Separating Agreement from Execution for Byzantine Fault Tolerant Services

Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, Mike Dahlin
Laboratory for Advanced Systems Research
Department of Computer Sciences
The University of Texas at Austin

ABSTRACT

We describe a new architecture for Byzantine fault tolerant state machine replication that separates *agreement* that orders requests from *execution* that processes requests. This separation yields two fundamental and practically significant advantages over previous architectures. First, it reduces replication costs because the new architecture can tolerate faults in up to half of the state machine replicas that execute requests. Previous systems can tolerate faults in at most a third of the combined agreement/state machine replicas. Second, separating agreement from execution allows a general *privacy firewall* architecture to protect confidentiality through replication. In contrast, replication in previous systems hurts confidentiality because exploiting the weakest replica can be sufficient to compromise the system. We have constructed a prototype and evaluated it running both microbenchmarks and an NFS server. Overall, we find that the architecture adds modest latencies to unreplicated systems and that its performance is competitive with existing Byzantine fault tolerant systems.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Distributed systems; D.4.6 [**Operating Systems**]: Information flow controls

General Terms

Performance, Security, Reliability

Keywords

Byzantine fault tolerance, confidentialiy, reliability, security, state machine replication, trustworthy systems

1. INTRODUCTION

This paper explores how to improve the trustworthiness of software systems by using redundancy to simultaneously enhance integrity, availability, and confidentiality. By integrity, we mean that a service processes users' requests correctly. By availability, we mean that a service operates without interruption. By confidentiality, we mean that a service only reveals the information a user is authorized to see.

Our goal is to provide these guarantees despite malicious attacks that exploit software bugs. Despite advances in applying formal verification techniques to practical problems [27, 36], there is little near-term prospect of eliminating bugs from increasingly complex software systems. Yet, the growing importance of software systems makes it essential to develop ways to tolerate attacks that exploit such errors. Furthermore, the increasing deployment of access-anywhere network services raises both challenges—access-anywhere can mean attack-from-anywhere—and opportunities—when systems are shared by many users, they can devote significant resources to hardening themselves to attack.

Recent work has demonstrated that Byzantine fault tolerant (BFT) state machine replication is a promising technique for using redundancy to improve integrity and availability [38] and that it is practical in that it adds modest latency [11], can proactively recover from faults [12], and can make use of existing software diversity to exploit "opportunistic n-version programming" [40].

Unfortunately, two barriers limit widespread use of these techniques to improve security. First, although existing general BFT systems improve integrity and availability, they hurt confidentiality. In particular, although either increasing the number of replicas or making the implementations of replicas more diverse reduces the chance that an attacker compromises enough replicas to bring down a service, each also increases the chance that at least one replica has an exploitable bug—and an attacker need only compromise the weakest replica in order to endanger confidentiality. Second, BFT systems require more than three-fold replication to work correctly [7]; for instance, tolerating just a single faulty replica requires at least four replicas. Even with opportunistic n-version programming and falling hardware costs, this replication cost is significant, and reducing the replication required would significantly increase the practicality of BFT replication.

In this paper, we explore a new general BFT replication architecture to address these two problems. The key principle of this architecture is to *separate agreement from execution.* State machine replication relies on first agreeing on a linearizable order of all requests [28, 29, 43] and then executing these requests on all state machine replicas. Existing BFT state machine systems tightly couple these two functions, but cleanly separating agreement and execution yields two fundamental advantages.

First, our architecture reduces replication costs because the new architecture can tolerate faults in up to half of the state machine replicas responsible for executing requests. In particular, while our system still requires $3f + 1$ *agreement replicas* to order requests using f-resilient Byzantine agreement, it only requires a simple majority of correct *execution replicas* to process the ordered requests. This distinction is crucial because execution replicas are likely to be much more expensive than agreement replicas both in terms of hardware—because of increased processing, storage, and I/O—and, especially, in terms of software. When n-version (or opportunistic n-version) programming is used to eliminate common-mode failures across replicas, the agreement nodes are part of a generic library that may be reused across applications, while the cost of replicating execution code must be paid by each different service.

Second, separating agreement from execution leads to a practical and general *privacy firewall* architecture to protect confidentiality through Byzantine replication. In existing state machine replication architectures, a voter co-located with each client selects from among replica replies. Voters are co-located with clients to avoid introducing a new single point of failure when integrity and availability are the goals [43], but when confidentiality is also a goal, existing architectures allow a malicious client to observe confidential information leaked by faulty servers. In our system, a redundant set of privacy firewall nodes restricts communication from the execution nodes (where confidential information is manipulated and stored) to filter out incorrect replies before they reach the agreement nodes or the clients.

Note that like the BFT systems on which we build [11, 12, 40], our approach can replicate state machines where execution nodes can perform arbitrary computations on or modifications of the system's internal state. In contrast, Byzantine storage systems based on quorum replication [33] can encrypt data to prevent servers from leaking it, but such protocols do not support more general services that require servers to manipulate the contents of stored data.

We have constructed a prototype system that separates agreement from execution and that implements a privacy firewall. Overall, we find its performance to be competitive with previous systems [11, 12, 40]. With respect to latency, for example, our system is 16% slower than BASE [40] for the modified Andrew-500 benchmark with the privacy firewall. With respect to processing overhead and overall system cost, several competing factors affect the comparison. On one hand, the architecture allows us to reduce the number of execution servers and the resources consumed to execute requests. On the other hand, when the privacy firewall is used, the system must pay additional cost for the extra firewall nodes and for a relatively expensive threshold signature operation, though the latter cost can be amortized across multiple replies by signing "bundles" containing multiple replies. Overall, our overheads are higher than previous systems when applications do little processing and when aggregate load (and therefore bundle size) is small, and our overheads are lower in the opposite situations.

In evaluating our privacy firewall architecture, we find that nondeterminism in some applications and in the network raises significant challenges for ensuring confidentiality because an adversary can potentially influence nondeterministic outputs to achieve a covert channel for leaking information. For the applications we examine, we find that agreement nodes can resolve application nondeterminism obliviously, without knowledge of the inputs or the application state and therefore without leaking confidential information. However, more work is needed to determine how broadly applicable this approach is. For nondeterminism that stems from the asynchrony of our system model and unreliability of our network model (e.g., reply timing, message order, and retransmission counts), we show that our system provides *output symbol set confidentiality*, which is similar to possibilistic non-interference [35] from the programming languages literature. We also outline ways to restrict such network nondeterminism, but future work is needed to understand the vulnerability of systems to attacks exploiting network nondeterminism and the effectiveness of techniques to reduce this vulnerability.

The main contribution of this paper is to present the first study to systematically apply the principle of separation of agreement and execution to (1) reduce the replication cost and (2) enhance confidentiality properties for general Byzantine replicated services. Although in retrospect this separation is straightforward, all previous general BFT state machine replication systems have tightly coupled agreement and execution, have paid unneeded replication costs, and have increased system vulnerability to confidentiality breaches.

In Section 2, we describe our system model and assumptions. Then, Section 3 describes how we separate agreement from execution and Section 4 describes our confidentiality firewall architecture. Section 5 describes and evaluates our prototype. Finally Section 6 puts related work in perspective and Section 7 summarizes our conclusions.

2. SYSTEM MODEL AND ASSUMPTIONS

We consider a distributed asynchronous system where nodes may operate at arbitrarily different speeds and where there is no *a priori* bound on message delay [28]. We also assume an unreliable network that can discard, delay, replicate, reorder, and alter messages. More specifically, our protocols ensure safety regardless of timing, crash failures, message omission, message reordering, and message alterations that do not subvert our cryptographic assumptions defined below. Note, however, that it is impossible to guarantee liveness unless some assumptions are made about synchrony and message loss [18]. Our system makes progress if the relatively weak *bounded fair links* assumption holds. We define a bounded fair links network as a network that provides two guarantees. First, it provides the *fair links* guarantee: if a message is sent infinitely often to a correct receiver then it is received infinitely often by that receiver. Second, there exists some delay T such that if a message is retransmitted infinitely often to a correct receiver according to some schedule from time t_0, then the receiver receives the message at least once before time $t_0 + T$; note that the participants in the protocol need not know the value of T. This assumption appears reasonable in practice assuming that network partitions are eventually repaired.

We assume a Byzantine fault model for machines and a strong adversary. Faulty machines can exhibit arbitrary behavior. For example, they can crash, lose data, alter data, and send incorrect protocol messages. Furthermore, we assume an adversary who can coordinate faulty nodes in arbitrary ways. However, we restrict this weak assumption about machine faults with two strong assumptions. First, we assume that some bound on the number of faulty servers is known; for example a given configuration might assume that at most f of n servers are faulty. Second, we assume that no machine can subvert the assumptions about cryptographic primitives described in the following paragraphs.

Our protocol assumes cryptographic primitives with several important properties. We assume a cryptographic *authentication certificate* that allows a subset containing k nodes out of a set $\mathcal{S}$ of nodes to operate on message X to produce an authentication certificate $\langle X\rangle_{\mathcal{S},\mathcal{D},k}$ that any node in some set of destination nodes $\mathcal{D}$ can regard as proof that k distinct nodes in $\mathcal{S}$ *said* [8] X. To provide a range of performance and privacy trade-offs, our protocol supports three alternative implementations of such authentication certificates that are conjectured to have the desired properties if a bound is assumed on the adversary's computational power: public key signatures [39], message authentication code (MAC) authenticators [11, 12, 47], and threshold signatures [16].

