with identity $ctrID$ during the period where the global clock value ranged from $globClk'$ to the current global clock value $globClk$. According to (1), each of the input nonces within the sublist contains within itself a signature that can be verified by using the client's public key. This allows the client to confirm that the corresponding increment certificates originated from an authentic request and not a fake increment.

The sublist can also be used by the client's device to detect whether the increments are based on valid counter values. Using (4), let $ctrVal_t$ be one of the values in the input nonce within the sublist. Then, counter value t resulted from an increment protocol which was initiated by one of the client's devices who thought that, just before the start of the increment protocol, the value of the counter is equal to $ctrVal_t$. Hence, if the counter with identity $ctrID$ has behaved like a valid counter, then, for each pair of consecutive input nonces with $ctrVal_t$ and $ctrVal_T$, $t < T$, within the sublist,

$$t \text{ should be equal to } ctrVal_T. \tag{5}$$

If this check passes, then the last value $ctrVal$ within the sublist is the current value of the virtual counter, $ctrID$, and the client's device transmits to the virtual counter manager (step 5 in Figure 1) a new confirmation certificate, described in (2), in which

$$confirm = (ctrID \| ctrVal \| globClk).$$

Increment-with-Validation protocol: The increment-with-validation protocol first executes the increment-without validation protocol. The resulting increment certificate contains the current global clock value as the incremented virtual counter value. In this sense, the increment certificate can also function as a current global clock certificate. The increment certificate can then be used in the read-with-validation protocol, in

place of the *ReadAndSignClock* primitive, to provide the validity proof for increment-with-validation.

3.5 Improvements

In this subsection we explain two techniques that can be used to improve the performance of the log-based scheme. The details of both techniques are presented in Appendix A of [45].

Sharing. One problem with the log-based scheme as described so far is that each read and increment primitive on a virtual counter requires the TPM to produce a signature using its AIK. As we will show in Section 6, read and increment signature operations typically takes around 1.3s using existing TPM 1.2 chips today. Moreover, TPM 1.2 chips also ***throttle*** increment operations on the TPM's built-in monotonic counter to prevent wear-down of the TPM's nonvolatile memory. This means that increment operations are only possible once every 2 to 5 seconds (depending on the manufacturer). As we will show in Sect. 6, if we only allow one virtual counter to be incremented for each increment of the global counter, then a single TPM can only handle a few virtual counters before the overall performance becomes unacceptably slow.

A solution to this problem is to allow multiple increment protocols of independent virtual counters to be executed at the same time, sharing a single global clock primitive. The general idea here is to collect the individual nonces of each increment protocol and to construct a single shared nonce which can then be used as an input to a single shared *IncAndSignClock* primitive.

Time-Multiplexing. The log-based scheme has another significant drawback: if a virtual counter v is not incremented while other counters are incremented many times, then the validity proof for v would need to include the log of all increments of all counters (not just v) since the last increment of v. The length of this log can quickly grow very large.

A solution to this problem is to time-multiplex the global clock. That is, instead of allowing increments at each possible global clock value for each client, each client associates with each of his virtual counters a ***fixed schedule*** of global clock values that are allowed to be virtual counter values. The main advantage of time multiplexing is that the log of increment certificates in a validity proof of a virtual counter can be reduced to those for which the corresponding verification algorithm retrieves a value which is allowed, according to the schedule of the virtual counter.

The disadvantage of time-multiplexing is a possible increase in the latency between the request and

finish of an increment. In order to reduce the effects of this problem we may use an ***adaptive schedule***. Under this scheme, a virtual counter's schedule is allowed to change. For example, the client's devices may agree to a back-off strategy. Immediately after a successful increment, the legal increment slots for a virtual counter would be close together. As the client's virtual counter remains idle, the legal increment slots are spaced farther and farther apart according to a known deterministic formula.

Finally, note that while there is an advantage to multiplexing increments, the same advantage does not apply to counter reads. Reads do not increase the length of the validity proof, so multiplexing them will not reduce the size of the read certificate. In fact, delaying reads by multiplexing may increase the size of the validity proof by allowing more clients to perform increments before the validity proof is generated. For these reasons, reads in the time-multiplexed scheme are not multiplexed, but handled on-demand.

4 Implementation

From an implementation point-of-view, the main goal of VMCStore is to develop and analyze various TPM-based trusted storage algorithms specified in [45], and discussed in Section 3. However, the system itself has been developed with a very modular design in mind, with the hope that this framework can be used to experiment with and test future algorithms as well. This section will give an overview of the various modules of VMCStore, and how they interact with each other. More information on data structures referenced in this section can be found in Section 5.

4.1 System Overview

VMCStore is written using Java 1.5 and TPM/J. TPM/J is an object-oriented API using Java for low-level access to the TPM [34]. While VMCStore is portable to any operating system capable of running a JVM, TPM-compatibility is the responsibility of TPM/J. At present, TPM/J has been shown to work under Windows XP, Linux, Mac OS X, and Windows Vista.

The VMCStore framework is divided into four primary modules: CounterHost, CounterGateway, Communication, and Client. The fifth module, Simulation, is any higher-level user or application that wishes to interact with the trusted storage system. The sixth module, Storage, is not specified by VMCStore, since it is heavily application-dependent. The interaction among these six modules can be seen in (Figure 3).

- *CounterHost*: Manages the actual incrementing and reading of the counters (e.g., the *virtual monotonic counters* implemented on the TPM 1.2).
- *CounterGateway*: Manages access to the CounterHost, directing client request threads to the appropriate timeslot.
- *Communication*: Manages the marshaling and unmarshaling of requests and responses between the Client and the CounterGateway. Two implementations have been included, providing communication over sockets or RMI.
- *Client*: Provides an interface between the user and the CounterHost, managing counter information, marshaling requests, and verifying proofs of freshness. If a Storage server is present, the Client will usually manage sending and retrieving files and timestamps from the storage server.

- *Simulation*: The user or application that interacts with the VMCStore framework. The package included in VMCStore imitates the user, providing a way to perform distributed experiments for collecting data and statistics about system performance.

- *Storage*: Stores the user data with the appropriate timestamp from the CounterHost. The Storage server is application-dependent.

As mentioned previously, VMCStore has been implemented with the goal of allowing the user to experiment and test his own algorithms. The algorithms discussed in Section 5.4 have been included as examples, and are referenced below by their collective package name, *logbased*. Sample communication protocols (e.g., RMI, sockets) have also been included. However, any of these modules can be swapped-out with user defined modules and algorithms.

4.2 CounterHost

The CounterHost manages the lowest-level interaction with the counters (e.g., reading and incrementing), as well as the creation of validity proofs for the client.

4.2.1 Interfaces

The key interface for this module is CounterHost. Three methods are defined by the interface, closely following the protocol defined in Section 3.4.

- *ReadCertificate readCounter(ByteArrayable nonce, ByteArrayable counterID)*:
  Returns a ReadCertificate for the specified virtual counter, using the given nonce. ReadCertificate is a wrapper interface which extends CountStamp, and should include the signed counter information from a TPM read operation. The ReadCertificate may contain additional information, such as a validity proof.

- *IncCertificate incrementCounter(ByteArrayable nonce, ByteArrayable counterID)*:
  Returns an IncCertificate for the specified virtual counter, using the given nonce. IncCertificate is a wrapper interface which extends CountStamp, and should include the signed counter information from a TPM increment operation. For increment-with-validation operations, the IncCertificate should also contain a validity proof.

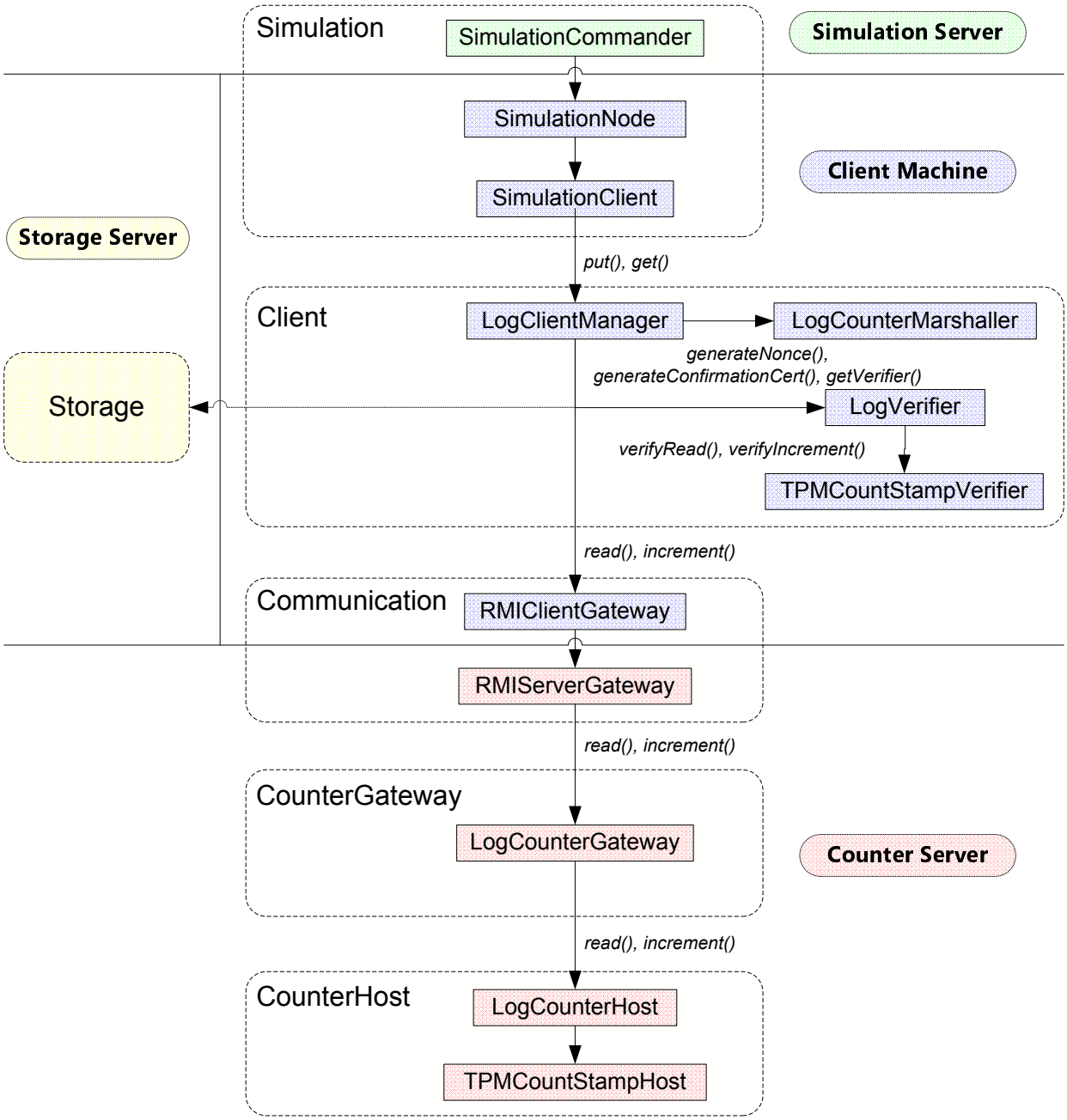

Figure 3: Modular diagram of the VMCStore system. The classes listed inside the modules are examples of those included for the logbased algorithms. Other possible combinations (e.g. using sockets in place of RMI) are included with VMCStore.

- *void sendConfirmationCertificate(ConfCertificate cert)*:

  Provides a way for clients to send confirmation certificates to the virtual counter manager (CounterHost). ConfCertificates, defined in Section 5.2, include the identity of the virtual counter, the time of the last successful increment, and a CountStamp defining the current time of the confirmation certificate. The ConfCertificate is also signed by the client, and provides a *verifyCertificate* method to confirm its validity.

4.2.2 Log-based Implementation

The *logbased* package provides a *LogCounterHostFactory*, which should be used to create *LogCounterHost* instances that follow the increment, read, and validation protocols defined by the algorithms in Sections 3.4 and 3.5. Available algorithms are defined by the *LogAlgorithm* enum.

The LogCounterHostFactory provides one method for creating LogCounterHost instances: *getLogCounterHostInstance()*. This method requires a LogAlgorithm enum, as well as a TPM driver, the global counter ID, and a signing key.

TPMCountStampHost provides a thin layer between the LogCounterHost and the TPM/J CountStamp functions. Both the read and increment operations of the TPMCountStampHost return a TPMCountStamp. Since the TPMCountStamp is operation-independent, and implements neither ReadCertificate nor IncCertificate, the TPMCountStampHost is not a true CounterHost. However, this class is simply intended as a wrapper and is never exposed to the LogCounterGateway.

4.3 CounterGateway

The CounterGateway sits between the clients and the CounterHost. It manages client requests, and is responsible for ensuring that clients only perform increments during legal increment timeslots.

4.3.1 Interfaces

The *CounterGateway* interface provides three methods, which closely mirror the CounterHost interface methods. The CounterGateway methods prepackage the return data for network communication (see Section 4.4.2):

- *ByteArrayableWrapperInterface <ReadCertificate> readCounterRemote(ByteArrayableWrapperInterface<? extends ByteArrayable> counterID, ByteArrayableWrapperInterface<? extends ByteArrayable> nonce)*
- *ByteArrayableWrapperInterface<IncCertificate> incrementCounter(ByteArrayableWrapperInterface<? extends ByteArrayable> counterID, ByteArrayableWrapperInterface<? extends ByteArrayable> nonce)*
- *void sendConfirmationCertificate(ByteArrayableWrapperInterface<? extends ConfCertificate>)*

CounterGateway extends *CommunicationGateway*, removing the throws IOException clauses included by its super-interface (see Section 4.4).

4.3.2 Log-based Implementation

The *logbased* package provides a LogCounterGatewayFactory, which should be used to create LogCounterHost instances that follow the algorithms defined in Section 5.4. The LogCounterGatewayFactory provides one method for creating LogCounterGateway instances: *getLogCounterGatewayInstance(LogAlgorithm, LogCounterHost)*.

