Proactive Obfuscation

TOM ROEDER
Microsoft Research
and
FRED B. SCHNEIDER
Cornell University

Proactive obfuscation is a new method for creating server replicas that are likely to have fewer shared vulnerabilities. It uses semantics-preserving code transformations to generate diverse executables, periodically restarting servers with these fresh versions. The periodic restarts help bound the number of compromised replicas that a service ever concurrently runs, and therefore proactive obfuscation makes an adversary's job harder. Proactive obfuscation was used in implementing two prototypes: a distributed firewall based on state-machine replication and a distributed storage service based on quorum systems. Costs intrinsic to supporting proactive obfuscation in replicated systems were evaluated by measuring the performance of these prototypes. The results show that employing proactive obfuscation adds little to the cost of replica-management protocols.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*Security and protection (e.g., firewalls)*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/server*; C.4 [**Performance of Systems**]: Fault Tolerance; D.4.5 [**Operating Systems**]: Reliability—*Fault tolerance*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

General Terms: Design, Reliability, Security

Additional Key Words and Phrases: Byzantine fault tolerance, distributed systems, proactive recovery, quorum systems, state machine approach

ACM Reference Format:

Roeder, T. and Schneider, F. B. 2010. Proactive obfuscation. ACM Trans. Comput. Syst. 28, 2, Article 4 (July 2010), 54 pages.
DOI = 10.1145/1813654.1813655 http://doi.acm.org/10.1145/1813654.1813655

This work was performed in part while T. Roeder was at Cornell University.
This research was supported in part by AFOSR grant F9550-06-0019, National Science Foundation Grants 0430161 and CCF-0424422 (TRUST), and Microsoft Corporation.
Authors' addresses: T. Roeder, Microsoft Research, One Microsoft Way, Redmond, WA 98052; email: throeder@microsoft.com; F. B. Schneider, Department of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853; email: fbs@cs.cornell.edu.

DOI 10.1145/1813654.1813655 http://doi.acm.org/10.1145/1813654.1813655

1. INTRODUCTION

Independence of replica failure is crucial when using replication to implement reliable distributed services. But replicas that use the same code share many of the same vulnerabilities and, therefore, do not exhibit independence to attack. This article introduces a new method of restoring some measure of that independence, *proactive obfuscation*, whereby each replica is periodically restarted using a freshly generated, diverse executable. The diversity increases the work an adversary must undertake to compromise a service that employs replication.

We designed and implemented mechanisms to support proactive obfuscation. And we used these mechanisms to implement prototypes of two services: (i) a distributed firewall (based on the `pf` packet filter [OpenBSD c] in OpenBSD [OpenBSD a]) and (ii) a distributed storage service. Each service uses a different approach to replica management—we chose approaches commonly used to achieve a high degree of resilience to server failure.[1]

Proactive obfuscation employs semantics-preserving program transformations for automatically creating diverse executables during compilation, loading, or at run-time. We call this *obfuscation*. Address reordering and stack padding [Forrest et al. 1997; Bhatkar et al. 2003; Xu et al. 2003], system call reordering [Chew and Song 2002], instruction set randomization [Kc et al. 2003; Barrantes et al. 2005, 2003], heap randomization [Berger and Zorn 2006], and data randomization [Cadar et al. 2008] are all examples. They each produce *obfuscated executables*, which are believed more likely to crash in response to certain classes of attacks than to fall under control of an adversary. For instance, success in a buffer overflow attack typically depends on stack layout details, so replicas using differently obfuscated executables based on high-entropy address reordering or stack padding are likely to crash instead of succumbing to adversary control. One goal of obfuscation is to provide *code independence*: an adversary that successfully attacks a replica and learns the secret that is the basis for the obfuscation gains no advantage in attacking another replica that was obfuscated based on a different secret.

Some approaches to replica management also involve *data independence*: different replicas store different state. Some forms of obfuscation thus create data independence. Replicated systems that support data independence are less vulnerable to certain attacks. For example, some code vulnerabilities are exercised only when a replica is in a given state—if replicas have data independence, then an attack that depends on such a vulnerability will not necessarily succeed at all replicas.

Obfuscation techniques are becoming common in commercial operating systems. Windows Vista, Windows 7, OpenBSD, and Linux employ obfuscation, either by default or in easily-installed modules, to defend against attacks that exploit internal implementation details. Although obfuscation involving

[1] Specifically, the firewall uses state-machine replication and the distributed storage service uses a dissemination quorum system. The *primary/backup* approach is also sometimes used for replica management, but it only masks certain benign failures. So, applying proactive obfuscation to the primary/backup approach would yield a system that is not resilient to attack.

insufficient entropy has been shown [Shacham et al. 2004] to be a weak defense, sufficiently-randomized executables can prevent attacks.

We distinguish between two kinds of replica failures. A replica can be *crashed* and therefore incapable of producing outputs until it reboots. Or, a replica can be *compromised* because it has come under control of an adversary. In the fault-tolerance literature, this second kind of behavior is often called *Byzantine*. Common usage in the literature presumes Byzantine failures are independent. So, to emphasize that attacks may cause correlated failures, we instead use the term "compromised". A replica that is not crashed or compromised is *correct*. Clients of a distributed service may also be crashed, compromised, or correct.

A replicated system in this failure model has a *compromise threshold* that bounds the number of compromised replicas and a *crash threshold* that bounds the number of crashed replicas:

—When the compromise threshold is not exceeded, any replies the system sends to the clients are correct.

—When neither the compromise threshold nor the crash threshold is exceeded, the system produces correct replies to client requests.

Notice that obfuscation, and consequently proactive obfuscation, does not defend against attacks that exploit the semantics (intentional or not) of the system interface implemented by the replicas, since obfuscation by definition preserves this semantics. For example, the system interface might unintentionally include an operation that allows a host external to the system to take control of a replica; in this case, proactive obfuscation would not prevent the adversary from performing this operation to compromise a replica.

Proactive obfuscation defends against client requests that—without the independence provided by obfuscation—would compromise all replicas simultaneously; with proactive obfuscation, all replicas are instead likely to crash. During such crashes, the system is unable to respond to clients. So, an implementation of proactive obfuscation must also provide a way for crashed replicas to recover. Simultaneous crashes can then be treated as periods of unavailability that drop the corresponding attacker's request packet.

Servers running obfuscated executables might share fewer vulnerabilities, but such artificially-created independence erodes over time, because an adversary with access to an obfuscated executable can experiment and generate a customized attack. Eventually, all replicas could thus become compromised. Proactive obfuscation defends against such customized attacks by introducing *epochs*; a server is rebooted in each epoch, and, therefore, all n servers are rebooted after n epochs have elapsed. This approach is an instance of a *moving-target defense* [Ghosh et al. 2009], whereby a system under attack changes configuration over time to avoid compromise.[2]

The approaches to replica management used by our prototypes are designed to tolerate at most some threshold t of compromised replicas out of n total

[2] Some instances of this kind of defense are also explored in Sousa et al. [2008].

replicas. Using proactive obfuscation with epoch length Δ seconds implies that an adversary is forced to compromise more than t replicas in $n\Delta$ seconds in order to subvert the service. Moreover, we can make the compromise of more than t replicas ever more difficult by reducing Δ, although Δ is obviously bounded from below by the time needed to reobfuscate and reboot a single server replica.

Neither replication nor proactive obfuscation enhances the confidentiality of data stored by replicas. For some applications, confidentiality can be enforced by storing data on each server in encrypted form under a different per-server key. And cryptographic techniques have been developed for performing certain computations on such encrypted data. Proactive obfuscation does not interfere with the use of these techniques.

Further, neither replication nor proactive obfuscation defends against all denial of service (DoS) attacks. Adversaries executing DoS attacks rely on one of two strategies: saturating a resource, like a network, that is not under the control of the replicas, or sending messages that saturate resources at replicas. This second strategy includes DoS attacks that cause replicas to crash and subsequently reboot.

Finally, note that proactive obfuscation is intended to augment—not replace—techniques that reduce vulnerabilities in replica code. And proactive obfuscation is attractive because extant techniques (e.g., safe languages or formal verification) have proved difficult to retrofit on legacy systems. Network services, for instance, are often written in C, which is neither a safe language nor amenable to formal verification. Other services currently in use also are implemented in unsafe languages and do not employ formal verification.

To quantify the number of different kinds of systems today that might benefit from proactive obfuscation, we performed a brief survey of vulnerabilities published by the Zero-Day Initiative [Zero-Day Initiative], a security group run by 3Com that pays for responsible disclosure of security vulnerabilities. We examined the last 25 vulnerabilities published in 2009: these vulnerabilities were found in a wide range of systems, from Unix variants to Microsoft Windows, and even in Sun's Java runtime. The vulnerable code was written by many groups, ranging from Adobe and Apple to Microsoft. From the summary descriptions provided on the website, it appears that 23 of the 25 vulnerabilities could have been mitigated by known obfuscation techniques, like instruction-set randomization, system-call randomization, or memory randomization.

The run-time costs of proactive obfuscation were found to be small. During normal execution, our firewall prototype achieves at least 92% of the throughput of a comparable replicated system that does not employ proactive obfuscation; latency increases to 140% (a difference of several milliseconds). And for our storage-service prototype, throughput of the version with proactive obfuscation is identical at the servers to a comparable replicated system without proactive obfuscation; the latency at the servers is also identical.

Of course, replication is not appropriate for all systems. But for those systems where replication is justified, proactive obfuscation enhances trustworthiness by providing some resilience against attacks. In analogy with the term "fault tolerance", we say that services resilient to attack exhibit *attack tolerance*.[3] And this paper elucidates the proactive obfuscation approach to attack tolerance by giving:

- —A set of requirements an obfuscation technique must satisfy for use in proactive obfuscation; this provides criteria for evaluating obfuscation techniques—present and future.
- —Mechanisms for applying proactive obfuscation to arbitrary replicated systems.
- —Two prototype instantiations of the architecture and mechanisms.
- —Performance analysis of the prototypes; this demonstrates that the additional run-time costs of adding proactive obfuscation are modest. We also provide performance comparisons of different plausible implementations of the mechanisms.

We proceed as follows. Proactive obfuscation is presented in Section 2, along with mechanisms and an architecture for its implementation. Then, Section 3 gives an overview of the state machine approach to replica management and describes and evaluates a firewall prototype. Quorum systems are the subject of Section 4, along with a description and evaluation of a storage-service prototype. Finally, Section 5 contains a discussion and a summary of related work.

2. PROACTIVE OBFUSCATION FOR REPLICATED SYSTEMS

An *obfuscator* takes two inputs—a program P and a secret κ—and produces an *obfuscated program* $\hat{P}$ semantically equivalent to P.[4] Secret κ specifies which exact transformations are applied to produce $\hat{P}$ from P. We abstract from the details of the obfuscator by defining properties we require it to implement:

(2.1) *Obfuscation Independence.* For $t > 1$, the amount of work an adversary requires to compromise t obfuscated replicas is $\Omega(t)$ times the work needed to compromise one replica.

(2.2) *Bounded Adversary.* The time needed for an adversary to compromise $t+1$ obfuscated replicas exceeds the time needed to reobfuscate, reboot, and recover n replicas.

Obfuscation Independence (2.1) implies that different obfuscated executables exhibit some measure of independence, and therefore a single attack is unlikely to compromise multiple replicas. Obfuscation techniques being advocated for systems today attempt to approximate Obfuscation Independence

[3] Other authors use the term *intrusion tolerance* (see, e.g., Deng et al. [2004] and Sousa et al. [2007]). We prefer the term "attack tolerance" because it conveys the need to tolerate not only a successful attack (which results in a compromised replica—an "intrusion"), but also a failed attack (which might result in many crashed replicas).

[4] This definition admits changes to data and/or code in the program and/or system environment in which the program is executed. For example, in addition to performing address-space layout randomization, an obfuscator might change the layout of the filesystem on a given replica. Examples of changes to program data are explored by Sousa et al. [2008].

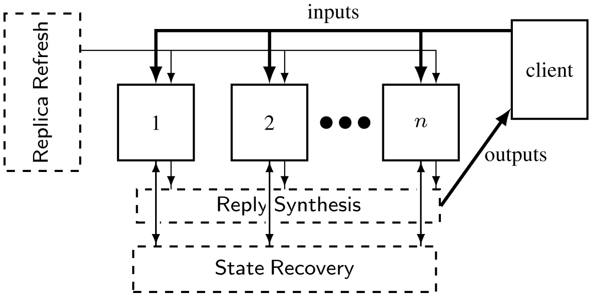

Fig. 1. Implementing proactive obfuscation.

(2.1). Given enough time, however, an adversary might still be able to compromise $t+1$ replicas. But Obfuscation Independence (2.1) and Bounded Adversary (2.2) together imply that periodically reobfuscating and rebooting replicas nevertheless makes it hard for adversaries to maintain control over more than t compromised replicas. In particular, by the time an adversary could have compromised $t + 1$ obfuscated replicas, all n will have been reobfuscated and rebooted (with the adversary evicted), so the work done by the adversary is mooted.

It might seem that an adversary could circumvent the consequences of Obfuscation Independence (2.1) and Bounded Adversary (2.2) by performing attacks on replicas in parallel; the adversary would send separate attacks independently to each replica. To prevent such parallel attacks, we employ an architecture that ensures any input processed by one replica is, by design, processed by all. Attacks sent in parallel to different replicas are ultimately processed serially by all replicas. The differently obfuscated replicas are likely to crash when they process most of these attacks, so the rate at which an adversary can explore different possible attacks is severely limited, and parallelism does not really benefit the adversary.

