

Small Trusted Primitives for Dependable Systems*

Petros Maniatis
Intel Labs Berkeley

Byung-Gon Chun
Intel Labs Berkeley

ABSTRACT

Secure, fault-tolerant distributed systems are difficult to build, to validate, and to operate. Conservative design for such systems dictates that their security and fault tolerance depend on a very small number of assumptions taken on faith; such assumptions are typically called the “trusted computing base” (TCB) of a system. However, a rich trade-off exists between larger TCBs and more secure, more fault-tolerant, or more efficient systems. In our recent work, we have explored this trade-off by defining “small,” generic trusted primitives—for example, an attested, monotonically sequenced FIFO buffer of a few hundred machine words guaranteed to hold appended words until eviction—and showing how such primitives can improve the performance, fault tolerance, and security of systems using them. In this article, we review our efforts in generating simple trusted primitives such as an attested circular buffer (called Attested Append-only Memory), and an attested human activity detector. We describe the benefits of using these primitives to increase the fault-tolerance of replicated systems and archival storage, and to improve the security of email SPAM and click-fraud prevention systems. Finally, we share some lessons we have learned from this endeavor.

Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed Systems; C.4 [Performance of Systems]: Fault tolerance; D.4.5 [Operating Systems]: Fault-tolerance; D.4.6 [Security and Protection]: Access controls; K.4.4 [Computers and Society]: Electronic Commerce

General Terms

Algorithms, Design, Reliability, Security

Keywords

Trusted primitives, dependability, activity attestation

1. INTRODUCTION

*This article appears in the SIGOPS Operating Systems Review, Volume 45, Number 1, January 2011. It surveys, abstracts, and discusses earlier work on A2M [31], Bonafide [32], and NAB [38]. That work was conducted with our collaborators from UC Berkeley, MIT, and Intel Labs, who share all credit for the technical contributions presented in this retrospective.

Large-scale systems are complex amalgams of many pieces of hardware and software. Quite frequently, those pieces come from many sources, have different purposes, and are under the control of different humans and organizations. Consider the example of a banking application; a bank customer obtains service via her home computer or smart-phone, the service itself operates on hardware and software primarily owned and operated by the bank, while communication between the two is conducted via a third-party network.

Establishing the dependability of an outcome—a computation result, a transaction, etc.—from such a system is tough even in the simplest cases. This dependability is contingent on the correctness of each component (does the banking application implement the algorithm intended by the bank?), on the ability of each component to weather faults (do banking transactions recover from temporary network disconnection, soft errors, or attacks?), and on the trustworthiness of the organizations contributing each component (does the bank intentionally skim off the top of each transaction?).

A typical dependability assessment is, at best, a guarantee conditioned on assumptions about these factors. Unfortunately, those assumptions are infrequently—if ever—justified. Implementations of algorithms in deployed systems are often buggy, systems often crumble under unanticipated conditions and attacks, and organizations or their insiders often act against their publicly stated mission and guarantees. Therefore, trusting a system (operating system, application, hardware, intervening network) in its *entirety*, although seductive in its simplicity, is completely unrealistic.

One way to improve the situation is to put the most critical functionality of a system in a special software or hardware component, built and deployed carefully enough to justify trust in its correctness, leaving less critical functionality and data to untrusted parts of the system. For example, the Trusted Platform Module (TPM) [12] is designed to ensure that the loaded software within a commodity computer is reported to external verifiers correctly, even in the face of possibly malicious operators with physical access to that computer or other software attacks. While a TPM chip is not sufficient to build a trustworthy system—after all, correctly booting a program will not magically make that program bug-free—it nevertheless improves the odds that a system works as expected: a whole class of problems no longer wreak havoc. Informally one might view this as a “re-allocation of the dependability budget” across the different

hardware and software components of a system.

In this article, we describe some of our research efforts towards three computational and storage primitives designed to improve the dependability of a broad range of applications. Our common goals for all three primitives were to make them easy to implement, cost-effective, and simple to interface with for broader applicability. Specifically, we summarize *Attested Append-Only Memory* (A2M) [31] for removing equivocation from replicated systems, *Moded Attested Storage* (MAS) [32] for weathering fault spikes in long-term storage systems, and *Not-A-Bot* (NAB) [38] for combating bot-attacks via human-activity detection. A2M enables Byzantine-fault-tolerant (BFT) systems to reduce the required replication factor without reducing their fault tolerance, or in some cases to improve the fault tolerance guarantees offered without extra overhead. MAS allows long-term storage systems to tolerate temporary fault spikes that usually doom more traditional replicated systems to failure forever after the fault spike. And NAB allows effective policy that prioritizes traffic (email, web requests, etc.) that is likely to have originated with a human user, over traffic that appears to come from automation; in the process, it dramatically improves spam filtering, denial-of-service defenses, and click-fraud prevention.

In what follows, we motivate trust in small primitives, describe our assumptions and fault model, and describe the three trusted primitives in more detail. We then present implementation choices for such primitives that trade-off performance with dependability. After reviewing related work, we conclude with lessons gathered during our research efforts, and our future research direction in this space.

2. WHY TRUST ANYTHING?

During the design of dependable systems, a natural question frequently arises: why should one place greater trust in one component than in another, or ideally than in the entire system? We seek to answer this question in this section.

Different components of nodes in a complex system exhibit different levels of fault tolerance, because each is built, validated, deployed, and maintained differently. One source of differentiation comes from *different assurance practices*. Hardware microprocessor designs undergo extensive formal verification before production. They tend to exhibit fewer bugs and security vulnerabilities in their implementation than typical software systems. Even in the software world, formally verified components can rigorously prove their correctness guarantees under specific execution models. As a result, these components are better protected from many runtime bugs and vulnerabilities [69, 73]. Formally verified software can be leveraged with some success and performance cost via the use of strongly typed languages such as Java and C#, which are touted as safer environments for building robust systems: they offer a formal guarantee that, as long as the compiler and execution runtime implement the language semantics correctly, no application will be vulnerable to some of the typical system plagues like buffer overflows. Similar guarantees are offered by language-based type-safe operating systems such as Singularity [42]. Of course, one may need to scrutinize those compilers and runtimes themselves, since their correctness is essential to

anything dependent on them [79].

A second source of differentiation comes from *care in the deployment of a system*: tight physical access controls, proactive hardware and software replacement, responsive system administration, well-designed firewalls and intrusion detection mechanisms contribute to keeping out the threats that can exploit any vulnerabilities present in the physical and logical interfaces of a system component. For example, a software component that is vulnerable to a particular exploit borne over SSH traffic can be shielded from that exploit if the firewalls between the Internet and that component drop all SSH packets before they reach it [86], or if it only communicates with other trusted components over a private network [33, 76].

A third source of differentiation comes from the *rolling procurement* characteristics of software and hardware technologies. A service that needs to remain dependable must weather change: hardware becomes obsolete, operating systems evolve, communication standards grow, and best-known cryptographic methods are broken and replaced by their successors. For example, a trusted logical component assumed to never fail would require the expensive proactive replacement of the cryptographic libraries or the trusted hardware platform used to implement it, as new cryptanalysis techniques become possible, faster hardware is introduced, and new processes for protecting processor packages from physical or electrical tampering become available. In contrast, a less trusted component could afford to trail the state of the art and use replication or other techniques to mask faults, only migrating to new software and hardware less frequently and potentially at a lower cost.

Finally, fault differentiation can come from *limited exposure*. Many high-assurance systems such as certification authorities keep their sensitive components (e.g., their signing keys) mostly or wholly off-line, limiting attack opportunities. Services that have limited or potentially batched updates but can be mostly read-only (or indeed off-line with on-line, untrusted proxies [36, 47, 51]), can be protected quite effectively in this fashion.

Interestingly, there are non-trivial dependencies among all these sources of differentiation. For example, a proven-correct system that is operated by a trustworthy organization is strictly more reliable than the same system operated by an unreliable organization. As goes the usual secure systems' truism, a complex system is as secure as its weakest link. This simple observation allows us to argue systems can be constructed by components with different dependability characteristics. For each such component, the expectation of greater dependability implies restrictions on its capabilities, for instance because it is still expensive to formally verify large, complex components [89]. Similarly, a component that protects its dependability by remaining mostly off-line can realistically be responsible only for functionality that can be performed infrequently and in batches.

3. SETUP

We use standard assumptions about the network model and about cryptography. In the network, packet drops, reorderings, and duplications can occur but retransmissions of a

message eventually deliver it. However, though finite upper bounds exist for message delivery and operation execution times, those bounds are not known to protocol entities. A faulty node cannot violate intractability assumptions about standard cryptography. Therefore, the adversary cannot produce pre-images or collisions for cryptographic hash functions¹ or forge previously unseen signatures for private signing keys he does not possess. For long-term uses of cryptography, we make a “rolling procurement” assumption, whereby cryptographic tools are replaced with up-to-date technologies, algorithms, and key sizes well before their compromise is even feasible, let alone practical. For a managed system, this is a reasonable and plausible approach.

In this paper, we consider fault models that depend on the cause of the node’s misbehavior. In particular, we distinguish between two cases: (i) the node’s owner is well-intentioned but unaware the node’s software has been compromised by a third-party (*faulty application model*), and (ii) the node’s behavior is Byzantine because of a malicious owner (*faulty operator model*). The nature of the trusted computing base is quite different in the two cases. In the first model, the trusted computing base is set up by the service owner; for instance, a bank owns all nodes and ensures, through physical security and other means, that only its nodes can provide the service. Our concern here is to combat software attacks such as worms and viruses against those centrally administered nodes. In the second model, we do not trust owners but trust a third party (e.g., a special service provider or a trusted hardware manufacturer) to set up the trusted computing base; for instance, a malicious storage server can manipulate all aspects of its node except what lies within the trusted device, which is the purview of the device provider.

