TrInc: Small Trusted Hardware for Large Distributed Systems

Dave Levin, *University of Maryland*
John R. Douceur, *Microsoft Research*
Jacob R. Lorch, *Microsoft Research*
Thomas Moscibroda, *Microsoft Research*

Abstract

A simple yet remarkably powerful tool of selfish and malicious participants in a distributed system is "equivocation": making conflicting statements to others. We present TrInc, a small, trusted component that combats equivocation in large, distributed systems. Consisting fundamentally of only a non-decreasing counter and a key, TrInc provides a new primitive: unique, once-in-a-lifetime attestations.

We show that TrInc is practical, versatile, and easily applicable to a wide range of distributed systems. Its deployment is viable because it is simple and because its fundamental components—a trusted counter and a key—are already deployed in many new personal computers today. We demonstrate TrInc's versatility with three detailed case studies: attested append-only memory (A2M), PeerReview, and BitTorrent.

We have implemented TrInc and our three case studies using real, currently available trusted hardware. Our evaluation shows that TrInc eliminates most of the trusted storage needed to implement A2M, significantly reduces communication overhead in PeerReview, and solves an open incentives issue in BitTorrent. Microbenchmarks of our TrInc implementation indicate directions for the design of future trusted hardware.

1 Introduction

As wide-area systems grow in scale, so do their exposure to threats. Much of the interesting distributed-systems research of the past decade has focused on the issues of security and adversarial incentive that are inherent to large-scale systems. This research has addressed a wide range of applications, including storage [2, 16, 19, 22, 28], communication [4, 45, 30], databases [40], content distribution [15, 24, 32, 36], grid computation [12], and games [3, 10], in addition to generic infrastructure [1, 5, 9, 18, 23, 43]. Virtually all of this work shares a common supposition, namely that the individual components in the system are completely untrusted.

Recently, the necessity of this supposition has been called into question. The Attested Append-only Memory (A2M) system by Chun et al. [7] showed that a small trusted module in each distributed component can significantly improve system security. In addition to founding this important new research direction, A2M made two key contributions: First, they proposed a particular abstraction for such a module, namely a *trusted log*. Second, they showed specifically that their proposed abstraction could improve the degree of fault tolerance to Byzantine faults in the server components of client-server systems.

Despite our appreciation for this work, we are concerned that distributed-protocol designers may be reluctant to start assuming the availability of such trusted modules. We have two reasons for this concern: First, the abstraction of a trusted log may require more storage space and complexity than researchers are comfortable assuming, particularly for an embedded module inside a potentially hostile component. Second, designers may have difficulty appreciating how broadly applicable a trusted module can be to distributed protocols.

In this paper, we continue the research direction begun by A2M, with an eye toward addressing these two issues. First, we have developed a significantly smaller abstraction: Instead of a trusted log, we propose a *trusted incrementer* (*TrInc*), which is little more than a monotonic counter and a key. Second, we demonstrate a more inclusive set of architectures, running a broader range of protocols, yielding a wider set of benefits: Our architectures include not only client-server systems but also peer-to-peer systems. Our protocols include not only Byzantine-fault-tolerant protocols but also PeerReview [13] and BitTorrent [8]. Our demonstrated benefits include not only improving fault tolerance but also reducing communication overhead and solving an open incentive problem.

We show that TrInc has several benefits over A2M. First, its smaller size and simpler semantics make it easier to deploy, as we demonstrate by implementing it on real, currently available trusted hardware. Second, we observe that TrInc's core functional elements are included in the Trusted Platform Module (TPM) [38] found on many modern PCs, lending credence to the idea that such a component could become widespread. Third, TrInc makes use of a shared symmetric session key among all participants in a protocol instance, which significantly decreases the cryptographic overhead.

The rest of this paper is structured as follows. §2 provides background on the underlying problem addressed by TrInc (and by A2M), as well as a primer on trusted hardware. §3 then presents the design of TrInc, and §4 analyzes its security. §§5, 6, and 7 respectively describe several protocols we modified to use TrInc, our trusted hardware implementation, and our evaluation thereof.

Property	Accountability layer		Trusted module	
	PeerReview [13]	Nysiad [14]	A2M [7]	TrInc
No centralized trust	✓*	✓*		
Easy to deploy	✓	✓		✓
Easy to apply to existing protocols	✓	✓†		✓‡
Immediate consistency		✓	✓	✓
No assumptions about protocol's determinism		✓†	✓	✓
No additional online assumptions			✓	✓
Additional communication overhead per protocol message, with witness sets of size W	$O(W^2)$	$O(W^2)$	$O(1)$	$O(1)$

Table 1: Summary of the properties of various equivocation-fighting systems. *While PeerReview and Nysiad do not require centralized trust, they do make use of a PKI. †Nysiad deals with nondeterminism by treating nondeterministic events as inputs; this requires protocol changes for nondeterministic state machines. ‡We found that, although TrInc requires a protocol redesign, the modifications are often localized, and vastly simplify security procedures.

2 Background and Related Work

2.1 Equivocation in distributed systems

Since 1982, it has been known that tolerating f Byzantine faults requires at least $3f + 1$ participants [20]. This stands in marked contrast to the case for f stopping faults, which more intuitively requires $2f + 1$ participants. A key insight behind A2M [7] was the observation that a single property of Byzantine faults is responsible for the difference between these two bounds. That property is *equivocation*, meaning the ability to make conflicting statements to different participants. A2M provides a mechanism that prevents participants from equivocating, thereby improving the fault tolerance of Byzantine protocols to f out of $2f + 1$.

We make the further observation that equivocation is a necessary property for many forms of cheating and fraud, not merely for classical Byzantine faults. For instance, in BitTorrent, recent work [21] has shown an exploit in which a peer can obtain an unfairly high download rate by lying about which chunks of a file it has received. This is equivocation, insofar as the peer acknowledges receiving a chunk from the peer that provided it, but then tells another peer that it does not have the chunk.

The following are three more brief examples:

- In a simultaneous-turn game, one can cheat by observing an opponent's move before making one's own move; this is equivocating about whether one has yet moved.
- In a distributed electronic currency system, one can counterfeit money by equivocating to different payees about whether one has spent a particular bill.
- In an election, the tallier can disrupt the vote by equivocating to a voter and an official about whether the voter's vote was recorded.

In §5.5, we will consider many other cases of malicious behavior that can be interpreted as equivocation.

2.2 Prior solutions to equivocation

Several recent efforts have addressed the problem of Byzantine faults in distributed systems. Although their approaches to the problem are very different, they have all effectively focused on the issue of equivocation. Table 1 summarizes our analysis of their properties.

PeerReview [13] is a system that employs witnesses to collect a tamper-evident record of all messages in a distributed system for subsequent checking against a reference implementation. Unlike the remaining approaches we will discuss, PeerReview does not provide fault tolerance. Instead, it provides eventual fault detection and localization, which the system's designers argue leads to fault deterrence. The tamper-evident record is a distributed collection of logs that are authenticated using hash chains. The purpose of the tamper-evidence is to detect equivocation about the messages recorded in a log. As shown in Table 1, the communication required to collectively manage the tamper-evident message log is quadratic in the size of the witness set.

Nysiad [14] is a mechanism that transforms crash-tolerant distributed systems into Byzantine-fault-tolerant ones. It does this by assigning a set of *guards* (comparable to witnesses) to each host in the system. The guards validate the messages sent by their associated hosts, using replicas of the hosts' execution engines. The potential for equivocation in Nysiad is that the host might send different messages to different guards or order its messages differently for different guards. To deal with this equivocation, the guards gossip among each other to agree on the order and content of messages sent by the host. As shown in Table 1, this gossip requires a count of messages that is quadratic in the number of guards. Relative to PeerReview, Nysiad has the benefit of immediate consistency, rather than eventual detection. Nysiad is also able to handle nondeterministic state machines, but doing so requires protocol changes to treat nondeterministic events as inputs.