Note that the number of crashes caused by an attack might exceed the crash threshold; the system will then suffer an outage. We assume the existence of a tamper-proof hardware mechanism that reboots crashed replicas, and we have designed recovery methods that allow the system to recover in this case. See Section 2.2.2 for details.

2.1 Mechanisms to Support Proactive Obfuscation

The time needed to reobfuscate, reboot, and recover all n replicas in a replicated system is determined by the amount of code and data at each replica and by the time needed to execute mechanisms for performing reboot and recovery. Figure 1 depicts an implementation of a replicated service and identifies three mechanisms needed for supporting proactive obfuscation: Reply Synthesis, Replica Refresh, and State Recovery. Reply Synthesis, Replica Refresh, and State Recovery have several obligations; we will describe implementations that discharge these obligations in subsequent sections. These implementations also depend on several assumptions, which we describe and justify below.

Clients send *inputs* to replicas. Each replica implements the same interface as a centralized service, processes these inputs, and sends its *outputs* to clients. To transform outputs from the many replicas into an output from the replicated service, clients employ an *output synthesis* function f_γ, where γ specifies the minimum number of distinct replicas from which a reply is needed. In addition to being from distinct replicas, the replies used by f_γ are assumed to be *output-similar*—a property defined below that differs for each approach to replica management and output synthesis function. Reply Synthesis is the mechanism that we postulate to implement this output synthesis function.

Some means of authentication must be available in order for Reply Synthesis to distinguish the outputs from distinct replicas; replica management also could need authentication for doing inter-replica coordination. These authentication obligations are summarized as:

> (2.3) *Authenticated Channels.* Each replica has authenticated channels from all other replicas and to all clients.

The Replica Refresh mechanism periodically reboots servers, informs replicas of epoch changes, and provides freshly obfuscated executables to replicas. For Replica Refresh to evict the adversary from a compromised replica, we require it to satisfy the following two obligations:

> (2.4) *Replica Reboot.* Any replica, whether compromised or not, can be made to reboot by Replica Refresh.

> (2.5) *Generation of Executables.* Executables used by recovering replicas are kept secret from other replicas prior to deployment and are generated by a correct host.

Replica Reboot (2.4) guarantees that an adversary cannot indefinitely prevent a replica from rebooting. Generation of Executables (2.5) ensures that replicas reboot using executables that have not been analyzed or modified by an adversary.

Replicas keep *state* that may change in response to processing client inputs. The State Recovery mechanism enables a replica to recover state after rebooting, so the replica can resume participating in the replicated service. Specifically, a recovering replica receives states from multiple replicas and converts them into a single state. Recovering replicas employ a *state synthesis* function g_δ for this, where δ specifies the minimum number of distinct replicas from which state is needed. Analogous to output synthesis, replies used by g_δ are assumed to be *state-similar*—a property defined separately for each approach to replica management and state synthesis function.

Synchronicity. Replica Refresh and State Recovery must complete in a bounded amount of time; otherwise, Bounded Adversary (2.2) will be violated. To guarantee that Replica Refresh and State Recovery complete in a bounded amount of time, some components in the system must be synchronous.[5] We

[5] Canetti et al. [1997] and Castro and Liskov [2005] impose similar constraints in their work on proactive recovery.

adopt the *architectural hybrid model* [Verissimo 2006; Sousa et al. 2006], whereby different components can have different levels of synchronicity; sets of components having similar strong-synchronicity properties are used to implement mechanisms satisfying strong-synchronicity properties.[6] The architectural hybrid model imposes two obligations:

—Provide implementations of components that satisfy the given synchronicity properties.

—Show that components continue to satisfy their properties despite any interactions with other components, even if the other components differ in their levels of synchronicity.

So, for each of our implementations of State Recovery and Replica Refresh, we must show that asynchrony in other mechanisms or components cannot interfere.[7]

The number of replicas needed to implement proactive obfuscation depends, in part, on the number of concurrently rebooting replicas; there must be enough non-rebooting correct replicas to run State Recovery. To implement this property, we assume an upper bound on the amount of state at each replica. We also make synchronous, but justifiable, assumptions about the processors and clocks for all hosts in the system as well as synchronous assumptions about message delays for some, but not all, networks.

For synchronous execution, replicas must have the same notion of time. So, we require the assumption that clocks on different replicas agree, within some bound.

(2.6) *Bounded-Rate Clocks.* The difference between rates of clocks on different correct hosts is bounded.

This assumption is easy to implement in a small distributed service. All replicas can be deployed on identical hardware, which means that all replicas will run on processors with similar clock rates. And these rates do not change drastically over time.

Similarly, the rate of execution of processors at different replicas is assumed to be comparable; if the processor at one replica can be arbitrarily slow or fast, then synchronous execution cannot be guaranteed.

(2.7) *Synchronous Processors.* Differences in the rate of instruction execution on different correct hosts are bounded.

Synchronous Processors (2.7) is an assumption that is easy to implement in our setting for the same reasons as Bounded-Rate Clocks (2.6). Replicas that use the same hardware have the same processors, hence the same instruction execution

[6] The architectural hybrid model also allows different components to have different failure models. But we employ the same failure model for all components: any host might be crashed, compromised, or correct.

[7] An alternative to demonstrating this non-interference is to assume that the necessary synchronous properties hold for the whole system for a bounded period at the beginning of each epoch. But this kind of assumption is unrealistic in practice, since it is not clear why synchronous assumptions should hold at such intervals.

rates. And synchronous software components executing on these processors can use well-known techniques from real-time systems to allocate enough cycles to execute in a bounded amount of time.

Finally, interacting replicas require a synchronous network for communication. We do not assume perfect synchrony for this network, but rather assume that replicas can guarantee timely delivery of messages.

> (2.8) *Timely Links.* There is a bound b on the number of times a message must be sent on a network before the message is received. For any message that is received, there is a bound ϵ on the amount of time it takes for this message to be transmitted and received.

Timely Links (2.8) is a strong assumption; it implies that one processor always succeeds in sending a message to another in at most ϵb seconds. We only assume this for networks used to implement State Recovery and Replica Refresh, since these mechanisms must execute in a bounded amount of time. These networks are separate from networks for client input and output.

It is possible to use trusted hardware to satisfy Timely Links (2.8)—specifically, by using a local switch as a network. Timely Links (2.8) now holds provided the local switch continues to deliver packets despite an arbitrary load being imposed by replicas sending messages. And that, in turn, can be satisfied provided

—the switch is itself synchronous and sufficiently well-provisioned;

—the switch implements asymmetric throttling of independent send and receive interfaces at each replica; and

—the receive interface at each replica is able to receive $\Omega(t)$ times the capacity that its send interface can generate.

Bounded-Rate Clocks (2.6), Synchronous Processors (2.7), and Timely Links (2.8) are used to guarantee a bound on the time involved in running Replica Refresh and subsequent State Recovery. Specifically, the epoch length must be chosen to exceed this bound, so that replicas have enough time to recover before others reboot. The epoch length thus determines the *window of vulnerability* for the service: the interval of time during which a compromise of $t + 1$ replicas leads to the service being compromised. These assumptions then allow mechanism implementations that together satisfy Bounded Adversary (2.2).

2.2 Mechanism Implementation

Implementing proactive obfuscation requires instantiating each of the mechanisms just described. Figure 2 depicts an architecture for an implementation.

Clients send inputs to replicas and receive outputs on the networks labeled *input network* and *output network*, respectively; the input network and output network can be lossy and asynchronous, not necessarily satisfying Timely Links (2.8). Reply Synthesis is performed by clients. State Recovery is performed by replicas using a network, labeled *internal service network*, that satisfies Timely Links (2.8). Replica Refresh is implemented either by a host (called the

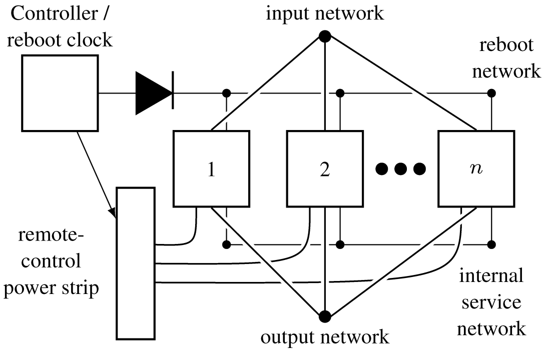

Fig. 2. The prototype architecture.

Controller and assumed to be correct) or by decentralized protocols that we give below.

If we design a Controller that never attempts to receive messages, then the Controller cannot be affected by hosts in its environment.[8] Because it cannot be affected by other hosts, the Controller cannot be successfully attacked. The Controller might still send messages on the *reboot network* connected to all replicas. This network will be used to provide boot code to replicas; it is assumed to satisfy Timely Links (2.8). Also, note that we require the Controller to satisfy Bounded-Rate Clocks (2.6) and Synchronous Processors (2.7). So, all operations performed by the Controller are synchronous, and the Controller can send any message to any replica in a bounded amount of time.

Whether using the Controller or decentralized protocols, Replica Reboot (2.4) is implemented by a *reboot clock* that comprises a timer for each replica. The reboot clock controls a synchronous, remote-control power strip, and it toggles power to individual replicas when the timer expires for that replica. Replicas are rebooted in order mod n, one per epoch. All of these operations are completed in a bounded amount of time, assuming Bounded-Rate Clocks (2.6) for the reboot clock; moreover, reboots complete in a bounded amount of time, due to Synchronous Processors (2.7). And these operations do not interact with non-synchronous components, so there is no problem of interference.

In this architecture, the Controller, the reboot clock, and the remote-control power strip are single points of failure. For simplicity of exposition, we do not describe how to implement fault-tolerant versions of these components. But the Controller is similar to the Controller in the intrusion-tolerant system designed by Arsenault et al. [2007] and can be made fault-tolerant by using similar techniques. The reboot clock and remote-control power strip are relatively straightforward hardware components and are not vulnerable to attack, since neither takes input from any potentially compromised source; hardware fault-tolerance is well understood, so we instead focus here on how to implement the other mechanisms necessary for replica attack-tolerance.

[8] The diode symbol in Figure 2 on the line from the Controller depicts the constraint that the Controller never receives messages on the reboot network.

Epoch change can be signaled to replicas either by messages from the Controller or by timeouts. In either case, the elapsed time between the first correct replica changing epochs and the last correct replica changing epochs is bounded, due to Bounded-Rate Clocks (2.6), Synchronous Processors (2.7), and Timely Links (2.8). Epochs are labeled with monotonically increasing *epoch numbers* that are incremented at each epoch change. For epoch changes with our decentralized protocols, we use timeouts at replicas; we cannot use the reboot clock, since it takes no input and cannot send messages to replicas.

2.2.1 *Reply Synthesis.* To perform Reply Synthesis using output synthesis function f_γ, clients must receive output-similar replies from γ distinct replicas. We have experimented with two different implementations.

In the first, each replica has its own private key, and clients authenticate digitally-signed individual responses from replicas; a replica changes private keys when recovering after a reboot. Clients thus need the corresponding new public key for a recovering replica in order to authenticate future messages from that replica. So, the service provides a way for a rebooting replica to acquire a certificate signed by the service for its new public key. A recovering replica reestablishes authenticated channels with clients by acquiring such a certificate and sending this certificate[9] on its first packet after reboot.[10] This method of Reply Synthesis and authentication requires clients to receive new keys in each epoch.

In our second Reply Synthesis implementation, the entire service supports a single public key that is known to all clients and replicas, and the corresponding private key is shared (using secret sharing [Shamir 1979]) by the replicas. Each replica is given a *share* of the private key and uses this share to compute *partial signatures* [Desmedt and Frankel 1990] for messages. The secret sharing is refreshed on each epoch change, using an operation called *share refresh*; however, the underlying public/private key pair for the service does not change. Consequently, clients do not need new public keys after epoch changes, unlike in the public-key per-server Reply Synthesis implementation given above. Recovering replicas acquire their shares by a *share recovery* protocol.

Each replica then includes a partial signature on responses it sends to clients. Only by collecting more than some threshold number of partial signatures can a client assemble a signature for the message. We use APSS, an asynchronous, proactive, secret-sharing protocol [Zhou et al. 2005] in conjunction with an $(n, t+1)$ threshold cryptosystem to compute partial signatures and perform assembly. Contributions from $t+1$ different partial signatures are necessary to assemble a valid signature, so a contribution from at least one correct replica is needed. Reply Synthesis is then implemented by checking assembled signatures using the public key for the service.

An optimization of this second Reply Synthesis implementation is possible. Here, a replica—not the client—assembles a signature from partial signatures

[9] Certificates contain epoch numbers to prevent replay attacks.

[10] Our implementation also allows clients to request certificates from replicas if they receive a packet containing a replica/epoch combination for which they have no certificate.

received from other replicas. This replica then sends the assembled signature with its output to the client. The optimization requires replicas to send partial signatures to each other, which increases inter-replica communication for each output, hence increases latency. But the optimization eliminates changes to client code designed to communicate with non-replicated services.

2.2.2 *State Recovery*.

Normally, each replica reaches its current state by receiving and processing inputs from clients. This, however, is not possible after reboot if previous client inputs are no longer available. Recall that reboots occur periodically for proactive obfuscation and also occur following crashes.