The LogCounterGateway implementations are all multi-threaded, treating each thread as a separate client. Operations are broken down into timeslots, and all threads scheduled for a given timeslot are allowed to proceed together. This allows the LogCounterGateway to collect together individual nonces to form a shared nonce, and parallelize the usage of TPM resources.

The *Multiplexed* and *Adaptive* algorithms require a further extension to this threading structure. Since clients must wait for a legal timeslot before their increment can complete, a *dummy thread* is included which increments the TPM when no other clients are available. This ensures that clients will never wait indefinitely for their timeslot to arrive.

The implementations of LogCounterGateway included in *logbased* also perform simple resource maximization by alternating increments and reads whenever possible. As discussed in section 6, TPM's typically require a cool-down period between increments to extend the life of the TPM's non-volatile RAM. Reads, however, can still be performed during this cool-down period. We found that alternating reads and increments provides good TPM utilization for the TPM's we have worked with.

LogCounterGateway also implements the CounterHost and SimulationGateway interfaces. Because they are intended to be used for simulation, the increment and read methods of LogCounterGateway also return

instances of StatPackagedCertificates, instead of ByteArrayableWrappers. More information on the SimulationGateway interface and StatPackagedCertificates can be found in section 4.6.

4.4 Communication

Communication is responsible for marshaling and unmarshaling requests and responses between the Client and the CounterGateway.

4.4.1 Interfaces

The primary communication interface is CommunicationGateway. This is the super-interface for CounterGateway, exposing the same methods, but also requiring implementations to include throws clauses for IOExceptions caused by network communication.

- *ByteArrayableWrapperInterface<ReadCertificate> readCounterRemote(ByteArrayableWrapperInterface<? extends ByteArrayable> counterID, ByteArrayableWrapperInterface<? extends ByteArrayable> nonce) throws IOException*
- *ByteArrayableWrapperInterface<IncCertificate> incrementCounter(ByteArrayableWrapperInterface<? extends ByteArrayable> counterID, ByteArrayableWrapperInterface<? extends ByteArrayable> nonce) throws IOException*
- *void sendConfirmationCertificate(ByteArrayableWrapperInterface<? extends ConfCertificate>) throws IOException*

There is also a SimulationCommunicationGateway which performs a function for SimulationGateway analogous to the relationship between CommunicationGateway and CounterGateway. See Section 4.3 for more information.

4.4.2 Sending Data Structures Over the Network: ByteArrayable

Most of the data structures defined by VMCStore implement the *ByteArrayable* interface defined by TPM/J (REF). ByteArrayable objects provide *toBytes()* and *fromBytes()* methods, which can be used to reduce an

object to a byte array, or reconstruct the object from a byte array. The ByteArrayable does not extend *Serializable*, so if a ByteArrayable object needs to be sent over the network, it must be reduced to a byte array, implement Serializable, or be wrapped within an object implementing the *ByteArrayableWrapperInterface*.

Two classes have been included in VMCStore for wrapping ByteArrayable objects: *ByteArrayableWrapper* and *StatPackagedCertificate*. ByteArrayableWrapper is simply a generic wrapper for ByteArrayables; StatPackagedCertificates are used by the Simulation modules for passing around simulation measurements along with increment and read certificates (see Section 4.6).

4.4.3 Implementations

VMCStore contains two built-in implementations of CommunicationGateway. Both of these packages also implement *SimulationCommunicationGateway* (see Section 4.6). Available implementations are defined by the *CommunicationType* enum.

The *Utils* class contains a method, *generateCommunicationGateway()*, which can be used to instantiate a CommunicationGateway from a Class object. The CommunicationGateway must expose a constructor that takes a hostname as a String and a port as an *int* for this method to work.

RMI. The first implementation takes advantage of the built-in Java RMI system (REF). The counter gateway-side class is *RMIServerGateway*. Java's *rmiregistry* program must be running on the counter gateway machine before RMIServerGateway can be started.

The client-side class is called *RMIClientGateway*. It will try to find an instance of the *RMIServerGatewayInterface* on the specified host.

All experiments in Section 6 were performed using the RMI communication protocols.

Sockets. A second implementation is included which communicates directly over Java *Socket* objects. Marshalling is handled by sending message objects, which extend *AbstractMessage*, with the necessary parameters.

The counter gateway-side class is *SocketServerGateway*. This class runs as a separate thread on the counter gateway server, accepting incoming connections and creating *ServerSocketThreads*. Future communication between the Client and the CounterGateway is directly between the *SocketClientGateway* and its ServerSocketThread.

The client-side class is called SocketClientGateway. It marshals requests as messages, and unmarshals responses for the client. If the socket connection is lost, SocketClientGateway will attempt to reestablish a connection.

While this implementation may be lighter-weight than the RMI implementation, our current implementation tends to be more fragile. Thus, we chose to use the RMI implementation for the experiments of Section 6.

4.5 Client

The Client module provides an interface for the user or higher-level application to use the VMCStore system. This module also marshals data so that it will be understood by the CounterHost, and verifies validity proofs received from the CounterHost. If the system includes a StorageManager, it will also be the Client which is responsible for communicating with the StorageManager.

4.5.1 Interfaces

The Client module defines three key interfaces: *ClientManagerInterface*, *CounterMarshallerInterface*, and *CounterVerifier*.

The ClientManagerInterface provides the methods that are exposed to the user or higher-level application. These methods are intentionally very generic to allow for a variety of different applications:

- *Object put(File)*
- *Object get(File)*
- *boolean contains(File)*

The CounterMarshallerInterface defines methods that a CounterMarshaller should provide to help the Client package data properly for sending to the CounterHost, and verify validity proofs from the CounterHost:

- *CounterVerifier getCounterVerifier()*:

  Returns a *CounterVerifier*, which can be used to verify validity proofs received from the CounterHost.

- *ByteArrayableWrapperInterface<? extends ByteArrayable> generateIncNonce(ByteArrayable counterID, PublicKey pk)*

- *ByteArrayableWrapperInterface<? extends ByteArrayable> generateIncNonce(ByteArrayable counterID, long counterValue, PrivateKey sk)*:
  Returns a nonce that can be used in an increment counter request. The first of these methods is used for counter registration; the second method is used for counters that have already been registered. See Section 5.4 for information on counter registration.

- *ByteArrayableWrapperInterface<? extends ByteArrayable> generateReadNonce(ByteArrayable counterID)*:
  Returns a nonce that can be used in a read counter request.

- *ByteArrayableWrapperInterface<? extends ConfCertificate> generateConfCertificate(ByteArrayable counterID, ScheduledCertificate opCert, PrivateKey sk)*:
  Generates and signs a confirmation certificate for sending to the CounterHost.

- *boolean confirmIncrements()*:
  Informs the Client whether or not confirmation certificates are required to be sent after a successful increment. Some algorithms, such as the *Adaptive* algorithm discussed in Sections 3.5 and 5.4, do require confirmation certificates for each increment.

The CounterVerifier is responsible for verifying the validity proof sent by the CounterHost. It is algorithm and implementation-dependent, and is usually packaged with the CounterMarshaller and should typically be accessed by CounterMarshaller.*getCounterVerifier()*.

- *int verifyRead(CountStamp readCert, ByteArrayable counterID, ByteArrayable nonce, VerificationKey verifyKey, PublicKey pk)*

- *int verifyIncrement(CountStamp incCert, ByteArrayable counterID, ByteArrayable nonce, VerificationKey verifyKey, PublicKey pk)*:
  Verifies the read or increment certificate (CountStamp), according to the information provided. The VerificationKey is provided by the CounterHost for verifying the signature on the CountStamp. PK is

the public key of the client, which may be needed for verifying certain structures in the validity proof (e.g., confirmation certificates).

4.5.2 Log-based Implementation

The *logbased* package contains a number of Client class modules designed to work with the CounterGateway and CounterHost *logbased* classes, as well as the included Simulation module.

The primary class that the Simulation module interacts with is the *LogClientManager* class. The purpose of the LogClientManager is to provide an entry point for performing statistical analysis of counter operations and communication. Since storage is highly application-dependent, the LogClientManager does not communicate with an external StorageManager, but simply stores file data internally. The assumption is that the cost of communication with the CounterGateway would be in addition to the cost communication with the storage manager. The feasibility of VMCStore then depends on the ratio of these two costs, which is a function of the application.

With this goal in mind – to perform statistical analysis of counter operations – LogClientManager implements *StatRecordClientManager*, an extension of the ClientManagerInterface. The *put()* and *get()* methods of LogClientManager returns *StatRecord* objects. See section 4.6 for more information on StatRecord.

The *logbased* package also includes a *LogCounterMarshallerFactory*. The static method *getLogCounterMarshallerInstance(LogAlgorithm)* can be used to obtain an algorithm-specific instance of the LogCounterMarhaller. The *LogAlgorithm* enum describes which algorithms are available.

Finally, the *logbased* package contains a LogVerifier class, which is used to verify LogReadCertificate, SharedIncCertificate, and VerifiedIncCertificate objects. LogVerifier contains a copy of TPMCountStampVerifier (actually part of *counterhost.tpm*), which checks the validity of the TPM transport log contained within the CountStamp.

4.6 Simulation

The Simulation module is not one of the core modules of the VMCStore framework, but is included as a demonstration of how an application can be built on top of VMCStore. The purpose of the Simulation module is to perform wide-area multi-client performance analyses of the VMCStore system.

4.6.1 Operational Overview

Figure 4 provides an overview of the hierarchical structure of the Simulation module. A central simulation server, running the *SimulationCommander*, is responsible for conducting, controlling, and collecting data from the experiments. At the beginning of an experiment, the SimulationCommander contacts an arbitrary number of *SimulationNodes*, any or all of which may be running on separate machines. The SimulationCommander tells each SimulationNode how many *SimulationClients* to generate, and any other important parameters for the experiment (e.g., algorithm type, running time).

As suggested, each SimulationNode can run an arbitrary number of clients. Each SimulationClient runs in a separate thread, owns a separate instance of LogClientManager, and establishes its own connection with the counter server.

While the experiment runs, each SimulationClient tracks a number of statistics relating to its operation and communication with the counter server. When the experiment finishes, the SimulationCommander asks each SimulationNode to send a report of the simulation data. In turn, each SimulationNode asks each of its SimulationClients to send a report of its own simulation data. The SimulationNode aggregates the SimulationClient data into a single report and sends it to the SimulationCommander. Finally, the SimulationCommander aggregates the reports from each SimulationNode and returns the result to the user.

LogCounterGateway also tracks certain statistics during its operation. At the end of an experiment, the SimulationCommander collects these statistics from the LogCounterGateway and includes them in the report to the user (see Interfaces).

SimulationNodes can be left running indefinitely on host machines, so multiple experiments can be run back-to-back. *MultiSimulation* takes advantage of this construction, providing an interface for running a series of experiments, by adding an extra layer on top of SimulationCommander.

4.6.2 Interfaces

The Simulation module has a number of communication requirements above and beyond the methods provided by the CounterGateway interface. These are captured by SimulationGateway:

- *String getStats()*:

  Returns the simulation statistics as a comma-separated value (CSV) file represented by a String.

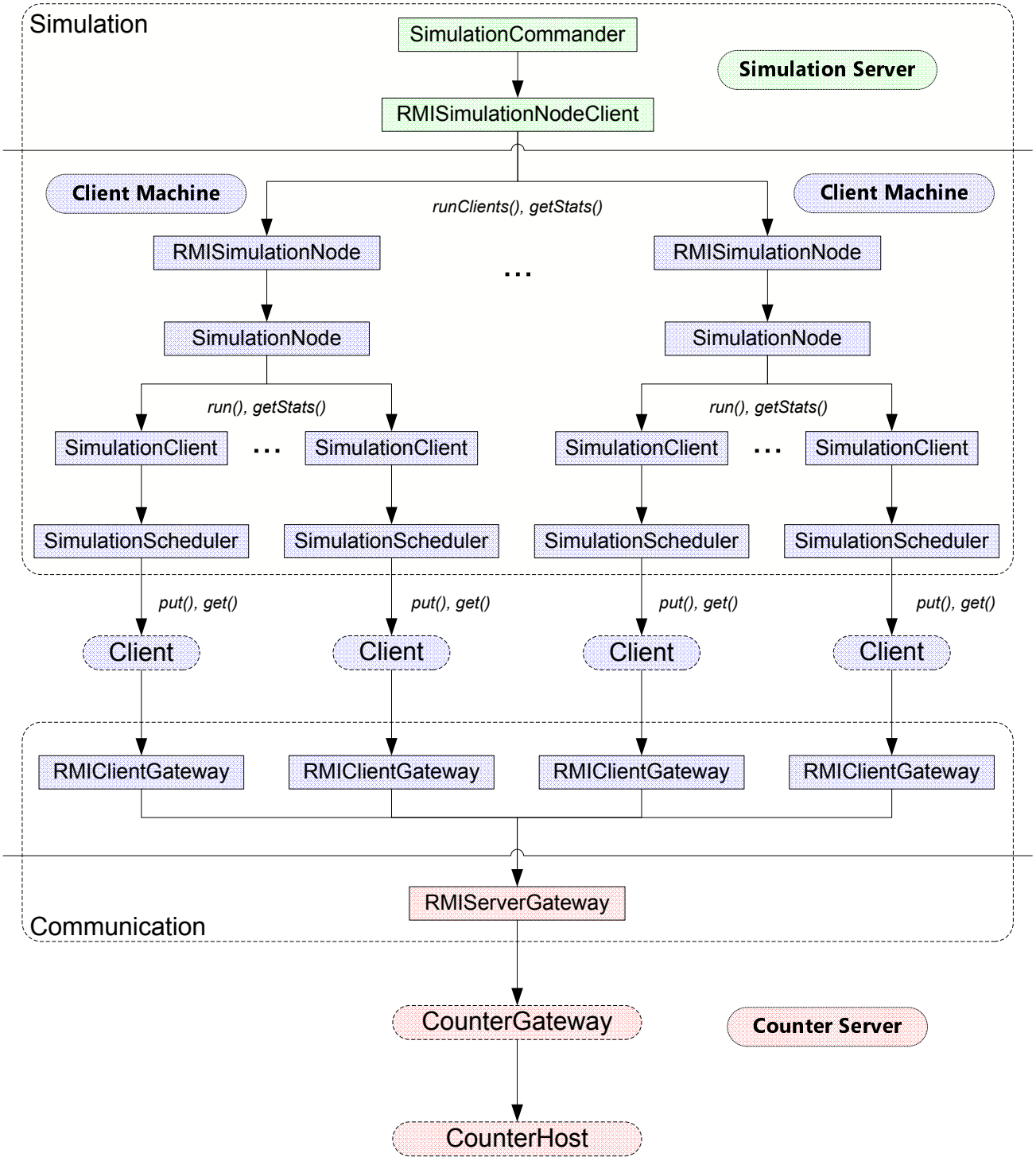

Figure 4: Detailed diagram of how the Simulation performs wide-scale performance tests of the VMCStore system. While there is only one simulation server and one counter server, there can be an arbitrary number of simulation nodes, each with an arbitrary number of clients. A socket-based implementation, included in VMCStore, can be used in place of the RMI classes.