Attested Append-only Memory, or A2M [7], is a

trusted module that is embedded in an untrusted machine, for the purpose of improving the fault tolerance of a distributed protocol. The A2M module provides the abstraction of a trusted log, which the machine can append to but not otherwise modify. This limitation prevents the machine from equivocating about whether it performed a particular action at a particular step, because once the action is recorded in the log, it cannot be overwritten. A2M uses cryptography to enforce its properties and to attest the log's contents to other machines. Relative to Nysiad and PeerReview, A2M does not require any additional online communication between machines beyond what is required in the base protocol. Consequently, the communication overhead is merely a constant factor due to the cryptographic attestations that accompany the protocol's messages.

As we will show in §3, TrInc is significantly smaller than A2M, making it easier to deploy. TrInc also has another advantage, namely that its use is less tightly coupled to the distributed protocol than use of A2M is. Specifically, because A2M's trusted log has finite storage, it provides a log-truncation operation, but opportunities to truncate the log may be limited by the protocol. Conversely, message sequencing in the protocol may be constrained by the available space in A2M's log. Perhaps in part to address this concern, A2M considered various implementations in addition to hardware, some of which would likely have plentiful storage for the log. These include a remote service, a software-isolated process, and a memory-isolated virtual machine. By contrast, the protocol modifications required to use TrInc tend to be quite localized. Furthermore, TrInc's use of a shared session key often simplifies the protocol.

2.3 Trusted hardware

There have been many trusted hardware designs that predate both TrInc and A2M. Perhaps most similar to TrInc is the abstraction of *virtual monotonic counters* [34]. These are similar to the four increment-only counters included in the current specification of the TPM [38]. Van Dijk et al. propose an algorithm by which to emulate multiple counters with a single trusted counter [39]. We believe a similar approach could ease TrInc's deployment by requiring fewer physical counters. Further, other systems have been proposed that make use of trusted hardware, such as for securing database systems [26] and auctions [31]. To the best of our knowledge, TrInc is the first trusted component designed to be used in large-scale, distributed systems.

3 TrInc Design

3.1 Design Goals

The fundamental security goal of TrInc is to remove participants' ability to equivocate. Consider the situation in which Mallory wishes to send conflicting messages to Alice and Bob. Common approaches to combating such equivocation require Alice and Bob to communicate with one another [13, 14, 20] or with a third party, so they can learn of the distinct messages sent to each. Unfortunately, this additional communication overhead can become a bottleneck for the overlying system, and constitutes the super-linear number of messages in PeerReview [13].

One goal of TrInc is to therefore minimize both communication overhead and the number of non-faulty participants required. With trusted hardware, it is possible to remove Mallory's ability to equivocate without *any* communication between Alice and Bob [7].

The other broad goal of TrInc is to be practical for distributed systems *today*. To be practical, a trusted component must be *small* so that it is feasible to manufacture and deploy. Arbitrary computation and a large amount of storage are difficult and costly to make tamper-resistant. Further, to be a practical primitive in distributed systems, the trusted component must have an API with which it is easy to build distributed systems.

3.2 Overview

To gain the benefits of TrInc, a user must attach a trusted piece of hardware we call a *trinket* to his computer. Unlike a typical TPM, which must attest to states of the associated computer, the trinket's API depends only on its internal state, so the trinket does not need access to the state of the computer. All it needs is an untrusted channel over which it can receive input and produce output, so even USB is quite sufficient.

When Mallory wishes to send a message m to Alice, she must include an attestation from her trinket that (1) binds m to a certain value of a counter, and (2) ensures Alice that no other message will ever be bound to that value of that counter, even messages sent to other users. A trinket enables such attestation by using a counter that monotonically increases with each new attestation. In this way, once Mallory has bound a message m to a certain counter value c, she will never be able to bind a different message m' to that value.

As we show in our case studies in §5, some protocols benefit from using multiple counters. In theory, anything done with multiple counters can be done with a single counter, but multiple counters allow certain performance optimizations and simplifications, such as assigning semantic meaning to a particular counter value. Furthermore, the user of a trinket may participate in multiple protocols, each requiring its own counter or counters. Therefore, a trinket provides the ability to allocate new counters. However, we must identify each of them uniquely so that a malicious user cannot create a new counter with the same identity as an old counter and thereby attest to a different message with the same counter identity and value.

As a performance optimization, TrInc allows its attestations to be signed with shared symmetric keys, which

vastly improves its performance over using asymmetric cryptography or even secure hashes. To ensure that participants cannot generate arbitrary attestations, the symmetric key is stored in trusted memory, so that users cannot read it directly. Symmetric keys are shared among trinkets using a mechanism that ensures they will not be exposed to untrusted parties.

3.3 Notation

We use the notation $\langle x \rangle_K$ to mean an attestation of x that could only be produced by an entity knowing K. If K is a symmetric key, then this attestation can be verified only by entities that know K; if K is a private key, then this attestation can be verified by anyone, or more accurately anyone who knows the corresponding public key. We use the notation $\{x\}_K$ to mean the value x encrypted with public key K, so that it can only be decrypted by entities knowing the corresponding private key.

3.4 TrInc state

Figure 1 describes the full internal state of a trinket, which we describe in more detail here. Each trinket is endowed by its manufacturer with a unique identity I and a public/private key pair $(K_{\text{pub}}, K_{\text{priv}})$. Typically, I will be the hash of K_{pub}. The manufacturer also includes in the trinket an attestation $\mathcal{A}$ that proves the values I and K_{pub} belong to a valid trusted trinket, and therefore that the corresponding private key is unknown to untrusted parties.

We leave open the question of what form $\mathcal{A}$ will take. This attestation is meant to be evaluated by users, not by trinkets, and so can be of various forms. For instance, it might be a certificate chain leading to a well-known authority trusted to oversee trinket production and ensure their secrets are well kept.

Another element of the trinket's state is the *meta-counter* M. Whenever the trinket creates a new counter, it increments M and gives the new counter identity M. This allows users to create new counters at will, without sacrificing the non-monotonicity of any particular counter. Because M only goes up, once a counter has been created it can never be recreated by a malicious user attempting to reset it.

Yet another element is Q, a limited-size FIFO queue containing the most recent few counter attestations generated by the trinket. It is useful for allowing users to recover from power failures, as we will describe later.

The final part of a trinket's state is an array of counters, not all of which have to be in use at a time. For each in-use counter, the state includes the counter's identity i, its current value c, and its associated key K. The identity i is, as described before, the value of the meta-counter when the counter was created. The value c is initialized to 0 at creation time and cannot go down. The key K contains a symmetric key to use for attestations of this counter; if $K = 0$, attestations will use the private key K_{priv} instead.

Global state:

Notation	Meaning
K_{priv}	Unique private key of this trinket
K_{pub}	Public key corresponding to K_{priv}
I	ID of this trinket, the hash of K_{pub}
$\mathcal{A}$	Attestation of this trinket's validity
M	Meta-counter: the number of counters this trinket has created so far
Q	Limited-size FIFO queue containing the most recent few counter attestations generated by this trinket

Per-counter state:

Notation	Meaning
i	Identity of this counter, i.e., the value of M when it was created
c	Current value of the counter (starts at 0, monotonically non-decreasing)
K	Key to use for attestations, or 0 if K_{priv} should be used instead

Figure 1: State of a trinket

3.5 TrInc API

Figure 2 shows the full API of a trinket, described in more detail in this subsection.