In order to support the required cryptographic primitives, we assume that correct nodes know their private keys (under signature and threshold signature schemes) or shared secret keys (under MAC authenticator schemes) and that if a node is correct then no other node knows its private keys. Further, we assume that if both nodes sharing a secret key are correct, then no other node knows their shared secret key. We assume that public keys are distributed so that the intended recipients of messages know the public keys needed to verify messages they receive.

Note that in practice, public key and threshold signatures are typically implemented by computing a cryptographic digest of a message and signing the digest, and MACs are implemented by computing a cryptographic digest of a message and a secret. We assume that a cryptographic digest of a message X produces a fixed-length collection of bits $D(X)$ such that it is computationally infeasible to generate a second message X' with the same digest $D(X') = D(X)$ and such that it is computationally infeasible to calculate X given $D(X)$. Because digest values are of fixed length, it is possible for multiple messages to digest to the same value, but the length is typically chosen to be large enough to make this probability negligible. Several existing digest functions such as SHA1 [44] are believed to have these properties assuming that the adversary's computational power is bounded.

To allow servers to buffer information regarding each client's most recent request, we assume a finite universe of *authorized clients* that send authenticated requests to the system. For signature-based authenticators, we assume that each authorized client knows its own public and private keys, that each server knows the public keys of all authorized clients, and that if a client is correct, no other machine knows its private key. For MAC-based authenticators, we assume each client/server pair shares a secret and that if both machines are correct, no other node knows the secret. For simplicity, our description assumes that a correct client sends a request, waits for the reply, and sends its next request, but it is straightforward to extend the protocol to allow each client to have k outstanding requests. The system tolerates an arbitrary number of Byzantine-faulty clients in that non-faulty clients observe a consistent system state regardless of the actions of faulty clients. Note that a careless, malicious, or faulty client can issue disruptive requests that the system executes (in a consistent way). To limit such damage, applications typically implement access control algorithms that restrict which actions can be taken by which clients.

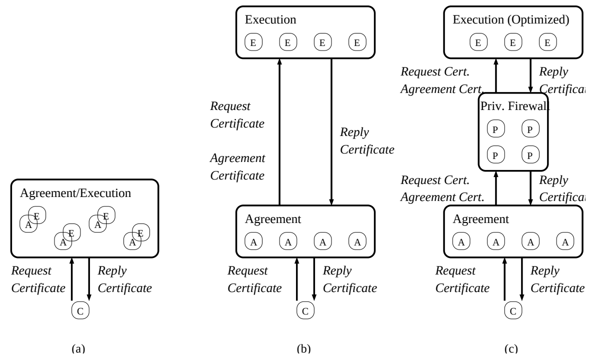

Figure 1: High level architecture of (a) traditional Byzantine fault tolerant state machine replication, (b) separate Byzantine fault tolerant agreement and execution, and (c) new optimizations enabled by the separation of agreement and execution.

The basic system replicates applications that behave as *deterministic* state machines. Given a current state C and an input I, all non-faulty copies of the state machine transition to the same subsequent state C'. We also assume that all correct state machines have a $checkpoint(C)$ function that operates on the state machine's current state and produces sequence of bits B. State machines also have a $restore(B)$ function that operates on a sequence of bits and returns a state machine state such that if a correct machine executes $checkpoint(C)$ to produce some value B, then any correct machine that executes $restore(B)$ will then be in state C. We discuss how to abstract nondeterminism and minor differences across different state machine replicas [40] in Section 3.1.4.

3. SEPARATING AGREEMENT FROM EXECUTION

Figure 1(a) illustrates a traditional Byzantine fault tolerant state machine architecture that combines agreement and execution [11, 12, 40]. In such systems, clients send authenticated requests to the $3f + 1$ servers in the system, where f is the maximum number of faults the system can tolerate. The servers then use a three-phase protocol to generate cryptographically verifiable proofs that assign unique sequence numbers to requests. Each server then executes requests in sequence-number order and sends replies to the clients. A set of $f + 1$ matching replies represents a *reply certificate* that a client can regard as proof that the request has been executed and that the reply is correct.

Figure 1(b) illustrates our new architecture that separates agreement and execution. The observation that enables this separation is that the agreement phase of the traditional architecture produces a cryptographically-verifiable proof of the ordering of a request. This *agreement certificate* can be verified by any server, so it is possible for execution nodes to be separate from agreement nodes.

Figure 1(c) illustrates two enhancements enabled by the separation of execution and agreement.

1. We can separately optimize the agreement and execution clusters. In particular, it takes a minimum of $3f + 1$ servers to reach agreement in an asynchronous system that can suffer f Byzantine faults [7]. But, once incoming requests have been ordered, a simple majority of correct servers suffices to mask Byzantine faults among execution servers—$2g+1$ replicas can tolerate g faults. Note that the agreement and execution servers can be separate physical machines, or they can share the same physical machines.

   Reducing the number of state machine execution replicas needed to tolerate a given number of faults can reduce both software and hardware costs of Byzantine fault tolerance. Software costs are reduced for systems using n-version programming to reduce the correlation of failures across replicas because the $3f+1$ agreement replicas run a generic library that can be reused across applications, and only $2g+1$ application-specific state machine implementations are needed. Hardware costs are reduced because fewer replicas of the request are processed, fewer I/Os are performed, and fewer copies of the state are stored.

2. We can insert a *privacy firewall* between the agreement nodes and the execution nodes to filter minority answers from replies rather than sending these incorrect replies to clients. This filtering allows a Byzantine replicated system to protect confidentiality of state machine data in the face of faults. Note that this configuration requires agreement and execution nodes to by physically separate and to communicate only via confidentiality nodes.

The rest of this section describes a protocol that separates agreement from execution. Section 4 then describes the privacy firewall.

3.1 Inter-cluster protocol

We first provide a cluster-centric description of the protocol among the client, agreement cluster, and execution cluster. Here, we treat the agreement cluster and execution cluster as opaque units that can reliably take certain actions and save certain state. In Sections 3.2 and 3.3 we describe how individual nodes within these clusters act to ensure this behavior.

3.1.1 Client behavior

To issue a request, a client sends a request certificate to the agreement cluster. In our protocol, request certificates have the form $\langle REQUEST, o, t, c\rangle_{c,\mathcal{A},1}$ where o is the operation requested, t is the timestamp, and c is the client that issued the request; the message is certified by the client c to agreement cluster $\mathcal{A}$ and one client's certification is all that is needed.[1] A correct client issues monotonically increasing timestamps; if a faulty client's clock runs backwards, its own requests may be ignored, but no other damage is done.

After issuing a request, a client waits for a reply certificate certified by at least $g+1$ execution servers. In our protocol a reply certificate has the form: $\langle REPLY, v, n, t, c, \mathcal{E}, r\rangle_{\mathcal{E},c,g+1}$ where v was the view number in the agreement cluster when it assigned a sequence number to the request, n is the request's sequence number, t is the request's timestamp, c is the client's identity, r is the result of the requested operation, and $\mathcal{E}$ is the set of execution nodes of which at least $g+1$ must certify the message to c.

[1]Note that our message formats and protocol closely follow Castro and Liskov's [11, 12].

If after a timeout the client has not received the complete reply certificate, it retransmits the request to all agreement nodes. Castro and Liskov suggest two useful optimizations to reduce communication [11]. First, a client initially sends a request to the agreement server that was the primary during the view v of the most recent reply received; retransmissions go to all agreement servers. Second, a client's request can designate a specific agreement node to send the reply certificate or can contain the token ALL, indicating that all agreement servers should send. The client's first transmission designates a particular server, while retransmissions designate ALL.

3.1.2 Agreement cluster behavior

The agreement cluster's job is to order requests, send them to the execution cluster, and relay replies from the execution cluster to the client. The agreement cluster acts on two messages—the intra-cluster protocols discussed later will explain how to ensure that these actions are taken reliably.

First, when the agreement cluster receives a valid client request certificate $\langle REQUEST, o, t, c\rangle_{c,\mathcal{A},1}$ the cluster proceeds with three steps, the first of which is optional.

1. Optionally, check $cache_c$ for a cached reply certificate with the same client c and a timestamp that is at least as large as the request's timestamp t. If such a reply is cached, send it to the client and stop processing the request. $cache_c$ is an optional data structure that stores the reply certificate for the most recent request by client c. $cache_c$ is a performance optimization only, required for neither safety nor liveness, and any $cache_c$ entry may be discarded at any time.

2. Generate an agreement certificate that binds the request to a sequence number n. In our protocol, the agreement certificate is of the form $\langle COMMIT, v, n, d, \mathcal{A}\rangle_{\mathcal{A},\mathcal{E},2f+1}$ where v and n are the view and sequence number assigned by the agreement three phase commit protocol, d is the digest of the client's request ($d = D(m)$), and the certificate is authenticated by at least $2f+1$ of the agreement servers $\mathcal{A}$ to the execution servers $\mathcal{E}$.

   Note that if the request's timestamp is no larger than the timestamp of a previous client request, then the agreement cluster still assigns the request a new sequence number. The execution cluster will detect the old timestamp t, assume such requests are retransmissions of earlier requests, and treat them as described below.

3. Send the request certificate and the agreement certificate to the execution cluster.

Second, when the agreement cluster receives a reply certificate $\langle REPLY, v, n, t, c, \mathcal{E}, r\rangle_{\mathcal{E},c,g+1}$ it relays the certificate to the client. Optionally, it may store the certificate in $cache_c$.