- *void resetStatistics()*:

  Clears any outstanding statistics remaining on the CounterGateway. This should usually be called before *prepareNewSimulation()*.

- *void prepareNewSimulation(String simID)*:

  Prepares the CounterGateway for a new Simulation, which will be referred to by the given simulation ID.

A second interface is provided, SimulationCommunicationGateway, the super-interface of SimulationGateway. The relationship between these two interfaces is analogous to the relationship between CounterGateway and CommunicationGateway (see Section 4.3).

4.6.3 Data Collection Structures

The Simulation module uses a number of different structures to collect and aggregate data. In-simulation, the LogCounterGateway and SimulationClients manage simulation data using the *StatPackagedCertificate*, *StatRecord* and *StatRecorder* classes. The SimulationCommander uses the MachineStatistics and GroupStatistics classes to perform the final aggregation.

The StatRecord class simply provides a mapping of Strings to doubles, where the String is a key that represents the statistic that was recorded, and the double is the recorded value. The StatRecorder class accepts StatRecord objects, and writes out the data to disk in CSV format, grouping records with the same key. The StatRecorder writes the data to disk immediately to minimize data loss in case the client crashes.

LogCounterGateway uses StatPackagedCertificates to send simulation data (e.g., TPM operation time) to the SimulationClient. The StatPackagedCertificate class implements ByteArrayableWrapperInterface, and holds a StatRecord and a CountStamp.

When the SimulationNode asks a SimulationClient for its statistics, the StatRecorder CSV file is read verbatim from disk and sent to the SimulationNode as a String. The SimulationNode collects all of these Strings from its SimulationClients and sends them as a String array to the SimulationCommander. Before aggregating the statistics, the SimulationCommander writes these individual CSV files to disk for backup.

While the StatRecord class only accepts a single instance of a statistic – reporting a second value under the same key throws an error – the MachineStatistics aggregates data over all instances of a statistic during

the run of an experiment. For each key, the MachineStatistics object maintains a *DoubleStat* (a TPM/J object), which calculates the mean, standard deviation, etc. of the statistic.

The GroupStatistics class takes aggregation one step further, accepting MachineStatistics and GroupStatistics objects, and calculating statistics across all given data. The aggregated data is then retrieved using the *toCSVMultiSim()* method, which returns a CSV file as a *StringBuilder*. This CSV file contains not only the final aggregated statistics, but also calls *toCSVMultiSim()* recursively on the child GroupStatistics objects to generate all of the intermediate aggregation statistics.

5 Theoretical Data Structures in VMCStore

This section provides a bridge between theory and implementation, explaining which data structures correspond to which classes in VMCStore. For more details on the implementation of these classes, please see Section 4.

5.1 Counters and the Virtual Counter Manager

In VMCStore, the virtual counter manager is implemented within the *TPMCountStampHost* class. This class uses TPM/J, a Java-based API for the TPM, to communicate with the TPM, as well as generate timestamps for the client [34].

Timestamps are derived from a TPM/J data structure called a *CountStamp*. CountStamps are generic containers, capable of holding a counter ID, a nonce, and a counter value, as well as other information. A subclass of CountStamp, called *TPMCountStamp*, can also hold the transport log of the counter operation performed at the TPM. The TPMCountStampHost returns a TPMCountStamp as the result of both increment and read operations.

5.2 Validation Proofs and Confirmation Certificates

The counter manager requires an additional layer to compose the validity proofs required to convince clients that counter operations were completed correctly. This layer is implemented as the *LogCounterManager*. The LogCounterManager is responsible for generating these validity proofs, as well as for verifying client signatures on incoming counter requests.

Validity proofs are packaged within a data structure called the *LogVerificationCertificate*. The LogVerificationCertificate contains the most recent confirmation certificate for the requested counter ID, as well as a log of all increment operations that have been performed on that TPM since that confirmation certificate was generated. If no confirmation certificate has yet been received (e.g., because the virtual counter has just been created), the first increment certificate and the increment log dating back to that certificate is returned. Note that the actual format of the log may vary from algorithm-to-algorithm; a multiplexed scheme only requires a subset of the log to prove counter validity. See Section 3.5 for more information.

Confirmation certificates are implemented by the *LogConfCertificate* class. The LogConfCertificate contains an instance of *LogConfCertData*, as well as a client signature over this data. The LogConfCertData

contains the counter ID of the virtual counter which is being confirmed, as well as the time the validation was completed, the last increment time of this counter, and a schedule of legal increment times.

5.3 Operations and Operation Certificates

VMCStore provides three of the four operations presented in the Protocol Overview section (Section 3.3). Since read-without-validation is not required by any of the algorithms presented, and not useful for proving validity, it is not provided by VMCStore.

The other three operations, however, are provided. Each one returns a different data structure, as described below:

- *Increment-without-validation*: Returns a *SharedIncCertificate*, which contains the increment nonce, TPMCountStamp, and a schedule of legal future increment times. Assuming the schedule is client-independent, the same increment certificate can be sent to all clients updating in a single timeslot (e.g., using the *Shared* algorithm).
- *Increment-with-validation*: Returns a *ValidatedIncCertificate*, which contains a SharedIncCertificate and a LogVerificationCertificate (see Section 5.2).
- *Read-with-validation*: Returns a *LogReadCertificate*, which contains a *SharedReadCertificate*, a SharedIncCertificate (from the last increment of the requested virtual counter), and a LogVerificationCertificate (see Section 5.2). The SharedReadCertificate contains the read nonce, as well as the TPMCountStamp for this read operation. (The SharedReadCertificate could be returned by itself to provide a read-without-validation operation.)

While two increment operations are provided, only one increment operation is allowed for each algorithm. An algorithm can decide whether increments should be validated by setting the *validateIncrement* option of the LogCounterHost class.

5.4 Algorithms, Nonces, and Schedules

Four log-based algorithms, derived from the Improvements section (Section 3.5), are provided by VMCStore. These algorithms are described below and, except for the *Adaptive* algorithm, are tested and analyzed in Section 6:

- *Simple*: The original single-client protocol defined in section 3.4.
- *Shared*: The multi-client increment/read protocol described by the **Sharing** improvement.
- *Multiplexed*: The fixed-schedule, multi-client protocol defined by the **Time-multiplexing** improvement.
- *Adaptive*: The adaptive-schedule, multi-client protocol defined by the **Time-multiplexing** improvement.

All schemes, except the *Adaptive* scheme, use the increment-without-validation protocol. Security concerns are introduced when the schedule is not fixed. The ultimate conclusion is that the virtual counter manager may only perform increments-with-validation when using the *Adaptive* scheme. A detailed analysis of the security risks of the adaptive multiplexed protocol can be found in Appendix B of [45].

In VMCStore, all protocols use the same nonce data structure for making increment and read requests to the virtual counter manager. For reads, this structure is the *SharedReadNonce*, composed of a list of *NonceWithCounterID* structures. Each NonceWithCounterID contains the counter ID of the virtual counter to be read, as well as an anti-replay nonce.

For increments, the *SharedIncNonce* class is used. A SharedIncNonce is composed of a list of *NonceWithSignature* objects, each containing a copy of *IncrementNonceData* as well as either a public key, or a signature over the data. If this increment represents the creation of a virtual counter (i.e., the first time the virtual counter is being incremented), this is indicated by passing a public key with the IncrementNonceData. Otherwise, a signature over the IncrementNonceData is sent to authenticate this increment request to the virtual counter manager.

The IncrementNonceData contains three pieces of information: the counter ID of the virtual counter to be incremented, an anti-replay nonce, and the time of the previous increment of the virtual counter. The previous increment time is included to prevent collisions where two devices of a single client try to increment a counter at the same time, and one may overwrite the changes of the other. If the virtual counter manager detects an erroneous previous increment time, it will elect to decline the increment and warn the client that there is a more recent update.

Note that although the *Simple* algorithm uses shared nonce structures, its protocol only allows one client to perform a read or increment at a time. For this algorithm, the shared nonces only contain the individual

nonce for the virtual counter involved in the operation.

Finally, as mentioned above, the SharedIncCertificate always contains a schedule of legal increment times. In VMCStore, these schedules are generated by the virtual counter manager and are generated for every algorithm. All schedule classes implement the *Schedule* interface, which provides basic functionality for iterating through and seeking legal increment times.

The *Simple, Shared,* and *Multiplexed* algorithms all use an implementation of Schedule called *MultiplexedSchedule*. A MultiplexedSchedule allows an increment on every n^{th} slot, where n is the multiplex factor. For the *Simple* and *Shared* algorithms, n is always 1; for the *Multiplexed* scheme, n may take any number, but for the experiments described in Section 6, n is always 5 or 10.

By default, the *Adaptive* algorithm uses a *LinearBackoffSchedule*, which works as its name implies, but the *Adaptive* algorithm can be customized to use any schedule which implements the Schedule interface.

6 Results

In this section we will analyze VMCStore and the practicality of its application to real-world scenarios. This section will first discuss the experiment and its parameters, and then continue with the expected results and the experimental results that were obtained.

6.1 The Experiment

The goal of the following experiment was to develop an intuition for the practical limits of a trusted storage application built around a TPM.

The experiment involves two entities: the server, and the clients. The server is a TPM-carrying machine, which acts as the virtual counter manager for the trusted storage system. The clients each own a virtual counter, and submits requests to the virtual counter manager for these virtual counters to be read or incremented, at probabilistically determined intervals.

To determine the limitations of VMCStore, the system measures operational parameters during the run of the experiment, such as operation time, bandwidth, etc. The hypothesis is that by increasing the number of clients and analyzing the change in measurements, an approximate upper bound for the number of clients VMCStore can maintain may be found. Furthermore, performing the experiment under different algorithms (see Sections 3.5 and 5.4), provides an intuition for potential tradeoffs that can be negotiated to make the system more practical for global-scale usage.

6.2 Experimental Parameters

This section describes the framework used to perform the experiment described above. This framework was used to generate all of the results discussed in the following section.

The server (virtual counter manager) was run on the in the Stata Center, on the MIT network (10MB/s), using a Gateway M-465 laptop. This laptop was running Linux, and contained a 1.83 GHz Core Duo CPU, 2GB RAM, and an ST Micro TPM 1.2 chip.

The clients were run on the PlanetLab network [42]. Using the model presented in Figure 4, 150 machines from this network were selected to act as SimulationNodes for the experiments. The SimulationClients were evenly distributed across these SimulationNodes. All communication was managed using VMCStore's RMI communication classes (see Section 4.4).

All digital signatures were produced using 2048-bit RSA keys, and hashing was done using SHA-1. We used TPM/J [34] as a Java interface to the TPM. Measurements showed that with the ST Micro TPM chip, an *IncSign* operation required about 1.3 seconds. A *ReadSign* operation was found to take roughly the same amount of time. We further found that the ST Micro chip throttles *IncSign* operations such that we could only execute one at most every 2.15 seconds.

The SimulationClients were run using the *PoissonScheduler* class. At runtime, this class creates a schedule of read and increment operations for the duration of the experiment. Operations times were determined over a Poisson distribution with a frequency of one operation every 30 seconds. Read and increment operations were selected with equal probability so that, on average, we would expect one increment and one read per client per minute. The schedule of client requests is generated at the beginning of the experiment. If a client misses a request deadline, because a previous request took too long, the next request is made as soon as the previous request returns.

The counter server was run as a multi-threaded application. Client requests were handled as independent threads, which were managed through the use of synchronized blocks, lists and queues. Server operations were handled on a separate thread from network communication, so that the TPM could perform operations in parallel with client requests and the sending of read and increment certificate responses. Each client also had a personal TCP socket connection with the server, so that requests and responses could be sent in parallel, maximizing the usage of the server network.

All experiments were run for a duration of 35 minutes. The first five minutes was used as a warm-up time to allow virtual counters to be initialized. The statistics represented below were derived from the final 30 minutes of the experiment. Experiments were conducted using the *Simple* algorithm, *Shared* algorithm, and *Multiplexed* algorithm (for multiplex values of 5 and 10). The *Adaptive* algorithm was not tested, as it is dependent on client request patterns, which are application-dependent.

6.3 Experimental Results

We begin with an analysis of the original algorithm presented in Section 3.4, and then continue by analyzing the extensions listed in Sections 3.5 and 5.4.

6.3.1 Operational Efficiency

To determine when the system saturates, we will use the idea of *operational efficiency*. We define operational efficiency to be ***the ratio of the number of operations actually completed to the expected number of operation requests, in an experiment***:

$$\gamma = \frac{operations_{completed}}{requests_{expected}}$$

If each client makes one operation request on average every 30 seconds, and all experiments run for 30 minutes, we should expect each client to request approximately 60 operations during the experiment. If an experiment saturates, the client should fall behind its request schedule and fail to complete all 60 operations. Therefore, according to this metric, $\gamma \approx 1$ should indicate that the system is practical and unsaturated (i.e., the server is able to complete all requested operation), while $\gamma < 1$ should indicate that the system is impractical and overloaded (i.e., unable to complete all of the client requests).

From the client's perspective, an overloaded server means that operations are consistently taking longer that 30 seconds to complete. Since the client makes requests on average every 30 seconds, and the request schedule is generated ahead of time, this means that at some point requests will be *late*. As mentioned in the Experimental Parameters section, a client will only send in a late request when it receives the result from the previous request. However, since the server is overloaded, future requests will also take more than 30 seconds to complete. This leads to a snowballing effect so that the lateness of later requests will grow without bound.