To facilitate recovery after a crash, each replica writes its state to non-volatile media after processing each client input; a replica recovering from a crash (but not from a reboot for proactive obfuscation) reads this state back as the last step of recovery. A replica thus recovers its state without sending or receiving any messages. So, replica crashes resemble periods of replica unavailability.[11]

Replicas rebooted for proactive obfuscation, however, cannot use their locally stored state, since this state might be corrupt and might cause a replica reading it to be compromised or to crash. In the worst case, all locally-stored state might have been deleted by an adversary. The obvious alternative to using local state is to obtain state from other correct replicas by executing a *recovery protocol*. However, obfuscation might mean that replicas use different internal state representations. Obfuscated replicas are therefore assumed to implement marshaling and unmarshaling functions to convert their internal state representation to and from some representation that is the same for all replicas.

Replicas implement a recovery protocol for State Recovery that completes in a bounded amount of time. Before executing State Recovery, a recovering replica i establishes authenticated channels to all replicas it communicates with. The recovery protocol then proceeds as follows:

(1) Replica i starts recording packets received from replicas.

(2) Replica i issues a *state recovery request* to all replicas. The actions taken by replicas upon receiving this state recovery request depend on the approach to replica management in use, but these actions must guarantee that correct replicas send state-similar replies to replica i in a bounded amount of time.

(3) Upon receiving δ state-similar replies, replica i applies state synthesis function g_δ.

(4) Replica i replays all packets recorded due to Step 1 as if they were received for the first time and stops recording.

[11] The period of unavailability begins just before receipt of the offending client input. This method of handling crashes only works for transient errors and for attacks that cause replicas to crash without writing state to disk. Other crashes are considered compromises and are addressed when a replica is rebooted. See Section 5.1 for a discussion of crashes caused as part of DoS attacks.

Notice that Step 4 might cause the recovering replica to replay messages already incorporated into states that it has reached and passed. But replication protocols normally guarantee that such delayed duplicate messages are ignored, and that will occur here, too.

State Recovery must terminate in a bounded amount of time; otherwise, a recovering replica from one epoch might still be recovering when the next replica is rebooted, violating one of our assumptions about epochs. If State Recovery fails to complete successfully for enough replicas, then availability will be lost, since there will not be enough replicas to handle client requests. To guarantee that State Recovery completes in a bounded amount of time, we assume that each replica stores bounded state. But also, the implementation of the state recovery protocol must complete in a bounded amount of time—even given interactions with asynchronous components that handle client requests. We consider each step in turn.

One problem that might keep Steps 2 and 3 from completing in time is that other replicas might crash an unbounded number of times due to attacks by an adversary. This could slow down State Recovery by an arbitrary factor. To solve this problem, a replica that has recovered its state after a crash does not perform any operations, other than to listen for a bounded amount of time to see if there are any state recovery requests from a recovering replica.[12] Only if there are no such messages, will the replica then restart execution of Reply Synthesis and continue as before. Otherwise, it processes the State Recovery messages from the recovering replica before handling messages from clients or other replicas. So, replicas crash and recover at most once during each period of State Recovery. Then, Bounded-Rate Clocks (2.6), Synchronous Processors (2.7), and Timely Links (2.8) together imply that Steps 2 and 3 complete in bounded time.

To avoid interference, we require some means to ensure that a replica recovering from a crash initially only processes certain messages. It suffices that replicas recovering from a crash have a way to sort messages and only process state recovery requests.[13] Also, a recovering replica receiving messages from the other replicas for its state recovery request must not be crashed by these messages; otherwise, State Recovery might require an unbounded amount of time to complete.[14]

Finally, we must guarantee that Step 4 completes in bounded time. That is, replicas must be able replay and process recorded packets while continuing

[12] The exact value for this bound depends on how frequently a recovering replica starts sending state recovery requests. But Timely Links (2.8) and Synchronous Processors (2.7) imply that there is a bound.

[13] One way to achieve this property would be to have an additional network that allows exactly one replica to send and all replicas to receive. On each epoch change, the switch for this network is updated (by the Controller or the reboot clock) to allow only the next replica mod n to send. Another way would be to have provably-correct signature checking code for the replicas.

[14] This is a strong property to assume, since an implementation must guarantee that messages sent from an adversary for State Recovery cannot cause a recovering replica to crash. One possible implementation would be to have provably-correct state-synthesis code; this is difficult but might be achievable, given a simple enough state-synthesis function.

to receive and record packets, and this processing must terminate in bounded time. So, there is a maximum speed at which a replicated system using proactive obfuscation can process inputs. This maximum speed depends on how quickly a recovering replica can process its recorded packets and must be enough slower so that Step 4 can terminate in bounded time.[15]

In addition to completing in bounded time, a recovering replica running State Recovery must recover a useful replica state. Informally, a replica state is useful if it allows a replica to catch up with the state of the system. To make this more formal, we refine our notion of replica states.

A replica state is a map from *state variables* to values; the state variables name components of a replica state. Each replica state is thus comprised of multiple *replica substates*, each mapping one state variable to a value. A replicated system also has a (single) *system state* mapping state variables to values, as well as *system substates* for each variable. We consider only the case where state variables and values are the same for a system substate and the corresponding replica substate.[16]

We say that a replica substate is *θ-consistent* if this replica substate is held by a threshold θ_c of correct replicas and was held by θ_b previously correct but currently rebooting replicas such that $\theta_c + \theta_b = \theta$. A replica state is said to be *θ-consistent* if each of its replica substates is θ-consistent. Notice that a θ-consistent replica state does not necessarily correspond to the state of any single replica in the current execution; this is because the θ-consistent replica substates for one state variable might occur at different replicas than those for a different state variable.

In any execution of the system and for any state variable v, we can define a reachability relation $\rightarrow_v$: if σ_v and σ'_v are replica substates for state variable v, then $\sigma_v \rightarrow_v \sigma'_v$ holds if σ'_v is reachable by applying messages to a replica with replica substate σ_v. Reachability immediately implies a partial order on replica substates for the same state variable. For the value of a state variable to be recovered, an approach to replica management must additionally guarantee that the reachability relation provides a total order on θ-consistent replica substates for this state variable; we call a state variable *well ordered* in this case and write $\sigma_v \leq_v \sigma'_v$ if $\sigma_v \rightarrow_v \sigma'_v$ holds. A system substate for a well-ordered state variable is defined to be the unique maximum θ-consistent replica substate for this state variable (or a special value representing "empty", if there is no such θ-consistent replica substate). The choice for threshold θ will depend on the approach to replica management, since it is part of the definition of the replicated system.

Recovering replicas must acquire a sufficiently current replica state. We say that a θ-consistent replica state σ is the *Least Dominant Consistent State* (LDCS) for a given execution if two conditions hold:

(1) each state variable v with non-empty replica substate σ_v in σ is well ordered, and

[15] In our implementations, this bound was not found to be a significant restriction.

[16] We leave the more general case as an open problem.

(2) for each replica state σ' at correct replicas, and for any θ-consistent replica substate σ'_v of σ' for well-ordered state variable v, $\sigma'_v \leq_v \sigma_v$ holds.

The LDCS is guaranteed to be unique, since it is comprised of the unique maximal θ-consistent replica substate for each well-ordered state variable.

The state recovery protocol will normally recover a replica state that is more advanced than the LDCS, since messages are being received by the system during the execution of state recovery. So, state recovery must implement the following property:

> (2.9) *Maximal State Recovery.* If σ is the LDCS at the time a replica i sends the state recovery request, then there is a time bound Δ and some replica state σ' such that for each state variable v, replica substate σ'_v is reached by correctly applying messages received after state recovery starts in a replica with corresponding replica substate σ_v, and replica i recovers σ' in Δ seconds.

A replica that acquires this replica state in a bounded amount of time after the beginning of the state recovery request protocol can be guaranteed to be able to catch up with all other replicas, given a low enough bound on the rate of execution of the other replicas.

2.2.3 *Replica Refresh.* Replica Refresh involves 3 distinct functions: (i) reboot and epoch change notification, (ii) reobfuscation of executables, and (iii) key distribution for implementing authenticated channels between replicas. We explored two different implementations of Replica Refresh. One is centralized, and the other is decentralized.

2.2.3.1 *Centralized Controller Solution.* A centralized implementation that employs a Controller for Replica Refresh can be quite efficient. For instance, a centralized implementation can provide epoch-change notification directly to replicas, can reobfuscate executables in parallel with replicas rebooting, and can generate keys and sign their certificates instead of running a distributed key refresh protocol.

Reboot and Epoch Change. To reboot a replica, the Controller toggles the remote-control power strip. Immediately after the reboot completes, the Controller uses the reboot network to send a message to all replicas, informing them of the reboot and associated epoch change. These operations use only one-way communication, so interference is not possible, and they complete in a bounded amount of time.

Reobfuscation of Executables. The Controller itself obfuscates and compiles executables of the operating system and application source code. By our assumption about the Controller, this guarantees that executables are generated by a correct host, as required by Generation of Executables (2.5). Executables are transferred to recovering replicas via the reboot network using a network boot protocol. To guarantee that no other replicas learn information about the executable and to prevent other replicas from providing boot code, the reboot network implements a separate confidential channel from the Controller to each replica. Replicas may not send packets on these channels. To summarize:

> (2.10) *Reboot Channels.* The Controller can send confidential information to each replica on the reboot network, no replicas can send any message on the reboot network, and clients cannot access the reboot network at all.

So, the reboot network is isolated and cannot be attacked. Therefore, any executable received on the reboot network comes from the Controller.

A simple but expensive way to implement Reboot Channels (2.10) is to have pairwise channels between each replica and the Controller. A less costly implementation involves using a single switch with ports that can be toggled on and off by the Controller. Then the Controller can communicate directly with exactly one replica by turning off all other ports on the switch. SNMP-controlled modern switches [Case et al. 1990] allow such control.

Instead of using TCP to communicate with the host providing an executable (since TCP-based methods like PXE boot [Intel Corporation 1999] require bidirectional communication with the Controller), a recovering replica monitors the reboot network until it receives a predefined sequence that signals the beginning of an executable. Then, the replica copies this executable and boots from it.[17] Notice that this operation is not vulnerable to interference from non-synchronous components, since it is independent of such components.

Key Distribution. The Controller performs key distribution to implement authenticated channels by generating a new public/private RSA [Rivest et al. 1978] key pair for each recovering replica i, certifying the public key to all replicas at the time i reboots. The new key pair along with public keys and certificates for each replica in the current epoch are written into an executable for i.[18] Reboot Channels (2.10) guarantees that other replicas cannot observe the executable sent from the Controller to a rebooting replica, so they cannot learn the new private key for i. There is also no danger of interference for this protocol, since it is entirely performed by the Controller.

2.2.3.2 *Decentralized Protocols Solution.* Decentralized schemes for Replica Refresh tend to be more expensive but can avoid the single point of failure of a centralized Controller.

Reboot and Epoch Change. We have not explored decentralized replica reboot mechanisms, because reboot depends on a remote-control power strip that is itself potentially a single point of failure. Decentralized epoch change

[17] A simpler boot procedure is possible, since a recovering replica that has not yet read input from any network is correct by assumption. Replicas known to be correct could be allowed to exchange messages with the Controller. This boot procedure requires modifying Reboot Channels (2.10) to provide the necessary control over which replicas can send using the reboot network.

[18] The Controller could include in this executable new public keys for replicas to be rebooted later. In our prototypes, these keys are not included in order to deprive adversaries access to the keys for as long as possible.

notification can be achieved, however, by using timeouts, as discussed at the beginning of Section 2.2. As for the centralized case, all operations are performed by the Controller, so there is no interference problem.

Reobfuscation of Executables. Replicas can each generate their own obfuscated executables in order to satisfy Generation of Executables (2.5). It suffices that each replica be trusted to boot from correct (i.e., unmodified) code; this trust is justified if the actions of the replica boot code cannot be modified:

(2.11) *Unmodifiable Boot Semantics.* The semantics of the boot procedure on replicas cannot be modified by an adversary.

And this assumption can be discharged if two conditions hold: (i) the BIOS is not modifiable,[19] and (ii) the boot code is stored on a read-only medium. Our prototypes assume (i) holds and discharge (ii) by employing a CD-ROM to store an OpenBSD system that, once booted, uses source on the CD-ROM to build a freshly obfuscated executable.[20]

After a newly obfuscated executable is built, it must be booted. This requires a way for a running kernel to boot an executable on disk or else a way to force a CPU to reboot from a different device after booting a CD-ROM (i.e., from the disk instead of the CD-ROM). The former is not supported in OpenBSD (although it is supported by kexec in Linux). The latter requires a way to switch boot devices, despite Unmodifiable Boot Semantics (2.11) with its implication that the code on the CD-ROM cannot change the BIOS in order to accomplish the switch.

In our prototypes, we resolve this dilemma by employing the reboot clock. It forces the server to switch between booting from the CD-ROM and from the hard disk, as follows. The BIOS on each server is set to boot from a CD-ROM if any is present and otherwise to boot from the hard disk. On reboot, the reboot clock not only toggles power to the server but also turns on power to the server's CD-ROM drive. The server boots, finds the CD-ROM (and therefore boots from that device), executes, and writes its newly obfuscated executable to its hard drive. The reboot clock then turns off power to the CD-ROM and toggles server power, causing the processor to reboot again. The server now fails to find a functioning CD-ROM, so it boots from the hard disk, using the freshly obfuscated executable. Decentralized reobfuscation completes in a bounded amount of time, due to Synchronous Processors (2.7); hence there is no interference problem.

Key Distribution. In our decentralized implementation, a recovering replica itself generates a new public/private key pair. It then establishes and disseminates a certificate for this new public key. Key generation can be performed by

[19] This can be implemented using a secure co-processor like the Trusted Platform Module [Trusted Computing Group].