In addition to these two message-triggered actions, the agreement cluster performs retransmission of requests and agreement certificates if a timeout expires before it receives the corresponding reply certificate. Unlike the traditional architecture in Figure 1(a), communication between the agreement cluster and execution cluster is unreliable. And, although correct clients should repeat requests when they do not hear replies, it would be unwise to depend on (untrusted) clients to trigger the retransmissions needed to fill potential gaps in the sequence number space at execution nodes. For each agreement certificate, the timeout is set to an arbitrary initial value and then doubled after each retransmission. To bound the state needed to support retransmission, the agreement cluster has at most P requests outstanding in the execution pipeline and does not generate agreement certificate n until it has received a reply with a sequence number of at least $n - P$.

3.1.3 Execution cluster behavior

The execution cluster implements *application-specific* state machines to process requests in the order determined by the agreement cluster.

To support exactly-once semantics, the execution cluster stores $Reply_c$, the last reply certificate sent to client c.

When the execution cluster receives both a valid request $\langle REQUEST, o, t, c\rangle_{c,\mathcal{A},1}$ and a valid agreement certificate $\langle COMMIT, v, n, d, \mathcal{A}\rangle_{\mathcal{A},\mathcal{E},2f+1}$ for that request, the cluster waits until all requests with sequence numbers smaller than n have been received and executed. Then, if the request's sequence number exceeds by one the highest sequence number executed so far, the cluster takes one of three actions.

1. If the request's timestamp exceeds the timestamp of the reply in $Reply_c$, then the cluster executes the new request, updates $Reply_c$, and sends the reply certificate to the agreement cluster.

2. If the request's timestamp equals $Reply_c$'s timestamp, the request is a client-initiated retransmission of an old request, so the cluster generates, caches, and sends a new reply certificate containing the cached timestamp t', the cached reply body r', the request's view v, and the request's sequence number n.

3. If the request's timestamp is smaller than $Reply_c$'s timestamp, the cluster must acknowledge the new sequence number so that the agreement cluster can continue to make progress, but it should not execute the lower-timestamped client request; therefore, the cluster acts as in the second case and generates, caches, and sends a new reply certificate containing the cached timestamp t', the cached reply body r', the request's view v, and the request's sequence number n.

The above three cases are relevant when the execution cluster processes a new sequence number. If, on the other hand, a request's sequence number is not larger than the highest sequence number executed so far, the execution cluster assumes the request is a retransmission from the agreement cluster, and it retransmits the client's last reply certificate from $Reply_c$; this reply is guaranteed to have a sequence number at least as large as the request's sequence number.

Note that in systems not using the privacy firewall architecture described in Section 4, a possible optimization is for the client to send the request certificate directly to both the agreement cluster and the execution cluster and for the execution cluster to send reply certificates directly to both the agreement cluster and the client.

3.1.4 Non-determinism

The state machines replicated by the system must be deterministic to ensure that their replies to a given request match and that their internal states do not diverge. However, many important services include some nondeterminism when executing requests. For example, in the network file system NFS, different replicas may choose different file handles to identify a file [40] or attach different last-access timestamps when a file is read [11]. To address this issue, we extend the standard technique [11, 12, 40] of resolving nondeterminism which has the agreement phase select and sanity check any nondeterministic values needed by a request. To more cleanly separate (generic) agreement from (application-specific) execution, our agreement cluster is responsible for generating nondeterministic inputs that the execution cluster deterministically maps to any application-specific values it needs.

For the applications we have examined, the agreement cluster simply includes a timestamp and a set of pseudo-random bits in each agreement certificate; similar to the BASE protocol, the primary proposes these values and the other agreement nodes sanity-check them and refuse to agree to unreasonable timestamps or incorrect pseudo-random number generation. Then, the abstraction layer [40] at the execution nodes executes a deterministic function that maps these inputs to the values needed by the application. We believe that this approach will work for most applications, but future work is needed to determine if more general mechanisms are needed.

3.2 Internal agreement cluster protocol

Above, we describe the high-level behavior of the agreement cluster as it responds to request certificates, reply certificates, and timeouts. Here we describe the internal details of how the nodes in our system behave to meet these requirements.

For simplicity, our implementation uses Rodrigues et al.'s BASE (BFT with Abstract Specification Encapsulation) library [40], which implements a replicated Byzantine state machine by receiving, sequencing, and executing requests. We treat BASE as a Byzantine agreement module that handles the details of three-phase commit, sequence number assignment, view changes, checkpoints, and garbage collecting logs [11, 12, 40]. In particular, clients send their requests to the BASE library on the agreement nodes to bind requests to sequence numbers. But, when used as our agreement library, the BASE library does not execute requests directly against the *application* state machine, which is in our execution cluster. Instead, we install a message queue (described in more detail below) as the BASE library's *local* state machine, and the BASE library "executes" a request by calling *msgQueue.insert(request certificate, agreement certificate)*. From the point of view of the existing BASE library, when this call completes, the request has been executed. In reality, this call enqueues the request for asynchronous processing by the execution cluster, and the replicated message queues ensure that the request is eventually executed by the execution cluster. Our system makes four simple changes to the existing BASE library to accommodate this asynchronous execution.

1. First, whereas the original library sends the result of the local execution of a request to the client, the modified library does not send replies to clients; instead, it relies on the message queue to do so.

2. Second, to ensure that clients can eventually receive the reply to each of their requests, the original BASE library maintains a cache of the last reply sent to each client and sends the cached value when a client retransmits a request. But when the modified library receives repeated requests from a client, it does not send the locally stored reply since the locally stored reply is the result of the enqueue operation, not the result of the executing the body of the client's request. Instead, it calls *msgQueue.retryHint(request certificate)*, telling the message queue to send the cached reply or to retry the request.

3. Third, BASE periodically generates consistent checkpoints from its replicas' internal state so that buffered messages can be garbage collected while ensuring that nodes that have fallen behind or that have recovered from a failure can resume operation from a checkpoint [11, 12, 40]. In order to achieve a consistent checkpoint at some sequence number n across message queue instances despite their asynchronous internal operation, the modified BASE library calls *msgQueue.sync()* after inserting message n. This call returns after bringing the local message queue state to a consistent state as required by the BASE library's checkpointing and garbage collection algorithms.

4. Fourth, the sequence number for each request is determined according to the order in which it is inserted into the message queue.

3.2.1 Message queue design

Each node in the agreement cluster has an instance of a message queue as its local state machine. Each message queue instance stores $maxN$, the highest sequence number in any agreement certificate received, and *pendingSends*, a list of request certificates, agreement certificates, and timeout information for requests that have been sent but for which no reply has been received. Optionally, each instance may also store $cache_c$, the reply certificate for the highest-numbered request by client c.

When the library calls *msgQueue.insert(request certificate, agreement certificate)*, the message queue instance stores the certificates in *pendingSends*, updates $maxN$, and multicasts both certificates to all nodes in the execution cluster. It then sets a per-request timer to the current time plus an initial time-out value. As an optimization, when a message is first inserted, only the current primary needs to send it to the execution cluster; in that case, all nodes retransmit if the timeout expires before they receive the reply. In order to bound the state needed by execution nodes for buffering recent replies and out of order requests, a pipeline depth P bounds the number of outstanding requests; an *insert()* call for sequence number n blocks (which prevents the node from participating in the generation of sequence numbers higher than n) until the node has received a reply with a sequence number at least $n - P$.

When an instance of the message queue receives a valid reply certified by $g + 1$ execution cluster nodes, it deletes from *pendingSends* the request and agreement certificates for that request and for all requests with lower sequence numbers; it also cancels the retransmission timer for those requests. The message queue instance then forwards the reply to the client. Optionally, the instance updates $cache_c$ to store the reply certificate for client c.

When the modified BASE library calls *retryHint(request certificate)* for a request r from client c with timestamp t, the message queue instance first checks to see if $cache_c$ contains a reply certificate with a timestamp of at least t. If so, it sends the reply certificate to the client. Otherwise, if a request and agreement certificate with matching c and t are available in *pendingSends*, then the queue resends the certificates to the execution cluster. Finally, if neither the *cache* nor the *pendingSends* contains a relevant reply or request for this client-initiated retransmission request, the message queue uses BASE to generate a new agreement certificate with a new sequence number for this old request and then calls *insert()* to transmit the certificates to the execution cluster.

When the retransmission timer expires for a message in *pendingSends*, the instance resends the request and agreement certificates and updates the retransmission timer to the current time plus twice the previous timeout interval.

Finally, when the modified BASE library calls *msgQueue.sync()*, the message queue stops accepting *insert()* requests or generating new agreement certificates and waits to receive a reply certificate for a request with sequence number $maxN$ or higher. Once it processes such a reply and *pendingSends* is therefore empty, the *sync()* call returns and the message queue begins accepting *insert()* calls again. Note that *cache* may differ across servers and is not included in checkpoints.

3.3 Internal execution cluster protocol

Above, we defined the high-level execution cluster's behavior in response to request and agreement certificates. Here, we describe execution node behaviors that ensure these requirements are met.

Each node in the execution cluster maintains the application state, a pending request list of at most P received but not yet executed requests (where P is the maximum pipeline depth of outstanding requests by the execution cluster), the largest sequence number that has been executed $maxN$, and a table *reply* where $reply_c$ stores the node's piece of its most recent reply certificate for client c. Each node also stores the most recent *stable checkpoint*, which is a checkpoint across the application state and the *reply* table that is certified by at least $g+1$ execution nodes. Nodes also store zero or more newer checkpoints that have not yet become stable.