6.3.2 *Simple*

The *Simple* algorithm allows only a single user to increment or read her virtual counter at any given timeslot. Intuitively, if the time that a client has to wait to perform an operation is greater than the average operation frequency defined by its Poisson process, the client should see a backlog of operations, and the system should saturate. By this intuition, we should expect the TPM to be the bottleneck for the *Simple* algorithm since, as mentioned above, the ST Micro TPM requires approximately 1.3 seconds to perform any given increment or read operation. Since clients perform, on average, one operation every 30 seconds, we expect the upper bound of the *Simple* algorithm system to be approximately $30s/1.3s \approx 23$ clients.

Figure 5 shows the operational efficiency of the VMCStore system using the *Simple* algorithm, for a

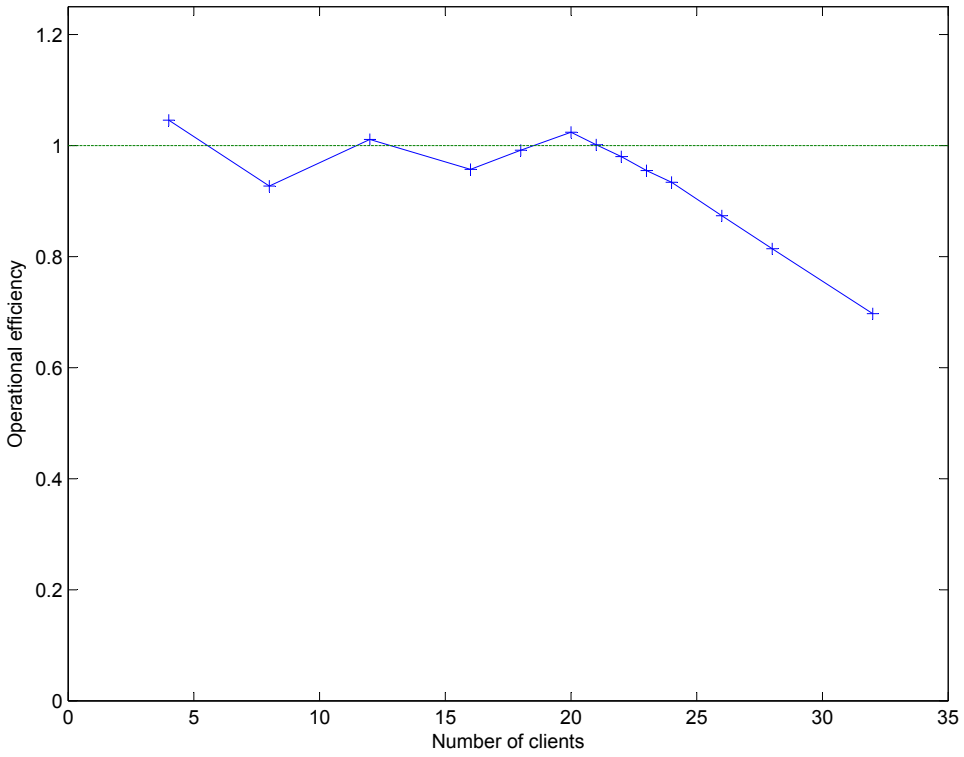

Figure 5: Operational efficiency of the *Simple* algorithm, for various clients. The decrease in efficiency suggests that the system's saturation point has been reached.

range of 4 to 30 clients. We can see from the graph that the efficiency of the *Simple* algorithm hovers around 1 until the number of clients reaches the low-to-mid 20's. The deviation in the smaller number of clients is probably within expectation; later results with many more clients tend to be much tighter. We also notice that the efficiency of the algorithm starts to drop below 1 around 22-24 clients, suggesting the system is becoming overloaded. This agrees with our expected saturation point of about 23 clients.

Looking at the data from another perspective, Figure 6 shows the total number of operations performed for each of these experiments. Since we expect the TPM to be the bottleneck, there should be an absolute limit to the number of operations it can perform, about $1800s/1.3s \approx 1384$ operations. While the number of operations grows linearly in the graph until we reach 20 to 22 clients, the TPM definitely appears to saturate at this point. As the number of clients continues to increase, the TPM approaches the maximum number of operations it can possibly support in 30 minutes; the maximum number measured was 1368 operations.

Figure 7 and Figure 8 show the average size of the increment and read certificates, per client, for each of the experiments. The *Simple* algorithm uses the increment without validation protocol to perform its increments. Since the size of the certificate produced by this protocol grows linearly with the number of

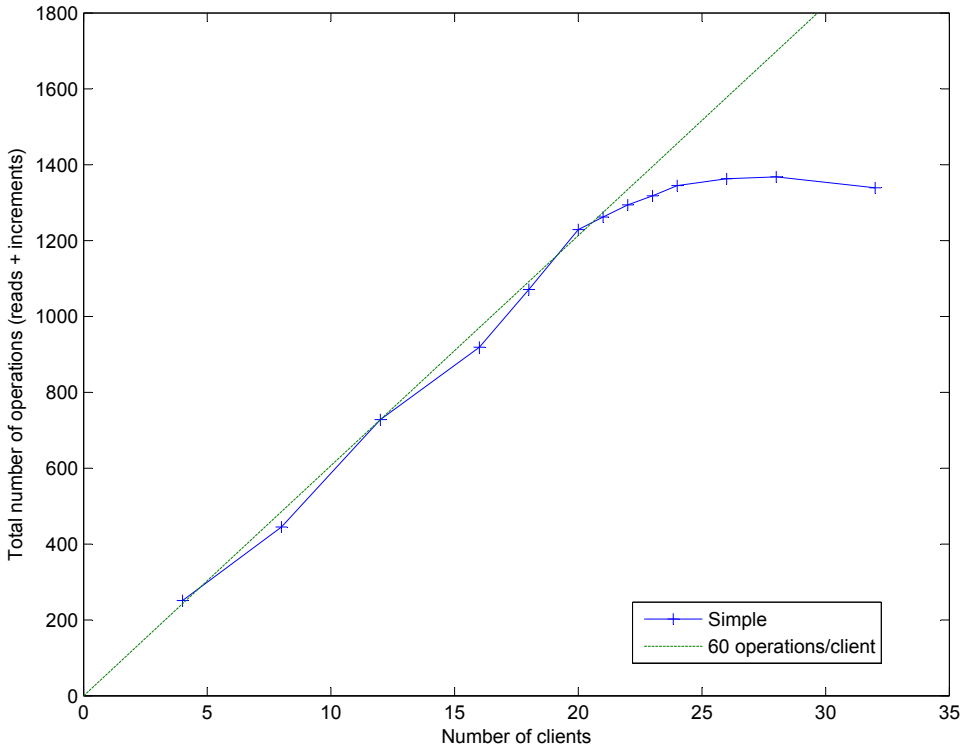

Figure 6: Total operations performed by the *Simple* algorithm, for various clients. As the number of clients increases, we see the TPM reach the limit for the number of operations it can perform in the timeframe of the experiment.

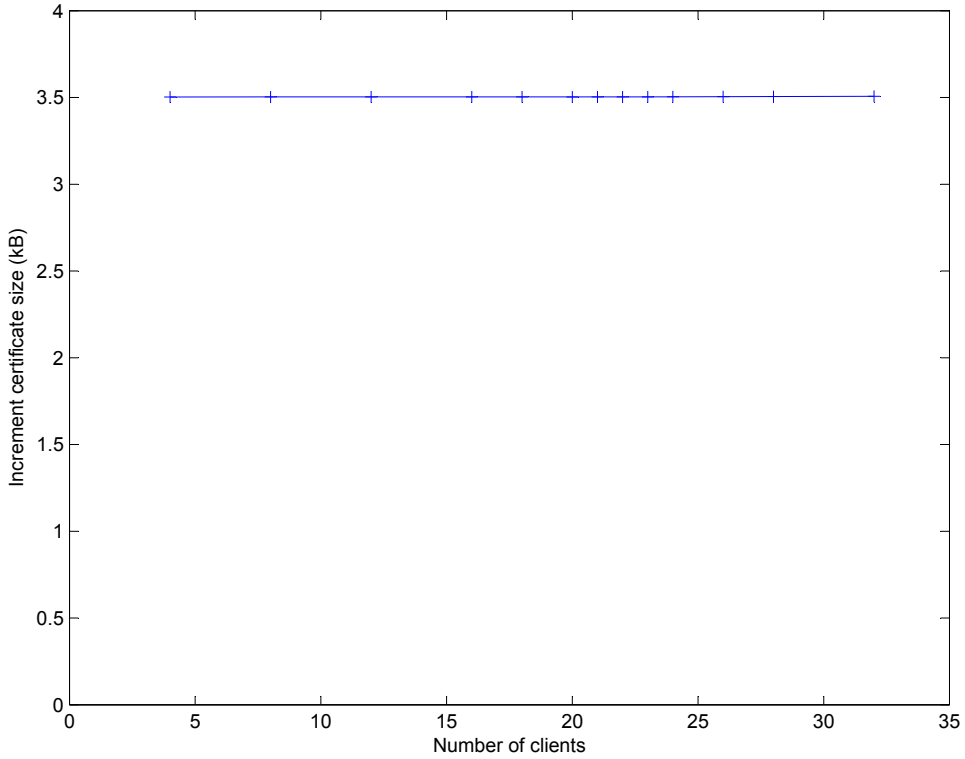

Figure 7: Average certificate size per increment of the *Simple* algorithm, for various clients.

concurrently incrementing clients, and we only allow one client to increment its counter during any given timeslot, the certificate should have approximately constant size, as confirmed in Figure 7.

On the other hand, the *Simple* algorithm uses the read with validation protocol for all read operations. We recall that the read certificate will contain a validity proof, as well as the read nonce. The validity proof contains a log of increment certificates since the last confirmation certificate was sent to the server. Since the *Simple* algorithm sends a confirmation certificate after each read, we expect the validity proof to grow with the number of expected increments between the time a client performs two reads. Since only one client can read or increment at a time, we should expect the validity proof to grow linearly with the number of clients. The read nonce has a constant size, since only one client can increment at a time, so the entire read certificate size should be an affine function that grows with the number of clients.

Indeed, the graph in Figure 8 confirms this relationship. There are some anomalies as the number of clients reaches saturation. These anomalies may be explained by the way the *Simple* algorithm queues increments and reads; while increment requests are strictly processed in FIFO order, reads are performed on-demand in a pseudo-random order (e.g., at the whim of the Java thread manager). Unlike the *Shared* and *Multiplexed* algorithms, it is possible for multiple consecutive reads to be processed, even if there is an

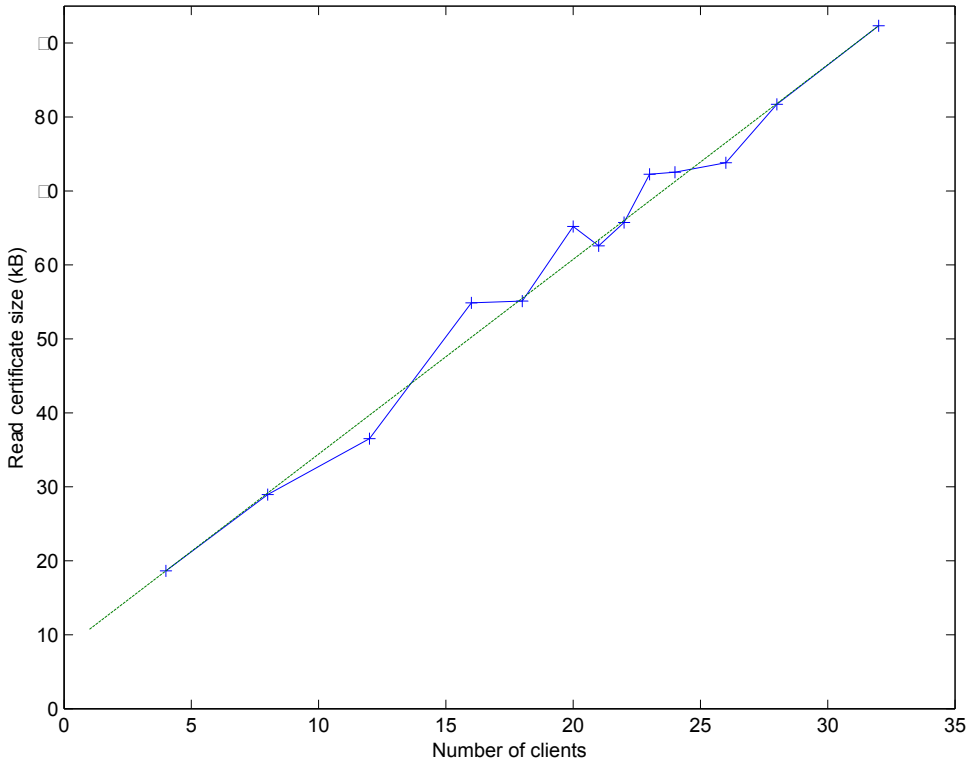

Figure 8: Average certificate size per read of the *Simple* algorithm, for various clients.

increment request in the queue.