3.5.1 Generating attestations

The core of TrInc's API is `Attest`. `Attest` takes three parameters: i, c', and h. Here, i is the identity of a counter to use, c' is the requested new value for that counter, and h is a hash of the message m to which the user wishes to bind the counter value. `Attest` works as follows:

Algorithm 1 `Attest`(i, c', h, n)

1. Assert that i is the identity of a valid counter.
2. Let c be the value of that counter, and K be the key.
3. Assert no roll-over: $c \leq c'$.
4. If $K \neq 0$, then let $a \leftarrow \langle I, i, c, c', h \rangle_K$; otherwise let $a \leftarrow \langle I, i, c, c', h \rangle_{K_{\text{priv}}}$.
5. Insert a into Q, kicking out oldest value.
6. Update $c \leftarrow c'$.
7. Return a.

Note that `Attest` allows calls with $c' = c$. This is crucial to allowing peers to attest to what their current counter value is without incrementing it. To allow for this while still keeping peers from equivocating, TrInc includes both the prior counter value and the new one. One can easily differentiate attestations intended to learn a trinket's current counter value ($c = c'$) from attestations that bind new messages ($c < c'$).

3.5.2 Verifying attestations

Suppose a user Alice with trinket A wants to send a message to user Bob with trinket B. She first invokes

Function	Operation
`Attest`(i, c', h)	Verifies that i is a valid counter with some value c and key K. Verifies that $c \leq c'$. Creates an attestation $a = \langle \text{COUNTER}, I, i, c, c', h \rangle_K$; if $K = 0$, uses K_{priv} instead of K. Adds a to Q. Sets $c = c'$. Returns a.
`GetCertificate`()	Returns a certificate of this trinket's validity: $(I, K_{\text{pub}}, \mathcal{A})$.
`CheckAttestation`(a, i)	Returns a boolean indicating whether a is the output of invoking `Attest` on a trinket using the same symmetric key as the one associated with counter i.
`CreateCounter`()	Increments M. Creates a new counter with $i = M$, $c = 0$, and $K = 0$. Returns i.
`FreeCounter`(i)	If i is the identity of a valid counter, deletes that counter.
`ImportSymmetricKey`($\mathcal{S}, i$)	Verifies that $\mathcal{S}$ is an encrypted symmetric key decryptable with K_{priv}. Decrypts it and installs the included key as K for counter i.
`GetRecentAttestations`()	Returns Q.

Figure 2: API of a trinket

`Attest` on her trinket using the message's hash, and thereby obtains an attestation a. Next, she sends the message to Bob along with this attestation. However, for Bob to accept this message, he needs to be convinced that the attestation was created by a valid trinket. There are two cases to consider: first, that the attestation used A's private key K^A_{priv}, and second, that the attestation used a shared symmetric key K.

In the first case, the API call `GetCertificate` will be useful. This call returns a certificate $\mathcal{C}$ of the form $(I, K_{\text{pub}}, \mathcal{A})$, where I is the trinket's identity, K_{pub} is its public key, and $\mathcal{A}$ is an attestation that I and K_{pub} belong to a valid trinket. Alice can call this API routine and send the resulting certificate $\mathcal{C}^A$ to Bob. Bob can then (a) learn Alice's public key K^A_{pub}, and (b) verify that this is a valid trinket's public key. After this, he can verify the attestation Alice attached to her message, and any future attestations she attaches to messages.

In the second case, the API call `CheckAttestation` is useful. When `CheckAttestation`(a, i) is invoked on a trinket, the trinket checks whether a is the output of invoking `Attest` on a trinket using the same symmetric key as the one associated with the local counter i. It returns a boolean indicating whether this is so. So, if Alice sends Bob an attestation signed with a shared symmetric key, Bob can invoke `CheckAttestation` on his trinket to learn whether the attestation is valid.

3.5.3 Allocating counters

Since a trinket may contain many counters, another important component of TrInc's API is the creation of these counters. TrInc creates new logical counters, and allows counters to be deleted, but never resets an existing counter. Logical counters are identified by a unique ID, generated using a non-deletable, monotonic *meta-counter* M. Every trinket has precisely one meta-counter, and when it expires, the trinket can no longer be used; we compensate for this by making M 64 bits, only incrementing M, and assigning no semantic meaning to M's value. TrInc exports a `CreateCounter` function that increments M; allocates a new counter with identity $i = M$, initial value 0, and initial key $K = 0$; and returns this new identity i. When the user no longer needs the counter, she may call `FreeCounter` to free it and thereby provide space in the trinket for a new counter.

3.5.4 Using symmetric keys

TrInc allows its attestations to be signed with shared symmetric keys, which vastly improves its performance over using asymmetric cryptography or even secure hashes. When a set of users are willing to use a single symmetric key for a certain purpose, we call this a *session*. Creating a session requires a *session administrator*, a user trusted by all participants to create a session key and keep it safe, i.e., to not reveal it to any untrusted parties.

To create a session, the session administrator simply generates a random, fresh symmetric key as the session key K. To allow a certain user to join the session, he asks that user for his trinket's certificate $\mathcal{C}$. If the session administrator is satisfied that the certificate represents a valid trinket, he encrypts the key in a way that ensures it can only be decrypted by that trinket. Specifically, he creates $\{\text{KEY}, K\}_{K_{\text{pub}}}$, where K_{pub} is the public key in $\mathcal{C}$. He then sends this encrypted session key to the user who wants to join the session.

Upon receipt of an encrypted session key, the user can join one of his counters to the session by using the API call `ImportSymmetricKey`($\mathcal{S}, i$). This call checks that $\mathcal{S}$ is a valid encrypted symmetric key, meant to be decrypted by the local private key. If so, it decrypts the session key and installs it as K for local counter i. From this point forward, attestations for this counter will use the symmetric key. Also, the user will be able to verify any trinket's attestation a using this symmetric key by invoking `CheckAttestation`(a, i).

3.5.5 Handling power failures

One practical concern is that of power failure. Unlike A2M, TrInc users need not query the trusted hardware to

obtain attestations. Instead, TrInc relies on the application (or a TrInc driver) to store attestations in untrusted, persistent storage. If there is a power failure between the time that the trinket advances its counter and the application writes it to disk, then the attestation is lost. This can be problematic for many protocols, which rely on the user being able to attest to a message with a particular counter value. For instance, if Charlie cannot produce an attestation for counter value v, Alice may suspect this is because Charlie has already told Bob about some message m associated with that counter value. Not wanting to be fooled about the absence of such a message, Alice may lose all willingness to trust Charlie.

To alleviate this, a trinket includes a queue Q containing the most recent attestations it has created. To limit the storage requirements, this queue only holds a certain fixed number k of entries, perhaps 10. In the event of a power failure, after recovery the user can invoke the API call `GetRecentAttestations` to retrieve the contents of Q. Thus, all a user must do to protect against power failure is make sure she writes a needed attestation to disk before she makes her kth next attestation request. As long as k is at least 1, the user can safely use the trinket for any application. Higher values of k are useful as a performance optimization, allowing greater pipelining between writing to disk and submitting attestations.

So far we have only discussed a power failure affecting the user, but a power failure can also affect the trinket. The `Attest` algorithm ensures that the attestation is inserted into the queue before the counter is updated, so the trinket cannot enter a situation where the counter has been updated but the attestation is unavailable. It can, however, enter the dangerous situation in which the attestation is in Q, and thus available to the user, but the counter has not been incremented. This window of vulnerability could potentially be exploited by a user to generate multiple attestations for the same counter value, if he could arrange to shut off power at precisely this intervening time. However, we guard against this case by having the trinket check Q whenever it starts up. At startup, before handling any requests, it checks all attestations in Q and removes any that refer to counter values beyond the current one.

3.5.6 A TrInc by any other name

The computational demands of a trinket are small. It must be able to do simple operations such as comparison, as well as cryptographic operations including hashing and both symmetric and asymmetric encryption and decryption. Such cryptographic operations are standard in trusted components such as the TPM [38]. However, we recognize that hardware manufacturers and users are often highly cost-conscious and may be willing to do without performance optimization to save hardware costs.