[20] Our prototypes actually boot from a read-only floppy, which then copies an OpenBSD system and source from a CD-ROM to the hard disk and runs it from there. We describe the implementation in terms of a single CD-ROM here to simplify the exposition.

a rebooting replica locally if we assume that each replica has a sufficient source of randomness. To establish and disseminate a certificate, we use a simplified version of a proactive key refresh protocol designed by Canetti et al. [1997]. This protocol employs threshold cryptography: each replica has *shares* of a private key for the service. A recovering replica submits a *key request* for its freshly generated public key to other replicas; they compute partial signatures for this key using their shares. These partial signatures can be used to reassemble a signature for a certificate. For verification of the reassembled signature on a certificate to work, we assume the public key of the service is known to all hosts.

A recovering replica must know the current epoch before running the recovery protocol, since it needs authenticated channels with other replicas, and the certificates used to establish these channels are only valid for a given set of epochs. A recovering replica learns the current epoch from its valid reassembled certificate.

To prevent too many shares from leaking to mobile adversaries, shares of the service key used to create partial signatures for submitted keys are refreshed by APSS at each epoch change, using the share refresh protocol.

To prevent more than one key from being signed per epoch, replicas use Byzantine Paxos [Castro and Liskov 1999],[21] a distributed agreement protocol, to decide on the key request to use for a given recovering replica; correct replicas produce partial signatures in this epoch only for the key specified in this key request. Since the replicas use Byzantine Paxos to decide which key to sign, the set of correct replicas will sign at most one key. Only one certificate can be produced for each epoch, since one correct replica must contribute a partial signature to a reassembled signature.[22]

An alternative which does not work is to allow replicas to choose keys to sign without employing agreement. This alternative does not work because only $t + 1$ partial signatures are required for signature reassembly, so up to $n - t$ keys might be signed per epoch. This is because there are at most t compromised replicas and at least $n - t$ correct replicas; the t compromised replicas could generate $n - t$ keys, submit each to a different correct replica, and themselves produce t partial signatures for each, since each compromised replica can produce multiple different partial signatures. So, there would be $t + 1$ partial signatures (hence a certificate) for $n - t$ different keys.

The key distribution scheme we employ does not guarantee that a recovering replica will succeed in getting a new key signed—only that some replica will. So a compromised replica might get a key signed in the place of a correct, recovering replica. However, if recovering replica i receives a certificate purporting to be for the current epoch but using a different key than i requested, then i knows that some compromised replica established the certificate in its

[21] Our implementation is actually based on a set of slides written by Lamport (L. Lamport, personal communication), but the work by Castro and Liskov presents a closely-related protocol with the same properties.

[22] Another solution would be to use an $(n, \lceil \frac{n+t+1}{2} \rceil)$ threshold cryptosystem, since then only one key could be signed. But the implementation of APSS used in our prototypes does not support this threshold efficiently.

place, and i can alert a human operator. This operator can check and reboot compromised replicas.

One way to prevent a compromised server from installing a different key would be to add another timeout per epoch to the reboot clock and an extra network connecting the replicas. The timeout controls a switch that disables message sending for all replicas on this switch, except by the replica that was just rebooted (the next replica mod n). And replicas only accept keys to sign from this network. So, only the recovering replica can get a key signed. The operations performed in decentralized key distribution only involve synchronous components; asynchronous protocols like Byzantine Paxos and APSS are employed, but these protocols will complete in a bounded amount of time when run on entirely synchronous replicas and networks.

2.3 Mechanism Performance

Assumptions invariably bring vulnerabilities. Yet implementations having fewer assumptions are typically more expensive. For instance, decentralized Replica Refresh protocols require more network communication (an expense) than centralized protocols, but dependence on a single host in the centralized protocols brings a vulnerability. The trade-offs between different instantiations of the mechanisms of Section 2.2 mostly involve incurring higher CPU costs for increased decentralization. Under high load, these CPU costs divert a replica's resources away from handling client inputs. We use throughput and latency, two key performance metrics for network services, to characterize these costs for each mechanism.

2.3.1 *Reply Synthesis.* Implementing Reply Synthesis with individual authentication between replicas and clients requires reestablishing keys with clients at reboot, but this cost is infrequent and small. The major cost of individual authentication in our prototype arises in generating digital signatures for output packets.

The threshold cryptography implementation of Reply Synthesis computes partial signatures for each output packet. And partial signatures take even more CPU time to generate than ordinary digital signatures. So, under high load, the individual authentication scheme admits higher throughput and lower latency than the threshold cryptography scheme.

Throughput can be improved in both of our Reply Synthesis implementations by batching output—instead of signing each output packet, replicas opportunistically produce a single signature for a batch of output packets up to a maximum batch size, called the *batching factor*. Batching allows cryptographic computations (in particular, digital signatures) used in authentication to be performed less frequently and thus reduces the CPU load on the replicas and the client.

2.3.2 *State Recovery.* The cost of State Recovery depends on how much state must be recovered. Large state transfers consume network bandwidth and CPU time, at both the sending and receiving replicas. So, when a recovering replica must recover a large state under high load, State Recovery leads to significant degradation of throughput and latency. The amount of state to recover also depends on the rate at which replicas handle client requests, since the recovering replica will need to catch up on processing inputs it missed while performing State Recovery.

2.3.3 *Replica Refresh.* The performance characteristics of Replica Refresh differ significantly between the centralized and decentralized implementations. Reboot and epoch change notification make little difference to performance—epoch change notification only takes a short amount of time, and reboot involves only the remote-control power strip. Centralized Reobfuscation of Executables is performed by the Controller directly, and the resulting executable is transferred over the reboot network, so this has little effect on performance. However, decentralized Reobfuscation of Executables significantly increases the window of vulnerability, because reobfuscation cannot occur while a replica is rebooting, since replicas perform their own reobfuscation. So, reboot and reobfuscation now must be executed serially instead of in parallel.

Choosing between centralized and decentralized key distribution is also crucial to performance. Decentralized key distribution uses APSS, which must perform share refresh at each epoch change. Our implementation of APSS borrows code from CODEX [Marsh and Schneider 2004]. And APSS share refresh requires significant CPU resources in the CODEX implementation of APSS, so we should expect to see a drop in throughput and an increase in latency during its execution. Further, a rebooting replica must acquire shares during recovery, and this share recovery protocol requires non-trivial CPU resources; we thus should expect to see a second, smaller, drop in throughput and increase in latency during replica recovery. The key distribution protocol itself only involves signing a single key and performing a single round of Byzantine Paxos, so its performance degradation is negligible.

3. STATE MACHINE REPLICA MANAGEMENT

The *state machine approach* [Lamport 1978; Schneider 1990] provides a way to build a reliable distributed service that implements the same interface as a program running on a single trustworthy host. Using it, a program is described as a *state machine*, which consists of *state variables* and deterministic[23] *commands* that modify state variables and may produce output.

Given a state machine m, a *state machine ensemble* SME(m) consists of n servers that each implement the same interface as m and accept client requests to execute commands. Each server runs a replica of m and coordinates client commands so that each correct replica starts in the same state, transitions through the same states, and generates the same outputs. Notice that correct replicas in the state machine approach store the same sequence of states and, therefore, the state machine approach does not have data independence.

Coordination of client commands to servers is not one of the mechanisms identified in Figure 1. For a service that employs the state machine approach,

[23] The requirement that commands be deterministic does not significantly limit use of the state machine approach, because nondeterministic choices in a service can often be captured as additional arguments to commands.

a client must employ some sort of Input Coordination mechanism to communicate with all replicas in a state machine ensemble. This mechanism will involve replicas running an *agreement algorithm* [Lamport et al. 1982] to decide which commands to process and in what order. Agreement algorithms proceed in (potentially asynchronous) rounds, where some command is *chosen* by the replicas in each round. In most practical implementations, a command is *proposed* by a replica taking the role of *leader*. The command that has been chosen is eventually *learned* by all correct replicas.

Replicas maintain state needed by the agreement algorithm and maintain state variables for the state machine. For instance, Byzantine Paxos requires each replica to store a monotonically increasing sequence number labelling the next round of agreement. In our prototype, replicas use sequence numbers partitioned by the epoch number; we represent the mapping from sequence number to epoch number as a pair that we call an *extended sequence number*. Extended sequence numbers are ordered lexicographically. Output produced by replicas and sent to clients consists of the output of the state machine along with the extended sequence number; clients use the extended sequence number to group replies for Reply Synthesis, then remove it.

The combination of the extended sequence number and state variables forms the replica state. For the state-machine approach, there is only a single replica substate; this substate maps all state-machine variables and the extended sequence number to values. The replica state at a correct replica that has just executed the command chosen for extended sequence number (e, s) is denoted $\sigma_{(e,s)}$. There is only one possible value for $\sigma_{(e,s)}$, since all correct replicas serially execute the same commands in the same order, due to Input Coordination, and we assume that all replicas start in the same replica state.

Although use of an agreement algorithm causes the same sequence of commands to be executed by each replica, client requests may be duplicated, ignored, or reordered before the agreement algorithm is run. Modern networks provide only a best-effort delivery guarantee, so it is reasonable to assume that clients would already have been designed to accommodate such perturbed request streams.

3.1 A Firewall Prototype

To explore the costs and trade-offs of our mechanisms for proactive obfuscation, we built a firewall prototype that treats `pf` (the OpenBSD packet filter) as a state machine and uses the techniques and mechanisms of Section 2. We chose `pf` as the basis of our prototype because it is used as a firewall in many real networks. Implementing our prototype requires choosing an agreement algorithm for Input Coordination. We also must instantiate the output and state synthesis functions and define the operations that replicas perform upon receiving a state recovery request.

Input Coordination. Our firewall prototype uses Byzantine Paxos to implement Input Coordination. The number of replicas required to execute Byzantine Paxos while tolerating t compromised replicas is known to be $3t + 1$ [Castro and Liskov 1999]. However, this number does not take into account rebooting replicas. A rebooting replica does not exhibit arbitrary behavior—it simply resembles a crashed replica. Sousa [Sousa 2006] shows that tolerating r rebooting and t compromised replicas requires $3t + 2r + 1$ total replicas, which means that only 2 additional replicas must be added to tolerate each rebooting one. In our prototype, $r = 1$ holds, so we employ $3t + 2 \times 1 + 1 = 3t + 3$ replicas in total.

Normally, leaders in Byzantine Paxos change according to a *leader recovery protocol* whenever a leader is believed by enough replicas to be crashed or compromised. This leads to system delays when a compromised leader merely runs slowly, because execution speed of the state machine ensemble depends on the speed at which the leader chooses commands for agreement. To reduce these delays, we use *leader rotation* [Dwork et al. 1988]: the leader for sequence number j is replica $j \bmod n$. Thus, leadership changes with each sequence number, rotating among the replicas.

With leader rotation, the impact of a slow leader is limited, since timeouts for changing to the next leader can be made very short. Replicas set a timer for each sequence number i; on timeout, replicas expect replica $(i + 1) \bmod n$ to be the leader. Compromised leaders cause a delay for only as long as the allowed time to select one next command and can only cause this delay for t out of every n sequence numbers.[24]

Leader rotation might also cause delays while replicas are rebooting if a rebooting replica is selected as the next leader. So, we extend the leader rotation protocol to handle rebooting replicas. Specifically, since there is a bounded period during which all correct replicas learn that a replica has rebooted, correct replicas can skip over rebooting replicas in leader rotation. This is implemented by assigning the sequence numbers for a rebooting replica to the next consecutive replica mod n. We call this *leader adjustment*; it allows Byzantine Paxos to run without many executions of the leader recovery protocol, even during reboots. During the interval in which some correct replicas have not changed epochs, replicas might disagree about which replica should be leader. But Byzantine Paxos works even in the face of such disagreement about leaders.[25]

Our implementation of Byzantine Paxos is actually used to agree on hashes of packets rather than full packet contents. Given this optimization, a leader might propose a command for agreement even though not all replicas have received a packet with contents that hash to this command. Each replica checks locally for a matching packet when it receives a hash from a leader. If such a

[24] Leader rotation might seem inefficient, because switching leaders in Byzantine Paxos requires executing the leader recovery protocol. But Byzantine Paxos allows a well-known leader to propose a command for sequence number *num* without running leader recovery, provided it is the first to do so. Since replica *num* mod n is expected by all correct replicas to be leader for sequence number *num*, it is a well-known leader and does not need to run leader recovery to run a round of agreement for sequence number *num*.

[25] The bound on the time needed for all correct replicas to learn about an epoch change is thus just an optimization with respect to Byzantine Paxos. Our implementation of Byzantine Paxos continues to operate correctly, albeit more slowly, even if there is no bound.

packet has not been received, then a matching input packet is requested from the leader.[26]

A replica might fall behind in the execution of Byzantine Paxos. Such replicas need some way to obtain messages they might have missed, and State Recovery is a rather expensive mechanism to invoke for this purpose. So, replicas can send RepeatRequest messages for a given type of message and extended sequence number. Upon receiving a RepeatRequest, a replica resends[27] the requested message if it has a copy.[28]

Synthesis Functions. The output synthesis and state synthesis functions in our firewall prototype assume at most t replicas are compromised, since then any value received from $t + 1$ replicas must have been sent by at least one correct replica.