When a node receives a valid request certificate certified by a client c and a valid agreement certificate certified by at least $2f + 1$ agreement nodes, it stores the certificates in the pending request list. Then, once all requests with lower sequence numbers have been received and processed, the node processes the request. If the request has a new sequence number (i.e., $n = maxN + 1$), the node takes one of two actions: (1) if $t > t'$ (where t is the request's timestamp and t' is the timestamp of the reply in $reply_c$), then the node handles the new request by updating $maxN$, executing the new request, generating the node's share of the full reply certificate, storing the partial reply certificate in $reply_c$, and sending the partial reply certificate to all nodes in the agreement cluster; or (2) if $t \leq t'$, then the node handles the client-initiated retransmission request by updating $maxN$ and generating, caching, and sending a new partial reply certificate containing the cached timestamp t', the cached reply body r', the request's view v, and the request's sequence number n. On the other hand, if the request has an old sequence number (i.e., $n \leq maxN$), the node simply resends the partial reply certificate in $reply_c$, which is guaranteed to have a sequence number at least as large as the request's sequence number.

3.3.1 Liveness and retransmission

To eliminate gaps in the sequence number space caused by the unreliable communication between the agreement and execution clusters, the system uses a two-level retransmission strategy. For a request with sequence number n, retransmissions by the agreement cluster ensure that eventually at least one correct execution node receives and executes request n, and an internal execution cluster retransmission protocol ensures once that happens, all correct execution nodes eventually learn of request n or of some stable checkpoint newer than n. In particular, if an execution node receives request n but not request $n-1$, it multicasts to other nodes in the execution cluster a retransmission request for the missing sequence number. When a node receives such a message, it replies with the specified request and agreement certificates unless it has a stable checkpoint with a higher sequence number, in which case it sends the proof of stability for that checkpoint (see below.)

3.3.2 Checkpoints and garbage collection

The execution nodes periodically construct checkpoints to allow them to garbage collect their pending request logs. Note that the inter-cluster protocol is designed so that garbage collection in the execution cluster requires no additional coordination with the agreement cluster and vice versa. Execution nodes generate checkpoints at prespecified sequence numbers (e.g., after executing request n where n mod $CP\_FREQ = 0$). Nodes checkpoint both the application state and their $reply_c$ list of replies to clients, but they do not include their pending request list in checkpointed state. As in previous systems [11, 12, 40], to reduce the cost of producing checkpoints, nodes can make use of copy on write and incremental cryptography [5].

After generating a checkpoint, execution servers assemble a proof of stability for it. When server i produces checkpoint C for sequence number n, it computes a digest of the checkpoint $d = D(C)$ and authenticates its view of the checkpoint to the rest of the cluster by multicasting $\langle CHECKPOINT, n, d\rangle_{i,\mathcal{E},1}$ to all execution nodes. Once a node receives $g+1$ such messages, it assembles them into a full checkpoint certificate:$\langle CHECKPOINT, n, d\rangle_{\mathcal{E},\mathcal{E},g+1}$

Once a node has a valid and complete checkpoint certificate for sequence number n, it can garbage collect state by discarding older checkpoints, discarding older checkpoint certificates, and discarding older agreement and request certificates from the pending request log.

3.4 Correctness

Due to space constraints, we outline our main results here and defer the proof to an extended technical report [51]. We show that the high level protocol provides safety and liveness if the clusters behave as specified. The node actions specified in our agreement cluster and execution cluster designs ensure that the clusters discharge their requirements.

The system[2] is safe in that a client receives a reply $\langle REPLY, v, n, t, c, \mathcal{E}, r\rangle_{\mathcal{E},c,g+1}$ only if (a) earlier the client issued a request $\langle REQUEST, o_n, t, c\rangle_{c,\mathcal{A},1}$, (b) the reply value r reflects the output of state machine in state Q_{n-1} executing request operation o_n, (c) there exists some set of operations $\mathcal{O}$ such that state Q_{n-1} is the state reached by starting at initial state Q_0 and sequentially executing each request o_i $(0 \leq i < n)$ in $\mathcal{O}$ as the ith operation on the state machine, and (d) a valid reply certificate for any subsequent request reflects a state in which the execution of o_i is the i'th action by the state machine $(0 \leq i \leq n)$.

This safety follows from the fact that the agreement cluster only generates agreement certificates for valid client requests and never assigns the same sequence number to two different requests. Then, the execution cluster only executes requests in the order specified by the agreement certificates.

The system is live in that if a client c sends a request with timestamp t, where timestamp t exceeds any timestamp in any previous request by c, and the client repeatedly sends that request and no other request until it receives a reply, then eventually it will receive a valid reply for that request.

This liveness follows from the liveness of the underlying agreement subsystem, BASE. Once the agreement cluster assigns a sequence number to a request, the execution cluster must eventually receive and execute it. After that, if the client keeps sending its request, eventually it will receive the reply.

4. PRIVACY FIREWALL

Traditional BFT systems face a fundamental tradeoff between increasing availability and integrity on the one hand and strengthening confidentiality on the other. Increasing diversity across replicas (e.g., increasing the number of replicas or increasing the heterogeneity across implementations of different replicas [2, 25, 48]) improves integrity and availability because it reduces the chance that too many replicas simultaneously fail. Unfortunately, it also increases the chance that at least one replica contains an exploitable bug. An attacker may gain access to the confidential data by attacking the weakest replica.

Compounding this problem, as Figure 2(a) illustrates, traditional replicated state machine architectures delegate the responsibility of combining the state machines' outputs to a voter at the client. Fate sharing between the client and the voter ensures that the voter does not introduce a new single point of failure; to quote Schneider [43], *"the voter—a part of the client—is faulty exactly when the client is, so the fact that an incorrect output is read by the client due to a faulty voter is irrelevant"* because a faulty voter is then synonymous with a faulty client.

Although such a model is appropriate for services where availability and integrity are the goals, it fails to address confidentiality for access-anywhere services. In particular, if an attacker manages to compromise one replica in such a system, the compromised replica may send confidential data

[2]Here we describe the properties of a system whose agreement cluster includes a cache ($cache_c$) storing the reply to client c's highest-timestamped request. If this optional cache is not included, the property is similar but more complex in that replies to retransmissions may contain a sequence number n that exceeds the sequence number m that was used to serialize execution.

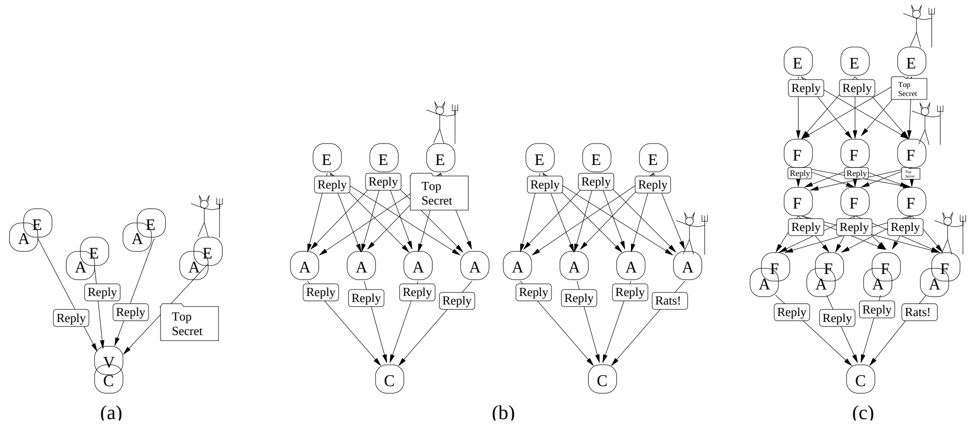

Figure 2: Illustration of confidentiality filtering properties of (a) traditional BFT architectures, (b) architectures that separates agreement and execution, and (c) architectures that separate agreement and execution and that add additional privacy firewall nodes.

back to the attacker.

Solving this problem seems difficult. If we move the voter away from the client, we lose fate sharing, and the voter becomes a single point of failure. And, it is not clear how to replicate the voter to eliminate this single point of failure without miring ourselves in endless recursion ("Who votes on the voters?").

As illustrated by Figure 2(b), the separation of agreement from execution provides the opportunity to reduce a system's vulnerability to compromising confidentiality by having the agreement nodes filter incorrect replies before sending reply certificates to clients. It now takes a failure of both an agreement node and an execution node to compromise privacy if we restrict communications so that (1) clients can communicate with agreement nodes but not execution nodes and (2) request and reply bodies are encrypted so that clients and execution nodes can read them but agreement nodes cannot. In particular, if all agreement nodes are correct, then the agreement nodes can filter replies so that only correct replies reach the client. Conversely, if all execution nodes are correct, then faulty agreement nodes can send information to clients, but not information regarding the confidential state of the state machine.

Although this simple design improves confidentiality, it is not entirely satisfying. First, it can not handle multiple faults: a single fault in both the agreement and execution clusters can allow confidential information to leak. Second, it allows an adversary to leak information via steganography, for instance by manipulating membership sets in agreement certificates.