For conciseness, throughout the paper we use the authentication notation of Yin et al. [90], according to which we denote by $\langle X \rangle_{S,D,k}$ an authentication certificate that any node in a set D can regard as proof that k distinct nodes in S said X . For example, a traditional digital signature on X from p that is verifiable by the entire replica population R would be $\langle X \rangle_{p,R,1}$, two signatures from p and q put together would be $\langle X \rangle_{\{p,q\},R,2}$, and a MAC from p to q with a shared key would be $\langle X \rangle_{p,q,1}$. As a convention, we use p to denote the singleton set $\{p\}$, and ∞ as shorthand for the universal set of all principals. When we use this notation to describe collective certificates made up of individual signatures, as for the second example above, we usually remove any signer identification from the collective certificate format: for example, the certificate $\langle X \rangle_{\{p,q\},R,2}$ above could correspond to the individually signed messages $\langle p, X \rangle_{p,R,1}$ and $\langle q, X \rangle_{q,R,1}$.

We use $h()$ to denote a one-way collision-resistant hash function such as SHA-256, and \parallel to denote the bit-string concatenation operator.

4. EQUIVOCATION PROTECTION: A2M

¹A one-way—or pre-image resistant—hash function h is one for which there is no polynomial-time algorithm that, given α , can find a previously unknown β such that $\alpha = h(\beta)$. A collision-resistant hash function h is one for which there is no polynomial-time algorithm that can find two values α and β for which $h(\alpha) = h(\beta)$.

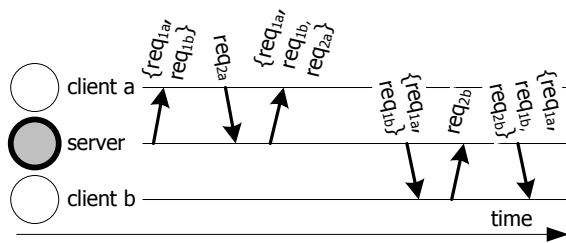


Figure 1: A forking attack example of two clients and one malicious server. The server convinces clients a and b of different system states.

The first primitive we consider targets the problem of equivocation in untrusted services. In particular, we consider client-server systems where a service is accessed and shared by multiple clients connected over a public network. The service can be implemented as a single server (e.g., a file server) or multiple servers (e.g., replicated state machines). Clients request *authenticated* operations from the service, the service executes those operations—which may change the service state—and returns responses to the requesting clients.

The fault-tolerance properties of such systems can be divided into *safety* guarantees, properties that must be true at all times, and *liveness* guarantees, properties that must become true within finite time from all execution states of the system. For replicated services the target safety guarantee is usually *linearizability* [41]: completed client requests appear to have been processed in a single, totally ordered, serial schedule that is consistent with the order in which clients submitted their requests and received their responses. The corresponding liveness guarantee is that a correct client’s request is eventually processed. It is well established that if servers have no trusted components, then no replicated system can provide these safety and liveness guarantees when more than a third of its replicas are faulty [50].

4.1 The Problem: Equivocation

In deterministic, replicated systems that aim to guarantee linearizability, lying is bad enough, but lying in different ways to different people is even worse. The “prototype” problem of agreeing on a single system execution, known as the “Byzantine generals problem,” has been demonstrated unsolvable in a population of three parties when one is faulty [50], precisely because of equivocation. Beyond agreement, especially when there is a single server to contend with, equivocation can wreak just as much havoc. Next, we present two detailed examples of equivocation attacks against single-server and replicated systems, to motivate our focus on eliminating equivocation through trusted primitives.

4.1.1 Servers Equivocating to Clients

We consider a log-structured storage server shared by multiple clients as an illustrative example. For example, in a straw-man design for SUNDR [54], to request an operation, a client first acquires a lock at the server and downloads the entire operation log, a time-ordered collection of signed client operations. The client checks whether the log is correct by verifying the signatures and by checking that the log contains all of its own operations in order; it then creates

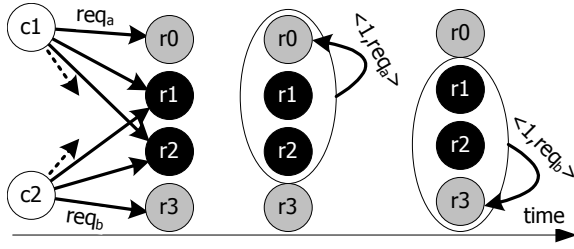


Figure 2: An example that shows the violation of linearizability in PBFT when two replicas are faulty out of four replicas. Black circles are faulty replicas (one of them is the primary), gray circles are correct replicas, and white circles are clients. When two clients c_1 and c_2 submit requests req_a and req_b to the replicas at roughly the same time, but only manage to reach one correct replica each, the two faulty replicas can convince the two correct replicas to assign the same sequence number to different requests.

what must be the server’s current state by starting with an initial state and then applying the logged operations in order, as a correct server would have in a linearized system. The client executes its operation based on the created state, thus finding out the result of this operation. The client then appends its signed operation to the end of the log, sends the updated log back to the server, and releases the lock.

A faulty server can mount a forking attack [54] by concealing operations, which causes the system’s state to diverge into multiple possibilities for different clients. Suppose two clients access a server as shown in Figure 1. Client a performs req_{1a} , client b performs req_{1b} , and client a performs req_{2a} . The latest state of the server becomes $\{req_{1a}, req_{1b}, req_{2a}\}$ as far as client a is concerned. Now, client b retrieves the log of the server to perform a new operation req_{2b} . The faulty server drops req_{2a} off the tail of the log, only returning $\{req_{1a}, req_{1b}\}$. Client b executes its operation and has the log state $\{req_{1a}, req_{1b}, req_{2b}\}$. The system state is now forked with regards to these two clients. The cause of the problem is the ability of the faulty server to misrepresent its operation log to the two clients, equivocating on what its state is according to who is asking.

4.1.2 Servers Equivocating to Servers

To demonstrate equivocation problems among servers, we consider BFT replicated state machines. In particular, we choose Practical Byzantine Fault Tolerance (PBFT) [28] for its profound impact on the systems literature. For the purposes of this illustration, a PBFT client is satisfied with a result to its request if it receives at least $\lfloor \frac{N-1}{3} \rfloor + 1$ replies from distinct replicas out of the N total replicas, all with a matching result; a PBFT replica can commit a request to its local state as long as a quorum of $2\lfloor \frac{N-1}{3} \rfloor + 1$ replicas agree on the request’s ordering in history. More detailed PBFT background can be found elsewhere [28].

Given this behavior, PBFT guarantees safety (linearizability) and liveness, as long as no more than $\lfloor \frac{N-1}{3} \rfloor$ replicas are faulty; if more than $\lfloor \frac{N-1}{3} \rfloor$ replicas are faulty, PBFT does not guarantee safety (and liveness is meaningless without safety): faulty replicas can fool non-faulty replicas to

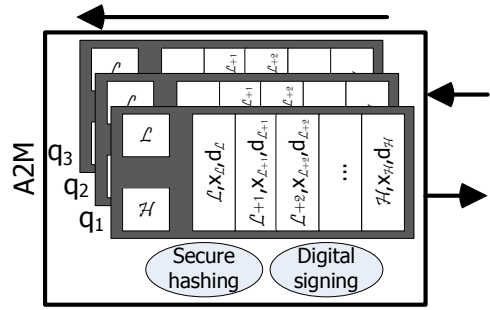


Figure 3: Structure of an *attested append-only memory* (A2M), which contains a set of distinct logs (q_i) that map sequence numbers (in the range of \mathcal{L}_i to \mathcal{H}_i) to values.

commit different request histories, and different clients may accept replies corresponding to different request histories, violating linearizability.

To illustrate, consider $N = 4$; replicas r_1 and r_2 are faulty, and non-faulty replicas r_0 and r_3 cannot temporarily communicate with each other (Figure 2). Client a sends req_a to the system. The two faulty replicas convince r_0 to commit and execute req_a first, since the three of them form a quorum of $3 = 2\lfloor \frac{N-1}{3} \rfloor + 1$. Later client b sends req_b to the system. The two faulty replicas convince r_3 to commit and execute req_b first, since r_3 never saw req_a . Faulty servers r_1 and r_2 equivocate to non-faulty servers r_0 and r_3 .

Furthermore, the ability of faulty servers to equivocate to non-faulty servers also allows the service to equivocate to clients, as in the previous section. For example, clients a and b experience two different histories through their accepted replies, thus violating linearizability: req_a is the first committed request for one, while req_b is the first request for the other. The problem arises because of the faulty replicas equivocating to clients. The faulty replicas are allowed to tell client a , with r_0 ’s help, that req_a is committed in their history at sequence number 1, and also to tell client b , with r_3 ’s help, that req_b is committed in their history at the same sequence number.

4.2 Primitive: A2M

In the previous section, we argued that the adversary’s ability to equivocate undetected—e.g., to claim to have two different histories depending on which host it is talking to—is a fundamental weapon against safety, both in single-server and replicated services. Here we describe an *attested append-only memory* (A2M), a simple attestation-based abstraction that, when trusted, can remove the ability of adversarial replicas to equivocate without detection. Using an A2M implementation within the trusted computing base, a protocol can assume that a seemingly correct host can give only a single response to every distinct protocol request—for some protocol specific definition of “distinct” request—, even when that same request is retransmitted multiple times by different clients or replicas, and even if that response is undetectably faulty. A2M essentially offers reliable services a bit-commitment scheme [68] for sequential logs, placed within the trusted computing base.

Informally, an A2M equips a host with a set of trusted, un-

deniable, ordered logs (illustrated in Figure 3). Each such log has an identifier q (unique within the same computer) and consists of a sequence of values, each annotated with (1) a log-specific sequence number that is incremented from 0 with every new value appended to the log, and (2) an incremental cryptographic digest of all log entries up to itself. Only a suffix of the log is stored in A2M, starting with the slot in the “low” position $\mathcal{L} \geq 0$ and ending with the last slot in the “high” position $\mathcal{H} \geq \mathcal{L}$.