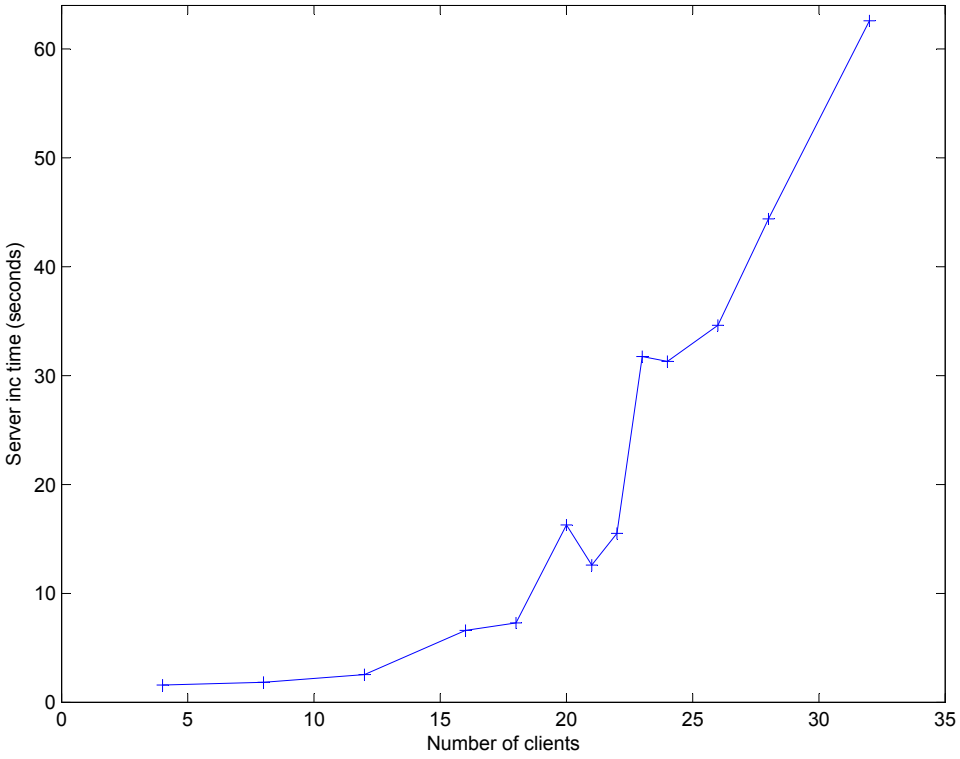

Figure 9: Average time spent by a client at the server during an increment of the *Simple* algorithm, for various clients.

Finally, for completeness, we present Figures 9 and 10, showing the average time that a client thread spends on the server during an increment of read operation. That is, these graphs represent the time that a client thread waits at the server for its operation to proceed, plus the TPM operation time. Network transit time is not included in these graphs. (The network bandwidth of the clients was not standardized, and proved a difficult parameter to use as a performance metric.)

To understand these graphs, consider three regions. If the *Simple* algorithm is well under-saturated (i.e., the server can easily handle all the requests), then we should expect few conflicting requests on the server, and a client should be able to perform its operation with minimal waiting time (i.e., close to the 1.3s required for the TPM time). This is the 4 to 12 client region, where both read and increment operations take under 2 seconds.

As the number of clients approaches saturation, we should expect many more conflicting client requests. Clients have a higher probability of having to wait for other clients to finish their operations, and the server time increases. This second region would be represented around the range of 16 to 20 clients.

Finally, as the server hits saturation, we expect the TPM to be processing operations all the time. While each client still schedules to make one request every 30s on average, the increase in clients means that over a 30s period, the total number of request would tally to more than 30s of work. This implies that a wait queue will build up on the server. As soon as a client finishes its operation, if it has missed the deadline for the next operation, it will immediately send the request to the server, adding itself to the wait queue.

In this third region, then, the size of the wait queue grows linearly with the number of clients. Since only one client is processed at a time, this leads to a linear wait time on the server, and the (approximately) linear behavior we see in the region with 22+ clients. (We are unable to explain the anomaly in the read time for 32 clients, at this time.)

It may seem surprising at first that the average server increment time is significantly higher than the average server read time. Once again, this is probably a result of increments and reads being handled asymmetrically, and how it is possible for multiple reads to be processed between two consecutive increments. Considering the increment time alone, however, if all clients were performing increments, with 30 clients we would expect to have to wait $29 * 1.3s = 37.7s$ on the server, before we can process our increment. If

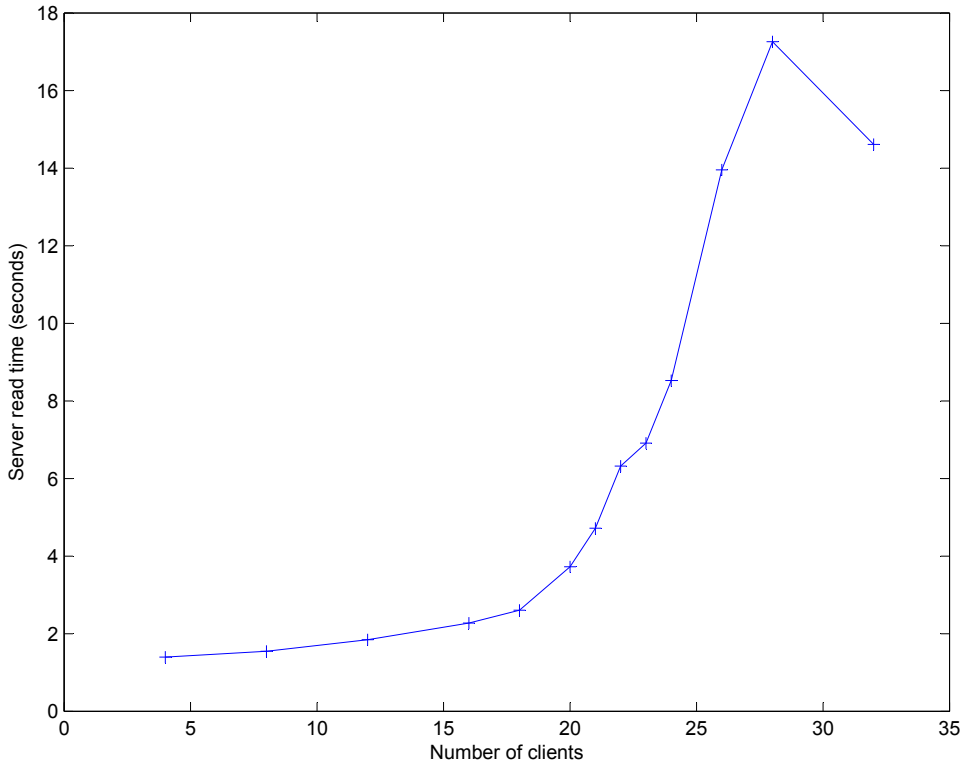

Figure 10: Average time spent by a client at the server during a read of the *Simple* algorithm, for various clients.

reads take higher priority than writes, it is possible that this wait time will increase even further. While this is difficult to quantify theoretically, since the Java thread manager is non-deterministic, given this explanation, the 62.6s wait time for increments in the 30-client experiment may make more sense.

To summarize our analysis of the *Simple* algorithm, we remind the reader of the results of Figures 5 and 6, which suggests that the TPM reaches a limit for the number of operations it can perform. Figures 9 and 10 provided further evidence that the *Simple* algorithm reaches a point where it cannot process all of the client requests in a timely matter, resulting in the build-up of wait queues at the server. Both of these results suggest that the *Simple* algorithm can only manage about 20 to 22 clients, close to our original estimate of 23 clients.

The next section will demonstrate how TPM sharing can be used to move the system bottleneck from the TPM to the network, significantly increasing the number of clients we can manage.

6.3.3 *Shared*

As shown in the previous section, the TPM serves as the bottleneck for the *Simple* algorithm limiting the number of clients it can manage to about 20. We will show that by allowing multiple clients to be processed for each TPM increment or read, the *Shared* algorithm increases the number of clients that can be managed, and moves the bottleneck to the network.

Figure 11 shows the operational efficiency of the *Shared* algorithm for various numbers of clients. As can be seen from this graph, the *Shared* algorithm maintains an efficiency of near 1 until the number of clients reaches between 300 and 400, at which point the system saturates, and the efficiency sharply decreases.

Figure 12 provides another view of this data, showing the total number of operations performed by the *Shared* algorithm for each experiment. While this graph is a little misleading, because the data points are spread far apart, we can make three observations:

a Until the system saturates, the number of operations performed grows linearly with the number of clients,

b The system saturates somewhere between 300 and 500 clients, and

c Adding more clients to a saturated system actually decreases the number of operations the system can perform.

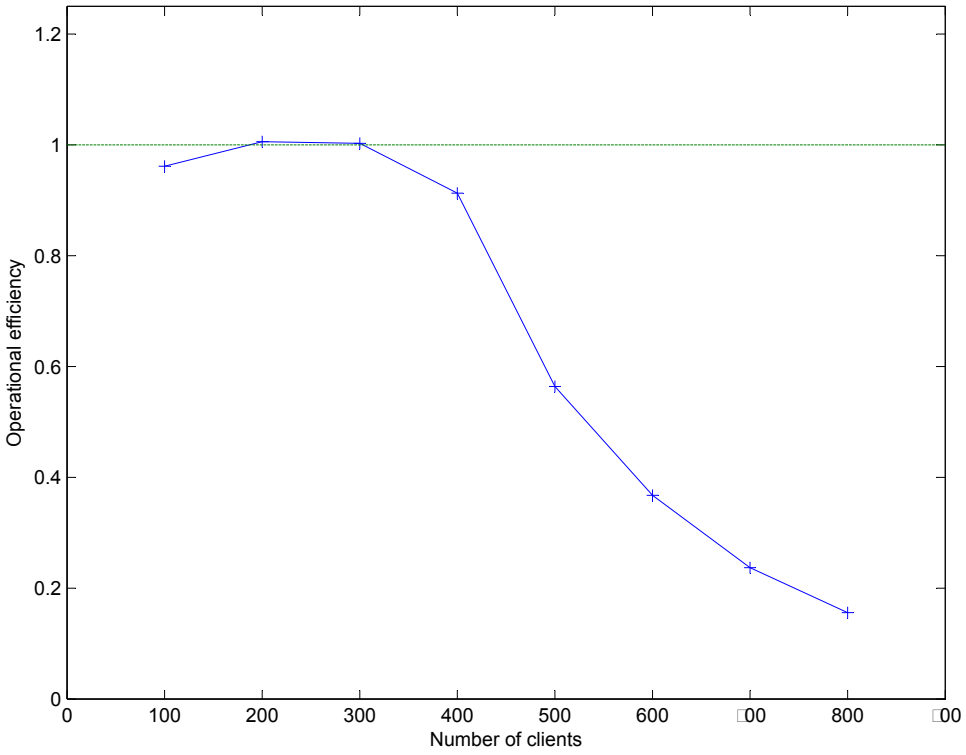

Figure 11: Operational efficiency of the *Shared* algorithm, for various clients. Notice that the *Shared* algorithm can support many more clients than the *Simple* algorithm before the system becomes saturated and the efficiency decreases.

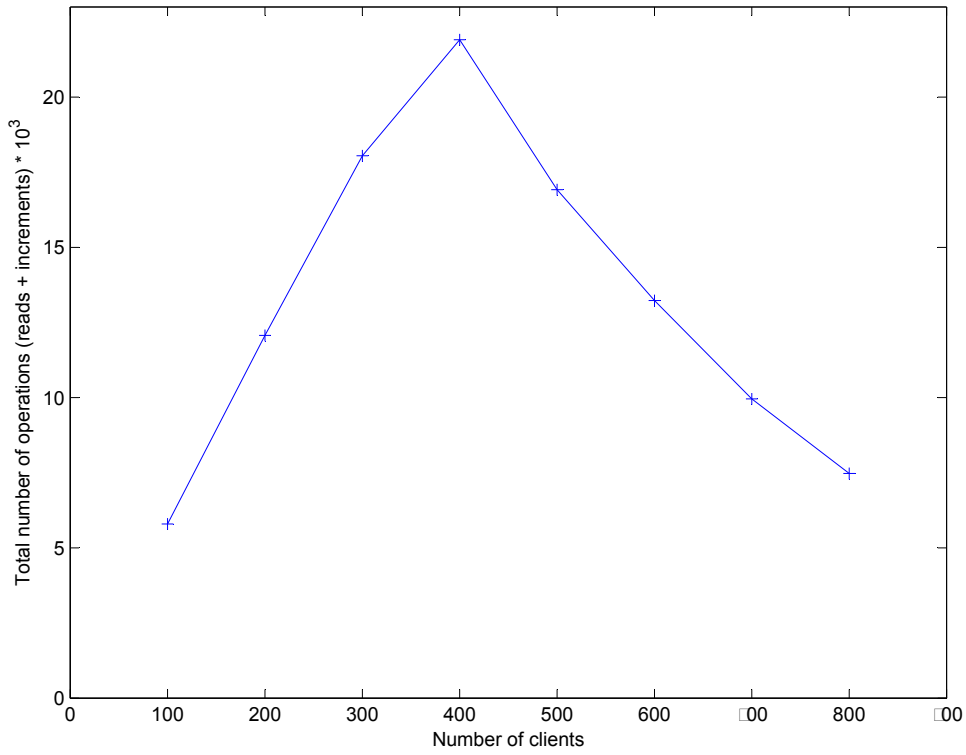

Figure 12: Total operations performed by the *Shared* algorithm, for various clients. Once again, we see a peek in the number of operations the system can perform somewhere between 300 to 500 clients.

This third point will be expanded upon in the analysis of the following graphs.

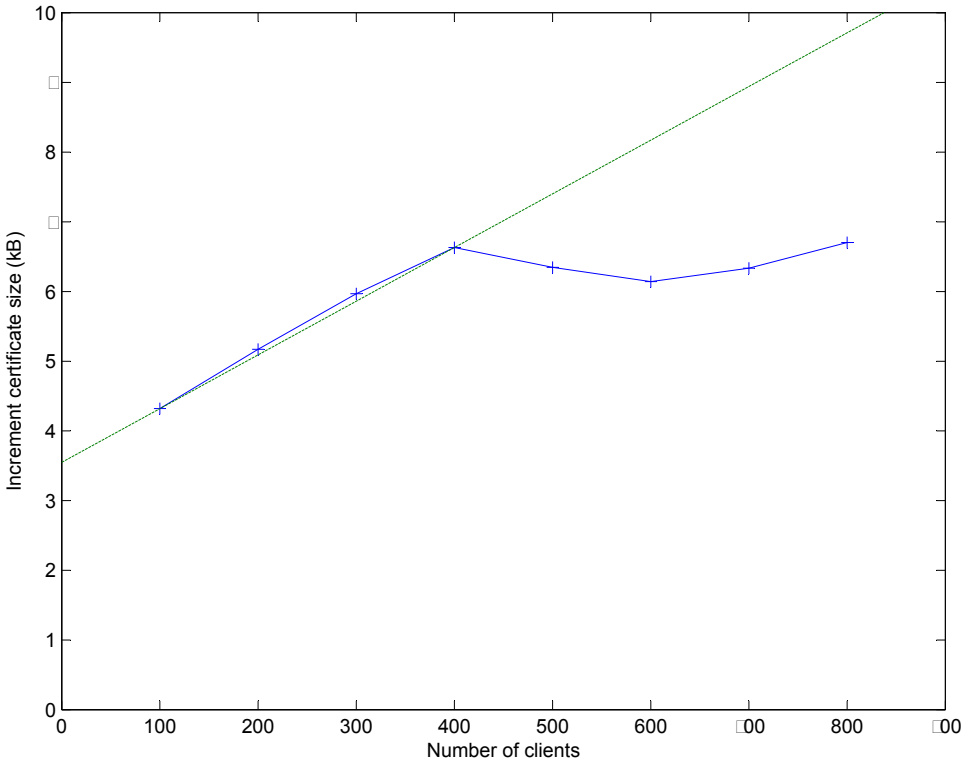

Figure 13: Average certificate size per client, per increment of the *Shared* algorithm, for various clients.