In the rest of this section, we describe a general confidentiality filter architecture—the *privacy firewall*. If the agreement and execution clusters have a sufficient number of working machines, then a privacy firewall of $h+1$ rows of $h+1$ filters per row can tolerate up to h faults while still providing availability, integrity, and confidentiality. We first define the protocol. We then explain the rationale behind specific design choices. Finally, we state the end-to-end confidentiality guarantees provided by the system, highlight the limitations of these guarantees, and discuss ways to strengthen these guarantees.

4.1 Protocol definition

Figure 2(c) shows the organization of the privacy firewall. We insert *filter nodes* F between execution servers E and agreement nodes A to pass only information sent by correct execution servers. Filter nodes are arranged into an array of $h+1$ rows of $h+1$ columns; if the number of agreement nodes is at least $h+1$, then the bottom row of filters can be merged with the agreement nodes by placing a filter on each server in the agreement cluster. Information flow is controlled by restricting communication to only the links shown in Figure 2(c). Each filter node has a physical network connection to all filter nodes in the rows above and below but no other connections. Request and reply bodies (the o and r fields in the request and reply certificates defined above) are encrypted so that the client and execution nodes can read them but agreement nodes and firewall nodes cannot.

Each filter node maintains $maxN$, the maximum sequence number in any valid agreement certificate or reply certificate seen, and $state_n$, information relating to sequence number n. $State_n$ contains *null* if request n has not been seen, contains *seen* if request n has been seen but reply n has not, and contains a reply certificate if reply n has been seen. Nodes limit the size of $state_n$ by discarding any entries whose sequence number is below $maxN - P$ where P is the pipeline depth that bounds the number of requests that the agreement cluster can have outstanding (see Section 3.1.2).

When a filter node receives from below a valid request certificate and agreement certificate with sequence number n, it ignores the request if $n < maxN - P$. Otherwise, it first updates $maxN$ and garbage collects entries in *state* with sequence numbers smaller than $maxN - P$. Finally it sends one of two messages. If $state_n$ contains a reply certificate, the node multicasts the stored reply to the row of filter nodes or agreement nodes below. But, if $state_n$ does not yet contain the reply, the node sets $state_n = seen$ and multicasts the request and agreement certificates to the next row above. As an optimization, nodes in all but the top row of filter nodes can unicast these certificates to the one node above them rather than multicasting.

Although request and agreement certificates flowing up can use any form of certificate including MAC-based authenticators, filter nodes must use threshold cryptography [16] for reply certificates they send down. When a filter node in the top row receives $g + 1$ partial reply certificates signed by different execution nodes, it assembles a complete reply certificate authenticated by a single threshold signature representing the execution nodes' split group key. Then, after a top-row filter node assembles such a complete reply certificate with sequence number n or after any other filter node receives and cryptographically validates a complete reply certificate with sequence number n, the node checks $state_n$. If $n < maxN - P$ then the reply is too old to be of interest, and the node drops it; if $state_n = seen$, the node multicasts the reply certificate to the row of filter nodes or agreement nodes below and then stores the reply in $state_n$; or if $state_n$ already contains the reply or is empty, the node stores reply certificate n in $state_n$ but does not multicast it down at this time.

The protocol described above applies to any deterministic state machine. We describe in Section 3.1.4 how agreement nodes pick a timestamp and random bits to obliviously transform non-deterministic state machines into deterministic ones without having to look at the content of requests. Note that this approach may allow agreement nodes to infer something about the internal state of the execution cluster, and balancing confidentiality and non-determinism in its full generality appears hard. To prevent the agreement cluster from even knowing what random value is used by the execution nodes, execution nodes could cryptographically hash the input value with a secret known only to the execution cluster; we have not yet implemented this feature. Still, a compromised agreement node can determine the time that a request enters the system, but as Section 4.3 notes, that information is already available to agreement nodes.

4.2 Design rationale

The privacy firewall architecture provides confidentiality through the systematic application of three key ideas: (1) redundant filters to ensure filtering in the face of failures, (2) elimination of non-determinism to prevent explicit or steganographic communication through a correct filter, and (3) restriction of communication to enforce filtering of confidential data sent from the execution nodes.

4.2.1 Redundant filters

The array of $h + 1$ rows of $h + 1$ columns ensures the following two properties as long as there are no more than h failures: (i) there exists at least one *correct path* between the agreement nodes and execution nodes consisting only of correct filters and (ii) there exists one row (the *correct cut*) consisting entirely of correct filter nodes.

Property (i) ensures availability by guaranteeing that requests can always reach execution nodes and replies can always reach clients. Observe that availability is also necessary for preserving confidentiality, because a strategically placed rejected request could be used to communicate confidential information by introducing a termination channel [42].

Property (ii) ensures a faulty node can either access confidential data or communicate freely with clients but not both. Faulty filter nodes above the correct cut might have access to confidential data, but the filter nodes in the correct cut ensure that only replies that would be returned by a correct server are forwarded. And, although faulty nodes below the correct cut might be able to communicate any information they have, they do not have access to confidential information.

4.2.2 Eliminating non-determinism

Not only does the protocol ensure that a correct filter node transmits only correct replies (vouched for by at least $g + 1$ execution nodes), it also eliminates nondeterminism that an adversary could exploit as a covert channel by influencing nondeterministic choices.

The contents of each reply certificate is a deterministic function of the request and sequence of preceeding requests. The use of threshold cryptography makes the encoding of each reply certificate deterministic and prevents an adversary from leaking information by manipulating certificate membership sets. The separation of agreement from execution is also crucial for confidentiality: agreement nodes outside the privacy firewall assign sequence numbers so that the non-determinism in sequence number assignment can not be manipulated as a covert channel for transmitting confidential information.

In addition to these restrictions to eliminate non-determinism in message bodies, the system also restricts (but as Section 4.3 describes, does not completely eliminate) non-determinism in the network-level message retransmission. The per-request *state* table allows filter nodes to remember which requests they have seen and send at most one (multicast) reply per request message. This table reduces the ability of a compromised node to affect the number of copies of a reply certificate that a downstream firewall node sends.

4.2.3 Restricting communication

The system restricts communication by (1) physically connecting firewall nodes only to the nodes directly above and below them and (2) encrypting the bodies of requests and replies. The first restriction enforces the requirement that all communication between execution nodes and the outside world flow through at least one correct firewall. The second restriction prevents nodes below the correct cut of firewall nodes from accumulating and revealing confidential state by observing the bodies of requests and replies.

4.3 Filter properties and limitations

In an extended technical report [51] we prove that there exists a *correct cut* of firewall nodes through which all information communicated from the execution servers passes and that this correct cut provides *output set confidentiality* in that any sequence of outputs of our correct cut is also a legal sequence of outputs of a correct unreplicated implementation of the service accessed via an asynchronous unreliable network that can discard, delay, replicate, and reorder messages. More formally, suppose that C is a correct unreplicated implementation of a service, S_0 is the abstract [40] initial state, I is a sequence of input requests, and O the resulting sequence of output replies transmitted on an asynchronous unreliable network to a receiver. The network can delay, replicate, reorder, and discard these replies; thus the receiver observes an output sequence O' that belongs to a set of output sequences $\mathcal{O}$, where each O' in $\mathcal{O}$ includes only messages from O. Because the correct cut of our replicated system is output set confidential with respect to C, then given the same initial abstract state S_0 and input I its output O'' also belongs to $\mathcal{O}$. This property follows from the operation of each firewall node, which ensures that each output message is a deterministic function of the preceeding inputs, and from the firewall replication and topology, which ensures that a correct cut filters all requests and replies (safety) and that the correct path between agreement nodes and execution nodes is always available (liveness).

Because our system replicates arbitrary black-box state machines, the above definition describes confidentiality with respect to the behavior of a single correct server. The definition does not specify anything about the internal behaviors of or the policies enforced by the replicated state machine, so it is more flexible and less strict than the notion of *non-interference* [42], which is sometimes equated with confidentiality in the literature and which informally states that the observable output of the system has no correlation to the confidential information stored in the system. Our output set confidentiality guarantee is similar in spirit to the notion of possibilistic non-interference [35], which characterizes the behavior of a nondeterministic program by the *set* of possible results and requires that the set of possible observable outputs of a system be independent of the confidential state stored in the system.

A limitation is that although agreement nodes do not have access to the body of requests, they do need access to the identity of the client (in order to buffer information about each client's last request), the arrival times of the requests and replies, and the encrypted bodies of requests and replies. Faulty agreement or filter nodes in our system could leak information regarding traffic patterns. For example, a malicious agreement node could leak the frequency that a particular client sends requests to the system or the average size of a client's requests. Techniques such as forwarding through intermediaries and padding messages can reduce a system's vulnerability to traffic analysis [13], though forwarding can add significant latencies and significant message padding may be needed for confidentiality [46].

Also note that although output set confidentiality ensures that the set of output messages is a deterministic function of the sequence of inputs, the nondeterminism in the timing, ordering, and retransmission of messages might be manipulated to furnish covert channels that communicate information from above the correct cut to below it. For example, a compromised node directly above the correct cut of firewalls might attempt to influence the timing or sequencing of replies forwarded by the nodes in the correct cut by forwarding replies more quickly or less quickly than its peers, sending replies out of order, or varying the number of times it retransmits a particular reply. Given that any resulting output sequence and timing is a "legal" output that could appear in an asynchronous system with a correct server and an unreliable network, it appears fundamentally difficult for firewall nodes to completely eliminate such channels.