An A2M log offers methods to **append** values, to **lookup** values within the log or to obtain the **end** of the log, as well as to **truncate** and to **advance** the log suffix stored in memory. There are no methods to replace values that have already been assigned.

- **append**(q, x) takes a value x , appends it to the log with identifier q , increments the highest assigned sequence number \mathcal{H} by 1, populates the slot at that position with x , and computes $d_{\mathcal{H}} = h(\mathcal{H} \| x \| d_{\mathcal{H}-1})$, a cumulative digest, where $d_0 = 0$. It does not cause any values to be forgotten, i.e., it does not affect \mathcal{L} ; if the log is unable to allocate storage to the new entry, the method fails.
- **lookup**(q, n, z) $\rightarrow \langle \text{LOOKUP}, q, n, z, x, w, n', d \rangle_{A2M_q, \infty, 1}$ is a method that takes log identifier q , a sequence number n and a nonce z (for freshness), and returns a LOOKUP attestation. w is the type of the attestation: if sequence number n has not been assigned yet (i.e., $n > \mathcal{H}$) then w is UNASSIGNED and $n' = \mathcal{H}$; if n was assigned once but has now been forgotten (i.e., $n < \mathcal{L}$), then w is FORGOTTEN and $n' = \mathcal{L}$; if slot n has been skipped over via the **advance** method (see below) then w is SKIPPED and n' is the sequence number of the **advance** call that caused the skip; finally, if n is a slot that was filled via **append** or **advance** (see below), then w is ASSIGNED and $n' = n$. x and d are the assigned log value and digest when w is ASSIGNED) and 0 otherwise.
- **end**(q, z) is similar to **lookup**, but returns the last entry of the given log (currently in position \mathcal{H}). Attestations from **lookup** and **end** have the same format except for the request name END in the beginning.
- **truncate**(q, n), where $n \in (\mathcal{L}, \mathcal{H}]$, forgets all log entries with sequence numbers lower than n , setting \mathcal{L} to n . All subsequent **lookup** requests for entries below n will be henceforth of type $w = \text{FORGOTTEN}$.
- **advance**(q, n, d, x) allows log q to skip ahead by multiple sequence numbers. It takes a sequence number $n > \mathcal{H}$, a digest d , and a value x . It operates similarly to **append**, but instead of using $d_{\mathcal{H}-1}$ in the digest computation, it uses the given d ; skipped sequence numbers are reported as SKIPPED in **lookups**. Any subsequent **lookup**(q, n'', z) request for a sequence number n'' that was skipped by this **advance** will return an attestation of the form $\langle \text{LOOKUP}, q, n'', z, x, \text{SKIPPED}, n', d \rangle_{A2M_q, \infty, 1}$, which contains information about the **advance** method that caused the skip, until the slot is finally FORGOTTEN.

4.3 A2M Usage

Equipped with A2M in its trusted computing base, a reliable service can mitigate the effects of Byzantine faults in its untrusted components, by relying on some small fallback information about individual operations or histories of operations that cannot be tampered with.

During setup, the untrusted component (e.g., a server) must make known to all possible verifiers (e.g., clients or other servers) the authentication keys for its A2M module and the identifier of the A2M log used for each distinct purpose. As far as a verifier is concerned, the A2M authentication key and log identifier are part of the untrusted component’s identity. Therefore, a particular A2M-enabled component is allowed to use only its associated A2M.

An untrusted component \mathcal{C} can commit individual data items or operations by **appending** them to an A2M log. For example, to prove that it has committed to a data item D , the component can execute **append**($q, h(D)$). The data item is hashed before appending to facilitate A2M implementations in which every log slot has a fixed length.

An interested verifier can establish that the data item is, indeed, in the untrusted component’s committed state by demanding the attestation $\langle \text{LOOKUP}, q, n, z, x, \text{ASSIGNED}, n, d \rangle_{A2M_{\mathcal{C}}, \infty, 1}$ for some sequence number n and nonce z , where $x = h(D)$. This conclusively establishes that the untrusted component indeed put the data item D somewhere into its committed log². The sequence number n can be further constrained (e.g., it can be associated with individual protocol steps) to ensure that the untrusted component only commits a single data item for that protocol step; in this sense, multiple verifiers who are mutually disconnected can be assured that the component cannot equivocate on the contents of its n -th slot. To ensure that the untrusted component has a particular data item as the last element in its log, a verifier can provide the untrusted component with a random nonce z and demand and END attestation instead.

The untrusted component is not bound to committing to individual data items in sequential log slots; it can use **advance** to skip some sequence numbers, e.g., for periods of disconnection or unavailability. Invocation of **advance** does not “unprove” things that the A2M has attested to before. It merely gives up the ability to attest to a real value for the skipped sequence numbers.

When interested in entire histories of data items (e.g., request logs), verifiers can make use of not only the committed data item itself, but also the cumulative digest d . Thanks to the collision-resistant properties of the hash function used, there is a single sequence of data items appended to log q for which the cumulative digest is d . Therefore, equal digests in LOOKUP attestations from two untrusted servers imply the two servers have committed to the same history thus far.

To revisit the scenario of a storage server that maintains a log for committed client requests but maliciously drops some off the end when talking to a victim client (Section 4.1.1), consider forcing the server to maintain that log in A2M. Client b can demand a fresh END attestation from the server’s A2M log, along with the history itself, and ensure that the included digest is indeed the cumulative digest of the his-

²We use $A2M_p$ to denote the authentication principal corresponding to host p ’s A2M module. Trusting A2M means that p cannot forge authenticators by $A2M_p$ without A2M’s cooperation, and that even then, it can only coerce A2M to generate such authenticators as per the A2M interface.

tory; this guarantees to b that the server has not omitted any requests from the end of its committed log in its response, eliminating this particular problem. Similarly, to revisit the replicated scenario in which malicious replicas profess to different committed requests to different non-faulty replicas, convincing them to commit divergent requests (Section 4.1.2), consider requiring replicas to place such messages into an A2M message log before transmitting them. Now a non-faulty replica, before it allows itself to be convinced by another replica’s message, ensures that the message is attested in a LOOKUP attestation drawn from the message sender’s A2M message log. In this way, the faulty replica cannot equivocate to two different non-faulty replicas.

4.4 Applicability: A2M-enabled Replication

A full treatment of how A2M can improve fault-tolerant systems is out of the scope of this article and is covered elsewhere [31]. Here, we summarize how we have used A2M.

A2M-PBFT-E is an A2M variant of Castro and Liskov’s Practical Byzantine Fault Tolerance (PBFT) protocol [26]. Similar to PBFT, A2M-PBFT-E guarantees safety and liveness with up to $\lfloor \frac{N-1}{3} \rfloor$ faulty replicas out of N total; however, whereas PBFT offers no guarantees whatsoever when this upper bound of faulty replicas is crossed, A2M-PBFT-E can still guarantee safety without liveness when the number of faulty replicas is more than $\lfloor \frac{N-1}{3} \rfloor$ but no more than $2\lfloor \frac{N-1}{3} \rfloor$. This is an important advantage for applications, such as high-volume banking, in which correctness (captured by safety) under heavy faults is desirable, even if it is not accompanied by availability (captured by liveness).

A2M-PBFT-EA is an extension of PBFT that can guarantee both safety and liveness with up to $\lfloor \frac{N-1}{2} \rfloor$ replica faults: whereas PBFT needs a three-fold replication to tolerate a given number of faults, A2M-PBFT-EA needs only two-fold replication. The additional complexity of A2M-PBFT-EA may be justifiable in applications that require both low replication *and* high fault tolerance, as might be the case for critical applications with very high replication costs, such as dependable software for space missions.

Finally, we also demonstrated the use of A2M with A2M-Storage, a single-server storage service like SUNDR [54], and A2M-Q/U, an A2M-enabled quorum-based replicated state machine protocol based on Q/U [17], in both cases increasing fault tolerance and/or decreasing the replication factor.

The cost of A2M is dependent on how it is implemented (Section 7). To illustrate, an NFS server running an A2M-enabled fault-tolerant backing store is only 4% slower compared to that of NFS on top of a traditional PBFT-replicated implementation, or about 24% compared to NFS on top of an unreplicated, untrusted server.

5. RATCHETED STORAGE: MAS

The second primitive we consider extends the problem of Byzantine-fault-tolerant storage to long-term applications. The Moded Attested Storage (MAS) continues where A2M left off, making it possible to build storage systems that survive temporary violations of fault assumptions, which incapacitate typical fault-tolerant systems.

5.1 The Problem: Fault-Threshold Violations

Current fault-tolerant replicated service designs are often unsuitable for long-term applications, such as archival storage for digital artifacts, which is gaining importance for business [71], regulatory [15,16], and cultural [59] reasons. This unsuitability results from the typical fault assumptions on which the correctness of such systems is conditioned (e.g., the 1/3-rd fault threshold of replicated BFT systems).

In typical, “near-term” applications, such a uniform-threshold-based fault assumption can be reasonable and achievable. For example, one can argue that in a well-maintained population of diverse, high-assurance replica servers, by the time a third of the population is broken into or just grows faulty, the operators of faulty replicas can repair them. Thus, the repair reduces the number of faulty replicas, averts a threshold breach, and thereby keeps the system’s fault assumption inviolate.

Unfortunately, this reasoning falls apart for applications and deployments with a long-term horizon, say many decades. Whereas a population of replica servers can be plausibly “well-maintained” enough for a few years, it is difficult to protect perfectly from momentary threshold breaches over the long haul. Even improbable correlated faults become probable given enough time [20]; what is more, changes in human operators, organizational priorities, and fluctuating budgets over time make the probability of at least temporary threshold breach uncomfortably high. Once that threshold is breached, for however brief a period, the system’s fault assumption is violated, and correctness can no longer be guaranteed at any point in time thereafter.