Figures 13 and 14 show the average increment and read certificate sizes per client, per operation of the *Shared* algorithm. Starting with the increment certificates, we first note that the *Shared* algorithm uses the increment without validation protocol. Since the VMCStore shared nonce grows linearly with respect to the number of concurrently incrementing clients, we should expect the graph of the size of the increment certificate to be affine. Indeed, until the system hits saturation, we can see from Figure 13 that the graph does grow as expected.

Once the system hits saturation, requests take more than 30s to complete. Recall that clients will only send a request with a missed deadline once the result of the previous request is returned from the server. The frequency at which a client can make requests is then bounded by the time it takes for operations to complete in an overloaded system. This implies that request frequency of saturated clients decreases, and that a saturated client's operations are spread over a longer period.

The expected number of concurrently operating clients is given by $\frac{number\ of\ clients}{operation\ period}$. When not in saturation, the expected operation period is 30s, and the number of concurrently operating clients grows linearly with the total number of clients. In saturation, the operation period is no longer a constant, but dependent

on the time it takes for operations to return, which is a function of the number of clients, and the network bandwidth, among other things. While it is difficult to predict the expected operation period when the system is overloaded, it will be greater than 30s, so the number of concurrently operating clients grows slower than in the unsaturated region as seen in Figure 13.

In Figure 14, we once again see a linear relation between the number of clients and the size of the read certificate. While this is a read with validation certificate, until we hit saturation we expect a constant number of increments between two consecutive reads of the client. The linear behavior is a result of the shared read nonce, which grows linearly with the number of concurrently reading clients. Comparing this graph with Figure 8, we notice that although both these graphs grow linearly with the number of clients, the read certificate size grows much more slowly int the *Shared* algorithm than in the *Simple* algorithm. This is because the nonces included in the shared nonce (which increase in the *Shared* algorithm) are much smaller than the increment certificates added to the validity proofs (which increase in the *Simple* algorithm).

While the validity proof size is constant when the system is not saturated, once the *Shared* algorithm hits saturation, a client's read frequency decreases, and the period between consecutive reads of that client increases. Now, the validity proof size is not constant, but increases as a function of the time it takes for an

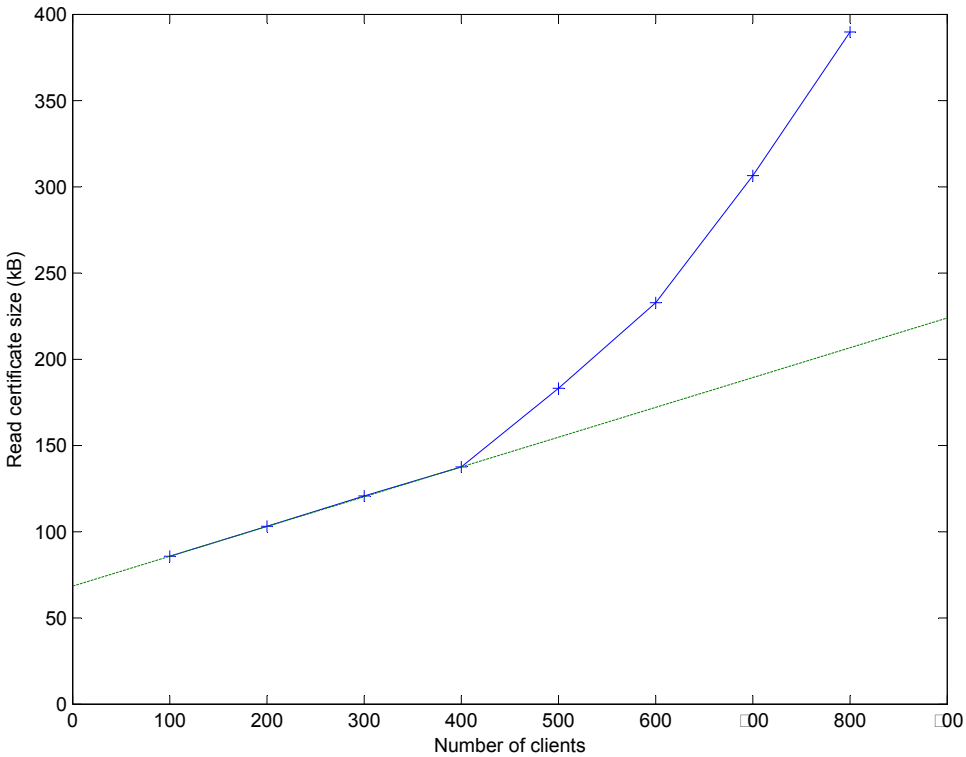

Figure 14: Average certificate size per client, per read of the *Shared* algorithm, for various clients.

operation to return. Even though the shared nonce size will decrease because fewer clients read concurrently (see above), this is overshadowed by the increase in the proof size, leading to the exploding read certificate sizes seen in the saturated regions of Figure 14.

Finally, note that if the network is saturated, then increasing the size of the read certificates should decrease the total number of read operations we can perform. Looking at Figure 12 once again, we notice that the number of operations does decrease in the saturated region, providing us with evidence that the network is the bottleneck in a *Shared* system.

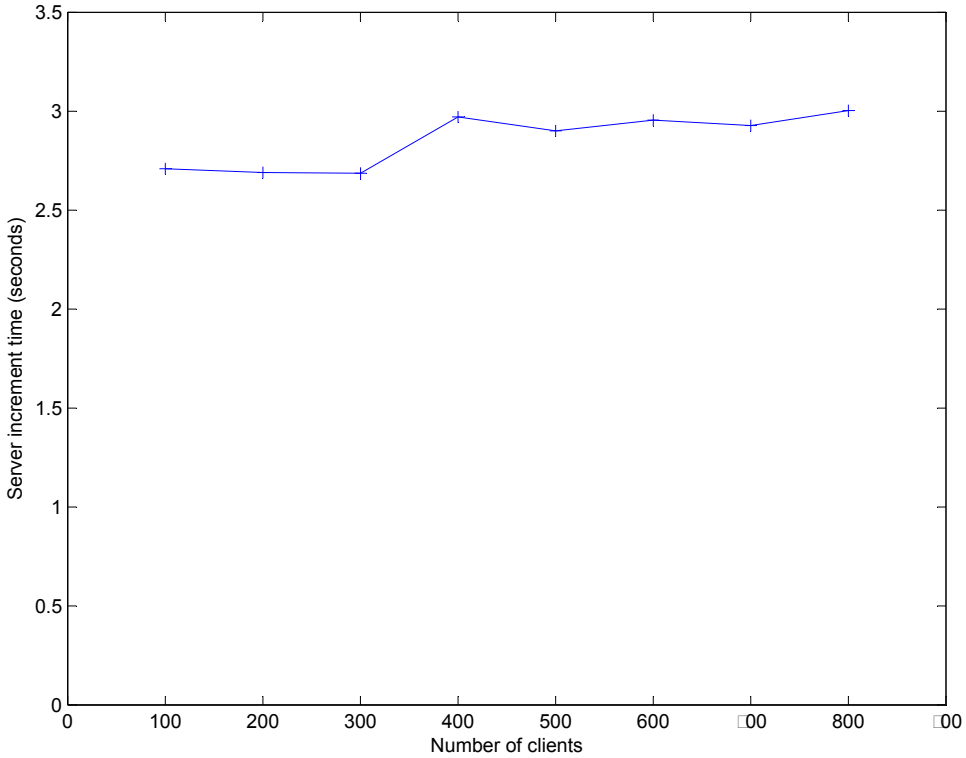

Figure 15: Average time spent by a client at the server during an increment of the *Shared* algorithm, for various clients.

Figures 15 and 16 present the average time that a client thread is present on the server during an increment or read operation. Once again, we remind the reader that this only includes waiting and operation time on the server, not network transfer time. Since, in the *Shared* algorithm, all requested reads or increments are processed concurrently, we should expect the average server time to be nearly constant, even in saturation. This time will include the operation time (1.3s for both increments and reads), as well as some wait time.

Both Figure 15 and 16 match these expected results. The average read and increment time on the server for all experiments – even those in saturation – was in the range 2.68s to 3.03s, reflecting some wait time

and processing time on top of the TPM operation time.

Comparing these graphs with the *Simple* results (Figures 9 and 10), we see a striking difference. In the *Simple* algorithm, we know the TPM was the bottleneck, and that the server operation time was proportional to the number of clients. However, for the *Shared* algorithm, it is clear from these graphs that the TPM is no longer the bottleneck. Indeed, it is unlikely the CPU is the bottleneck either, because we would expect the increased processing time to be reflected in Figures 15 and 16.

Our conclusion is that the network becomes the bottleneck for the *Shared* algorithm. Saturation implies that a client is no longer able to perform operations at its expected rate of once every 30s. However, Figures 15 and 16 show that the client spends only a few seconds waiting and operating on the server. This implies that the bottleneck is either the client itself, or somewhere between the server and the client. It is improbable that the client is the bottleneck, since the major processing operation on the client is probably proof verification, and the proof size remains constant until saturation is reached.

However, as shown in Figures 13 and 14, certificate size grow linearly with the number of clients. Also, we see from Figure 12 that the number of operations performed grows linearly with the number of clients.

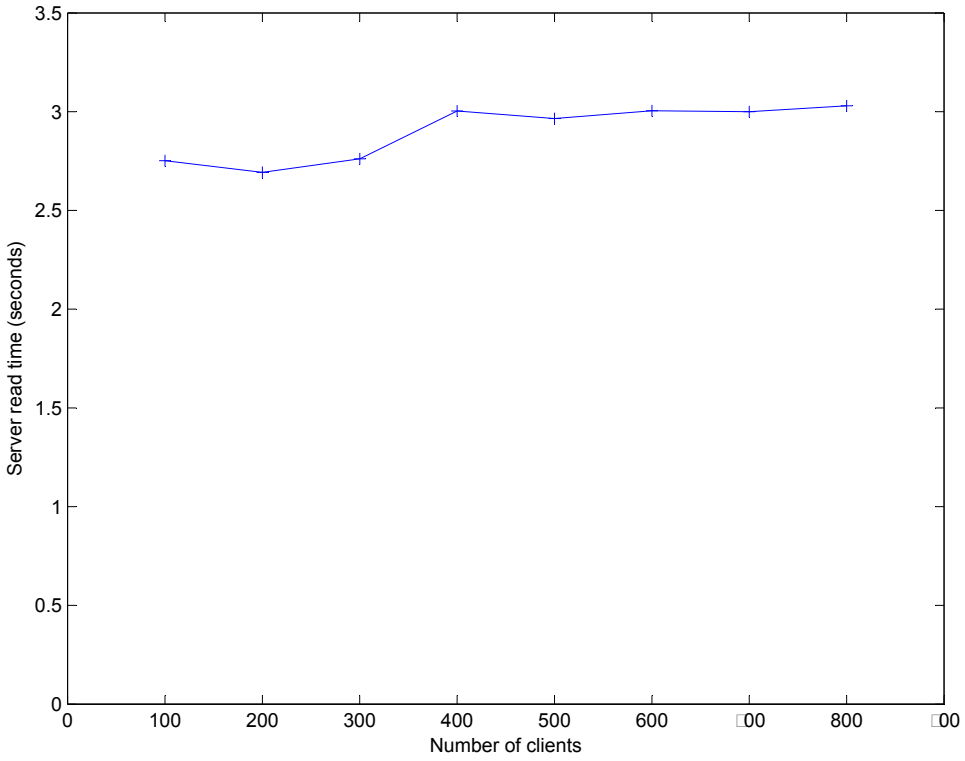

Figure 16: Average time spent by a client at the server during a read of the *Shared* algorithm, for various clients.

This implies that the network bandwidth usage grows quadratically with the number of clients. Experiments showed that the server would typically send around 3.0GB to 3.5GB data over 30-minute experiments near saturation. It seems likely that the network bandwidth was reaching saturation, and serves as the bottleneck for the *Shared* algorithm.

In summary, we have shown that by processing client reads and increments concurrently, the *Shared* algorithm removes the TPM bottleneck and improves the number of clients it can manage to around 400. We have also shown that it is likely that the network bandwidth is now the bottleneck of the system. Under this assumption, we will now demonstrate how we can use multiplexing to reduce the network load, increasing the number of clients we can handle at the expense of longer server wait times.

6.3.4 *Multiplexed*

The previous section demonstrated how the *Shared* algorithm uses concurrent processing of clients to shift the bottleneck of the system from the TPM to the network bandwidth, increasing the number of clients that can be managed. As discussed in Section 3.5, client increment requests can be multiplexed to reduce the size of the validity proof, at the expense of increased server wait times. This section will demonstrate the results of these modifications, applied by the *Multiplexed* algorithm.

Two versions of the *Multiplexed* algorithm were run for the experiments in this section: one with a multiplex factor of 5 (*Multiplexed-5*), and one with a multiplex factor of 10 (*Multiplexed-10*). In all of the following graphs, the results for these two experiments are graphed against the result of the *Shared* algorithm, which is equivalent to the *Multiplexed* algorithm with a factor of 1.