There are two output synthesis functions, one for each Reply Synthesis implementation. In both, γ is set to $t+1$.[29] Replies are considered to be *output-similar* for the individual authentication implementation if they contain identical outputs. So, output synthesis using individual authentication returns any output received in output-similar replies from $t + 1$ distinct replicas. Replies are considered to be *output-similar* for the threshold cryptography implementation if their partial signatures together reassemble to give a correct signature on this output (since this implies they contain identical outputs). So, output synthesis using threshold cryptography also returns any output received in output-similar replies from $t + 1$ distinct replicas.

For either Reply Synthesis implementation, clients need only receive $t + 1$ output-similar replies. So, if at most r replicas are rebooting, and t are compromised, then it suffices for only $2t + r + 1$ replicas to send replies to a client, since then there will be at least $2t + r + 1 - t - r = t + 1$ correct replicas that reply. And replies from $t + 1$ correct replicas for the same extended sequence number are always output-similar. In our prototype implementation, the leader for a given extended sequence number and the $2t + r$ next replicas mod n are the only replicas to send packets to the client for this extended sequence number.

For state synthesis, δ is also set to $t + 1$, and replies are defined to be *state-similar* if they contain identical replica states. So, state synthesis returns a

[26] Compromised leaders are still able to invent input packets to the prototype. But a compromised leader could always have invented such input packets simply by having a compromised client submit them as inputs.

[27] Input packets forwarded by the leader and replies to RepeatRequest messages in our implementation of Byzantine Paxos would normally be sent on the internal service network, but replicas in our prototype use spare capacity on the reboot network to send these messages. Replicas are technically disallowed from sending on the reboot network by Reboot Channels (2.10); our implementation effectively treats this spare capacity as an extra logical network.

[28] In our prototype, old messages are only kept for a small fixed number of recent sequence numbers. In general, the amount of state to keep depends on how fast the state machine ensemble processes commands. Since replicas can always execute State Recovery instead, the minimum number of messages to keep depends on how many messages are needed to run State Recovery, as discussed below.

[29] Note, however, that replicas might need to receive $\gamma + t$ replies to have γ that are output-similar.

replica state if it has received this replica state in state-similar replies from $t + 1$ distinct replicas.

State Recovery Request. Implementing a state recovery protocol requires defining the threshold θ of correct or rebooting replicas that must hold a replica substate for this replica substate to correspond to a system substate. We set θ to be 1; a replica substate is a system substate if it is held by any correct or rebooting replica. The order on θ-consistent replica substates is the order defined on replica states, since the replica state contains exactly one replica substate.

The state recovery request protocol used in State Recovery requires a channel that satisfies Timely Links (2.8), a bound T on the amount of time needed to contact all replicas, and a bound η on the time needed to marshal and send state from one replica to another. The protocol operates as follows.

(1) The recovering replica contacts each replica with a state recovery request.
(2) Each replica that receives this request performs two actions:
   (a) This replica sends its current state and its immediately previous state to the recovering replica, along with the messages that were processed for the replica to reach its current state from the immediately previous state.[30]
   (b) This replica also sends to the recovering replica all the messages that it receives for the agreement protocol over the next T seconds.
(3) After $T + \eta$ seconds, the recovering replica uses the states and messages it has received to recover.

For this algorithm to work, replicas must keep not only their current state, but also their immediately previous state (if any), along with all messages that they have received and verified for the current round of agreement.[31]

A recovering replica using this state recovery request protocol always recovers the state with the highest sequence number available at any correct replica at the time the last replica is contacted in the protocol. And this state is always recovered in a bounded amount of time. To state this property formally, we let s be the highest sequence number held by a correct replica at time T seconds after the beginning of the state recovery protocol.

[30] Instead of sending two successive states, the sending replica could send one state and a diff that allows the receiving replica to recover the other.

[31] An optimization is for replicas to reply immediately with their replica state the first time they receive a state recovery request from a recovering replica, instead of running the protocol. If a recovering replica i does not receive $t + 1$ identical replica states from these responses, then i can send a second request; upon receiving the second request, replicas run the above protocol. This optimization suffices for State Recovery as long as recovering replicas always get enough identical states to be able to recover in a bounded amount of time to the LDCS at the time the state recovery request was sent.

Our firewall prototype implements this optimization, and the system has never executed a second request, because recovering replicas always got $t + 1$ identical replica states on their first request in the experiments we ran and were able to recover quickly to a replica state greater than or equal to the LDCS.

(3.1) *SME Highest State Recovery.* $T + \eta$ seconds after the beginning of state recovery request protocol, the recovering replica will have received enough information to recover to the state with sequence number s.

See the Appendix for a proof of this property. SME Highest State Recovery (3.1) implies Maximal State Recovery (2.9), since s is guaranteed to be at least the Least Dominant Consistent State at the time the state recovery protocol starts, and s is also guaranteed to be a replica state at some correct replica, hence s was reached by applying messages to the LDCS, as required. So, this protocol succeeds at recovering a state for the recovering replica in a bounded amount of time, even when Byzantine Paxos operates asynchronously, as long as replicas can marshal representations of their state in a bounded amount of time.

As noted in Section 2.2.2, for State Recovery to be able to complete in bounded time, a recovering replica must also be able to replay its recorded packets and catch up with the other replicas in the system in bounded time. The processing of replayed packets might require replicas to send RepeatRequest messages and request packets they missed while recording. So, after receiving a State Recovery Request and before determining that a recovering replica has finished State Recovery, replicas must keep enough packets to bring recovering replicas up to date using RepeatRequest messages. The number of packets stored depends on the number q of extended sequence numbers processed by replicas during State Recovery. The value of q is bounded, since the firewall is assumed to have a bounded maximum throughput, and Maximal State Recovery (2.9) guarantees that State Recovery completes in bounded time.

For RepeatRequest messages to guarantee that packet replay completes in a bounded interval, the rate at which commands for extended sequence numbers are learned via RepeatRequest messages must be faster than the rate at which commands are handled by the firewall, hence recorded by the recovering replica. This guarantees that the recovering replica eventually processes all the commands it has recorded and can stop recording. So, the maximum throughput of the firewall must be chosen to take into account the time needed to learn a command for an extended sequence number via RepeatRequest messages.[32] Let B be a bound on the number of extended sequence numbers that a recovering replica will need to learn via RepeatRequest messages after State Recovery. If replicas store messages for at least B extended sequence numbers, then recovering replicas will be able to catch up with other replicas in a bounded amount of time using State Recovery.

3.2 Performance of the Firewall Prototype

The performance of the firewall prototype depends on how mechanisms are implemented. To quantify this, we ran experiments on the different instances of our firewall prototype that result from:

[32] This time is bounded, given Bounded-Rate Clocks (2.6), Synchronous Processors (2.7), and Timely Links (2.8)

Table I. The Versions of Our Firewall Prototype

Version name	Input Coordination	Reply Synthesis	Replica Refresh		
			epoch	reobf	key
Ftol-Baseline	$3t + 1$	indvdl auth	none	none	none
Centralized	$3t + 2r + 1$	indvdl auth	cntrl	cntrl	cntrl
Decentralized	$3t + 2r + 1$	indvdl auth	cntrl	cntrl	dist
Reboot Clock	$3t + 2r + 1$	indvdl auth	dist	dist	dist
Threshold Client	$3t + 2r + 1$	thresh crypto	dist	dist	dist

—Input Coordination performed either by Byzantine Paxos using $n = 3t + 1$ or $n = 3t + 2r + 1$; the version with $3t + 2r + 1$ allows for r rebooting replicas, while the version with $3t + 1$ does not.

—Reply Synthesis either based on individual authentication or based on threshold cryptography.

—Replica Refresh implemented either using a centralized Controller or using decentralized protocols.

In all experiments, State Recovery employs the protocol of Section 2.2.2 with state recovery request and state synthesis as described in Section 3.1. Input Coordination always uses Byzantine Paxos. The *Ftol-Baseline* version provides Byzantine Paxos but no rebooting replicas: it does not perform proactive obfuscation or any form of proactive recovery, thereby defining a baseline for performance of a fault-tolerant system without attack-tolerance. The result is five different versions of our firewall prototype, listed in Table I.

3.2.1 *Experimental Setup.* Our implementations are written in C using OpenSSL 0.9.7j [OpenSSL Project]; we also use OpenSSL for key generation. We take $t = 1$ and $n = 6$; all hosts are 3 GHz Pentium 4 machines with 1 GB of RAM running OpenBSD 4.0. We can justify setting $t = 1$ provided Bounded Adversary (2.2) is satisfied in this case; this requires that all $n = 6$ replicas be reobfuscated and rebooted before $t + 1 = 2$ replicas are compromised. The epoch length in our prototype is on the order of several minutes, so we believe this assumption to be reasonable. The Ftol-Baseline version only needs $3t + 1$ replicas to run Byzantine Paxos; it has $n = 4$.

A host called the *outside client* is connected to the input network of the firewall prototype. A host called the *inside client* is connected to the output network. The OpenBSD kernel of the inside client is modified for Reply Synthesis so that a packet passes through the inside client's network stack only if $\gamma = t+1$ output-similar packets have been received. This allows applications on the inside client to run unmodified. Replicas are connected to the output network and input network by hubs—all replicas use the same MAC and IP address and receive all packets sent by the outside client and inside client.

For ease of implementation, Input Coordination, Reply Synthesis, and State Recovery execute in user space; we built a pseudo-device that transfers packets from the kernel, as in Mogul's firewall design [Mogul 1989]. The pseudo-device allows programs running in user space to take and replace packets on the network stack, similar to Linux netfilter [Netfilter Project].

The pf code provides a pseudo-device called pfsync [OpenBSD b] that marshals and unmarshals an abstract state representation (pfsync was designed for synchronizing a backup to a primary pf firewall). The output of pfsync is a data structure that contains information about the state of the firewall.

The prototype employs three obfuscation methods: (i) system call reordering obfuscation [Chew and Song 2002] permutes the order of system call numbers and embeds them into a larger space of identifiers, most of which do not map to valid system calls; (ii) memory randomization is implemented by default in OpenBSD; and (iii) Propolice [Etoh] inserts and checks a random value after the return value of functions to protect against stack-smashing attacks. However, any obfuscation method applicable during compilation, linking, or loading could be used in our prototype. Recall that our interest is not with details of obfuscation but with mechanisms that must be available in conjunction with obfuscation in order to create and maintain replica independence.

The elapsed time between reboots bounds the window of vulnerability for cryptographic keys used by each replica. Our prototype uses 512-bit RSA keys, because the risk is small that an adversary will compute a private key from a given 512-bit public key during the relatively short (30 minute) window of vulnerability in which secrecy of the key matters.

We also use 512-bit RSA keys for the Ftol-Baseline version to allow direct performance comparisons with versions that implement attack tolerance by supporting proactive obfuscation. The Ftol-Baseline version does not need to use obfuscated replicas, but our implementation of the Ftol-Baseline uses obfuscated replicas because we found no runtime differences in speed between obfuscated and unobfuscated versions.[33]

Replicas batch input and output packets when possible, up to batch size 43—this is the largest batch size possible for 1500-byte packets if output batches are sent to clients as single packets, since the maximum length of an IP datagram is 64 kB.[34] But recall that agreement is done on hashes of client inputs and not the inputs themselves, so batching input packets involves batching hashes. Replicas also sign batched output packets for the client.

Finally, replicas in our prototype do not actually write their state to disk after executing each command, because the cost of these disk I/O operations would obscure the costs we are trying to quantify.

3.2.2 *Performance Measurements.*

Each reported value is a mean of at least five runs; error bars depict the sample standard deviation of the measurements around this mean.

3.2.2.1 *Input Coordination.*

To quantify how throughput is affected by the use of $3t+1$ or $3t+2r+1$ replicas in Input Coordination, we performed experiments in which there were no compromised, crashed, or rebooting replicas, so

[33] This result will not hold for all obfuscators, since code that depends on locality of reference can be slowed significantly by some obfuscations.

[34] Implementing higher batching factors requires using or implementing a higher-level notion of message fragmentation and reassembly.

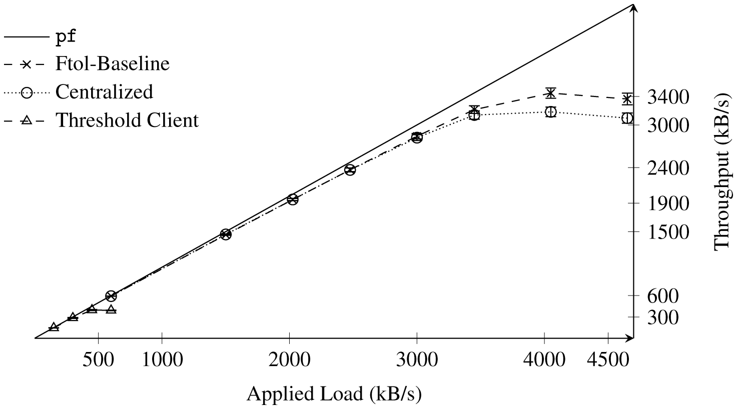

Fig. 3. Overall throughput for the firewall prototype.

Replica Refresh and State Recovery could be disabled without averse effect. We consider two prototype versions that differ only in the number of replicas: the Ftol-Baseline version and the Centralized version. Both use RSA signatures for Reply Synthesis.

Throughput for Input Coordination. During each experiment, the outside client sends 1500-byte UDP packets (the MTU on our network) to a particular port on the firewall; firewall rules cause these packets to be forwarded to the inside client. Varying the timing of input packet sends enables us to control bandwidth applied by the outside client. Figure 3 shows throughput for each of the versions.[35] The curve labeled `pf` gives the performance of the `pf` firewall running on a single server; it handles an applied load of up to at least 12.5 MB/s—considerably higher bandwidths than we tested.