It may, however, be possible to systematically restrict such channels by engineering the system to make it more difficult for an adversary to affect the timing, ordering, and replication of symbols output by the correct cut. The use of the *state* table to ensure that each reply is multicast at most once per request received is an example of such a restriction. This rule makes it more difficult for a faulty node to encode information in the number of copies of a reply sent through the correct cut and approximates a system where the number of times a reply is sent is a deterministic function of the number of times a request is sent. But, with an asynchronous unreliable network, this approximation is not perfect—a faulty firewall node can still slightly affect the probability that a reply is sent and therefore can slightly affect the expected number of replies sent per request (e.g., not sending a reply slightly increases the probability that all copies sent to a node in the correct cut are dropped; sending a reply multiple times might slightly reduce that probability). Also note that for simplicity the protocol described above does not use any additional heuristic to send replies in sequence number order, though similar processing rules could be added to make it more difficult (though not impossible in an asynchronous system) for a compromised node to cause the correct cut to have gaps or reorderings in the sequence numbers of replies it forwards.

Restricting the nondeterminism introduced by the network seems particularly attractive when a firewall network is deployed in a controlled environment such as a machine room. For example, if the network can be made to deliver messages reliably and in order between correct nodes, then the correct cut's output sequence can always follow sequence number order. In the limit, if timing and message-delivery nondeterminism can be completely eliminated, then covert channels that exploit network nondeterminism can be eliminated as well. In the extended technical report [51], we describe a variation of the protocol that assumes that agreement nodes and clients continue to operate under the asynchronous model but that execution and firewall nodes operate under a synchronous model with a time bound on state machine processing, firewall processing, and message delivery between correct nodes. This protocol variation extends the *state* table to track when requests arrive and uses this information and system time bounds to restrict when replies are transmitted. Then, as long as the time bounds are met by correct nodes and links between correct nodes, the system is fully confidential with respect to a single correct server in that the only information output by the correct cut of firewall nodes is the information that would be output by a single correct server. If, on the other hand, the timing bounds are violated, then the protocol continues to be safe and live and provides output symbol set confidentiality.

5. EVALUATION

In this section, we experimentally evaluate the latency, overhead, and throughput of our prototype system under microbenchmarks. We also examine the system's performance acting as a network file system (NFS) server.

5.1 Prototype implementation

We have constructed a prototype system that separates agreement and replication and that optionally provides a redundant privacy firewall. As described above, our prototype implementation builds on Rodrigues et al.'s BASE library [40].

Our evaluation cluster comprises seven 933Mhz Pentium-III and two 500MHz Pentium-III machines, each with 128MB of memory. The machines run Redhat Linux 7.2 and are connected by a 100 Mbit ethernet hub.

Note that three aspects of our configuration would not be appropriate for production use. First, both the underlying BASE library and our system store important persistent data structures in memory and rely on replication across machines to ensure this persistence [3, 11, 14, 32]. Unfortunately, the machines in our evaluation cluster do not have

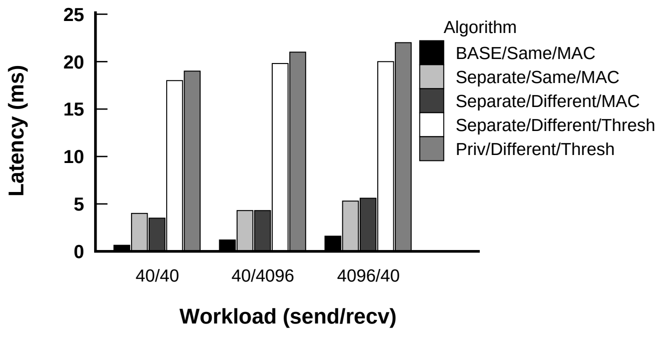

Figure 3: Latency for null-server benchmark for three request/reply sizes.

uninterruptible power supplies, so power failures are a potentially significant source of correlated failures across our system that could cause our current configuration to lose data. Second, our privacy firewall architecture assumes a network configuration that physically restricts communication paths between agreement machines, privacy filter machines, and execution machines. Our current configuration uses a single 100 Mbit ethernet hub and does not enforce these restrictions. We would not expect either of these differences to affect the results we report in this section. Third, to reduce the risk of correlated failures the nodes should be running different operating systems and different implementations of the agreement, privacy, and execution cluster software. We only implemented these libraries once and we use only one version of the application code.

5.2 Latency

Past studies have found that Byzantine fault tolerant state machine replication adds modest latency to network applications [11, 12, 40]. Here, we examine the same latency microbenchmark used in these studies. Under this microbenchmark, the application reads a request of a specified size and produces a reply of a specified size with no additional processing. We examine request/reply sizes of 40 bytes/40 bytes, 40 bytes/4 KB, and 4 KB/40 bytes.

Figure 3 shows the average latency (all run within 5%) for ten runs of 200 requests each. The bars show performance for different system configurations with the *algorithm/machine configuration/authentication algorithm* indicated in the legend. BASE/Same/MAC is the BASE library with 4 machines hosting both the agreement and execution servers and using MAC authenticators; Separate/-Same/MAC shows our system that separates agreement and replication with agreement running on 4 machines and with execution running 3 of the same set of machines and using MAC authenticators; Separate/Different/MAC moves the execution servers to 3 machines physically separate from the 4 agreement servers; Separate/Different/Thresh uses the same configuration but uses threshold signatures rather than MAC authenticators for reply certificates; finally, Priv/Different/Thresh adds an array of privacy firewall servers between the agreement and execution cluster with a bottom row of 4 privacy firewall servers sharing the agreement machines and an additional row of 2 firewall servers separate from the agreement and execution machines.

The BASE library imposes little latency on requests, with request latencies of 0.64ms, 1.2ms, and 1.6ms for the three workloads. Our current implementations of the library that separates agreement from replication has higher latencies when running on the same machines—4.0ms, 4.3ms, and 5.3ms. The increase is largely caused by two inefficiencies in our current implementation: (1) rather than using the agreement certificate produced by the BASE library, each of our message queue nodes generates a piece of a new agreement certificate from scratch, (2) in our current prototype, we do a full all-to-all multicast of the agreement certificate and request certificate from the agreement nodes to the execution nodes, of the reply certificate from the execution nodes to the agreement nodes, and (3) our system does not use hardware multicast. We have not implemented the optimizations of first having one node send and having the other nodes send only if a timeout occurs, and we have not implemented the optimization of clients sending requests directly to the execution nodes. However, we added the optimization that the execution nodes send their replies directly to the clients. Separating the agreement machines from the execution machines adds little additional latency. But, switching from MAC authenticator certificates to threshold signature certificates increases latencies to 18ms, 19ms, and 20ms for the three workloads. Adding a two rows of privacy firewall filters (one of which is co-located with the agreement nodes) adds a few additional milliseconds.

As expected, the most significant source of latency in the architecture is public key threshold cryptography. Producing a threshold signature takes 15ms and verifying a signature takes 0.7ms on our machines. Two things should be noted to put these costs in perspective. First, the latency for these operations is comparable to I/O costs for many services of interest; for example, these latency costs are similar to the latency of a small number of disk seeks and are similar to or smaller than wide area network round trip latencies. Second, signature costs are expected to fall quickly as processor speeds increase; the increasing importance of distributed systems security may also lead to widespread deployment of hardware acceleration of encryption primitives. The FARSITE project has also noted that technology trends are making it feasible to include public-key operations as a building block for practical distributed services [1].

5.3 Throughput and cost

Although latency is an important metric, modern services must also support high throughput [49]. Two aspects of the privacy firewall architecture pose challenges to providing high throughput at low cost. First, the privacy firewall architecture requires a larger number of physical machines in order to to physically restrict communication. Second, the privacy firewall architecture relies on relatively high-overhead public key threshold signatures for reply certificates. Two factors mitigate these costs.

First, although the new architecture can increase the total number of machines, it also can reduce the number of application-specific machines required. Application-specific machines may be more expensive than generic machines both in terms of hardware (e.g., they may require more storage, I/O, or processing resources) and in terms of software (e.g., they may require new versions of application-specific software.) Thus, for many systems we would expect the application costs (e.g., the execution servers) dominate. Like router and switch box costs today, agreement node and privacy filter boxes may add a relatively modest amount to overall system cost. Also, although filter nodes must run on $(h+1)^2$ nodes (and this is provably the minimal number to ensure confidentiality [51]), even when the privacy firewall architecture is used, the number of machines is relatively modest when the goal is to tolerate a small number of faults. For example, to tolerate up to one failure among the execution nodes and one among either the agreement or privacy filter servers, the system would have four generic agreement and privacy filter machines, two generic privacy filter machines, and three application-specific execution machines. Finally, in configurations without the privacy firewall, the total number of machines is not necessarily increased since the agreement and execution servers can occupy the same physical machines. For example, to tolerate one fault, four machines can act as agreement servers while three of them also act as execution replicas.

Second, a better metric for evaluating the hardware costs of the system than the number of machines is the overhead imposed on each request relative to an unreplicated system. On the one hand, by cleanly separating agreement from execution and thereby reducing the number of execution replicas a system needs, the new architecture often reduces this overhead compared to previous systems. On the other hand, the addition of privacy firewall filters and their attendant public key encryption add significant costs. Fortunately, these costs can be amortized across batches of requests. In particular, when load is high the BASE library on which we build bundles together requests and executes agreement once per bundle rather than once per request. Similarly, by sending bundles of requests and replies through the privacy firewall nodes, we allow the system to execute public key operations on bundles of replies rather than individual replies.