Therefore, we propose three versions of TrInc that make different trade-offs between cost and performance, summarized in Table 2. The bronze version simply offers correctness with no performance optimizations, by leaving out the ability to use symmetric keys. The silver version is as we have described it. The gold version adds one additional optimization: the use of fast persistent memory such as battery-backed RAM. This optimization makes attestations especially fast since they need not incur the cost of writing to the slow flash memory often found in modern TPMs.

	Persistent Memory	Asym. Crypto	Symm. Crypto	Fast Memory
Bronze TrInc	✓	✓		
Silver TrInc	✓	✓	✓	
Gold TrInc	✓	✓	✓	✓

Table 2: Versions of TrInc with different performance.

3.6 Local adversaries

Mutually distrusting principals on a single computer will share access to a single trinket, creating the potential for conflict between them. Although they cannot equivocate to remote parties, they can hurt each other. They can impersonate each other by using the same counter, and they can deny service to each other by exhausting shared resources within the trinket. Resource exhaustion attacks include allocating all available counters, submitting requests at a high rate, and rapidly filling the queue Q to prevent the pipelining performance optimization.

The operating system can solve this problem by mediating access to the trinket, just as it mediates access to other devices. In this way, the OS can prevent a principal from using counters allocated to other principals, and can use rate limiting and quotas to prevent resource exhaustion. Developing a detailed API and policy for such mediation is beyond the scope of this paper, and is left for future work. However, note that a remote party need not care about how or whether such local mediation is done. Equivocation to remote parties is impossible, even if an adversary has root access to the machine, since cryptography allows the trinket to communicate securely even over an untrusted channel.

4 Analysis of TrInc

We now present a brief discussion of why TrInc is sufficient for a broad class of distributed protocols and why it is nearly minimal in size.

4.1 Equivocation

When a trinket creates an attestation with distinct old and new counter values of c and c', we say that attestation *covers* the half-open interval $(c, c']$. TrInc prevents equivocation by ensuring that no two attestations will cover overlapping intervals. This property could be violated only if:

- the counter is decremented,
- the cryptosystem is broken,

- more than one counter has the same identity, or
- more than one trinket has the same identifier.

By construction, it is not possible to decrement the counter nor to assign the same identity to multiple counters. By hypothesis, cryptographic primitives are effectively unbreakable. Finally, no two trinkets will be created with the same identifier, at least not by a trusted manufacturer; recall that users can verify whether the trinket comes from a trusted manufacturer by observing the certificate chain in $\mathcal{A}$.

4.2 Timeliness

When a trinket creates an attestation with the same old and new counter values, there is no change to the trinket's state; however, the attestation demonstrates the current value of the counter. Thus, if a machine attests to a value of a remotely supplied nonce, the remote machine can be certain that the attestation was generated after the nonce was supplied. Since this attestation carries the current counter value, the remote machine can thus also be sure that the local machine's counter is no lower than this value.

Therefore, when the local machine provides attestations of counter values up to the nonce-attested value, the remote machine can be certain that these attestations are timely.

4.3 Minimality

Suppose, during the execution of a protocol, a participant sends n messages requiring attestation, but her attesting module has fewer than $\log_2(n)$ bits of storage. The attesting module must be willing to provide all n attestations, or else it will cause the protocol to halt prematurely. However, since the module can be in fewer than n distinct states, by the pigeonhole principle it must be willing to attest to two different messages while in the same state. Since this state is as it was before the first message, it cannot reflect the trinket's having attested to the first message. This means a malicious user could take advantage of the trinket's inability to remember its first attestation when requesting the second attestation, and thereby obtain an attestation inconsistent with the earlier one. This is clearly inconsistent with the goals of a trusted module, so we come to a contradiction, and conclude that such a module requires at least $\log_2(n)$ bits of storage. In other words, it needs sufficient storage to accommodate a message counter.

Furthermore, an attesting module needs for its attestations to be unforgeable. Otherwise, the user could generate attestations without using the module, and thereby attest to both sides of an equivocation. TrInc achieves this unforgeability with simple cryptographic primitives.

In summary, the core components of TrInc, a counter and cryptography, seem to be essential for equivocation prevention.

5 Designing Systems with TrInc

5.1 Overview

When designing a protocol that incorporates TrInc, we find it important to address the following questions:

5.1.1 What does TrInc's counter represent?

In the applications we have considered, TrInc's counter represents a natural "progression" of the system. In BitTorrent, for instance, the counter represents the number of blocks a given peer has received, a value which is naturally monotonically increasing. In Byzantine Fault Tolerance (BFT), the counter represents which view a replica is in. Ultimately, the choice of what the counter represents is dependent on what data peers will need to attest to.

5.1.2 To what data do peers attest?

There are two broad types of attestations that TrInc offers. *Advance* attestations increase the trinket's counter, thus binding a message to a counter. *Status* attestations attest to the current counter without advancing it.

Advance attestations Advance attestations are largely protocol-dependent, including such elements as the set of pieces received in BitTorrent, or the root of a Merkle tree of file hashes in a file server. The specific data to which to attest often requires a careful analysis of the selfish or malicious ways in which peers could equivocate. It is important to ensure that the impossibility of equivocating about what was assigned to a particular counter value translates into the impossibility of equivocating at the higher desired semantic level.

For instance, suppose an attestation consists solely of a number n of pieces received in BitTorrent and a list of n peers. In this case, a participant Mallory can cheat in the following way. After receiving the first piece a from Alice, she replies with an attestation that her one-piece set contains only a. Next, after receiving her next two pieces b from Bob and c from Charlie, she sends them both an identical attestation that her two-piece set is b and c. In this way, Mallory gets away with hiding the fact that she has received piece a, despite not being able to get different attestations for the same value of $n = 2$. As we will see later, in §5.4, we prevent this by having an attestation include the last piece received.

Status attestations Most distributed systems do not have an implicit system-wide "counter." Rather, peers progress at varying rates: BitTorrent peers download at rates largely dependent on their own upload rates, DHT peers store varying amounts of data, and so on. Status attestations enable peers to determine others' current counter values. The data in a status attestation is generally a nonce, to ensure freshness in peers' reports of their counters. Coupled with a counter that has semantic meaning, status attestations can provide peers with up-to-date information about their neighbors. In BitTorrent, for instance, knowing how much of a file a neighbor has downloaded can help determine whether to bootstrap him

Algorithm 2 Implementation of A2M with TrInc

Init()

1. Create low and high counters:
$$\mathcal{L}_q \leftarrow \texttt{CreateCounter}();\ \mathcal{H}_q \leftarrow \texttt{CreateCounter}()$$
2. Return $\{\mathcal{L}_q, \mathcal{H}_q\}$

Append(queue q, value x)

1. Bind $h(x)$ to a unique counter (the current "high counter"):
$$a \leftarrow \texttt{Attest}(\mathcal{H}_q.\text{id},\ \mathcal{H}_q.\text{ctr} + 1,\ h(x))$$
2. Store the attestation in untrusted memory:
$$q.\texttt{append}(a, x)$$

Lookup(queue q, sequence number n, nonce z)

1. If $n < \mathcal{L}_q$, the entry was truncated. Attest to this by returning an attestation of the supplied nonce using the low-counter:
$$\texttt{Attest}(\mathcal{L}_q.\text{id},\ \mathcal{L}_q.\text{ctr},\ h(\text{FORGOTTEN}||z))$$
2. If $n > \mathcal{H}_q$, the query is too early. Attest to this by returning an attestation of the supplied nonce using the high-counter:
$$\texttt{Attest}(\mathcal{H}_q.\text{id},\ \mathcal{H}_q.\text{ctr},\ h(\text{TOOEARLY}||z))$$
3. Otherwise, return the entry in q that spans n, i.e., the one such that $a.c < n \leq a.c'$. Note that if $n < a.c'$, this means n was skipped by an `Advance`.