The Centralized version throughput reaches a maximum of about 3180 kB/s, whereas the Ftol-Baseline version reaches a maximum throughput of about 3450 kB/s. So, the Centralized version reaches about 92% of the throughput of the Ftol-Baseline version under high load. This suggests that the cost of the additional replication over Ftol-Baseline needed to support proactive obfuscation in an already-replicated system is not excessive.

The Ftol-Baseline version and the Centralized version throughputs peak due to CPU saturation. Some of the CPU costs result from the use of digital signatures to sign packets both for Input Coordination and for Reply Synthesis. These per-packet costs are amortized across multiple packets by batching, so these costs could be reduced significantly, either by batching or by faster authentication mechanisms. The other major cost, that cannot be reduced in our implementation, arises from copying packets between the kernel and our mechanism implementations which run in user space. Multiple system calls to our pseudo-device are performed for each packet received by the kernel. Reducing this cost is possible by moving the code for proactive obfuscation into the kernel.

[35] Discussion of the Threshold Client in Figure 3 curve appears in Section 3.2.2.2.

To quantify the cost of these calls to pseudo-devices, we ran microbenchmarks in which we passed multiple 1500-byte packets to and from our pseudo-device and measured the round-trip time. Our pseudo-device implementation requires a mean time of 3.75 μs to copy to the kernel and 4.72 μs to copy from the kernel into our application. Each packet handled by our system is copied to and from our pseudo-device twice, so the mean overhead is 16.94 μs. At 4.5 MB/s of applied load, CPU utilization is at 100%, and the cost of packet copying becomes an overhead of 52.04 ms per second, a contribution of about 5% of the CPU utilization. So, the kernel-copy overhead, while not negligible, is dominated by the other CPU costs of processing packets in our prototype.

Throughput decreases for both the Ftol-Baseline and the Centralized versions after the CPU becomes saturated. The throughput decrease accelerates because higher applied loads means that replicas spend more time dropping packets. Dropped packets consume non-trivial CPU resources, since all packets in the firewall prototype are copied into user space and deleted from the kernel before being handled.

The choice of batching factor and the choice of timeout for leader recovery both affect throughput when a replica has crashed. To quantify this effect, we ran an experiment similar to the one for Figure 3, but with one replica crashed. While the replica was crashed, throughput in the Centralized version drops to 1133 ± 10 kB/s.[36] The decrease in throughput when one replica is crashed occurs because the failed replica cannot act as leader when its turn comes, and therefore replicas must wait for a timeout (chosen to be 200 ms in the prototype) each 6 sequence numbers, at which point the next consecutive replica runs the leader recovery protocol and acts as leader for this sequence number.

Latency for Input Coordination. Latency in the firewall prototype also is affected by the number of replicas. In the experiment used to produce Figure 3, latency was measured at 39 ms ± 3 ms for the Centralized version, whereas latency in the Ftol-Baseline version was 28 ms ± 6 ms under the same circumstances. This difference is due to running fewer replicas in the Ftol-Baseline version, with replicas in Ftol-Baseline needing to handle fewer replies from replicas per message round in the execution of Input Coordination. This 40% difference is small in absolute terms, so it is likely to be undetectable when the service is accessed across a wide-area network.

Unlike throughput, however, latency is not affected by the batching factor, since latency depends only on the time needed to execute the agreement algorithm.[37] And batching is opportunistic, so replicas do not wait to fill batches. This also keeps batching from increasing latency.

[36] Linear changes in the batching factor provide proportional changes in the throughput during replica failure: the same experiment with a batching factor of 32 leads to a throughput of 873 ± 18 kB/s.

[37] Of course, larger batching factors cause replicas to transmit more data on the network for each packet, and this increases the time to execute agreement. But this increase is negligible in all cases we examined.

To understand the latency when one replica is crashed, we ran an experiment where the outside client sent 1500-byte packets, but with one replica crashed. With a crashed replica, latency increases to 342 ms ± 60 ms for the Centralized version. This increase is because packets normally handled by the failed replica must wait for a timeout and leader recovery before being handled. This slowdown reduces the throughput of the firewall, causing input-packet queues to build up on replicas. Latency for each packet then increases to include the time needed to process all packets ahead of it in the queue. And some packets in the queue have to wait for the timeout. In the Centralized version, the timeout is set to 200 ms, so the latency during failure is higher, as would be expected.

3.2.2.2 *Reply Synthesis*

Throughput for Reply Synthesis. Throughput for different Reply Synthesis implementations is already given in Figure 3—in an experiment where no Replica Refresh occurs, any differences between the Centralized version and the Threshold Client version can be attributed solely to their different implementations of Reply Synthesis: the Centralized version uses RSA signatures, whereas the Threshold Client version uses threshold RSA signatures.

Figure 3 confirms the prediction of Section 2.3.1 that the Threshold Client version exhibits significantly lower throughput, due to the high CPU costs required for generating partial signatures using the threshold cryptosystem in the CODEX implementation of APSS. Compare the maximum throughput of 397 kB/s with 3180 kB/s measured for the Centralized version, which does not use threshold cryptography.

Latency for Reply Synthesis. Latency for the Threshold Client version (measured in the same experiment as for throughput) is 413 ms ± 38 ms as compared with 39 ms ± 3 ms for the Centralized version. Again, this difference is due to high CPU overhead of threshold RSA signatures.

3.2.2.3 *Replica Refresh*. We evaluate the three tasks comprising Replica Refresh separately, for both the centralized and the decentralized implementations. Due to the high costs of threshold cryptography, we use RSA signatures for Reply Synthesis throughout these experiments. We set the outside client to send at 3300 kB/s, slightly above the throughput-saturation threshold.

We measured no differences in throughput or latency in our experiments for the two different implementations of replica reboot and epoch-change notification.

The time required to generate an obfuscated executable affects elapsed time between reboots. Obfuscating and rebuilding a 22 MB executable (i.e., all our kernel and user code) using the obfuscation methods employed by our prototype takes about 13 minutes with CD-ROM-based executable generation at the replicas and takes 2.5 minutes with a centralized Controller; reboot takes about 2 minutes in both. Both versions allow about 30 seconds for State Recovery, which is more than sufficient for the amount of state being maintained by the firewall in our experiments.

It is difficult to quantify how much of the rebuild time is due to obfuscation. The obfuscations implemented by OpenBSD are integrated with the compiler and operating system, and our simple syscall obfuscation takes at most a couple of seconds per replica to run. So, we divide the proactive costs into two parts: (i) reboot and refresh, which is required for any system that employs proactive recovery, and (ii) reobfuscation and rebuild, which is needed for proactive obfuscation alone.

A Controller can perform reobfuscation for one replica while another replica is rebooting, so reobfuscation and reboot here can be overlapped. This means that a new replica can be deployed approximately every 3 minutes. There are 6 replicas, so a given replica is obfuscated and rebooted every 18 minutes. In comparison, with decentralized protocols, reobfuscation, reboot, and recovery in sequence take about 15 minutes, so a given replica is obfuscated and rebooted every 90 minutes.

The cost of using our decentralized protocols for generating executables has a significant effect on the performance of the Reboot Clock version, but this effect is limited to extending the window of vulnerability caused by the extra time needed for CD-ROM-based executable generation. The performance of the Reboot Clock version is otherwise identical to the performance of the Decentralized version, so we do not present any measurements of the Reboot Clock version.

Key distribution for Replica Refresh involves generating, signing, and disseminating a new key for a recovering replica. In the decentralized implementation, replicas refresh their shares of the private key for the service at each epoch change and participate in a share recovery protocol for the recovering replica after reboot. The costs of generating, signing, and disseminating a new key are small in both versions, but the costs of share refresh and share recovery are significant.

Throughput for Replica Refresh. To understand the throughput achieved during share refresh and share recovery, we ran an experiment in which one replica is rebooted. We measured throughput of two versions that differ only in how key distribution is done: the Centralized version uses a centralized Controller while the Decentralized version requires the rebooting replicas to generate their own keys and use the key distribution protocol of Section 2.2.3.2 to create and distribute a certificate for this key.

Figure 4 shows throughput for these two versions in the firewall prototype while the outside client applies a constant UDP load at 3300 kB/s. During the first 50 seconds, all replicas process packets normally, but at the time marked “epoch change”, one replica reboots and the epoch changes.[38] In the Decentralized version, non-rebooting replicas run the share refresh protocol at this point; the high CPU overhead of this protocol in the CODEX implementation

[38] The drop occurs on the graph slightly before the 50-second mark due to time differences between the script controlling the outside client and the measurement of throughput at the servers.

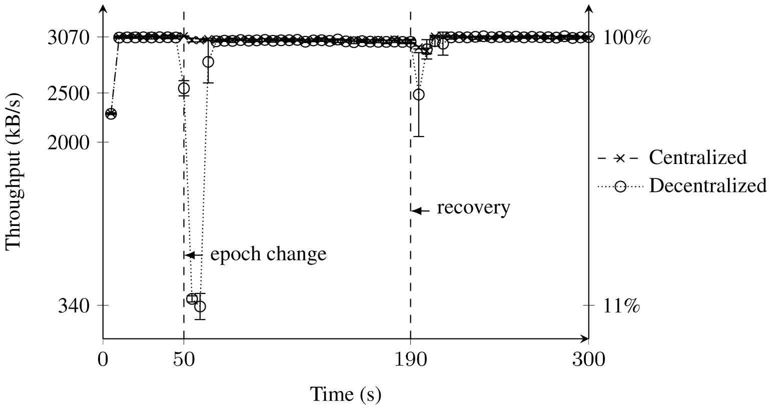

Fig. 4. Two key distribution methods for the firewall prototype at an applied load of 3300 kB/s.

of APSS causes throughput to drop to about 340 kB/s, which is about 11% of the maximum. In the Centralized version, replicas have no shares to refresh, and both versions perform leader adjustment to take the rebooting replica into account, so there is no throughput drop.

In both versions, the recovering replica executes the State Recovery mechanism at the point marked “recovery” in Figure 4. In the Decentralized version, the recovering replica also runs its share recovery protocol. Throughput drops more in the Decentralized version than the Centralized version due to the extra CPU overhead of executing share recovery.

Latency for Replica Refresh. Latency increases during share refresh. The same experiment as for measuring throughput shows that the Centralized version has a latency of 37 ms ± 2 ms after an epoch change, similar to its latency of 39 ms ± 3 ms when there are no failures. But latency in the Decentralized version goes up to 2138 ms ± 985 ms during the same interval. The high latency occurs because few packets can be processed while APSS performs share refresh. Latency also increases slightly during share recovery in the Decentralized version to 65 ms ± 26 ms.

3.2.2.4 *State Recovery*. State Recovery does not significantly degrade throughput, as shown in Figure 4 for the Centralized version at the point labeled “recovery”. The low cost of state recovery is due to the small amount of state stored by our firewall for each packet stream; each stream is represented by a structure that contains 240 bytes. And the outside client uses multiple 75 kB/s packet streams to generate load for the firewall. So, there are 44 streams at an applied load of 3300 kB/s. This corresponds to 10560 bytes of state; recovery then requires each replica to send 8 packets with at most 1500 bytes each. The overhead of sending and receiving 8 packets from $t + 1 = 2$ replicas and updating the state of `pf` at the recovering replica is small. And the cost of using RepeatRequest packets to catch up with other replicas is likewise minimal.

4. QUORUM SYSTEM REPLICA MANAGEMENT

A *quorum system* [Thomas 1979; Gifford 1979; Herlihy 1986] stores *objects* containing *object state*; an object supports *operations* to modify its object state. A quorum system is defined by a collection $\mathcal{Q}$ of *quorums*—sets of replicas that satisfy an *intersection property* guaranteeing some specified overlap between any pair of quorums. Each replica stores object states.

Clients of a quorum system perform an operation on an object by reading that object's state from a quorum of replicas, combining those states, and executing the operation using the result, then writing an updated object state back to a (potentially different) quorum of replicas. We follow a common choice [Malkhi and Reiter 1998] for the semantics of concurrent operations on a given object:

(1) Reads that are not concurrent with any write generate the latest object state written to a quorum, according to some serial order on the previous writes.

(2) Reads that are concurrent with writes either *abort*, which means they do not generate an object state, or they return an object state that is at least as recent as the last object state written to a quorum.

On abort, the operation can be retried.

The object state stored by a replica is labeled with a totally ordered, monotonically-increasing *sequence number*, and the label is kept as part of the object state. The label is generated by the client that wrote the object state. Replicas only store a new object state for an object o if the new object state is labeled with a higher sequence number than the object state already stored by this replica for o.

An intersection property on quorums ensures that a client reading from a quorum obtains the most recently-written object state. For instance, when there are no crashed or compromised replicas, we could require that any two quorums have a non-empty intersection; then any quorum from which a client reads an object overlaps with any quorum to which a client writes that object. So, a client always reads the latest object state written to a quorum when there are no concurrent writes.

Byzantine quorum systems [Malkhi and Reiter 1998] are defined by a pair $(\mathcal{Q}, \mathcal{B})$; collection $\mathcal{Q}$ of replica sets is as before, and collection $\mathcal{B}$ defines the possible sets of compromised replicas. By assumption, only replicas in some set B in $\mathcal{B}$ may be compromised. In a *threshold fail-prone* system, at most some threshold t of replicas can be compromised, so $\mathcal{B}$ consists of all replica sets of size less than or equal to t.