To illustrate, let us revisit the equivocation example given in Section 4.1.2 (see Figure 2). We established before that equivocating faulty replicas r_1 and r_2 can cause the states of correct replicas r_0 and r_3 to diverge. Because only $2(= f + 1)$ matching replies are required to convince a client of a result, even if the fault assumption is again met because one faulty replica is repaired, the remaining faulty replica will always be able to corroborate r_0 ’s view of the world to some clients and r_3 ’s view of the world to other clients, keeping up the charade indefinitely. Crash-fault tolerant replicated state machines based on the Paxos [49] protocol do not deal with Byzantine faults explicitly (i.e., assume a Byzantine-fault threshold of 0) and they can have similar problems if a Byzantine fault crops up rarely and briefly.

Though not general for all possible replicated state machine protocols, this illustration serves to demonstrate the common trend: once the fault assumption is violated (the same as a threshold breach in traditional BFT protocols) the system cannot offer its correctness guarantees again, even if the fault assumption is later restored.

The fault bound in the original PBFT protocol applies over the lifetime of the system, assuming that once a replica becomes faulty it does not recover. PBFT’s authors devised PBFT-PR [27], an enhanced protocol with some hardware support that attempts to repair faulty replicas. As a result, PBFT reduces the length of the *vulnerability window* of the system during which the fault bound might be breached; even though more than f faults may occur during the life-

time of the system, as long as faults are repaired frequently enough so that no more than f faults are ever simultaneously present, the system maintains its guarantees. PBFT-PR achieves this repair using *proactive recovery* [27]: a hardware watchdog on every replica periodically reboots it with a fresh software installation from a read-only medium, flushing any runtime code damage caused since the last reboot. Upon reboot, the protocol cleans up the service state before it goes back into regular operation. Now the window of vulnerability³ is the period of time between two successive, successful proactive recovery phases across the replica population, which is much shorter than the lifetime of the system. However, if the f fault bound is violated within a vulnerability window, the protocol fails once again.

5.2 Primitive: MAS in Tiered Fault Models

The intuition behind MAS is to combine equivocation protection, as offered by A2M, with the ability to restrict write access to the primitive only during “more dependable” periods of system operation. As described in Section 2, different components can have different dependability characteristics. First, a complex but formally unverified software artifact might be likely to exhibit vulnerabilities; all that stands between it and a bug is a lapse in the judgment of a human programmer. In contrast, a formally verified software artifact, especially one that is very infrequently open to updates, might take much longer to exhibit vulnerabilities: it will not exhibit bugs against which it was verified, but perhaps the assumptions under which its correctness was verified might cease to hold upon a radical technology change.

This observation leads us to a *tiered fault framework* that partitions system components into different classes, according to how critical they are and, commensurately, to how trustworthy they must be. For instance, software and hardware that changes the state of a service is more vulnerable than software and hardware that reads only the state, which is more vulnerable than components that check only service status (on or off). The first class can potentially destroy the service (wipe out all state); the second class cannot destroy the service but can leak its contents; the third class leaks only as much information as can be conveyed by whether the service is up or down. Correspondingly, the first class must be simpler, better analyzed, more carefully deployed than the second class, and both more so than the third class.

A tiered fault framework assigns a separate *fault tier* to each component class. Like more traditional models, the fault assumption within each tier is threshold-based, but the actual threshold differs from tier to tier. For instance, the fault assumption for the population of write-operation components may be a 1/3 threshold as with typical BFT systems, whereas the fault threshold for the population of read-operation components may be higher. There is no magic in this formulation: each fault tier is itself subject to a fault threshold. However, this multi-tier approach enables us to structure a system so as to operate longer without violating its overall fault assumptions.

At the root of trust lies MAS, a device akin to A2M that

³The window of vulnerability varies depending on system conditions. For example, if some replicas’ state is corrupted, the window becomes large.

also contains a time source (this can be a regular, monotonic, crystal-based clock source with an upper bound on drift, or an external unconditionally trusted time source received by the device such as GPS). MAS also contains a hardware watchdog, which uses the time source to trigger proactive recovery periodically, by causing the host to reboot from read-only media. Finally, the hardware watchdog sets the device *mode* (a bit in our simple implementation). The mode corresponds to the fault tier under which the system currently operates; the device mode can change only from more vulnerable class (higher tier) to less vulnerable class (lower tier): in our example, from state update mode, to state read mode, etc. MAS’s mode is reset to the highest tier *only* when the watchdog is triggered. The MAS mode is akin in spirit to the ratcheting layers of the IBM 4758 class of secure co-processors [34].

Our specific MAS design contains a single mode bit, an externally unreadable time source, and a set of storage slots, each of which is identified by an identifier q . The write interface to a MAS is `Store(q, v)` where v is a value; this stores v to the slot with identifier q . This interface allows requests only when the mode bit indicates a U phase is ongoing. The read interface of MAS allows access all the time. It allows the attested, fresh retrieval of any slot; a `Lookup(q, z)`, where q is a slot identifier and z is a nonce used for freshness, returns `(LOOKUP, q, v, z, t, m) $_{i'}$` , where v is the value currently occupying the slot with identifier q of the MAS, t is the internal time in the device, m is the current mode bit, and i' is the hardware device principal. If the slot is empty, then $v = \text{EMPTY}$ in the returned attestation.

Note here that a minimal extension of A2M would implement MAS: it is trivial to extend the A2M interface with a watchdog and mode, by including the current mode to all A2M attestations, and using A2M’s `advance` and `end` to implement MAS’s `Store` and `Lookup` respectively. The resulting primitive would be of almost the exact same complexity as A2M, with the addition of a time source component.

5.3 Applicability: Bonafide

We have demonstrated the uses of a MAS and a tiered fault model in the context of Bonafide, an authentic long-term key-value store. Such a facility can be useful, for instance, as a directory for finding sensitive data given a human-memorable name. One example is a directory for self-certifying names of stored files given a file’s human-friendly name. Such a service can close the loop for previously proposed reliable long-term archival services such as Glacier [40] and OceanStore [48], which can withstand Byzantine attacks only as long as a client holds a self-certifying name for a data item.

Bonafide is a replicated service running on $3f + 1$ replicas. The replicas operate in alternating synchronous phases of two types: a *service phase* (S phase) and a subsequent *state-update phase* (U phase). During the i -th S phase, `Get` requests can query the service state committed (i.e., fetch bindings that were added) up to the $(i - 1)$ -st U phase. `Add` requests are buffered and executed after the end of the i -th S phase, during the i -th U phase. That is, service state changes occur in batch *only* during the U phase.

Fault bound	Component	When	How used
0	Watchdog	Periodic	Invoked
	MAS	S phase	Read
		U phase	Written/Read
1/3 Byz. ⁴	Update	U phase	Replicated store Serve ADDS
None	Service	S phase	Serve GETS Buffer ADDS Audit and repair

Table 1: The components in Bonafide and their associated fault tiers.

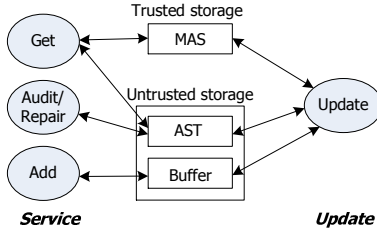


Figure 4: A Bonafide node contains a MAS that stores the AST root digest, a buffer to hold Add requests temporarily, and an AST that maintains committed bindings. The get, add, audit/repair processes run during the S phase, and the update process runs during the U phase. The arrows show what state the processes access.

The system maps to the tiered fault model by assigning to the lowest, most error-prone fault tier the *service process* (during the S phase), a mechanism for responding to the clients’ read-only requests (e.g., looking up existing key-value bindings) and for buffering—but not executing—new key-value additions. The middle fault tier contains the *update process* (running during the U phase), which performs in batch all buffered key-value additions, but runs periodically and only briefly. The highest tier contains the MAS, which keeps the error-prone service process safe and protects the integrity of the update process. Table 1 summarizes the fault tiers in Bonafide.

Under this tiered fault assumption, Bonafide guarantees service safety, that is, *integrity of returned data*. However, to guarantee *durability* as well (i.e., that no data are lost) the system should create copies of data faster than they are lost, as in Carbonite [29]. Also, to ensure *liveness* (i.e., non-starvation) S phases with at most 2/3 faulty replicas must occur once in a while (more precisely, within a finite number of phases at all points in time). This is to ensure that an Add request must be resubmitted by a client a finite number of times before it is eventually served by a U phase.

A Bonafide node contains a MAS as well as a buffer to hold Add requests temporarily and a main data structure that maintains committed bindings (Figure 4). In Bonafide, the service state—the key-value pairs—is maintained as a variation of a hash tree [64], which computes a cryptographic digest of the whole state from the leaves up, storing it at the tree root. The results of individual state queries (i.e., key lookups in the tree) can be validated against that root digest; as long as the digest is kept safe from tampering, individual lookups can be performed by an untrusted service component without risking an integrity violation. This

state is replicated at each replica in the system in untrusted storage (bottom tier) but its root digest (of size on the order of 1 Kbit in today’s hardware) is stored in each node’s MAS. Each replica’s MAS module lies in its most trusted fault tier: we assume that while in service no MAS module returns contents other than those that were stored at it. We use a MAS for the root digest of the service state, since it cryptographically protects the integrity of any answers about that state provided by even an untrusted component.

The service state is updated during the U phase invoked by a trusted watchdog in the most trusted fault tier. In the U phase, all buffered writes are agreed upon by non-faulty replicas using a state machine replication protocol and then reflected in the service state, replacing the integrity digest in each replica’s MAS. The U phase is in the next most trusted fault tier in Bonafide: we assume that no more than a third of the replicas’ update software can fail simultaneously, to ensure that the state machine replication protocol safety and liveness guarantees can be upheld within a single U phase.