End(queue q, sequence number n, nonce z)

1. Retrieve the latest entry from the given log:
$$\{a, x\} \leftarrow q.\texttt{end}()$$
2. Attest that this is the latest entry with a high-counter attestation of the supplied nonce:
$$a' \leftarrow \texttt{Attest}(\mathcal{H}_q.\text{id},\ \mathcal{H}_q.\text{ctr},\ z)$$
3. Return $\{a', \{a, x\}\}$

Truncate(queue q, sequence number n)

1. Remove the entries from untrusted memory:
$$q.\texttt{truncate}(n)$$
2. Move up the low counter:
$$a \leftarrow \texttt{Attest}(\mathcal{L}_q.\text{id},\ n,\ \text{FORGOTTEN})$$

Advance(queue q, sequence number n, value x)

1. Append a new item with sequence number n:
$$a \leftarrow \texttt{Attest}(\mathcal{H}_q.\text{id},\ n,\ h(x))$$
2. Store the attestation in untrusted memory:
$$q.\texttt{append}(a, x)$$

with free pieces (because he is new to the swarm) or to initiate a trade with him (because he has many interesting pieces of the file).

5.2 Case study 1: A2M

Attested Append-only Memory (A2M) [7] is another proposed trusted hardware design with the intent of combating equivocation. A2M offers *trusted logs*, to which users can only append. The fundamental difference between the designs of A2M and TrInc are in the amount of state and computation required from the trusted hardware. To demonstrate that TrInc's decreased complexity is enough, we present, as our first case study, how to build A2M using TrInc.

5.2.1 A2M overview

A2M's state consists of a set of logs, each containing entries with monotonically increasing sequence numbers. A2M supports operations to add (`append` and `advance`), retrieve (`lookup` and `end`), and delete (`truncate`) items from its logs. The basis of A2M's resilience to equivocation is `append`, which binds a message to a unique sequence number. For each log q, A2M stores the lowest sequence number, $\mathcal{L}_q$, and the highest sequence number, $\mathcal{H}_q$, stored in q. A2M appends an entry to log q by incrementing the sequence number $\mathcal{H}_q$ and setting the new entry's sequence number to be this incremented value. The low and high sequence numbers allow A2M to attest to failed lookups; for instance, if a user requests an item with sequence number $s > \mathcal{H}_q$, A2M returns an attestation of $\mathcal{H}_q$.

5.2.2 Trusted logs with TrInc

In our TrInc-based design of A2M, we store logs in untrusted memory, as opposed to within a trinket. As in A2M, we make use of two counters per log, representing the highest ($\mathcal{H}_q$) and lowest ($\mathcal{L}_q$) sequence number in the respective log q.

We present the detailed protocol in Algorithm 2, and summarize some of its characteristics here. Note the power of TrInc's simple API; our design is built predominately on calls to a trinket's `Attest` function. Our protocol uses advance attestations for moving the high sequence number when appending to the log, and for moving the low sequence number when deleting from the log. We perform status attestations of the low counter value to attest to failed lookups, and of the high counter to attest to the end of the log. No additional attestations are necessary for a successful `lookup`, even if the `lookup` is to a skipped entry. Conversely, A2M requires calls to the trusted hardware even for successful lookups.

5.2.3 Properties of a TrInc-based A2M

Chun et al. [7] demonstrate how to apply A2M to BFT [20], SUNDR [22], and Q/U [1]. Our implementation of A2M in TrInc demonstrates that TrInc, too, can be applied to these systems.

Implementing trusted logs using TrInc has several benefits over a completely in-hardware design like A2M. Because TrInc stores the logs in untrusted storage, we decouple the usage demand of the trusted log from the amount of available trusted storage. Conversely, limited by the amount of trusted storage, A2M must make

more frequent calls to `truncate` to keep the logs small. Some systems, such as PeerReview [13], benefit from large logs, making TrInc a more suitable addition, which we consider next.

5.3 Case study 2: PeerReview

Accountability systems, such as PeerReview [13] and Nysiad [14], strive to augment existing protocols to make them tolerant to Byzantine faults. This is a powerful approach, as it allows system designers to focus on the system at hand, rather than consider Byzantine faults at all layers of the system. The general approach is to have participants in the system communicate with and audit one another, resulting in what is sometimes, unfortunately, a massive amount of additional communication overhead.

Our main observation in this case study is that the means by which these systems combat equivocation constitutes the bulk of their communication overhead. By applying TrInc to PeerReview, we are able to vastly reduce PeerReview's communication overhead.

5.3.1 PeerReview review

PeerReview [13] is a system that enables accountability in general distributed protocols. Unlike BFT, which ensures that bad behavior never has an effect, PeerReview allows bad behavior to affect the system but ensures that the improper act will eventually be detected. This allows a system to correct for bad behavior after the fact, and also deters bad behavior to begin with.

PeerReview works on any protocol in which each participant acts according to a deterministic state machine. PeerReview assigns each participant a set of *witnesses*, machines whose job it is to detect bad behavior by that participant. The participant is required to log all of the messages it sends and receives, and report these to the witnesses. The witnesses then run the participant's state machine to ensure the participant's outgoing messages were consistent with proper operation.

A participant might try to cheat by sending different messages to the witnesses than it sends to other participants. For this reason, when a participant receives a message from another, it forwards this message to the sender's witnesses, so they can ensure this message actually appears in the sender's log.

As a practical matter, full messages do not have to be transmitted to witnesses thanks to the use of a *tamper-evident log*. Each log entry is associated with a sequence number, and the log itself is represented by a recursive hash reflecting all log entries. When a participant sends a message, it includes a signed statement that this message has a particular sequence number and that the log had a particular recursive hash when this message was logged. In this way, the receiver only needs to report this authenticator to the witness.

PeerReview's tamper-evident log has another important use. When a participant or witness discovers bad behavior in a participant, the authenticators signed by the malefactor stand as clear proof of the misbehavior. Thus, a faulty witness cannot improperly accuse a participant, and an incompletely trusted witness can be believed when it presents evidence of a participant's misbehavior.

5.3.2 Simplifying PeerReview with TrInc

By augmenting PeerReview with TrInc, we are able to simplify much of PeerReview's protocol. We detail here the modifications we make to PeerReview in augmenting it with TrInc.

Trusted logs As demonstrated with A2M, TrInc can easily supply a trusted log without the assistance of a witness set. Our first modification is to include such a trusted log. Whenever a participant sends or receives a message, it logs that message with an attestation from its trinket. A participant should only process a received message if it is accompanied by an attestation that the message has been logged by the sender's trinket.

Audits Each witness w for a participant p keeps track of n, a log sequence number, and s, the state that p should have been in after sending or receiving the message in log entry n. It initializes n to 0 and s to the initial state of participant p.

Whenever w wants to audit p, it sends it n and a nonce. The participant returns an attestation of its current log entry number n' using the nonce, and also returns a log entry and attestation for every index i such that $n < i \leq n'$. Note that witnesses need only obtain these entries directly from p, and not from other peers with whom p has communicated. The witness then runs the reference implementation, starting at state s, and progressing through the log entries between n and n'. If the reference implementation sends the same messages that are in the log, then the witness simply updates n to n' and updates s to the state of the reference implementation at that point. If not, then the witness has proof it can present of the participant's failure to act properly.

5.3.3 Properties of a TrInc-enabled PeerReview

The benefits from applying TrInc to PeerReview are evident when considering what the protocol *no longer has to do*.

Challenge/response Enabled with TrInc, PeerReview's challenge/response protocol is no longer needed for a participant to verify a hash chain of log entries. The fact that TrInc signs the messages is sufficient. The only time a participant i has to challenge another participant j is when it sends participant j a message and receives no acknowledgment of it. In this case, the challenge works as in regular PeerReview.