Intuitively, an increased multiplex factor will result in smaller validity proofs, which should allow us to transmit proofs to more clients. However, as the nonce size increases linearly with the number of clients, and since the shared nonce is sent to each client, the bandwidth from nonces will increase quadratically as the number of clients increase. So while the decreased proof size will allow us to handle more clients, the increased nonce size, for both reads and increments, will limit this improvement. Furthermore, if the multiplex factor causes the operation time to be longer than the operation frequency, this will cause the system to saturate.

This suggests that there is an optimal multiplex factor for a given set of client and bandwidth parameters. We do not present a method for computing that here, but leave it as an open research topic. Indeed, even for

the experiments below, we do not know the optimal multiplex factor.

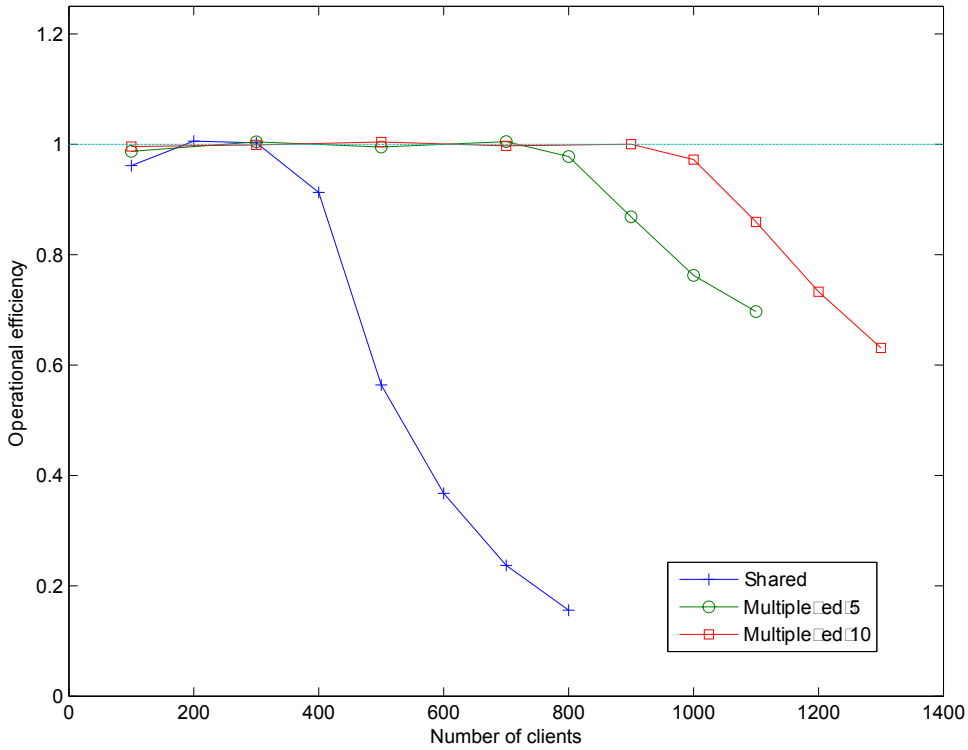

Figure 17: Comparison of the operational efficiency of the *Multiplexed* and *Shared* algorithms, for various clients.

Figures 17 and 18 compare the operational efficiency and total operations performed by the *Shared* and *Multiplexed* algorithms. As seen in Figure 17, the efficiency of the *Multiplexed-5* algorithm breaks around 700 to 800 clients, while the *Multiplexed-10* algorithm breaks around 900 to 1000 clients. Looking at the total operation results in Figure 18, we see roughly the same behavior. As expected, the *Multiplexed* algorithms can handle more clients than the *Shared* algorithm, but the relation is not linear with respect to the multiplex factor.

Figures 19 and 20 display the average increment and read certificate sizes for the *Multiplexed* algorithms. We note first that all increments are increments without validation, and all reads are reads with validation. Looking first at the increment certificate sizes in Figure 19, we notice that, in unsaturated regions, there is a tight linear relationship between the number of clients and the increment certificate size. We remind the reader that this is because the increment certificate contains a shared nonce, the size of which is directly proportional to the number of concurrently incrementing clients, which itself is proportional to the total number of clients. Regarding the behavior in the saturated region, we remind the reader that, while the

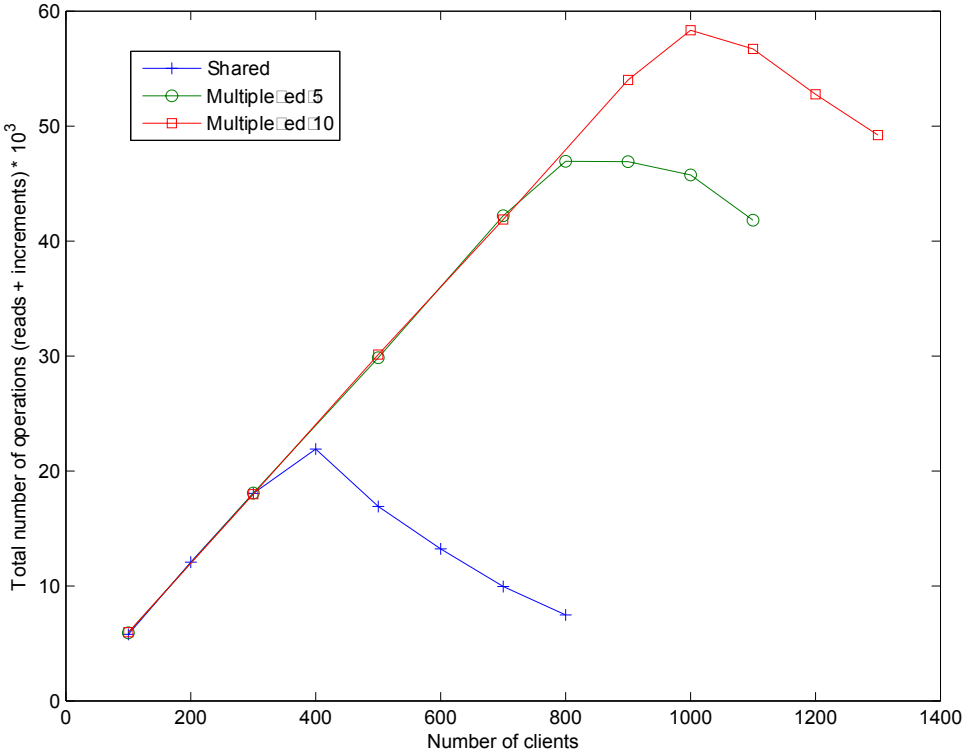

Figure 18: Comparison of the total operations performed by the *Multiplexed* and *Shared* algorithms, for various clients.

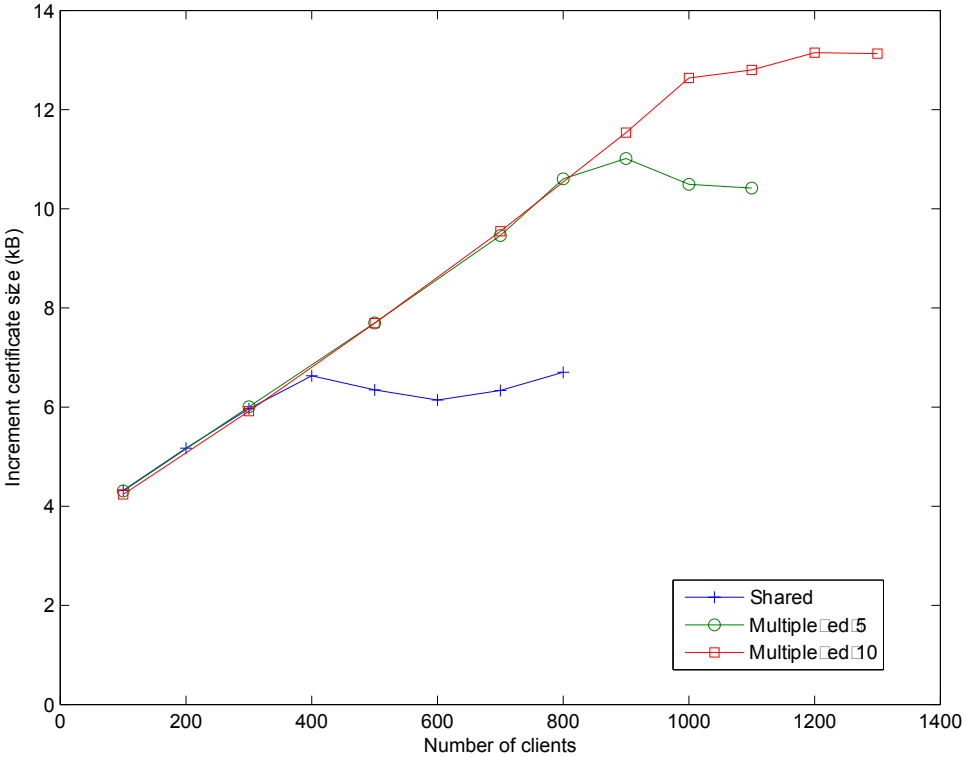

Figure 19: Comparison of the average certificate size per client, per increment of the *Multiplexed* and *Shared* algorithms, for various clients.

number of clients increases, the client increment frequency decreases. See the discussion of Figure 13 for more details.

Regarding the read certificate sizes in Figure 20, we first notice that in the unsaturated region for all three curves, read certificate sizes for all three algorithms grow linearly, and the *Multiplexed* algorithms grow significantly more slowly than the *Shared* algorithm. The read certificate size is not inversely proportional to the multiplex factor, but this is expected, since the read certificate contains a shared nonce, which grows linearly with the number of clients and is roughly the same for all three algorithms.

Indeed, this growing shared nonce also means that the benefit factor of the multiplexing decreases as the number of clients increases: with 100 clients, the *Shared* read certificate is 5.65x larger than the *Multiplexed-10* certificate; at 400 clients, this factor has been reduced to 3.66x. This is one of the reasons why the number of clients the *Multiplexed* algorithm can handle does not scale linearly with the multiplex factor.

Finally, we present Figures 21 and 22, representing the average client increment and read times, measured at the server. We remind the reader that this includes wait time at the server and TPM operation time, but no network transfer time. We expect the server increment time to scale approximately linearly with the

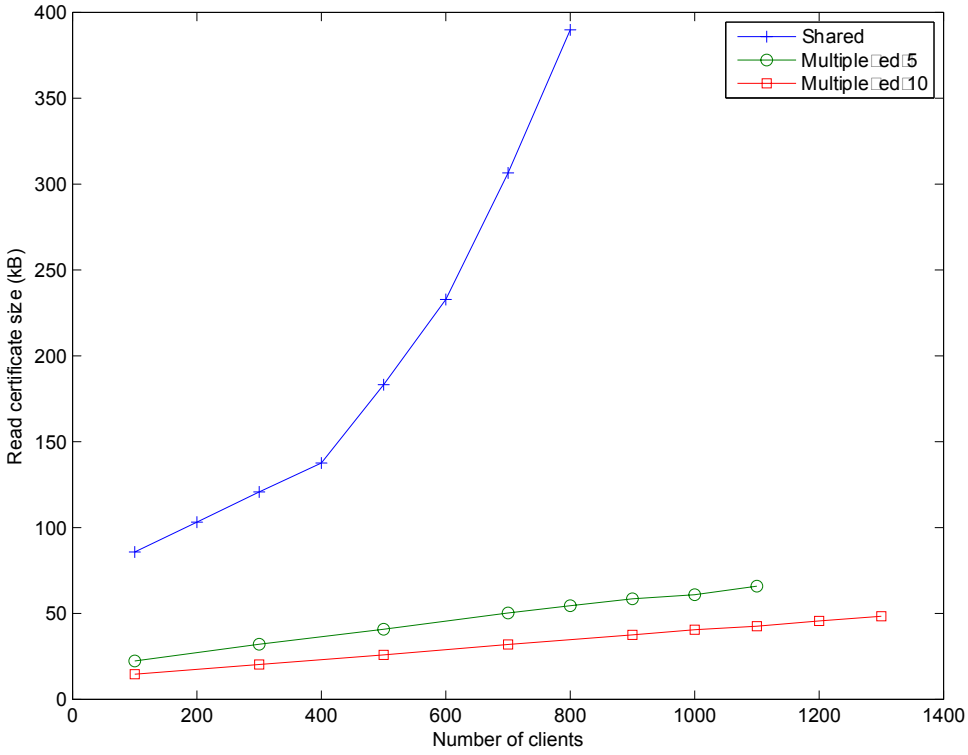

Figure 20: Comparison of the average certificate size per client, per read of the *Multiplexed* and *Shared* algorithms, for various clients.

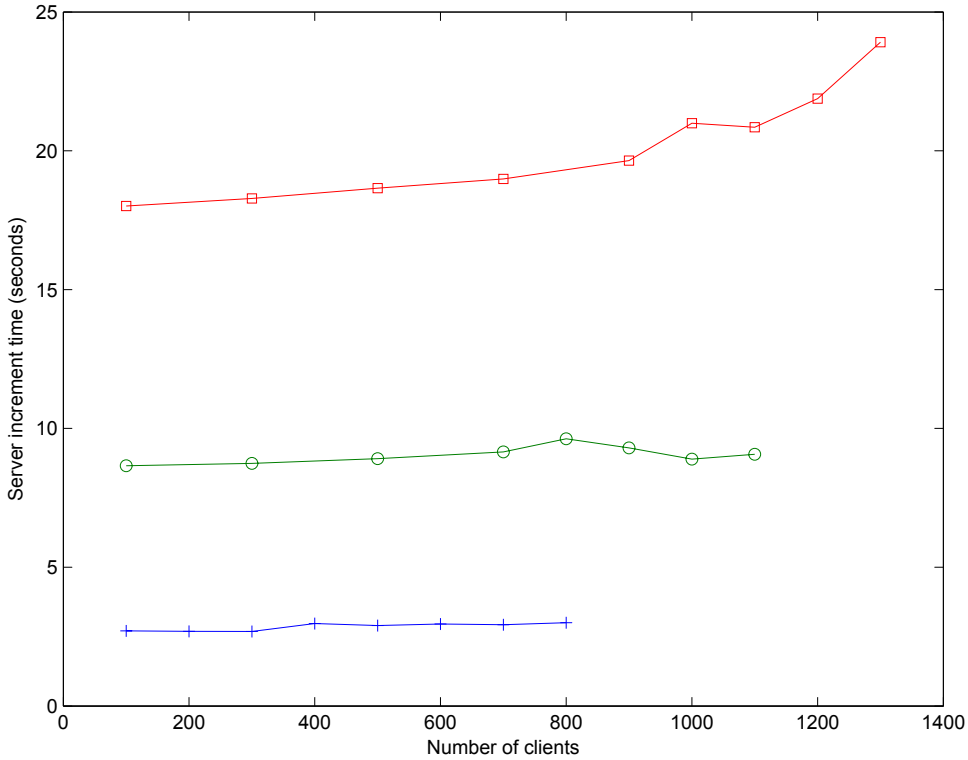

Figure 21: Comparison of the average time spent by a client at the server during an increment of the *Multiplexed* and *Shared* algorithms, for various clients.