The service state is served to clients during the S phase. Responses to Get/Add requests are accepted by clients when $f + 1$ replicas return to the client the same result, and each result is consistent with the corresponding replica’s service state digest in its MAS module. The $f + 1$ number comes from the fault bound of the update tier, which assumes no more than f update processes can be faulty in any single U phase; as a result, no more than f update processes can put an incorrectly updated digest into their own MAS. If the same response to a client request is provided by at least $f + 1$ untrusted service processes, but backed by the $f + 1$ trusted state digests in the MAS, the client is guaranteed to be getting what at least one correct replica provides. At worst, the client will receive no valid responses or obviously invalid responses from the replicas and try again. Also, the service state is audited (for latent storage faults or other bit loss) and repaired during the S phase.

The full details about MAS, the tiered fault model and Bonafide have been presented elsewhere [32].

6. HUMAN ACTIVITY ATTESTATION: NAB

The trusted primitives we have presented thus far are service-facing in a sense: they capture a contained system that interacts with the back end of a service, not so much the users of that service. The primitive we present next is more focused on end users. Specifically, we turned to a trusted *human activity attester*, the fundamental trusted component of the *Not-A-Bot* (NAB) system, whose primary aim is to differentiate between computing activity directly triggered by human action from activity initiated by automation.

6.1 The Problem: Who Is A Bot?

A group of compromised machines that are coordinated remotely by attackers is called a *botnet*. Botnets the major originators of email spam, distributed denial-of-service (DDoS) attacks, and click-fraud on advertisement-based web sites today. By one measure, the current top six botnets alone are responsible for more than 85% of all spam mail [8], amounting to more than 120 billion messages per day that infest more than 95% of all inboxes [9,45]. Botnet-generated DDoS attacks account for about five percent of all web traf-

fic [3], occurring at a rate of more than 4000 distinct attacks per week on average [66]. A problem of a more recent vintage, click-fraud, is a growing threat to companies that draw revenue from web ad placements [11]; bots are said to generate 14–20% of all ad clicks today [2].

As a result, if it were possible for servers to differentiate legitimate, “human-generated” inputs (e.g., email, link clicks, web requests) from automated, “bot-generated” inputs (e.g., spam, click fraud streams, DDoS traffic), many of the aforementioned problems would be significantly mitigated. This observation is not new, but there is currently no good way to obtain such tags *automatically* without explicit human input.

Requiring human input (say, by answering CAPTCHAs [84]) is unlikely to solve the problem. On one hand, solving CAPTCHAs is onerous and disruptive for most humans. On the other hand, CAPTCHAs are not inextricably linked to the intended request and requester and can, therefore, be delegated to other machines or human sweatshops [10]. Furthermore, whereas CAPTCHAs are used today for coarse-grained actions such as email account creation, they are considered too intrusive to be used for finer granularity requests such as sending email or retrieving web URLs. So, in practice, the challenge response is “amortized” over multiple requests (i.e., all email sent from the CAPTCHA-created mail account). Even if an actual human created the account, nothing prevents the bots in that human’s desktop from sending email indiscriminately using that account.

Yet, the problem with obtaining evidence of human activity automatically, say from an as yet unspecified human activity attester, is that the client machine may have been compromised, so one cannot readily trust any information provided by software running on the compromised machine. There are four main requirements for such an attester. First, attestations must be generated in response to human requests automatically, with minimal user involvement. Second, such attestations must not be transferable from the client on which they are generated to attest traffic originating from another client. Third, adoption of the attester must benefit users that deploy it without hurting those that do not. Fourth, the attester must preserve the existing privacy and anonymity semantics of applications while delivering these benefits, which implies that a solution based on globally unique identities—even if those were feasible today despite strong evidence of the contrary [22]—would not be appropriate. All of these requirements must be satisfied without the need to trust the user’s host OS or applications (e.g., while a compromised machine actively tries to subvert the attester functionality). Worse yet, the attester must be small enough to be trustworthy.

6.2 Primitive: Human Activity Attestation

The attester’s sole function is to generate an attestation when an application requests one. An attestation request contains only the application-specific content to attest to (e.g., the email message to send out). The attester may provide the attestation or refuse to provide an attestation at all. We discuss two important decisions: when to grant an attestation and what to attest.

6.2.1 When To Grant An Attestation

The attester’s decision is one of determining the human’s presence and intent: was there a human operating the computer, and did she really intend to send the particular email for which the application is requesting an attestation? Since the attester lacks a direct link to the human’s intentions, it must guess based on the trusted inputs available: the keyboard and mouse⁵. We considered three key design points for such a guessing module.

The best-quality guess is not a guess at all: the attester could momentarily take over the keyboard, mouse, and display device, and prompt the user with a specific question to attest or not attest to a particular email, similarly to how User Account Control works in recent Microsoft Windows systems. While technically feasible to implement, users have found explicit prompts annoying in practice [13]. What is worse, user fatigue inevitably leads to an always-click-OK user behavior [87], which defeats the purpose of attestation.

Instead, we guess human intent using *timing*: how recently before a particular attestation request was the last keyboard or mouse activity observed? We call this a “ $t - \delta$ ” attester, if δ_m denotes the time since the last mouse activity and δ_k denotes the time since the last keyboard activity. For example, the email application requests an attestation specifying that a keyboard or mouse click should have occurred within the last Δ_k or Δ_m milliseconds respectively, where the $\Delta_{\{k,m\}}$ represents the application-specified upper-bound. The attester generates attestations that indicate this time lag, or refuses if that lag is longer than $\Delta_{\{k,m\}}$ milliseconds.

One drawback of the $t - \delta$ attester is that it allows a bot to generate attestations for its own traffic by “harvesting” existing user activity. We mitigate this avenue for bot exploitation via rate limiting activity attestations (rates of hundreds of milliseconds are reasonable for human-initiated activity), and via user notification when her legitimate applications have trouble obtaining activity attestations, which is an explicit indication of local bot activity.

6.2.2 What To Attest

The second attester design decision is what to attest, i.e., how much to link a particular attestation to the issuer, the verifier, and the content. NAB generates responder-specific, content-specific, and, where appropriate, challenger-specific attestations. Attestations are certificates of human activity that contain a signature over the entire request content. For example, an email attestation contains the signature over the entire email, including the “From:” address (i.e., the responder), the email body (i.e., the content), and the “To:” address (i.e., the challenger).

Challenger-specific attestation helps in ensuring that unwitting, honest humans do not furnish attestations for bad purposes. A verifier expecting an attestation from human A ’s attester will reject an attestation from human B that might be provided instead. In the spam example, this is tantamount to explicit sender authentication.

⁵We did not consider other sensors such as cameras, microphones, or other application-specific data, but those might prove very helpful for our purposes.

The attester API is simple: there is only a single request/reply pair of calls between the OS and the attester. An application’s attestation request contains the hash of the message to be attested (i.e., the contents of an email message or the URL of a browser click), the type of attestation requested, and the process id of the requesting process. If the attester verifies that the type of attestation being requested is consistent with user activity seen on the keyboard/mouse channels, it signs the attestation and, depending on the attestation type, includes δ_m and δ_k , which indicate how long ago a mouse click and a keyboard click respectively were last seen.

The same API is used for all applications. The only customization allowed is whether to include the values of the δ_m or δ_k , depending on the attestation type. The attester can protect user privacy by using a group signature scheme for anonymous attestations, extending the Direct Anonymous Attestation service [24] developed for recent TPMs.

We have currently defined and implemented two attestation types, one that hides fine-grained detail about the user’s activity (“type 0”) and one that does not (“type 1”). The former prevents fine temporal tracking of a user’s keyboard and mouse patterns and is offered as a privacy enhancement; essentially it fixes a time threshold to a standard value (1 *sec*) and only issues an attestation if activity has occurred within that period. The latter allows finer-grained attestation requests. Even for interactive applications such as web browsing, private attestations suffice, but we provide both types for flexibility.

An attestation has the form $\langle d, n, \delta_m, \delta_k, \sigma, C \rangle$. It contains a cryptographic content digest d (e.g., a SHA-1 hash) of the application-specific payload attested; a nonce n for attestation freshness and to disallow improper reuse; the $\delta_{\{k,m\}}$ values (for type 1 attestations); the signature $\sigma = \text{sign}(K_{\text{priv}}, \langle d, n, \delta_m, \delta_k \rangle)$; and a certificate C from the TPM guaranteeing the attester’s integrity, the version of the attester, the attestation identity key of the TPM that measured the attester integrity, and the signed attester’s public key K_{pub} . The certificate C is generated during booting of the attester and is stored and reused until reboot.

6.2.3 The Verifier of Attestations

The verifier is co-located with the server processing requests. When invoked, the verifier is passed both the attestation and the request. The attestation and request contain all the necessary information to validate the request.

The verifier first checks the validity of the attester’s public key used for signing the request, by traversing the public-key chain in the certificate C . If valid, it then recomputes the hash of the request’s content and verifies whether the signed hash value in the attestation matches the request’s contents. Further, for attestations that include the $\delta_{\{k,m\}}$ values, the verifier also checks whether $\delta_{\{k,m\}}$ are less than the application-specified $\Delta_{\{k,m\}}$. The verifier then checks to ensure that the attestation is not being double-spent.

A bot running in an untrusted domain cannot masquerade as a trusted attester to the verifier because a TPM will not release the signed K_{pub} to the bot without the correct code hash. Further, the bot derives no benefit from tampering

with the δ values a user specifies in her requests, because the verifier enforces the application-specified upper-limit on $\delta_{\{k,m\}}$.

NAB provides two important security guarantees. First, it ensures that attestations cannot be double-spent. Second, it ensures that a bot cannot steal key clicks and accumulate attestations beyond a fixed time window, which reduces the aggregate volume and burstiness of bot traffic. The verifier uses the nonce in the attestation for these two guarantees. The verifier stores the nonces for a short period (10 minutes for web requests, one month for email). We find this nonce overhead to be small in practice.