Our prototype implements a *dissemination quorum system* [Malkhi and Reiter 1998]; this is a Byzantine quorum system where object state is *self-verifying* and, therefore, there is a public *verification* function that *succeeds* for an object state only if this object state was written by a client and has not since been changed by a compromised replica. For instance, an object state signed by a client with a digital signature is self-verifying, since signature verification succeeds only if the object state is unmodified from what the client produced.

A dissemination quorum system where $\mathcal{B}$ is a threshold fail-prone system for threshold t must satisfy the following properties [Malkhi and Reiter 1998]:

(4.1) *Threshold DQS Availability.* $\forall Q \in \mathcal{Q} \ : \ n - t \geq |Q|$

(4.2) *Threshold DQS Correctness.* $\forall Q_1, Q_2 \in \mathcal{Q} \ : \ |Q_1 \cap Q_2| > t$

Threshold DQS Availability (4.1) and Threshold DQS Correctness (4.2) are satisfied if $n = 3t + 1$ holds and, for any quorum Q, $|Q| = 2t + 1$ holds, since then $\forall Q_1, Q_2 \in \mathcal{Q} \ : \ |Q_1 \cap Q_2| = t + 1 > t$ and $\forall Q \in \mathcal{Q} \ : \ n - t = 2t + 1 = |Q|$ both hold, as required.

Given these properties, a client can read the latest object state by querying and receiving responses from a quorum. Threshold DQS Availability (4.1) guarantees that some quorum is available to be queried. And Threshold DQS Correctness guarantees that any pair of quorums overlaps in at least $t+1$ replicas, hence at least one correct replica is in the overlap. The latest object state is written to a quorum, so any quorum that replies to a client contains at least one correct replica that has stored this latest object state. To determine which object state to perform an operation on, a client chooses the object state that it receives with the highest sequence number. This works because object states are totally ordered by sequence number and are self-verifying, so the client can choose the most-recently written state from only those replies containing object state on which the verification function succeeds.

If object states were not self-verifying, then compromised replicas could invent a new object state and provide more than one copy of it to clients—these clients would not be able to decide which object state to use when performing an operation. With object states required to be self-verifying, the worst that compromised replicas can do is withhold an up-to-date object state.

Dissemination quorums guarantee that the latest object state is returned if objects are not written by compromised clients. If compromised replicas knew the signing key for a compromised client, then these replicas could sign an object state with a higher sequence number than had been written. Clients then would choose the object state created by the compromised replicas. One way to prevent such attacks—and the approach we adopt—is by allowing only one client per object o to write object state for o but allowing any client to read it.[39]

When at most r replicas in a threshold fail-prone system with threshold t might be rebooted proactively, a quorum system must take these rebooting replicas into account. Maintaining Threshold DQS Availability (4.1) requires that there be a quorum that clients can contact even when $t + r$ replicas are unavailable. Formally, we can write this property as follows [Sousa 2006]:

(4.3) *Proactive Threshold DQS Availability.*

$$\forall Q \in \mathcal{Q} \ : \ n - (t + r) \geq |Q|$$

[39] Allowing only a single client per object to write object state is reasonable for many applications. For instance, web pages are often updated by a single host and accessed by many.

Setting $n = 3t + 2r + 1$ and defining quorums to be any sets of size $n - (t + r) = 2t + r + 1$ suffices to guarantee Proactive Threshold DQS Availability (4.3) because $n - (t + r) = |Q|$ holds, and this satisfies Proactive Threshold DQS Availability (4.3) directly. Given that r replicas might be rebooting, hence unavailable, it might seem that requiring only $t + 1$ replicas in quorum intersections would be insufficient to guarantee that clients receive the most up-to-date object state. Notice that Proactive Threshold DQS Availability (4.3) guarantees that a quorum Q' of $2t + r + 1$ replicas is always available, where Q' does not contain any rebooting replicas. So, an intersection between Q' and any other quorum consists only of replicas that are not rebooting. And an overlap of $t + 1$ replicas from Q' is sufficient to guarantee that at least one correct replica replies with the most up-to-date object state, as long as no writes are executing concurrently. So, Proactive Threshold DQS Correctness (4.4) is the same as Threshold DQS Correctness (4.2):

(4.4) *Proactive Threshold DQS Correctness.* $\forall Q_1, Q_2 \in \mathcal{Q} \; : \; |Q_1 \cap Q_2| > t$

The values of the quorum size ($|Q| = 2t + r + 1$) and the number of replicas chosen above ($n = 3t + 2r + 1$) suffice to satisfy this property, since any two replica sets of size $2t + r + 1$ out of $3t + 2r + 1$ replicas overlap in at least $t + 1 > t$ replicas.

4.1 A Storage-Service Prototype

To confirm the generality of the various proactive obfuscation mechanisms we implemented for the firewall prototype, we employed them to implement a storage-service prototype using a dissemination quorum system over a threshold fail-prone system with threshold t and one concurrently rebooting replica. The prototype supports read and write operations on objects with self-verifying object state. Object states stored by replicas are indexed by an *object ID*. The object state for each object can only be written by a single client, so the object ID contains a client ID. Clients sign object state with RSA signatures to make the object state self-verifying; the client ID is associated with the client's public key.

The storage service supports two operations, which are implemented by the replicas as follows. A *query* operation for a given object ID returns the latest corresponding object state or a *unknown-object message* when no replica currently stores an object state for this object ID. An *update* operation is used to store a new object state; a replica only performs an update when given an object state having a higher sequence number than what it currently stores. If a replica applies an update, then it sends a *confirmation* to the client; the confirmation contains the object ID and the sequence number from the updated object state. Otherwise, it sends an *error* containing the object ID and the sequence number to the client.

Adding proactive obfuscation to this service requires instantiating the output synthesis and state synthesis functions as well as defining the action taken by a replica on receiving a state recovery request.

Quorum systems have the potential to create data independence, since each replica might store different data. But a quorum system in which any message sent to one replica is sent to all replicas does not have this property; most replicas will store the same data. Our architecture for proactive obfuscation sends every message to all replicas, so our quorum system implementation exhibits little data independence.

Synthesis Functions. Both the output and state synthesis functions require replies from a quorum.

For the implementation of a Reply Synthesis mechanism that authenticates individual replicas, the output synthesis function operates as follows. Object states are defined to be output-similar if they have the same object ID. So, output synthesis for object states returned by queries waits until it has received correctly-signed, output-similar object states from a quorum. Then it returns from that set the object state with the highest sequence number or returns an unknown-object message if all replies contain unknown-object messages. Confirmations are defined to be output-similar if they have the same object ID and sequence number. So, for updates, the output synthesis function returns a confirmation for an object ID and sequence number when it has received output-similar confirmations for this object ID and sequence number from a quorum. Otherwise, it returns abort if no quorum returned confirmations (so, some replies must have been errors instead of confirmations). In both cases, γ is set to the quorum size.[40]

The implementation of a Reply Synthesis mechanism using threshold cryptography would be incompatible with a dissemination quorum system. A client of a dissemination quorum system must authenticate replies from a quorum of replicas, and different correct replicas in that quorum might send different object states in response to the same query—dissemination quorum systems only guarantee that at least one correct replica in a quorum returns an object state with the most up-to-date sequence number. This weaker guarantee violates the assumption of the threshold cryptography Reply Synthesis mechanism that at least $t+1$ replicas send an identical value. So, we are restricted to using the individual authentication implementation of Reply Synthesis in our storage-service prototype.[41]

For state synthesis, object states are defined to be state-similar if they have the same object ID. Unknown-object messages from replicas are state-similar with each other and with object states if they have the same object ID. We say that sets containing object states and unknown-object messages are state-similar if, for any object state with a given object ID in one of the sets, each other

[40] There is no communication between replicas in the quorum system, so replicas that have already performed the update keep the updated object state even if no quorum returned confirmations to the client.

[41] Other types of Byzantine quorum system [Malkhi and Reiter 1998] require more overlap between quorums. In some, $2t+1$ replicas must appear in the intersection of any two quorums, hence the intersection will include at least $t+1$ correct replicas that return the most up-to-date object state. The threshold cryptography implementation of Reply Synthesis would work in such a Byzantine quorum system, since the threshold for signature reassembly is $t+1$.

set has an object state or an unknown-object message with the same object ID. So, the state synthesis function waits until it receives correctly-signed, state-similar sets from a quorum and, for each object state o in at least one set, returns the object state for o with the highest sequence number. This means that δ is also set to the quorum size.

Since object states are self-verifying, they cannot be modified by replicas; this means that all replicas must return object states sent by clients. Therefore, there is no need for marshaling and unmarshaling an abstract state representation, unlike in the firewall prototype.

State Recovery Request. In the storage-service prototype, a recovering replica sends a state recovery request to all replicas and waits until it has received replies from a quorum; upon receiving a state recovery request, a correct replica sends to the recovering replica the object state for each object it has stored.

To show that State Recovery will work with this protocol, we must define replica substates and threshold θ. For a quorum system, a replica substate is an object with a given unique object ID; then the total order on θ-consistent replica substates for the same object ID is simply the total order on the sequence numbers. The value θ is set to $t+r+1$, the number of correct or rebooting replicas in a quorum.[42] These definitions guarantee that each object state in the LDCS was written correctly to the quorum system; they also imply that the LDCS contains all the latest correctly-written object states at a given time.

Since an object state is self-verifying, it does not need to be signed by the sending replica. For each object state with object ID o that was received from one replica i but not from another replica j, the recovering replica creates an unknown-object message with object ID o in the reply from j.[43] This makes the object IDs found in each reply set the same, and, therefore, makes the replies received by the recovering replica state-similar.

The State Recovery protocol of Section 2.2.2, using the definition of the state recovery request above, satisfies Maximal State Recovery (2.9). To see why, notice that Proactive Threshold DQS Availability (4.3) guarantees that some quorum is available even when one replica is rebooting and even when actions by the quorum system are asynchronous. The recovery request from a rebooting replica goes to a quorum, and Proactive Threshold DQS Correctness (4.4) guarantees that, for each object o, this quorum intersects in at least one correct replica with the quorum that had object state for o with the current maximum

[42] Note that if all replicas are correct, then it is possible for a compromised client to write two different θ-consistent replica substates with the same object ID and sequence number, since $2(t + r + 1) \leq 3t + 2r + 1$. This might prevent the replica substate from being recovered correctly, as noted in Section 2.2.2. However, all objects written correctly to quorums will have a total order, hence recovery will work as expected.

[43] A replica replying to a state recovery request also sends a signed list of the object IDs that it will send. This list allows the recovering replica to know which object states to expect. So, the recovering replica knows when it has received all the object states from a quorum. At this time, it can safely add unknown-object messages for each object state that was received from some, but not all, replicas in the quorum. An added unknown-object message need not be signed, since it is only used by the replica that creates it.

sequence number for o at the time the recovery protocol started. So, this correct replica answers with object state for o with a sequence number that is at least the value of the current maximum sequence number for o when the recovery protocol started. The state synthesis function will thus return an object state with at least this sequence number, and any later object recovered will have been the result of handling subsequent messages, as required. The recovering replica then processes incoming operations that it has queued during execution of the State Recovery protocol, so it recovers with an up-to-date state. Bounded-Rate Clocks (2.6), Synchronous Processors (2.7), and Timely Links (2.8), along with the assumed bound on state size, guarantee that this protocol terminates in a bounded amount of time.

Recovering replicas in the storage-service prototype implement a simple optimization for State Recovery: since correct client inputs are signed and have a totally-ordered sequence number, a recovering replica can process inputs that it receives while executing State Recovery instead of recording the inputs and processing them after. If a recovering replica stores an object state with a higher sequence number than the one it eventually recovers, then the older state is dropped instead of being stored after recovery. This means that State Recovery terminates for a recovering replica immediately once this replica receives δ state-similar replies. And this optimization does not interfere with completing State Recovery in a bounded amount of time, because there is an assumed maximum rate for inputs received by the replicas, and there is a bound on the amount of time needed to process any input, due to Synchronous Processors (2.7) and Bounded-Rate Clocks (2.6).[44]

4.2 Performance of the Storage-Service Prototype

The performance of the storage-service prototype depends on the mechanism implementations used for Reply Synthesis, State Recovery, and Replica Refresh. Input Coordination is not needed for quorum systems, and Reply Synthesis has no implementation here using threshold cryptography. All versions of our prototype use the State Recovery implementation from Section 2.2.2 with the state recovery request as described in Section 4.1.

Table II enumerates the salient characteristics of the four versions of the prototype we analyzed. We include a *Ftol-Baseline* version that implements a quorum system using only $3t + 1$ replicas, does not take rebooting replicas into account, and therefore is not designed to be attack-tolerant. As in the firewall prototype, replicas in the Ftol-Baseline version are obfuscated.

4.2.1 *Experimental Setup.* To measure performance of the storage-service prototype, we use the experimental setup described in Section 3.2.1, with $t = 1$ and $n = 6$; a quorum is any set of 4 replicas. Since the Ftol-Baseline version does not perform proactive obfuscation or reboot replicas, it only needs $n = 3t+1 = 4$ replicas and uses quorums consisting of $2t + 1 = 3$ replicas.

[44] Note that local read and write operations on a replica are subject to Synchronous Processors (2.7), so code executing state recovery request protocol can read the local replica state in a bounded amount of time.