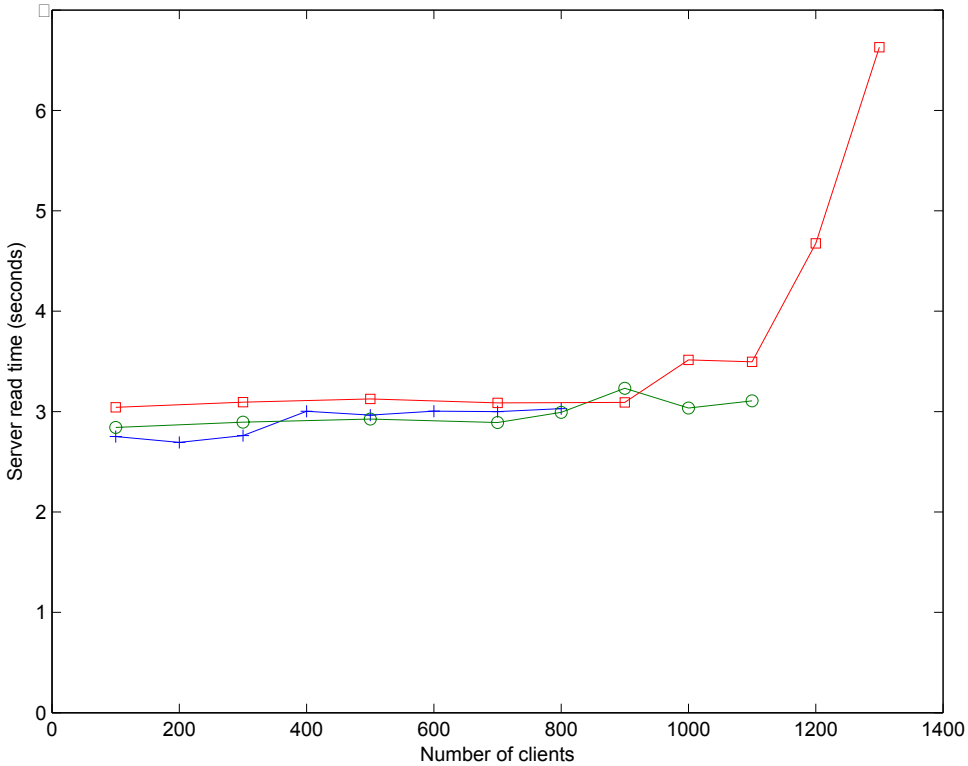

Figure 22: Comparison of the average time spent by a client at the server during a read of the *Multiplexed* and *Shared* algorithms, for various clients.

multiplex factor. However, because reads are not constrained by multiplexing, the server read time should be the same for the *Multiplexed* algorithm as for the *Shared* algorithm.

Looking first at Figure 21, we first see that increments in the *Multiplexed* algorithms take significantly more time on the server than increments for the *Shared* algorithm. While this does not appear to be a linear relationship at first, we point out that the relationship should actually be affine: the wait time should increase linearly with the multiplex factor, but the 1.3s TPM increment time is constant. Ignoring the increment time, the *Shared* algorithm then takes $\approx$ 1.4s, while *Multiplexed-5* takes $\approx$ 7.3s, and *Multiplexed-10* takes $\approx$ 16.7s.

Furthermore, we notice that the server increment times are still mostly independent of the number of clients. We recall that this is because the TPM is no longer the bottleneck of the system, but all requested reads or increments can be processed concurrently.

Multiplexed-10 does demonstrate abnormal behavior above 1000 clients. We note that this is in its saturated region, but do not fully understand the cause of this behavior.

Finally, we look at the average server read times in Figure 22. As expected, the server read times for all three algorithms are roughly the same. The *Multiplexed-10* algorithm again showed abnormal behavior above 1000 clients, but we are not able to explain this at this time.

In summary, we have demonstrated how the *Multiplexed* algorithm can multiplex client increments to increase the number of clients that VMCStore can manage. By increasing the multiplex factor, the *Multiplexed* algorithm can decrease the size of validity proofs, relieving network bandwidth pressures, but at the cost of increasing the time that client increments have to wait at the server. For a given system, the optimal multiplex factor would need to be analyzed by considering the number of clients, the available network bandwidth, the application, and client access patterns.

6.4 Conclusion

The *Shared* and *Multiplexed* algorithms provide a significant improvement over the *Simple* algorithm, in terms of the upper bound for number of clients that can be supported, by moving the bottleneck of the system from the TPM to the network.

Still, the question remains, how practical would a VMCStore-like trusted storage system be in the real world? From the server perspective, we argue that this largely depends on the available network bandwidth.

From the client perspective, there are two main issues: the certificate size, and the server operation time.

These experiments were run under the observation that any time or bandwidth incurred during communication with the virtual counter manager would be on top of the costs of communicating with the storage server. The practicality of the system is then dependent on the parameters of the counter server, as well as the details of the storage system.

For example, if client is using the storage system for managing large files (e.g. 10MB+), the overhead of spending a few seconds at the virtual counter manager and retrieving an 80kB certificate may be acceptable. However, if the client is using the storage system to store many small files, maintaining a virtual counter for each file may be prohibitive.

In conclusion, we have shown that it is possible to build a trusted storage system on an untrusted server, around a TPM. Furthermore, we believe that the results of the experiments we have performed on the VM-CStore system suggest that a TPM-based trusted storage system may be practical for certain applications, even for a relatively large number of clients.

7 Future Work and Conclusions

While the VMCStore framework provides preliminary evidence that the TPM could be used as the basis for a trusted storage system, many research areas are still open that would make such a system more practical.

As noted in Section 6, the network bandwidth tends to be the system bottleneck as the number of clients increases. Currently, the network output grows quadratically with the number of concurrent clients. One improvement for the shared nonces would be to use an authenticated hash tree [28] as opposed to the flat log of individual nonces which is currently used. While this would not reduce proof size significantly, the nonce sizes would only grow $O(\log n)$ with the number of clients, as opposed to their current linear relation.

The required network bandwidth could also be reduced by optimizing the read and increment certificates, reducing them to a minimal set of required information. In particular, the current TPM/J CountStamp data structure contains unnecessary redundant data (e.g., it contains both the encrypted and unencrypted forms of the increment command and its result). It may be possible to optimize certain VMCStore data structures as well: the LogReadCertificate probably does not need to return the SharedIncCertificate of the most recent increment if the most recent confirmation certificate is returned. Finally, we suspect there may be more efficient ways, instead of relying on Java serialization, to transfer the VMCStore data structures over the network.

Finally, while VMCStore provides ***tamper-evident*** trusted storage, it provides little-to-no protection against failures and ***denial-of-service*** attacks. A TPM-based trusted storage system would require a layer of replication to become commercially feasible. One such replication scheme, which would transform VMCStore from a tamper-evident to a ***temper-tolerant*** system, can be found in Appendix C of [45].

In conclusion, this thesis has presented VMCStore, a framework for a trusted storage system based solely on trusting a TPM 1.2. Preliminary experimental results, performed on PlanetLab, have demonstrated that such a system could be used to implement and manage *virtual monotonic counters*, providing trusted storage for a significant number of clients. VMCStore uses the concept of *log-based validation proofs* to prove the validity of its counters. A variety of algorithms based on these log-based validation proofs were developed and tested, demonstrating flexibility in the system, bottlenecks and potential areas of improvement.

References

[1] A. Anagnostopoulos, M. Goodrich, and R. Tamassia. Persistent Authenticated Dictionaries and Their Applications. In *4th International Conference on Information Security (ISC)*, 2001.

[2] T. Arnold and L. van Doorn. The IBM PCIXCC: A new cryptograhic co-processor for the IBM eServer. *IBM Journal of Research and Development*, 48:475–487, 2004.

[3] A. Buldas, P. Laud, and H. Lipmaa. Accountable Certificate Management using Undeniable Attestations. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 9–17, 2002.

[4] A. Buldas, P. Laud, and H. Lipmaa. Eliminating Counterevidence with Applications to Accountable Certificate Management. *Journal of Computer Security*, 10:273–296, 2002.

[5] A. Buldas, P. Laud, H. Lipmaa, and J. Villemson. Time-stamping with binary linking schemes. In *Advances in Cryptology - CRYPTO '98*, pages 486–501, 1998.

[6] A. Buldas, H. Lipmaa, and B. Schoenmakers. Optimally efficient accountable time-stamping. In *Public Key Cryptography '2000*, pages 293–305, 2000.

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Third Symposium on Operating Systems Design and Implementation*, February 1999.

[8] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[9] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental Multiset Hash Functions and their Application to Memory Integrity Checking. In *Advances in Cryptology - Asiacrypt 2003 Proceedings*, volume 2894 of *LNCS*. Springer-Verlag, 2003.

[10] P. T. Devanbu and S. G. Stubblebine. Stack and queue integrity on hostile platforms. *Software Engineering*, 28(1):100–108, 2002.

[11] D. Eastlake and P. Jones. RFC 3174: US secure hashing algorithm 1 (SHA1), Sept. 2001.

[12] S. Even, O. Goldreich, and S. Micali. On-line/off-line digital signatures. *Journal of Cryptology*, 9(1):35–67, 1996.

[13] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and Merkle Trees for Efficient Memory Integrity Verification. In *Proceedings of Ninth International Symposium on High Performance Computer Architecture*, New-York, February 2003. IEEE.

[14] M. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing. In *Proceedings of DARPA Information Survivability Conference and Exposition*, pages 68–82, 2001.

[15] Hewlett Packard. Embedded Security for HP ProtectTools. http://h18004.www1.hp.com/products/security/embedded_security.html, 2007.

[16] M. Kallahala, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the Second Conference on File and Storage Technologies (FAST 2003)*, 2003.

[17] P. Kocher. On certificate revocation and validation. In *Proceedings of Financial Cryptography 1998*, pages 172–177, 1998.

[18] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):241–248, 1979.

[20] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[21] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[22] U. Maheshwari, R. Vingralek, and W. Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of OSDI 2000*, 2000.

[23] P. Maniatis. *Historic Integrity in Distributed Systems*. PhD thesis, Stanford University, Aug. 2003.

[24] P. Maniatis and M. Baker. Enabling the Archival Storage of Signed Documents. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 31–45, 2002.

[25] P. Maniatis and M. Baker. Secure History Preservation through Timeline Entanglement. In *Proceedings of the 11th USENIX SEcurity Symposium*, 2002.

[26] D. Mazières and D. Shasha. Don't trust your file server. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.

[27] D. Mazières and D. Shasha. Building Secure File Systems out of Byzantine Storage. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing*, pages 108–117, 2002.

[28] R. C. Merkle. Protocols for public key cryptography. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[29] S. Micali. Efficient certificate revocation. Technical Report MIT/LCS/TM-542b, 1996.

[30] Microsoft Corporation. BitLocker Drive Encryption . http://technet.microsoft.com/en-us/windowsvista/aa905065.aspx, 2007.

[31] C. Mitchell, editor. *Trusted Computing*. The Institution of Electrical Engineers, 2005.

[32] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proceedings 7th USENIX Security Symposium (San Antonio, Texas)*, 1998.

[33] Omen Wild. Enforcer Homepage. http://enforcer.sourceforge.net/, 2004.

[34] L. F. G. Sarmenta and contributors. TPM/J: Java-based API for the Trusted Platform Module (TPM). http://projects.csail.mit.edu/tc/tpmj/, Dec. 2006.

[35] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a tpm without a trusted os. In *Proceedings of the first ACM*

workshop on Scalable trusted computing, Applications and compliance, pages 27–42, New York, NY, USA, 2006. ACM Press.

[36] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *The Seventh USENIX Security Symposium Proceedings, USENIX Press*, pages 53–62, January 1998.

[37] B. Schneier and J. Kelsey. Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1998.

[38] S. W. Smith and S. H. Weingart. Building a High-Performance, Programmable Secure Coprocessor. *Computer Networks (Special Issue on Computer Network Security)*, 31(8):831–860, April 1999.

[39] G. E. Suh. *AEGIS: A Single-Chip Secure Processor*. PhD thesis, Massachusetts Institute of Technology, Aug. 2005.

[40] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the* 17^{th} *Int'l Conference on Supercomputing (MIT-CSAIL-CSG-Memo-474 is an updated version)*, New-York, June 2003. ACM.

[41] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *Proceedings of the* 32^{nd} *Annual International Symposium on Computer Architecture*, New-York, June 2005. ACM.

[42] The Trustees of Princeton University. PlanetLab — An open platform for developing, deploying, and accessing planetary-scale services. https://www.planet-lab.org/, 2007.

[43] Trusted Computing Group. TCG TPM Specification version 1.2, Revisions 62-94 (Design Principles, Structures of the TPM, and Commands). https://www.trustedcomputinggroup.org/specs/TPM/, 2003-2006.

[44] Trusted Computing Group. Trusted Computing Group: TPM FAQ. https://www.trustedcomputinggroup.org/faq/TPMFAQ/, 2007.

[45] M. van Dijk, L. F. G. Sarmenta, J. Rhodes, and S. Devadas. Securing Shared Untrusted Storage by using TPM 1.2 Without Requiring a Trusted OS. Technical report, MIT CSAIL CSG Technical Memo 498, May 2007.

[46] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.