The combination of application-specific verifier policy and content-bound attestations can also be used to mitigate bursty attacks. For example, a URL can include an identifier that encodes the link freshness. Since attestations include the identifier, the verifier can discard out-of-date requests, even if they have valid signatures.

6.3 Applicability: Email & Web Defenses

We present a brief overview of two uses of NAB: email anti-spam and web denial-of-service defense. Other applications and further details can be found elsewhere [38].

The mechanism for sending email in the common case is straightforward: the entire email message, including headers and attachments, constitutes the request. Interestingly, the same basic mechanism is extensible to other email usage scenarios, such as text or web-based email, email-over-ssh, batched and offline email, and script-generated email.

At the email recipient side, the attestation verifier uses a verified attestation in much the same way as traditional spam filter, to sort the useful email from spam. The biggest problem with Bayesian spam filters such as spamassassin today is that they either flag too much legitimate email as spam, or flag too little spam as such.

If legitimate requests are expected to carry attestations, the verifier can set spam filters aggressively to flag questionable unattested messages as spam, but use positive evidence of human activity to “whitelist” questionable attested messages. For example, the verifier can sit on the sender ISP’s server alongside a Bayesian spam filter like spamassassin. The filter is configured at an aggressive, low threshold (e.g., -2 instead of the default 5 for spamassassin), because the ISP can force its users to send email with attestations, in exchange for relaying email through its own servers.

This low spamassassin “required score” threshold tags most unattested spam as unwanted. However, in the process, it might also tag some valid email as spam. In order to correct this mistake, the sender-side verifier “salvages” messages with a high spam filter score that carry a valid attestation, and relays them; high-score, unattested email is discarded as spam. This step ensures that legitimate human-generated email is forwarded unconditionally, even if the sender’s machine is compromised.

The recipient’s inbox server can similarly ensure that a legitimate email is not misclassified by improving the spam

score for attested email by a small number (e.g., 3). This number should be high enough that all legitimate email is classified correctly, while spam with or without attestations is still caught.

The mechanism for attesting to web requests is also simple: when a user clicks on a URL, the browser requests an attestation on the entire page URL. After the browser fetches the page content, it uses the same attestation to retrieve any included objects within the page.

We consider scenarios where DDoS is effected by overloading servers, and not by flooding networks. The verifier resides in a firewall or load balancer, and observes the response time of the web server to determine whether the server is overloaded [85]. Here, unlike in spam, the verifier does not drop requests with invalid or missing attestations. Instead, it prioritizes requests with valid attestations over those that lack them. Prioritizing, rather than dropping, makes sense because some valid requests may actually be generated automatically by machines (for example, automatic page refreshes on news sites like `cnn.com`).

We implemented the attester and application-specific policies within the Xen hypervisor. By analyzing traces of keyboard and mouse activity from 328 users at Intel, together with adversarial traces of spam, DDoS, and click-fraud activity, we estimate that NAB reduces the amount of spam that currently passes through a tuned spam filter by more than 92%, while not flagging any legitimate email as spam [38].

7. PERFORMANCE–TRUSTWORTHINESS TRADE-OFFS

The fundamental premise behind an implementation of a trusted primitive \mathcal{P} is that it is harder to subvert than the main application. Different implementation scenarios (illustrated in Figure 5) lead to different threat models and degrees of trust in the resulting system, and are appropriate for different applications. The discussion this far has focused on the benefits of putting particular constrained functionality—what we abstract as \mathcal{P} here—into a *logically* trusted component, not on how that component is ultimately implemented in practice. We switch our focus to this latter aspect of the problem here.

We consider the following implementation scenarios for \mathcal{P} : a separate service offered by a trusted provider or a hardened component (Figure 5(a)), a software-isolated module (Figure 5(b)), a trusted virtual machine (Figure 5(c)), a trusted virtual machine monitor (Figure 5(d)), and trusted hardware (Figure 5(e)). These implementations are viable in the face of different threats. All five implementations work under the faulty application model (external attacks against server software) but only (a) and (e) work under the faulty operator model (malicious operators that operate and can manipulate entire servers).

In the simplest case, \mathcal{P} can be a software abstraction implemented as a service visible to applications via an RPC-like interface (Figure 5(a)). For instance, it could be a service offered by a trusted provider, such as Amazon’s S3 [1], or by a separate, hardened component with significantly greater assurances in the face of software errors than the main applica-

tion software and hardware. This is similar to notarization-like approaches [39, 58, 91] that rely on a trusted write-once medium external to the main system. Though the entire application stack can fail (application, operating system, and hardware), as long as \mathcal{P} is running on a trusted system the application can be protected. The big drawback with this implementation scenario is its network-bound nature—in fact, many of its prior instances in practice use this external write-once medium once a day or so—as well as the requirement that everyone needs on-line access to the trusted \mathcal{P} service provider. Applications with fairly slow request rates such as shared backup services, long-term digital preservation, or certificate authorities may be able to absorb the high-latency interaction with \mathcal{P} in their relatively infrequent state changes. MAS may fit well into this model, but the typically frequent interaction of the application with A2M and NAB—worse yet, the need for a trusted path to NAB from the user’s input devices—make this an inappropriate implementation for them.

Figure 5(b) presents a more decentralized approach, in which the \mathcal{P} implementation relies on the software-based isolation between \mathcal{P} and an \mathcal{P} -enabled application. This approach takes advantage of programming language type and memory safety for isolation. Therefore, \mathcal{P} can be implemented as a library. For instance, in the Singularity [43] operating system, the \mathcal{P} module would be a program that runs as a separate software-isolated process in the same address space. If the Singularity isolation mechanism is trusted, it is possible to trust \mathcal{P} even if the \mathcal{P} -enabled application is untrusted. Similarly, in the Java Virtual Machine (JVM) [7], an application using \mathcal{P} runs in a sandbox, which constitutes a safe execution environment. The assumption is that if the JVM interpreter, JVM core classes, and an operating system that runs the JVM can be trusted, \mathcal{P} can be trusted, even if the \mathcal{P} -enabled Java application is not. Though the isolation is no longer physical as with the scenario of Figure 5(a), communication between the application and \mathcal{P} is fast since they are both in the same address space.

Figure 5(c) presents the \mathcal{P} implementation that relies on the inherent fault isolation properties of a virtual machine monitor (VMM). In the figure, the \mathcal{P} module is a user-space program running on a small, verifiable operating system on top of a VMM. As long as the VMM and the mini-OS are trusted to be exploit-free, possibly by paring down mini-OS complexity through disaggregation [67], it is possible to trust the \mathcal{P} abstraction, even if the application and its general-purpose operating system are compromised. For instance, the virtual Trusted Platform Module (vTPM) [23] has this architecture. Though the isolation is no longer physical as with the scenario of Figure 5(a), communication between the application and \mathcal{P} is only subject to VMM-optimized RPCs, which systems such as Xen [21] make very efficient.

Further reducing the trusted footprint, the \mathcal{P} implementation could be placed within the VMM, as in Figure 5(d). Here, the assumption is that a small VMM (or, indeed, a microkernel) can be carefully implemented (or formally verified [46]) as bug-free, isolating the correctness of the \mathcal{P} implementation from potential operating system or application errors above the VMM. For instance, Xen’s trusted hypervisor interfaces [21] could host such an implementation sce-

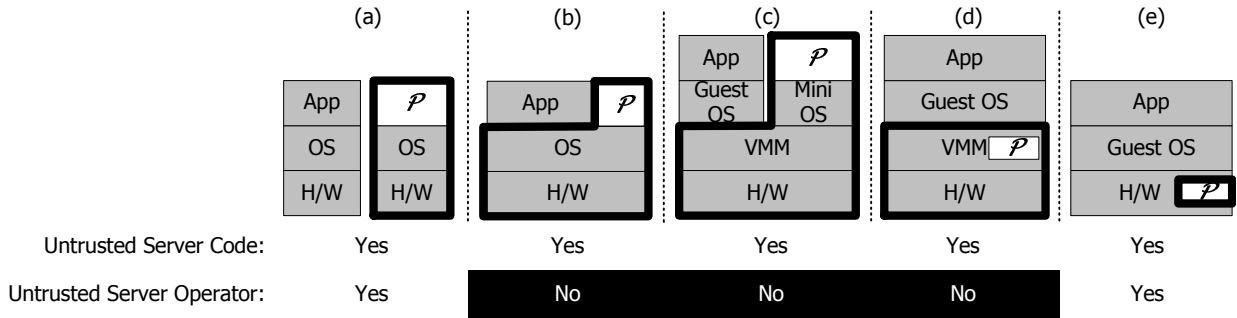


Figure 5: Implementation scenarios. Thick boxes delineate the trusted computing base. (a) trusted service, (b) trusted software isolation, (c) trusted VM, (d) trusted VMM, and (e) trusted hardware.

nario. Both VMM approaches reduce the cost of contacting \mathcal{P} and can yield efficient performance for applications.

Finally, Figure 5(e) places the \mathcal{P} within the hardware itself. Since it tends to be much harder to coerce a hardware module to operate against its specification than it is for software modules, especially without physical access to the hardware, this scenario provides the greatest level of trust in \mathcal{P} . Hardware implementation options might be to extend a standard Trusted Platform Module (TPM) or an Intel Active Management Technology (AMT) chip [5], or to use a programmable secure coprocessor such as IBM’s commercially available PCIXCC [18] board, a programmable PCI-X card with cryptographic primitives as well as physical and electrical tamper-resistance. Tamper resistance offers increased *physical security*: even a malicious host operator armed with electrical probes cannot coerce \mathcal{P} to give responses that are inconsistent with its specification or to reveal its authentication key material, except for extremely expensive physical cryptanalytic attacks that are unrealistic for most practical situations. Moreover, whereas in the past tamper resistance implied low performance, products such as the PCIXCC coprocessor make a hardware \mathcal{P} implementation potentially the best performing one—albeit most expensive—among our scenarios. Nevertheless, pervasive hardware implementations of new programming abstractions tend to be slow to arrive, slow to change, and slow to turn into commodities, making this a more tenuous scenario, except for the most sensitive applications.