To put these two factors in perspective, we consider a simple model that accounts for the application execution costs and cryptographic processing overheads across the system (but not other overheads like network send/receive.) The *relativeCost* of executing a request is the cost of executing the request on a replicated system divided by the cost of executing the request on an unreplicated system. For our system and the BASE library,

$$relativeCost = \frac{numExec \cdot proc_{app} + overhead_{req} + \frac{overhead_{batch}}{numPerBatch}}{proc_{app}}.$$

The cryptographic processing overhead has three flavors: MAC-based authenticators, public threshold-key signing, and public threshold-key verifying. To tolerate 1 fault, the BASE library requires 4 execution replicas, and it does 8 MAC operations per request[3] and 36 MAC operations per batch. Our architecture that separates agreement from replication requires 3 execution replicas and does 7 MAC operations per request and 39 MAC operations per batch[4]. Our privacy firewall architecture requires 3 execution replicas and does 7 MAC operations per request and 39/3/6 MAC operations/public key signatures/public key verifications per batch.

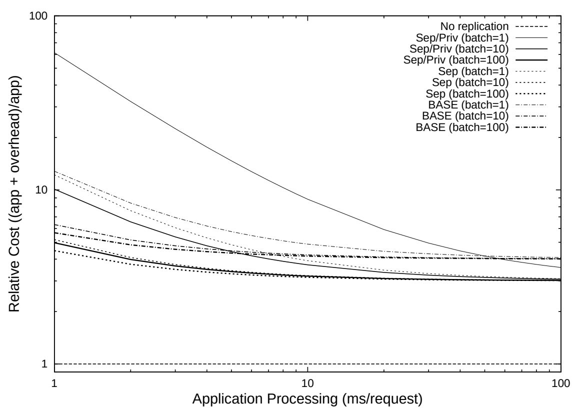

Figure 4: Estimated relative processing costs including application processing and cryptographic overhead for an unreplicated system, privacy firewall system, separate agreement and replication system, and BASE system for batch sizes of 1, 10, and 100 requests/batch.

Given these costs, the lines in Figure 4 show the relative costs for BASE (dot-dash lines), separate agreement and replication (dotted lines), and privacy firewall (solid lines) for batch sizes of 1, 10, and 100 requests/batch. The (unreplicated) application execution time varies from 1ms per request to 100ms per request on the x axis. We assume that MAC operations cost 0.2ms (based on 50MB/s secure hashing of 1KB packets), public key threshold signatures cost 15ms (as measured on our machines for small messages), and public key verification costs 0.7ms (measured for small messages.)

Without the privacy firewall overhead, our separate architecture has a lower cost than BASE for all request sizes examined. As application processing increase, application processing dominates, and the new architectures gain a 33% advantage over the BASE architecture. With small requests and without batching, the privacy firewall does greatly increase cost. But with batch sizes of 10 (or 100), processing a request under the privacy firewall architecture costs less than under BASE replication for applications whose requests take more than 5ms (or 0.2ms).

The simple model discussed above considers only encryption operations and application execution and summarizes total overhead. We now experimentally evaluate the peak throughput and load of our system. In order to isolate the overhead of our prototype, we evaluate the performance of the system when executing a simple Null server that receives 1 KB requests and returns 1 KB replies with no additional application processing. We program a set of clients to issue requests at a desired frequency and vary that frequency to vary the load on the system.

Figure 5 shows how the latency for a given load varies with bundle size. When bundling is turned off, throughput is limited to 62 requests/second, at which point the execution servers are spending nearly all of their time signing replies. Doubling the bundle size to 2 approximately doubles the throughput. Bundle sizes of 3 or larger give peak

[3]Note that when authenticating the same message to or from a number of nodes the work of computing the digest on the body of a message can be re-used for all communication partners [11, 12]. For the small numbers of nodes involved in our system, we therefore charge 1 MAC operation per message processed by a node regardless of the number of sources it came from or destinations it goes to.

[4]Our unoptimized prototype does 44 MAC operations per batch both with and without the privacy firewall.

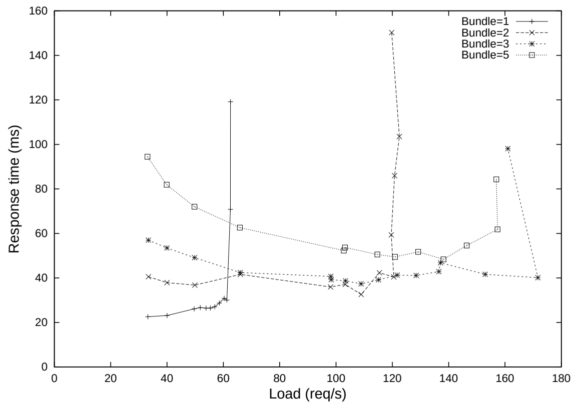

Figure 5: Microbenchmark response time as offered load and request bundling varies.

Phase	No Replication	BASE	Firewall
1	7	7	19
2	225	598	1202
3	239	1229	862
4	536	1552	1746
5	3235	4942	5872
TOTAL	4244	8328	9701

Figure 6: Andrew-500 benchmark times in seconds.

throughputs of 160-170 requests/second; beyond this point, the system is I/O limited and the servers have idle capacity. For example, with a bundle size of 10 and a load of 160 requests/second, the CPU utilization of the most heavily loaded execution machine is 30%. Note that our current prototype uses a static bundle size, so increasing bundle sizes increases latency at low loads. The existing BASE library limits this problem by using small bundles when load is low and increasing bundle sizes as load increases. Our current prototype uses fixed-sized bundles to avoid the need to adaptively agree on bundle size; we plan to augment the interface between the BASE library and our message queues to pass the bundle size used by the BASE agreement cluster to the message queue.

5.4 Network file system

For comparison with previous studies [11, 12, 40], we examine a replicated NFS server under the modified Andrew500 benchmark, which sequentially runs 500 copies of the Andrew benchmark [11, 21]. The Andrew benchmark has 5 phases: (1) recursive subdirectory creation, (2) copy source tree, (3) examine file attributes without reading file contents, (4) reading the files, and (5) compiling and linking the files.

We use the NFS abstraction layer by Rodrigues, et al. to resolve nondeterminism by having the primary agreement node supply timestamps for modifications and file handles for newly opened files. We run each benchmark 10 times and report the average for each configuration. In these experiments, we assume hardware support in performing efficient threshold signature operations [45].

Figure 6 summarizes these results. Performance for the benchmark is largely determined by file system latency, and

Phase	BASE	faulty server	faulty ag. node
1	12	19	33
2	1426	1384	1553
3	1196	1010	1102
4	1755	1898	2180
5	5374	6050	7071
TOTAL	9763	10361	11939

Figure 7: Andrew-500 benchmark times in seconds with failures.

our firewall system's performance is about 16% slower than BASE. Also note that BASE is more than a factor of two slower than the no replication case; this difference is higher than the difference reported in [40] where a 31% slowdown was observed. We have worked with the authors of [40] and determined that much of the difference can be attributed to different versions of BASE and Linux used in the two experiments.

Figure 7 shows the behavior of our system in the presence of faults. We obtained it by stopping a server or an agreement node at the beginning of the benchmark. The table shows that the faults only have a minor impact on the completion time of the benchmark.

6. RELATED WORK

Byzantine agreement [31] and Byzantine fault tolerant state machine replication has been studied in both theoretical and practical settings [7, 10, 20, 24, 38, 43]. The recent work of Castro, Liskov, and Rodrigues [11, 12, 40] has brought new impulse to research in this area by showing that BFT systems can be practical. We use their BASE library [40] as the foundation of our agreement protocol, but depart significantly from their design in one key respect: our architecture explicitly separates the responsibility of achieving agreement on the order of requests from the processing the requests once they are ordered. Significantly, this separation allows us to reduce by one third the number of application-specific replicas needed to tolerate f Byzantine failures and to address confidentiality together with integrity and availability.

In a recent paper [30], Lamport deconstructs his Paxos consensus protocol [29] by explicitly identifying the roles of three classes of agents in the protocol: *proposers*, *acceptors*, and *learners*. He goes on to present an implementation of the state machine approach in Paxos in which "each server plays all the roles (proposer, acceptor and learner)". We propose a similar deconstruction of the state machine protocol: in Paxos parlance, our clients, agreement servers, and execution servers are performing the roles played, respectively, by proposers, acceptors, and learners. However, our acceptors and learners are physically, and not just logically, distinct. We show how to apply this principle to BFT systems to reduce replication cost and to provide confidentiality.

Our system also shares some similarities with the systems [6, 37] using stateless witness to improve fault tolerance. However, our system differs in two respects. First, our system is designed to tolerate Byzantine faults instead of fail-stop failures. Second, our general technique replicates arbitrary state machines instead of specific applications such as voting and file systems.

Baldoni, Marchetti, and Tucci-Piergiovanni advocate a three-tier approach to replicated services, in which the replication logic is embedded within a software middle-tier that sits between clients and end-tier application replicas [4]. Their goals are to localize to the middle tier the need of assuming a timed-asynchronous model [15], leaving the application replicas to operate asynchronously, and to enable the possibility of modifying on the fly the replication logic of end-tier replicas (for example, from active to passive replication) without affecting the client.

In a recent workshop extended abstract [50] we sketched an early design of our privacy firewall. We move beyond that work in two ways. First, we propose a different overall architecture for achieving confidentiality in BFT services. Instead of placing the privacy firewall between the client and the BFT service, we now exploit the separation of agreement from execution and position the firewall between the two clusters. This change is important because it eliminates a potential covert channel by preventing nodes with confidential information from affecting the order in which requests are processed. Second, we report on our experience building a prototype of our system.