Consistency TrInc further removes the need for witness-to-witness communication. In PeerReview, if p receives an authenticator from q, then p's witnesses must forward it to q's witnesses. This is not necessary in a TrInc-augmented PeerReview because there would be no way for those other participants to avoid sending the au-

thenticators themselves to their witnesses. Another way to look at it is that it is not necessary for a participant to pass on authenticators it receives to witnesses, so it is not necessary for a witness to do this on behalf of participants.

To summarize, we find that by applying TrInc to PeerReview, we are able to vastly decrease the amount of communication overhead. We demonstrate this empirically in Section 7.

5.4 Case study 3: BitTorrent

The previous two systems demonstrate that TrInc is a minimal counterpart to a related trusted component, and that it can reduce the overhead of achieving accountability in a distributed setting. Our third case study demonstrates TrInc's versatility. We show how TrInc can be applied to solving an open incentive problem [21] in the immensely popular BitTorrent system [8].

5.4.1 A brief overview of BitTorrent

BitTorrent [8] is a decentralized file swarming system whose goal is to disseminate large files to a large number of downloaders. Rather than rely on a highly provisioned server, BitTorrent peers trade small *pieces* of a file with one another, thereby contributing to the system while gaining from it. *Bitfields* represent which pieces of a file a peer has. Peers trade bitfields in order to gain one another's interest; a peer is *interested* in peers who have pieces that it does not. Since peers only upload to peers in whom they are interested, peers have incentive to be as interesting to as many others as possible.

5.4.2 Piece under-reporting

BitTorrent peers can sometimes have incentive to under-report what pieces they have to their neighbors, since by doing so they can limit the degree to which their neighbors find interest in one another [21]. For instance, suppose peer i has neighbors j and k, both of whom want pieces p and q from i. If i were to tell them both about both pieces, one might demand p and the other might demand q. After obtaining them, they might gain interest in one another and exchange p and q among themselves, thus *decoupling* from i. Thus, i may prefer to under-report by sending to j and k a bitfield that contains p but not q. As a result, both neighbors request and obtain p, gaining no interest in one another; only then does i reveal that he also has piece q, forcing j and k to download it from i.

Such under-reporting leads to a tragedy of the commons, since although strategic under-reporters' download times improve, the system as a whole suffers [21]. Since its recent discovery, strategic under-reporting has yet to be solved; we demonstrate how to solve it with TrInc.

5.4.3 Solving under-reporting with TrInc

We observe that *under-reporting in file swarming systems is an act of equivocation*. Using the above example, when peer i received piece q from peer ℓ, i must have sent an acknowledgment, stating to ℓ that he received the piece. However, by under-reporting q to peers j and k, i is effectively contradicting a statement he made earlier to ℓ.

Algorithm 3 Fighting equivocation in BitTorrent

Upon receipt of piece p:
1. Add p to bitfield B
2. $a_{curr} \leftarrow \texttt{Attest}(i, |B|, h(p, B))$

Upon sending piece p to neighbor j:
1. Request an attestation from j with a random nonce.
2. Do not send any piece other than p to j until j admits to having p.

Periodically, for each neighbor j:
1. Request an attestation of j's current bitfield with a random nonce.

Upon receiving an attestation request with nonce z:
1. $a_{tmp} \leftarrow \texttt{Attest}(i, |B|, z)$.
2. Reply with (a_{curr}, a_{tmp}).

Our goal is therefore to remove BitTorrent peers' ability to undetectably equivocate. We present in Algorithm 3 a TrInc-based protocol for fighting equivocation in BitTorrent. In this protocol, a peer attests to his bitfield, incrementing a trinket counter for each piece he receives. Also, peers periodically request up-to-date attestations from their neighbors, to maintain fresh state.

Because they join the swarm at different times and download at different rates, peers' counters are not synchronized. In Algorithm 3, the TrInc counter does not correspond to some system-wide "round" the protocol is in, as it would in, say, BFT machine replication. Instead, peer i's counter represents how many pieces i has downloaded. This is a natural fit for the counter, because it is a monotonically increasing number, and because the type of malicious behavior we want to prevent corresponds to pretending it is not monotonic.

Algorithm 3 demonstrates the importance of choosing the correct data to which to attest. Suppose, for instance, peers were to attest *only* to their bitfields. Clearly, when s sends an attested bitfield to neighbor n, s must include the piece n sent him, p_n, in the bitfield, otherwise n will observe an under-report. Were s to attest only to the bitfield, then s could under-report as follows, where B_{old} represents the bitfield before receiving pieces p_a, p_b, and p_c, and $\oplus$ denotes adding a piece to the bitfield:

- To a: $B_{old} \oplus p_a$
- To b and c: $B_{old} \oplus p_b \oplus p_c$

The problem arises because the data to which s is attesting does not enforce monotonicity at the semantic level we desire. Specifically, though the counter cannot decrease, it does not have to correspond to the number of

distinct pieces acknowledged, allowing a malicious participant to misstate the number of distinct pieces he has acknowledged.

In our solution, a peer attests not only to the hash of his bitfield B, but also to the most recent piece he has received, p. Neighbor n therefore expects an advance attestation including both p_n and a bitfield containing p_n. As a result, every piece must have a unique advance attestation, ensuring that s's counter must be as large as the number of pieces he has acknowledged receiving.

5.4.4 Properties of a TrInc-augmented BitTorrent

Our TrInc-based solution to equivocation in BitTorrent solves two difficult incentives-related problems. First, peers have incentive to truthfully reveal the pieces they have whenever they are asked to. TrInc removes the ability to equivocate, and step-omission failures (remaining silent) result in getting no further pieces from a neighbor. Peers can therefore obtain long-lived trades with others only by truthfully reporting their pieces.

Second, our solution adds additional security to BitTorrent's bootstrapping mechanism. In BitTorrent, peers *optimistically unchoke* new participants, sending them pieces without requiring anything in return, to introduce them into the system. BitThief [24] exploits this by pretending not to be able to make progress [35]. However, such artifice is not possible with TrInc since with it a peer cannot hide the rate at which he is downloading pieces.

Note, however, that what we propose is not a complete solution to problems with bootstrapping. Even with TrInc-enabled BitTorrent, a peer can steal a single piece from each other peer. Our goal of applying TrInc here is to ensure truthfulness in long-lived peerings, which (surprisingly) does not arise automatically.

5.5 Other applications

We see many other potential applications for TrInc. We briefly described three such apps in Section 2.1: simultaneous-turn games, electronic currency, and elections. Here, we detail several others:

Secure DNS is intended to protect the integrity of the Internet domain name system. One identified threat [6] is that a resolving name server could be compromised and forge incorrect responses. The official solution to this threat is *data origin identification* in the DNS Security Extensions (DNSSEC), which uses public-key signatures to authenticate name updates. However, this solution does not address a threat in which the compromised name server replies to a query with out-of-date data, which would still bear a valid signature. Modifying DNSSEC with TrInc could address this problem by preventing the resolving name server from equivocating about whether it has received an update. Once it acknowledges receipt to the authoritative name server, it can no longer pretend it has not received the update.

Secure Origin BGP (soBGP) [44] is intended to protect the integrity of Internet routing updates. Like DNSSEC, soBGP uses public-key signatures to authenticate updates. Also like DNSSEC, soBGP is vulnerable to a threat in which a compromised router advertises out-of-date routes, which would still bear valid signatures. TrInc could address this problem by preventing a router from equivocating about whether it has received a routing update.

Distributed hash tables (DHTs), such as Chord [37], Bamboo [33], and Kademlia [27], are vulnerable to misbehaving nodes. In particular, a node can lie about which region of the keyspace it is responsible for. As nodes join and leave the DHT, these regions of responsibility change (sometimes quite rapidly [33]) in response to reconfiguration messages. A node can equivocate about whether it has received a particular message, which may allow it to claim responsibility for a region of the keyspace it does not own. TrInc could be used to prevent this equivocation.