8. RELATED WORK

In this section, we survey related work on refactoring existing applications so as to separate small trusted functionality from the remainder. For the detailed uses of presented trusted primitives, we refer the interested reader to our conference publications [31, 32, 38].

8.1 Trusted Execution

Trusted hardware has been used extensively as a mechanism to get reliable information about the state of a computer (*trusted boot*) and to ensure remotely that the computer’s state is as expected for a particular application (*secure boot*). IBM’s 4758 class of secure coprocessors [34] and the Trusted Computing Group’s Trusted Platform Module [12] are the primary exemplars of this functionality. Recent commodity microprocessors have added support for TPMs to enable trusted and secure boot not only at power-on, but also later

during normal operation (e.g., Intel’s Safer Mode Extension instructions, marketed at Intel TXT [37]). Software-based methods have also been described for a computer to attest remotely to its current execution state (e.g., the Pioneer project [75]). Parno et al. have recently published a very comprehensive survey of the space [70].

It is important to note that trusted and secure boot by themselves are only a mechanism to know *what* is executing at a platform, not whether what is executing is actually worth the trust. The properties of what is running can either be formally verified (e.g., via formal proofs, as with the seL4 microkernel [46]), or checked by a (trusted) sandbox-like layer, such as a reference monitor (e.g., for dataflow properties [92] or adherence to known functionality [88]). The checker also needs to be checked itself; validating the checker is starting to see some interest in the literature, especially in the security context [60, 61].

8.2 Fault-Model Differentiation

In the literature, trust levels are differentiated by components, failure types, and failure thresholds. The wormholes model is a *hybrid system model* where the system is decomposed into payload subsystems with weak assumptions and wormhole subsystems with strong assumptions and the two communicate through wormhole gateways [80, 82]. Wormholes such as the Timely Computing Base (TCB) [81] and the Trusted Timely Computing Base (TTCB) [33] provide concrete services such as timely execution and trusted block agreement to payload subsystems. TCB and TTCB are synchronous and fail by crashing. At a somewhat more theoretical level, *hybrid fault models* differentiate failure types on homogeneous systems: some nodes can have benign faults and others can have Byzantine faults [65, 78]. Furthermore, Byzantine faults are classified into malicious symmetric and malicious asymmetric faults [78]. Modified versions of the classic agreement algorithms can lead to more flexible fault tolerance guarantees [19, 78]. Our work has similarly hybridized system components using tiers with different functionalities and fault characteristics.

Researchers have applied *different fault thresholds* to different sites or clusters in several proposed designs. The multi-site threshold model differentiates two types of faults—site and process faults—in multi-site systems [44]. The model uses a fault threshold for the number of sites and a vector of fault thresholds, each of which is assigned to a site to account

for a different process-fault probability depending on sites. Yin et al. [90] proposed an architecture that separates execution from agreement: two groups of replicas— N agreement and M execution replicas—by dividing functionalities. This architecture can tolerate $\lfloor \frac{N-1}{3} \rfloor$ faults and $\lfloor \frac{M-1}{2} \rfloor$ faults, thus assigning different thresholds for the clusters. This partition is done based on functionalities.

8.3 Non-equivocation

The benefits of combating equivocation via a trustworthy component have been explored before, for example in Fleet [57], which uses Timed Append-Only Arrays (TAOAs): single-writer, multi-reader objects to which clients can append values and from which clients can read values. TAOAs are emulated by a distributed client-server protocol built atop a b -masking quorum system [56], which are themselves a layer that may fail. This is similar to our update operation in NAB, but weaker than the local-only, trusted primitives like A2M and MAS, which are simpler and can be built more reliably than a fault-threshold-bound protocol like TAOA.

The TrInc project [53] uses a trusted monotonic counter primitive to implement A2M and, by extension, all applications we have built on top of A2M, making some trade-offs between durability and primitive minimality (and, therefore, reliability). Veronese et al. use the native (but slow) monotonic counters in TPM chips to protect replicated state machines from equivocation [83]. More broadly, in our own prior work we have considered trusted coordinators for on-chip Byzantine fault tolerance [30], and the recent Prophecy system is a PBFT-like replicated state machine that uses a trusted serialization and caching component between clients and faulty servers [74].

8.4 Human-Activity Attestation

CAPTCHAs [84] are the currently most popular mechanism for proving human presence to remote verifiers. However, they suffer from four major drawbacks that render them less attractive for mitigating botnet attacks. First, CAPTCHAs as they are used today are transferable and not bound to the content they attest, and are hence vulnerable to man-in-the-middle attacks; second, CAPTCHAs are semantically independent of the application (i.e., unbound to the user’s intent), are hence exposed to human solver attacks [10]; third, they are obtrusive, which restricts their use for fine-grained attestations (by definition, CAPTCHAs require manual human input), and hence cannot be automated, unlike NAB; and, fourth, even though they are meant to be easy for humans to solve, they are still hard to get right consistently [25], further hurting usability. Also, we are witnessing continued successes in breaking the CAPTCHA implementations of several sites such as Google, Yahoo, and MSN [4], leading some to question even their long-term viability [14], at least in their current form.

The approach of using hardware to enable human activity detection has been described before in the context of on-line games, using untrusted hardware manageability engines (such as Intel’s AMT features) [72].

9. LESSONS LEARNED

The trusted primitives we describe here were not the only ones we explored, but the more successful ones. However,

while searching for trusted primitives, we gained invaluable insights into balancing the conflicting goals of simplicity, broad applicability, and efficiency. We share the lessons learned from our experience in this section.

9.1 More is Less

We have argued repeatedly in this article that smaller, simpler primitives make it easier to justify trusting them. Yet the draw of increasing complexity to build more powerful primitives, or primitives that apply to more systems, is seductive. We considered, for instance, a more powerful human attestation primitive that would keep track of actual user keyclicks, and identify how much of a request is “supported” by keyclick evidence; for instance, compute how much of an email text being attested appeared as sequential keystrokes, using the computed measure as a confidence metric for the attestation. Yet that would have required significant algorithmic complexity, would still not capture the most sophisticated user interaction patterns (e.g., switching windows in the midst of composing a message), and would have provided little benefit. As our results showed, even with a simplistic human activity attester, the benefits for our applications are significant.

Interestingly, not all complexity is created equal with respect to “appropriate primitive sizing.” For instance, adding more arithmetic is not quite as troubling as adding more pointer arithmetic, and even that is not quite as troubling as adding new functional units. While arithmetic might imply complex algorithms, memory manipulations and uncouth practices like pointer arithmetic make it difficult to verify the soundness of an implementation, and new functional units bring with them their libraries, device drivers, software stacks, etc.

Consider, as an illustrative example, changing a primitive so that it directly communicates with other trusted components over a network—as would have been the case if we put all of PBFT, not just a trusted log, into the trusted computing base: much more logic would need to be trusted for that extension to fit into the primitive. A much cheaper alternative would be to “outsource” authenticated communication to the untrusted operating system or application via explicit input and output buffers. While the untrusted pieces may fail to send any messages, they cannot affect the soundness of the primitive. Another such expensive feature is temporal synchrony, which requires trusted time sources with bounded drift or possibly networking for synchronization.

All in all, we found the common-sense recommendations about appropriate trustworthy interfaces by Murray et al. [67] worthwhile. A possibly winning choice for writing practical trusted primitives is combining a small number of functionalities (by excluding, say, dynamic memory management, time-keeping, or networking) with a safe and possibly efficiently evaluable language for expressing semantics; for example, the runtime component for Datalog using in-memory tuples is relatively simple and has polynomial evaluation complexity [77]. This seems to be the trend for both high-integrity software development (e.g., the AdaSPARK commercial language and toolkit) and for *active* content running in shared platforms (as with P2 [55], SPLAY [52], and Comet [35]).

9.2 Less is Expensive

We just argued that the smaller the primitive, the better. Unfortunately, in practice smaller primitives for the same functionality result in “chattier” interactions with the applications using them, which amplifies the effects of any latency incurred by invoking the trusted primitive (Section 7). Also, removing from a trusted primitive some state—for instance, by storing it authenticated in untrusted storage, as was done by TrInc to reduce the footprint of A2M—may result in losses of durability or even liveness. Finally, reducing primitive size may result in reducing application efficiency. For example, A2M’s simple structure as monotonic logs meant that, for some A2M-enabled protocols, messages could only be processed in increasing sequence number, in contrast to the original protocol, which could process messages from different sequence numbers concurrently. This reduced the ability of the protocol to amortize communication delays by having multiple concurrent operations in flight.

One way to improve the situation is to improve the chances that a primitive’s implementation is correct, regardless of its size, as described in the previous section. Another, perhaps brute-force, way forward may just be to lower the overheads of executing trusted code, thereby reducing the penalty incurred by frequent interactions with the primitive. To some extent, enabling late-launch of trusted functionality with Intel’s TXT and AMD’s Presidio did improve the cost of launching trusted code: whereas before late launch running trusted code required that the machine be power-cycled, late launch now only requires resetting some Platform Configuration Registers in the TPM chip and some cryptography. Nevertheless, despite valiant efforts to extract as much controlled-launch functionality as possible with what is available today (e.g., by Flicker [63] and TrustVisor [62]), TPMs are still slow. Thankfully, at least one microprocessor manufacturer seems to be headed towards faster mechanisms for launching isolated, trusted code [6], perhaps at the speed of a context switch.

9.3 The Way Ahead

Applications in the home and in the enterprise alike are only getting more complex. In this article, we described some of the research efforts we have conducted with our collaborators towards identifying, implementing, and evaluating simple trusted primitives that increase the dependability of otherwise untrusted complex applications.