Most previous efforts to achieve confidentiality despite server failures restrict the data that servers can access. A number of systems limit servers to basic "store" and "retrieve" operations on encrypted data [1, 26, 33, 34, 41] or on data fragmented among servers [19, 22]. The COCA [52] online certification authority uses replication for availability, and threshold cryptography [16] and proactive secret sharing [23] to digitally sign certificates in a way that tolerates adversaries that compromise some of the servers. In general, preventing servers from accessing confidential state works well when servers can process the fragments independently or when servers do not perform any significant processing on the data. Our architecture provides a more general solution that can implement arbitrary deterministic state machines.

Secure multi-party computation (SMPC) [9] allows n players to compute an agreed function of their inputs in a secure way even when some players cheat. Although in theory it provides a foundation for achieving Byzantine fault tolerant confidentiality, SMPC in practice can only be used to compute simple functions such as small-scale voting and bidding because SMPC relies heavily on computationally expensive oblivious transfers [17].

Firewalls that restrict incoming requests are a common pragmatic defence against malicious attackers. Typical firewalls prevent access to particular machines or ports; more generally, firewalls could identify improperly formatted or otherwise illegal requests to an otherwise legal machine and port. In principle, firewalls could protect a server by preventing all "bad" requests from reaching it. An interesting research question is whether identifying all "bad" requests is significantly easier than building bug-free servers in the first place. The privacy firewall is inspired by the idea of mediating communication between the world and a service, but it uses redundant execution to filter mismatching (and presumptively wrong) outgoing replies rather than relying on *a priori* identification of bad incoming requests.

7. CONCLUSION

The main contribution of this paper is to present the first study to systematically apply the principle of separation of agreement and execution to BFT state machine replication to (1) reduce the replication cost and (2) enhance confidentiality properties for general Byzantine replicated services. Although in retrospect this separation is straightforward, all previous general BFT state machine replication systems have tightly coupled agreement and execution, have paid unneeded replication costs, and have increased system vulnerability to confidentiality breaches.

8. ACKNOWLEDGMENTS

This work was supported in part by the Texas Advanced Research Program, Cisco, and an IBM University Partnership Award. Alvisi was supported by the National Science Foundation (CAREER award CCR-9734185), an Alfred P. Sloan Fellowship, and the AFRL/Cornell Information Assurance Institute. Dahlin was also supported by a Sloan Research Fellowship. Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov have kindly provided their code for our experiments and spent much time in helping us to understand their code. We would like to thank Fred B. Schneider for many insightful discussions on the nature of state machine replication, and Michael Clarkson, Barbara Liskov, Keith Marzullo, Andrew Myers, Rodrigo Rodrigues, Steve Zdancewic, and the anonymous reviewers for valuable suggestions on how to improve the quality of this paper. Alvisi also would like to thank the support and hospitality of Cornell's Information Assurance Institute.

9. REFERENCES

[1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated available and reliable storage for incompletely trusted environments. In *5th Symp on Operating Systems Design and Impl.*, December 2002.

[2] P. Ammann and J. Knight. Data diversity: An approach to software fault tolerance. *IEEE Trans. Comput.*, 37(4):418–425, 1988.

[3] M. Baker. *Fast Crash Recovery in Distributed File Systems.* PhD thesis, University of California at Berkeley, 1994.

[4] R. Baldoni, C. Marchetti, and S. Piergiovanni. Asynchronous active replication in three-tier distributed systems, 2002.

[5] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementally at reduced cost. In *Eurocrypt97*, 1997.

[6] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Palo Alto, CA, USA, 10 1993.

[7] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. of the ACM*, 32(4):824–840, October 1985.

[8] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. In *ACM Trans. on Computer Systems*, pages 18–36, February 1990.

[9] R. Canetti. *Studies in Secure Multiparty Computation and Applications.* PhD thesis, Weizmann Institute of Science, 1995.

[10] R. Canetti and T. Rabin. Optimal Asynchronous Byzantine Agreement. Technical Report 92-15, Dept. of Computer Science, Hebrew University, 1992.

[11] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *3rd Symp. on Operating Systems Design and Impl.*, February 1999.

[12] M. Castro and B. Liskov. Proactive recovery in a Byzantine-Fault-Tolerant system. In *4th Symp. on Operating Systems Design and Impl.*, pages 273–288, 2000.

[13] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. In *Comm. of the ACM*, volume 24, pages 84–90, February 1981.

[14] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System

Crashes. In *7th Internat. Conf. on Arch. Support for Programming Languages and Operating Systems*, October 1996.

[15] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[16] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Advances in Cryptology: Crypto89, the Ninth Annual International Cryptology Conference*, pages 307–315, 1990.

[17] S. Even, O. Goldreich, and A. Lempel. A Randomized Protocol for Signing Contracts. *Comm. of the ACM*, 28:637–647, 1985.

[18] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed commit with one faulty process. *J. of the ACM*, 32(2), April 1985.

[19] J. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2):363–389, July 2000.

[20] J. Garay and Y. Moses. Fully Polynomial Byzantine Agreement for $n>3t$ Processors in $t+1$ Rounds. *SIAM Journal of Compouting*, 27(1), 1998.

[21] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Trans. on Computer Systems*, 6(1):51–81, February 1988.

[22] A. Iyengar, R. Cahn, C. Jutla, and J. Garay. Design and Implementation of a Secure Distributed Data Repository. In *Proc. of the 14th IFIP Internat. Information Security Conf.*, pages 123–135, 1998.

[23] R. Karp, M. Luby, and F. auf der Heide. Proactive Secret Sharing or How to Cope with Perpetual Leakage. In *Proc. of the 15th Annual Internat. Cryptology Conf.*, pages 457–469, 1995.

[24] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The securering protocols for securing group communication. In *Hawaii International Conference on System Sciences*, 1998.

[25] J. Knight and N. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. Softw. Eng.*, 12(1):96–109, 1986.

[26] J. Kubiatowicz, 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 *9th Internat. Conf. on Arch. Support for Programming Languages and Operating Systems*, November 2000.

[27] S. Kumar and K. Li. Using model checking to debug device firmware. In *5th Symp on Operating Systems Design and Impl.*, 2002.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7), July 1978.

[29] L. Lamport. Part time parliament. *ACM Trans. on Computer Systems*, 16(2), May 1998.

[30] L. Lamport. Paxos made simple. *ACM SIGACT News Distributed Computing Column*, 32(4), December 2001.

[31] L. Lamport, R. Shostack, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[32] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *13th ACM Symposium on Operating Systems Principles*, October 1991.

[33] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, pages 203–213, 1998.

[34] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *17th ACM Symposium on Operating Systems Principles*, December 1999.

[35] J. McLean. A general theory of composition for a class of 'possibilistic' security properties. *IEEE Trans. on Software Engineering*, 22(1):53–67, January 1996.

[36] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *5th Symp on Operating Systems Design and Impl.*, 2002.

[37] J-F. Paris and D. D. E. Long. Voting with regenerable volatile witnesses. In *ICDE*, pages 112–119, 1991.

[38] M. Reiter. The Rampart toolkit for building high-integrity services. In *Dagstuhl Seminar on Dist. Sys.*, pages 99–110, 1994.

[39] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and pulic-key cryptosystems. *Comm. of the ACM*, 21(2):120–126, 1978.

[40] R. Rodrigues, M. Castro, and B. Liskov. Base: Using abstraction to improve fault tolerance. In *18th ACM Symposium on Operating Systems Principles*, October 2001.

[41] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles*, 2001.

[42] A. Sabelfeld and A. Myers. Language-based information-flow security, 2003.

[43] F. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A tutorial. *Computing Surveys*, 22(3):299–319, September 1990.

[44] Secure hash standard. Federal Information Processing Standards Publication (FIPS) 180-1, April 1995.

[45] M. Shand and J. E. Vuillemin. Fast implementations of RSA cryptography. In E. E. Swartzlander, M. J. Irwin, and J. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 252–259, Windsor, Canada, 1993. IEEE Computer Society Press, Los Alamitos, CA.

[46] Q. Sun, D. Simon, Y. Wang, W. Russell, V. Padmanabhan, and L. Qiu. Statistical identification of encrypted web browsing traffic. In *Proc. of IEEE Symposium on Security and Privacy*, May 2002.

[47] G. Tsudik. Message authentication with one-way hash functions. *ACM Computer Comm. Review*, 22(5), 1992.

[48] U. Voges and L. Gmeiner. Software diversity in reacter protection systems: An experiment. In *IFAC Workshop SAFECOMP79*, May 1979.

[49] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *18th ACM Symposium on Operating Systems Principles*, pages 230–243, 2001.

[50] J. Yin, J-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Byzantine fault-tolerant confidentiality. In *Proceedings of the International Workshop on Future Directions in Distributed Computing*, pages 12–15, June 2002.

[51] J. Yin, J-P. Martin, A. Venkataramani, M. Dahlin, and L. Alvisi. Separating agreement from execution for byzantine fault tolerant services. Technical report, University of Texas at Austin, Department of Computer Sciences, August 2003.

[52] L. Zhou, F. Schneider, and R. van Renesse. COCA : A Secure Distributed On-line Certification Authority. *ACM Trans. on Computer Systems*, 20(4):329–368, November 2002.