Version control systems, such as CVS [41] and Subversion [29] are often run on remote servers. Thus, they are vulnerable to a threat model in which the server presents different views of the repository to different clients. Although this threat could be addressed at the block-store level [22], it might be more efficient to address it at the application level, in which case TrInc could prevent this equivocation.

Distributed auctions [42] are vulnerable to cheating participants. A bidder can try to manipulate others' bids by equivocating about the value of his current bid. An auctioneer can try to manipulate the bidding by equivocating about her reserve price for a particular auction. TrInc could protect against both of these classes of cheating, by preventing both bidders and auctioneers from equivocating.

Leader election protocols [25] rely on a quorum of participants to agree on a choice of leader. For a quorum of size q, it can legitimately happen that two groups of size $q - 1$ will nominate different leaders. In this case, one participant can equivocate about which leader to nominate, causing the protocol to select two leaders concurrently. TrInc could be used to prevent this equivocation.

Digital signatures are used in many cryptographic protocols, but commonly use slow asymmetric key operations [17]. However, TrInc allows faster symmetric key operations to be used instead. To do so, a signer merely has to have his trinket attest to the hash of the message to be signed using a shared symmetric key. Since this attestation can only be generated by a party with access to the symmetric key, and since the hardware includes the ID in any attestation, no other party (except the trusted session administrator) can have generated the attestation. Thus, it functions effectively as a digital signature, verifiable by anyone whose trinket has the same symmetric key installed.

Operation		Time (msec)
Noop		6.14 ± 0.15
Attest	(asymmetric, advance > 0)	230.24 ± 0.28
	(asymmetric, advance $= 0$)	198.21 ± 0.10
	(symmetric, advance > 0)	128.95 ± 0.08
	(symmetric, advance $= 0$)	105.90 ± 0.08
Verify Symmetric Attestation		85.81 ± 0.11

Table 3: TrInc microbenchmarks on a Gemalto .NET Smartcard, with 95% confidence intervals.

Operation	Time (msec) TrInc	A2M
Noop	6.99 ± 0.01	
Append	187.60 ± 0.15	551.93 ± 154
Lookup (Successful)	0.0122 ± 0.02	304.14 ± 6.87
Lookup (TooEarly)	162.24 ± 0.08	289.68 ± 2.23
Lookup (Forgotten)	162.35 ± 0.10	350.51 ± 1.43
End	162.31 ± 0.11	294.16 ± 2.04
Truncate	187.94 ± 0.10	28.99 ± 0.02
Advance	187.81 ± 0.12	288.20 ± 11.4

Table 4: TrInc-A2M microbenchmarks, with 95% confidence intervals.

6 TrInc Implementation

The application case studies demonstrate the strong theoretical properties of TrIncs. In this section, we study the performance of TrIncs *today*. To this end, we have implemented TrInc on Gemalto .NET SmartCards [11], and present microbenchmarks that measure TrInc's performance on these widely available pieces of trusted hardware.

6.1 Microbenchmarks

Our experimental setup consists of an Intel Core 2 Duo 1.6GHz machine with 3GB of RAM, and a smartcard connected via a USB card reader. We present our microbenchmarks in Table 3, with results averaged over 1,000 runs. In addition to TrInc's API, we include a noop to essentially measure the round-trip time between PC and smartcard.

Compare the `Attest` results on the card to those on the untrusted PC, where 3-DES took 0.017 ± 0.008 msec, and RSA took 8.6 ± 0.67 msec. It is no surprise that a smartcard does not perform as well, but the difference in relative performance between symmetric and asymmetric encryption is striking. On the PC, they differ by a factor of over 500, while on the card they differ by less than a factor of 2. While using symmetric instead of asymmetric operations improves TrInc's performance, we were surprised to see it was by this small a factor.

6.2 Why so slow?

The conclusion is clear: today's trusted hardware is *slow*! Indeed, it is much slower than would be allowed by most components of a distributed system. But why is it slow, and why do current applications that use trusted hardware not suffer as a result?

We believe this is attributable to the fact that *TrInc uses trusted hardware in a fundamentally different way than that for which the hardware is currently designed.* Today's trusted hardware is designed to bootstrap software, generally performing few operations during a machine's boot cycle. Conversely, TrInc makes use of trusted hardware *during* operation, in some cases multiple times for each message sent.

We proposed several versions in §3.5.6 that we believe would be viable directions for future designs of trusted hardware to take. In the interim, a logical solution is to design protocols that limit the number of necessary attestations, but such approaches are beyond the scope of this paper. Nevertheless, our empirical results in the following section indicate that making trusted hardware more suitable for use in distributed systems *today* is a valuable area of future work.

7 Application Evaluation

We now turn to macrobenchmarks, evaluating TrInc as it applies to our three case studies: A2M, PeerReview, and BitTorrent.

7.1 TrInc-A2M

In Section 5.2, we proposed a way to build A2M using TrInc. While demonstrating TrInc's ease of use and versatility, it also allows us to compare the two trusted-component designs. To this end, we have implemented A2M in the Gemalto .NET SmartCard, and a TrInc library—run on an untrusted machine—that accesses TrInc as prescribed in Algorithm 2.

We present microbenchmark comparisons in Table 4. As expected, TrInc performs `Appends` much more quickly, as it does not require as many writes to trusted storage. Where TrInc offers vast speed improvements over A2M is in successful `Lookups`; since these do not have to be either stored in trusted hardware or attested, they are merely local operations. Interestingly, A2M improves with `Truncate`, since A2M simply increases the log's low counter and postpones the attestation of the operation until a lookup that needs to return FORGOTTEN. TrInc amortizes this cost, in the expectation that there will be more FORGOTTEN lookups than truncations.

These results demonstrate that TrInc performs better on *today's* trusted hardware. As trusted components improve, particularly in terms of memory writes and cryptographic operations, it is likely that A2M and TrInc will perform comparably well. However, the slowness of today's trusted hardware brings to light the difference in complexity between A2M and TrInc. We believe TrInc's relative simplicity makes it a more suitable candidate even with future designs of trusted hardware.

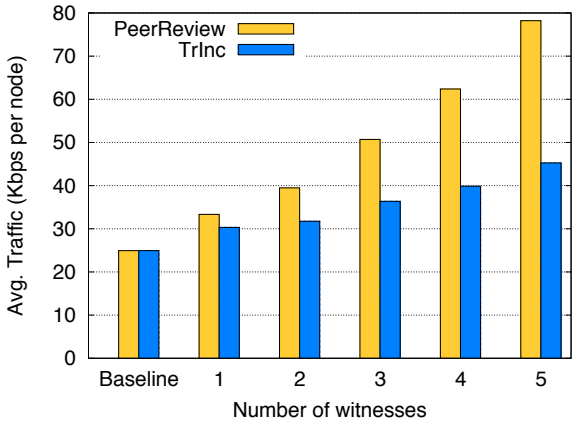

Figure 3: Reduction in PeerReview's message overhead due to TrInc.

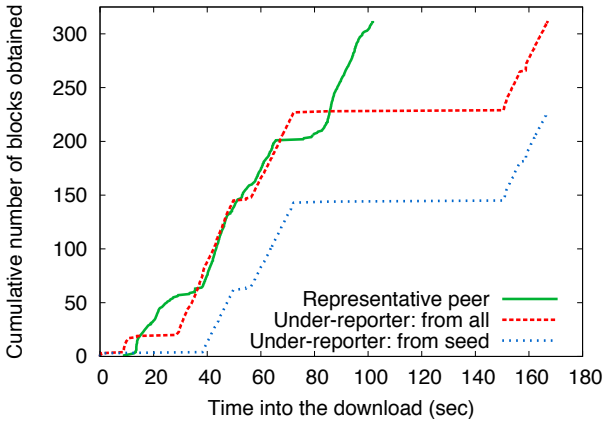

Figure 4: Rate of progress for various BitTorrent clients when TrInc is used.