It might be a while before something like a full-featured word processor can be pushed into the trusted computing base wholesale. In the meantime, we envision delivering a dependable service to users via the careful refactoring of complex applications into a (large) untrusted piece and a (small and/or simple) trusted primitive. Important in this endeavor would be tools that help application designers and information technology technicians to identify and carve out what piece of a complex application need be trusted to improve dependability, to study (only) that piece extracting correctness guarantees, and to run the application so that its trusted pieces are seamlessly and efficiently launched in a protected execution environment. Our on-going research pursues these goals.

Acknowledgments: First and foremost we would like to

acknowledge gratefully our co-authors in the work we survey here: Hari Balakrishnan, Ramakrishna Gummadi, John Kubitowicz, and Scott Shenker. We would also like to thank our colleague Jaeyeon Jung for providing us with detailed guidance on how to present this article best.

10. REFERENCES

- [1] Amazon S3. <http://aws.amazon.com/s3/>.
- [2] Click fraud rate rises to 14.1%. <http://redmondmag.com/columns/print.asp?EditorialsID=1456>.
- [3] Five percent of web traffic caused by ddos attacks. <http://www.builderau.com.au/news/soa/Five-percent-of-Web-traffic-caused-by-DDoS-attacks/0,339028227,339287902,00.htm>.
- [4] Gmail CAPTCHA cracked. <http://securitylabs.websense.com/content/Blogs/2919.aspx>.
- [5] Intel Active Management Technology (AMT). <http://www.intel.com/technology/platform-technology/intel-amt/index.htm>.
- [6] Intel cto envisions on-chip data centers. <http://www.informationweek.com/news/global-cio/interviews/showArticle.jhtml?articleID=221900325>.
- [7] Java. <http://java.sun.com/>.
- [8] Six botnets churning out 85% of all spam. <http://arstechnica.com/news.ars/post/20080305-six-botnets-churning-out-85-percent-of-all-spam.html>.
- [9] Spam reaches all-time high of 95% of all email. <http://www.net-security.org/secworld.php?id=5545>.
- [10] Spammers using porn to break CAPTCHAs. http://www.schneier.com/blog/archives/2007/11/spammers_using.html.
- [11] The first AdFraud workshop. <http://crypto.stanford.edu/adfraud/>.
- [12] Trusted Platform Module (TPM) specifications. <https://www.trustedcomputinggroup.org/specs/TPM/>.
- [13] Vista’s UAC security prompt was designed to annoy you. <http://arstechnica.com/news.ars/post/20080411-vistas-uac-security-prompt-was-designed-to-annoy-you.html>.
- [14] Windows Live Hotmail CAPTCHA cracked, exploited. <http://arstechnica.com/news.ars/post/20080415-gone-in-60-seconds-spam-bot-cracks-livehotmail-captcha.html>.
- [15] 104th Congress, United States of America. Public Law 104-191: Health Insurance Portability and Accountability Act (HIPAA), 1996.
- [16] 107th Congress, United States of America. Public Law 107-204: Sarbanes-Oxley Act of 2002, 2002.
- [17] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. of SOSp*, 2005.
- [18] T. W. Arnold and L. P. V. Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48(3/4):475–487, 2004.
- [19] M. H. Azmanesh and R. M. Kieckhafer. New hybrid fault models for asynchronous approximate agreement.

- IEEE Trans. on Computers*, 45(4):439–449, 1996.
- [20] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A Fresh Look at the Reliability of Long-term Digital Storage. In *Proc. of EuroSys*, 2006.
- [21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of SOSP*, 2003.
- [22] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Proc. of CRYPTO*, 1993.
- [23] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the trusted platform module. In *Proc. of USENIX Security*, 2006.
- [24] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *ACM CCS*, 2004.
- [25] E. Bursztein, S. Bethard, J. C. Mitchell, D. Jurafsky, , and C. Fabry. How Good are Humans at Solving CAPTCHAs? A Large Scale Evaluation. In *IEEE Security and Privacy (Oakland)*, 2010.
- [26] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of OSDI*, 1999.
- [27] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proc. of OSDI*, 2000.
- [28] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. on Computer Systems*, 20(4):398–461, 2002.
- [29] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatiowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of NSDI*, 2006.
- [30] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine Byzantine-fault tolerance. In *Proc. of USENIX ATC*, 2008.
- [31] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatiowicz. Attested Append-Only Memory: Making Adversaries Stick to their Word. In *Proc. of SOSP*, 2007.
- [32] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatiowicz. Tiered Fault Tolerance for Long-Term Integrity. In *Proc. of USENIX FAST*, 2009.
- [33] M. Correia, P. Verissimo, and N. F. Neves. The design of a COTS real-time distributed security kernel. In *European Dependable Computing*, 2002.
- [34] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34(10), 2001.
- [35] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An Active Distributed Key-Value Store. In *Proc. of OSDI*, 2010.
- [36] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Super-efficient verification of dynamic outsourced databases. In *Proc. of RSA*, 2008.
- [37] D. Grawrock. *Dynamics of a Trusted Platform*. Intel Press, 2008.
- [38] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-Bot: improving service availability in the face of botnet attacks. In *Proc. of NSDI*, 2009.
- [39] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.
- [40] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proc. of NSDI*, 2005.
- [41] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.
- [42] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing OS processes to improve dependability and safety. In *Proc. of EuroSys*, 2007.
- [43] G. Hunt and J. Larus. Singularity: Rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.
- [44] F. P. Junqueira and K. Marzullo. The virtue of dependent failures in multi-site systems. In *USENIX HotDep*, 2005.
- [45] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. Spamalytics: An empirical analysis of spam marketing conversion. In *ACM CCS*, 2008.
- [46] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. of SOSP*, 2009.
- [47] P. C. Kocher. On certificate revocation and validation. In *Financial Cryptography*, 1998.
- [48] J. Kubiatiowicz, D. Bindel, Y. Chen, S. Czerwinski, 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. of ASPLOS*, 2000.
- [49] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems*, 16(2), 1998.
- [50] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems*, 4(3):382–401, 1982.
- [51] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. on Computer Systems*, 1992.
- [52] L. Leonini, É. Rivière, and P. Felber. SPLAY: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Proc. of NSDI*, 2009.
- [53] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proc. of NSDI*, 2009.
- [54] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. of OSDI*, 2004.
- [55] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proc. of SOSP*, 2005.
- [56] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. of STOC*, 1997.

- [57] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Trans. on Knowledge and Data Engineering*, 12(2):187–202, 2000.
- [58] P. Maniatis and M. Baker. Secure History Preservation Through Timeline Entanglement. In *Proc. of USENIX Security*, 2002.
- [59] P. Maniatis, M. Roussopoulos, T. Giuli, D. S. H. Rosenthal, and M. Baker. The LOCKSS Peer-to-Peer Digital Preservation System. *ACM Trans. on Computer Systems*, 23(1):2–50, 2005.
- [60] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *Proc. of ISSTA*, 2009.
- [61] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing System Virtual Machines. In *Proc. of ISSTA*, 2010.
- [62] J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Security and Privacy (Oakland)*, 2010.
- [63] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proc. of EuroSys*, 2008.
- [64] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proc. of CRYPTO*, 1987.
- [65] F. Meyer and D. Pradhan. Consensus with dual failure modes. In *IEEE Fault-Tolerant Computing*, 1987.
- [66] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring Internet denial-of-service activity. *ACM Trans. on Computer Systems*, 24(2), 2006.
- [67] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security through Disaggregation. In *Proc. of VEE*, 2008.
- [68] M. Naor. Bit commitment using pseudorandomness. *Journal of Cryptology*, 4(2):151–158, 1991.
- [69] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proc. of OSDI*, 1996.
- [70] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping Trust in Commodity Computers. In *IEEE Security and Privacy (Oakland)*.
- [71] C. Preimesberger. Intel Faces Up to E-Mail Retention Problems in AMD Lawsuit. *eWeek.com*, Mar. 2007. Fetched on 10/9/2007 from <http://www.eweek.com/article2/0,1759,2101674,00.asp>.
- [72] T. Schluessler, S. Goglin, and E. Johnson. Is a bot at the controls?: Detecting input data attacks. In *Proc. of NetGames*, 2007.
- [73] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proc. of SOSP*, 2003.
- [74] S. Sen, W. Lloyd, and M. J. Freedman. Prophecy: Using History for High-Throughput Fault Tolerance. In *Proc. of NSDI*, 2010.
- [75] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proc. of SOSP*, 2005.
- [76] P. Sousa, N. F. Neves, and P. Veríssimo. Proactive Resilience through Architectural Hybridization. In *ACM Symposium on Applied Computing*, 2006.
- [77] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proc. of ICLP*, 1986.
- [78] P. Thambidurai and Y.-K. Park. Interactive consistency with multiple failure modes. In *Proc. of SRDS*, 1988.
- [79] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8), 1984.
- [80] P. Verissimo. Uncertainty and Predictability: Can they be reconciled? *LCNS: FuDiCo*, 2584, 2003.
- [81] P. Verissimo, A. Casimiro, and C. Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proc. of DSN*, 2000.
- [82] P. E. Verissimo. Travelling through wormholes: a new look at distributed systems models. *ACM SIGACT News*, 37(1), 2006.
- [83] G. S. Veronese, M. Correia, L. C. Lung, A. N. Bessani, and P. Verissimo. Minimal Byzantine Fault Tolerance. Technical Report TR-08-29, Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa, Portugal, 2008.
- [84] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In *Proc. of EuroCrypt*, 2003.
- [85] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS Defense by Offense. In *Proc. of SIGCOMM*, 2006.
- [86] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proc. of SIGCOMM*, 2004.
- [87] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proc. of USENIX Security*, 1999.
- [88] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proc. of OSDI*, 2008.
- [89] Y. Xie and A. Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *International Conference on Computer Aided Verification*, 2005.
- [90] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. of SOSP*, 2003.
- [91] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. In *Proc. of USENIX FAST*, 2007.
- [92] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: System Support for Derived Data Management. In *Proc. of VEE*, 2010.