7.2 TrInc-PeerReview

In Section 5.3, we demonstrated how including TrInc into the design of an accountability system such as PeerReview can decrease the amount of communication required between participants. This represents one of the fundamental strengths of including a small, trusted component into an otherwise untrusted system.

Applying TrInc to PeerReview removes the requirement for a peer p to communicate with the witness set of any other peer q, unless, of course, p happens in q's witness set. Using data from the original PeerReview study [13], we demonstrate in Figure 3 the extent to which TrInc reduces PeerReview's communication overhead. TrInc effectively removes the $O(W^2)$ witness-set-to-witness-set communication, for reasons described in Section 5.3. As a result, the amount of additional communication overhead scales linearly rather than quadratically with the size of the witness sets.

7.3 TrInc-BitTorrent

To evaluate our TrInc-based solution for BitTorrent, we simulated using a "gold-standard" trinket in the Azureus BitTorrent client. To do so, we modified BitTorrent's Have messages to include attestations to counters. We observed that Have messages, originally intended simply to inform others when a peer receives a piece, come frequently enough in practice to also satisfy peers' continual need for fresh attestations.

We modified the BitTorrent code to recognize these new messages, and to cut off peers thereby discovered to be under-reporting. However, we never have the seeder punish a peer in this way. It seems reasonable to have such a forgiving seeder since otherwise peers who suffer failures—for example, from a corrupted disk—could never request blocks after they have attested to them.

We ran our experiments on a local cluster consisting of 23 leechers, each with upload bandwidth capped at 50Kbps, and one seeder, with upload bandwidth capped at 80Kbps. We chose one host to act as a strategic piece revealer using an algorithm from a prior study [21]. We chose this host arbitrarily since, on the local cluster, we found them to be virtually indistinguishable in terms of performance.

Our experiments demonstrated a clear loss in performance from under-reporting. In a representative run, the under-reporting peer took 27% longer to download the file than the other peers did on average, and 33% longer than the median.

The under-reporter's download times would have been much worse if not for the forgiving seeder. We show in Figure 4 the total number of blocks the under-reporter received over time, compared to the number of blocks he received from the seeder. We plot a representative, truthful peer from the swarm as a point of comparison. Because other peers refused to send to the under-reporter until he revealed all the pieces in his possession, the seeder became the under-reporter's only remaining option. Indeed, the under-reporting peer obtained more pieces (73%) from the seeder than any other peer in the swarm (11% on average, 6% median).

These results indicate the power of applying a small amount of trust, and small attestations piggybacked on existing protocol messages, to a large-scale decentralized system.

8 Conclusions

In this paper, we presented TrInc, a simple yet powerful abstraction for improving security in distributed systems. TrInc is a trusted hardware module that holds a non-decreasing counter and a hidden cryptographic key. This combination, along with the computational machinery to support it, yields an abstraction that significantly improves various aspects of security in distributed systems.

TrInc was inspired by the seminal work of A2M, which introduced the idea of a trusted log for improv-

ing system security. Relative to A2M, TrInc has a significantly simpler abstraction: a counter instead of a log. We have also demonstrated a wider range of applications for, and benefits from, a trusted module than previously shown.

We have implemented TrInc on real, currently available trusted hardware. We have performed three detailed case studies of TrInc as applied to different distributed protocols. Our results show that this abstraction is easy to deploy, powerful, and versatile.

Acknowledgments

We would like to thank Josh Benaloh, Paul England, Sandro Forin, Atul Singh, Talha Bin Tariq, and Gideon Yuval for helpful discussions and assistance in evaluating TrInc on real trusted hardware. We also thank Adam Bender, Stefan Saroiu, the anonymous reviewers, and our shepherd, Greg Minshall, for their helpful comments on improving the presentation of this paper. Dave Levin was supported in part by a Microsoft Live Labs fellowship.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 59–74, 2005.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 1–14, 2002.

[3] N. E. Baughman and B. N. Levine. Cheat-proof playout for centralized and distributed online games. In *Proc. Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, pp. 104–113, 2001.

[4] A. Blanc, Y.-K. Liu, and A. Vahdat. Designing incentives for peer-to-peer routing. In *Proc. NetEcon*, 2004.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[6] R. Chandramouli and S. Rose. Secure domain name system (DNS) deployment guide. *Special Publication 800-81, NIST*, 2006.

[7] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 189–204, 2007.

[8] B. Cohen. Incentives build robustness in BitTorrent. In *P2PEcon*, 2003.

[9] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 177–190, 2006.

[10] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *NOSSDAV*, 2004.

[11] Gemalto. http://www.gemalto.com/.

[12] R. Gupta and A. K. Somani. CompuP2P: An architecture for sharing of compute power in peer-to-peer networks with selfish nodes. In *Proc. NetEcon*, 2004.

[13] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 175–188, 2007.

[14] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 175–188, 2008.

[15] D. Hughes, G. Coulson, and J. Walkerdine. Free riding on Gnutella revisited: The bell tolls? *IEEE Distributed Systems Online*, 6(6):1–18, 2005.

[16] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pp. 29–42, 2003.

[17] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. CRC Press, 2007.

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 45–58, 2007.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[20] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[21] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee. BitTorrent is an auction: Analyzing and improving BitTorrent's incentives. In *Proc. SIGCOMM Conference on Data Communication*, pp. 243–254, 2008.

[22] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 121–136, 2004.

[23] Q. Lian, Y. Peng, M. Yang, Z. Zhang, Y. Dai, and X. Li. Robust incentives via multi-level tit-for-tat. In *Proc. Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[24] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in BitTorrent is cheap. In *Proc. Workshop on Hot Topics in Networks (HotNets)*, pp. 85–90, 2006.

[25] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[26] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 135–150, 2000.

[27] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. Workshop on Peer-to-Peer Systems (IPTPS)*, pp. 109–114, 2002.

[28] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 31–44, 2002.

[29] W. Nagel. *Subversion Version Control: Using the Subversion Version Control System in Development Projects*. Prentice Hall, 2005.

[30] T.-W. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. In *Proc. NetEcon*, 2004.

[31] A. Perrig, S. Smith, D. Song, and J. D. Tygar. SAM: A flexible and secure auction architecture using trusted hardware. In *ICEC*, 2001.

[32] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 1–14, 2007.

[33] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. USENIX Annual Technical Conference*, pp. 127–140, 2004.

[34] 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 *Proc. Workshop on Scalable Trusted Computing (STC)*, 2006.

[35] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free-riding in BitTorrent networks with the large view exploit. In *Proc. Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[36] M. Sirivianos, J. H. Park, X. Yang, and S. Jarecki. Dandelion: Cooperative content distribution with robust incentives. In *Proc. USENIX Annual Technical Conference*, pp. 157–170, 2007.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. SIGCOMM Conference on Data Communication*, pp. 149–160, 2001.

[38] Trusted Computing Group. Trusted Platform Module Specifications. Online at https://www.trustedcomputinggroup.org/specs/TPM/.

[39] M. van Dijk, L. F. G. Sarmenta, C. W. O'Donnell, and S. Devadas. Proof of freshness: How to efficiently use an online single secure clock to secure shared untrusted memory. Technical report, 2006.

[40] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 59–72, 2007.

[41] J. Vesperman. *Essential CVS, 2nd Edition*. O'Reilly, 2006.

[42] J. M. Vidal. Multiagent coordination using a distributed combinatorial auction. In *AAAI Workshop on Auction Mechanisms for Robot Coordination*, 2006.

[43] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A Secure Economic Framework for P2P Resource Sharing. In *Proc. NetEcon*, 2003.

[44] R. White. Securing BGP through Secure Origin BGP. *Internet Protocol Journal*, 6(3), 2003.

[45] S. Zhong, J. Chen, and Y. R. Yang. Sprite: A simple, cheat-proof, credit-based system for mobile ad-hoc networks. In *Proc. Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, 2003.