Table II. The Versions of Our Storage-Service Prototype

Version name	Input Coordination	Replica Refresh		
		epoch	reobf	key
Ftol-Baseline	$3t+1$	none	none	none
Centralized	$3t+2r+1$	cntrl	cntrl	cntrl
Decentralized	$3t+2r+1$	cntrl	cntrl	dist
Reboot Clock	$3t+2r+1$	dist	dist	dist

We implemented the storage-service server and client in C using OpenSSL. The server and client make no assumptions about replication—the client is designed to communicate with a single instance of the server. All replication is handled by the mechanism implementations.

There is a single client connected to both the input and output networks; this client submits all operations to the prototype and receives all replies. With no Input Coordination, there is no batching of input packets, but replicas sign batched output for the client up to batch size 20—reasons for this batching size are given in Section 4.2.2. Replicas do not currently write to disk for crash recovery.

4.2.2 *Performance Measurements.* We measure throughput and latency. Each reported value is a mean of at least 5 runs; error bars depict the sample standard deviation of the measurements around this mean.

4.2.2.1 *Reply Synthesis*. We performed experiments to quantify the performance of the individual authentication implementation for Reply Synthesis. In all experiments, objects are 4-byte integers with 24-byte headers containing metadata such as the object ID and the sequence number. Each object also stores a signature; in our implementation, this signature occupies 128 bytes, since clients use 1024-bit RSA. The client performs queries, but not updates, on randomly chosen objects at a specified rate; this workload allows us to characterize an upper bound on the throughput of the service and a lower bound on its latency—the objects are as small as possible and queries do not incur CPU time on the client for generating signatures or on the server for verification (whereas updates would). Update operations by the client would only increase the cryptographic computation done at the client and server, hence slow both down.

It might seem 4-byte integer objects are too small to be interesting. But this choice actually represents the worst case for a real system. Any object can be represented by a set of 4-byte integer parameters, so there is no loss of generality, and dividing objects into 4-byte units imposes the maximum possible overhead on the system for representing objects.

Figure 5 graphs throughput for our Reply Synthesis experiments. Throughput of the Centralized version peaks at 1360 queries/second at an applied load of 1550 queries/second and then declines slightly as the applied load increases. This decline is due to the cost of increasing numbers of queries being dropped because they cannot be handled. The "QU" curve in Figure 5 shows the performance of a single server and thus characterizes the best possible case.

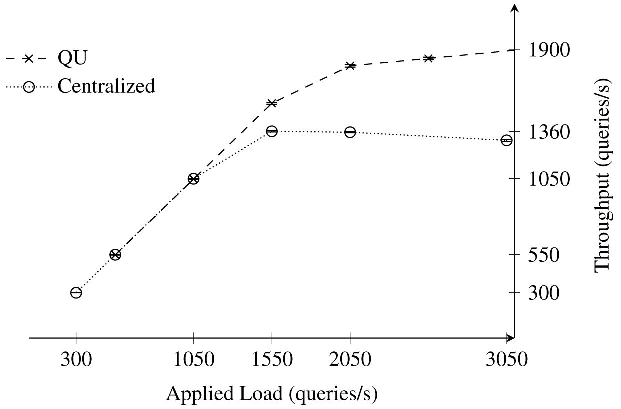

Fig. 5. Overall throughput for the storage-service prototype.

The throughput of the server starts saturating at about 1550 queries/second, and thereafter degrades slightly as applied load further increases. Throughput breakdown occurs in both the QU and Centralized cases due to CPU saturation. The difference in behavior at saturation is due, in part, to implementing the storage-service prototype in user space instead of in the kernel. Replicas copy packets from the kernel and manage them there, and even packets that are dropped are first copied from the kernel to user space. So, high applied loads induce a significant CPU overhead in the replicas. But in the single server version, packet queues are managed by the OpenBSD kernel, so dropping packets is less costly.[45]

The Ftol-Baseline version exhibited exactly the same performance as depicted in Figure 5 for the Centralized version. This is because quorum systems do not use Input Coordination, so each replica handles packets independently of all others. Thus, CPU saturation occurred in the Ftol-Baseline version at exactly the same number of queries per second as in the Centralized version. Client overhead in the Centralized version is 1/3 higher than in the Ftol-Baseline version, since quorum size increases in the Centralized version from 3 replicas to 4. So, clients must be adequately provisioned to handle the slightly increased load.

The latency of the storage-service prototype in these experiments is 21 ms ± 1 ms for applied load below the throughput saturation value. When the system is saturated, latency of requests that are not dropped increases to 205 ms ± 3 ms. Higher latency at saturation is due to bounded queues filling in the replica and in the OpenBSD kernel, since the latency of a packet includes the time needed to process all packets ahead of it in these queues. The queue implementations in the firewall and storage-service prototypes are different, so these latency behaviors are incomparable.

CPU saturation occurs for the Ftol-Baseline version at 1360 queries/second—the maximum load a replica can handle. Given this bound, we ran

[45] We expect to see the same effect of decreasing throughput in the QU plot, but at higher applied loads.

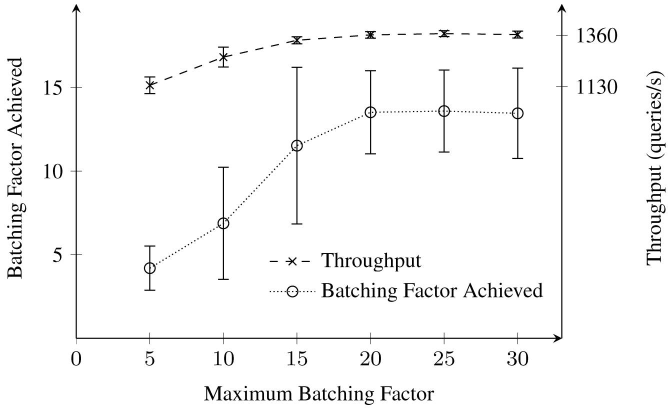

Fig. 6. Batching factor and throughput for the storage-service prototype under saturation.

our experiments for Replica Refresh and State Recovery at an applied load of 1400 queries/second, and thus we explored behavior at saturation. We allowed servers to reach a steady state in their processing of packets before starting our measurements.

To select a suitable batching factor, we performed an experiment in which a client applied a query load sufficient to saturate the service and varied the batching factor. Figure 6 shows the results. Throughput reaches a plateau at a maximum batching factor of 20. The batching factor achieved by replicas can also be seen in Figure 6—it also reaches a peak at a maximum batching factor of 20 and declines slightly thereafter (the throughput decline is due to the overhead of unfilled batches). So, we chose a batching factor of 20 for all our experiments.

Unlike in the firewall prototype, throughput here is unaffected by the crash of a single replica. Throughput in the firewall prototype decreases due to slowdown in Input Coordination, but the storage-service prototype does not use Input Coordination.

4.2.2.2 *Replica Refresh.* The Decentralized and Reboot Clock versions of our storage-service prototype employ different mechanism implementations for Replica Refresh. However, the only component of Replica Refresh that causes significant performance differences in throughput and latency between prototype versions is share refresh and share recovery for key distribution. The Decentralized version and the Reboot Clock version do not differ in their implementation of key distribution, so we only show results for the Decentralized version. The difference in reobfuscation between the two versions causes a difference in epoch length, as discussed for the firewall prototype in Section 2.2.3.

Throughput for Replica Refresh. Query throughput measurements given in Figure 7 confirm the results of Section 3.2.2.3, which compares centralized and decentralized key distribution for the firewall prototype. The experiment used

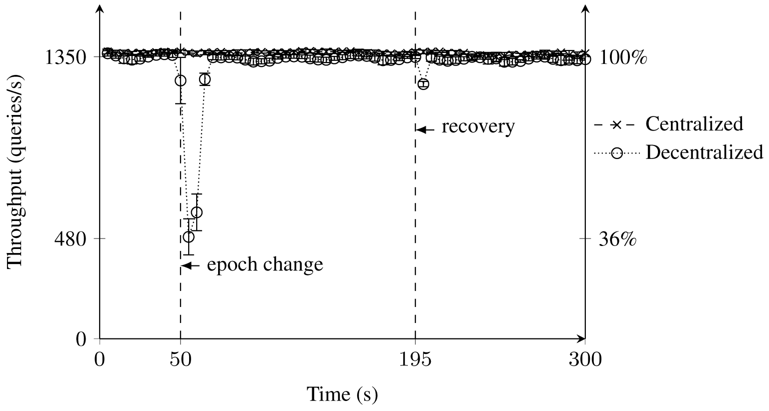

Fig. 7. Two key distribution methods for the storage-service prototype at an applied load of 1400 queries/second.

to generate these numbers is the same as for Reply Synthesis: a client sends random queries at a specified rate to the storage-service prototype, which replies with the appropriate object. We eliminate the costs of State Recovery by providing the appropriate state directly to the recovering replica—this isolates the cost of key distribution. As in the firewall prototype, the CPU overhead of APSS recovery causes a throughput drop at epoch change. Throughput decreases to about 36% (the firewall prototype dropped to 11%) for the Decentralized version, whereas the Centralized version continues at constant throughput for the whole experiment. We also observe a slight drop in the Decentralized version at recovery due to the share recovery protocol run for the rebooting replica. This drop is slightly shorter in duration than in the corresponding graph for the firewall prototype, since only share recovery is executed rather than State Recovery and share recovery in sequence.

The difference in throughput between the firewall and storage-service prototypes during share refresh can be explained by differences in CPU utilization. The storage-service prototype spends more of its CPU time in the kernel, whereas the firewall prototype spends most of its CPU time in user space performing cryptographic operations for agreement. We believe that kernel-level packet handling operations performed by the storage service mesh better with the high CPU utilization of APSS than the cryptographic operations performed by the firewall.

The same experiment run at 1300 queries/second (slightly below the saturation threshold) exhibits a throughput decrease during share refresh from 1300 queries/second to 750 queries/second; this is 57%. Throughput remains higher during share refresh in the non-saturated case than the saturated case, because the storage service does not use the CPU as much and, therefore, does not compete as much with APSS. Moreover, the higher load of queries in the saturated case forces the storage service to spend more CPU resources dropping packets than it must spend in the non-saturated case.

Latency for Replica Refresh. We do not give a graph of the latency for these experiments. The graph is not surprising in light of the graph for throughput; both graphs measure performance over time, and where there are dips in the throughput graph, there are peaks in the latency graph. Latency in the Decentralized case for the storage-service prototype increases to 646 ms ± 89 ms during share refresh, as opposed to 205 ms ± 3 ms for the same interval in the Centralized case. This latency is lower than what was seen in Section 3.2.2.3 for the firewall prototype during share refresh. As for throughput, we believe this is due to the kernel-level packet handling operations of the storage-service prototype competing better with the high CPU costs of APSS in user space. Latency also increases slightly during share recovery. The Decentralized version has a latency of 233 ms ± 3 ms during share recovery, whereas the Centralized version has a latency of 203 ms ± 1 ms during the same interval of time.

4.2.2.3 *State Recovery*. The number of object states that must be recovered after a reboot in the storage-service prototype significantly impacts throughput and latency. The size of the storage-service state increases linearly in the number of object states stored. Each object state in the prototype only contains a 4-byte integer and also has headers of length 24 bytes and a signature of length 128 bytes, so each object contains 156 bytes. So, a storage-service state with 68 object states has 10608 bytes (which corresponds to a firewall state with about 44 packet streams, since each stream has a state of size 240 bytes). This corresponds to an applied load of 3300 kB/s, by the same calculation as in Section 3.2.2.4. And, therefore, the amount of state held by replicas in the firewall prototype under saturation is held by replicas in the storage-service prototype when there are 68 object states.

Typically, a storage service would have far more than 68 object states. To confirm the analysis of Section 2.3.2 without using too many object states, our storage-service prototype uses a simple implementation of the state recovery request that does not batch object states for recovery but instead uses one round of communication for each object state from each replica. The cost of recovery could be reduced by batching object states and preventing identical object states from being sent by multiple replicas. But this does not change the linear relationship between recovery time and the number of object states.

Throughput for State Recovery. Figure 8 shows the effect of State Recovery on throughput when different numbers of object states must be recovered. As before, a single client sends queries to the service; the recovering replica must then recover all objects from other replicas. We use only the Centralized version, so that share recovery does not influence the measurements.

Throughput in Figure 8 drops during State Recovery to about 470 queries/s for time directly proportional to the number of objects being recovered—there is a linear relationship, where each object adds about 260 ms to the recovery time. This reduction in throughput is due to the CPU time replicas spend sending and receiving recovery messages for objects (instead of processing inputs from the client).

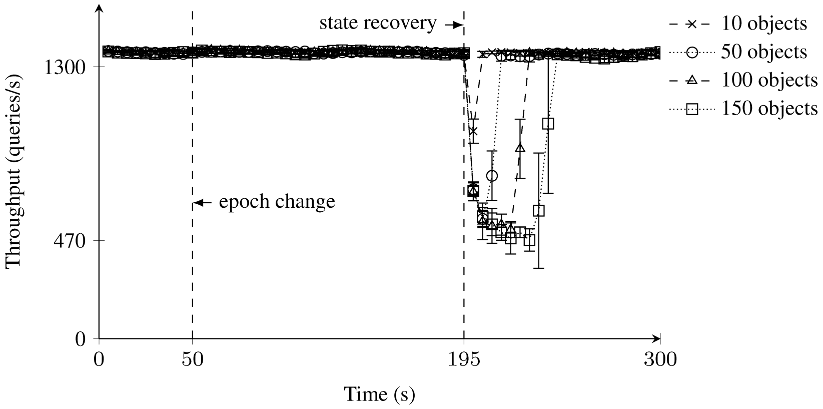

Fig. 8. Throughput under varying numbers of objects to recover for the storage-service prototype.

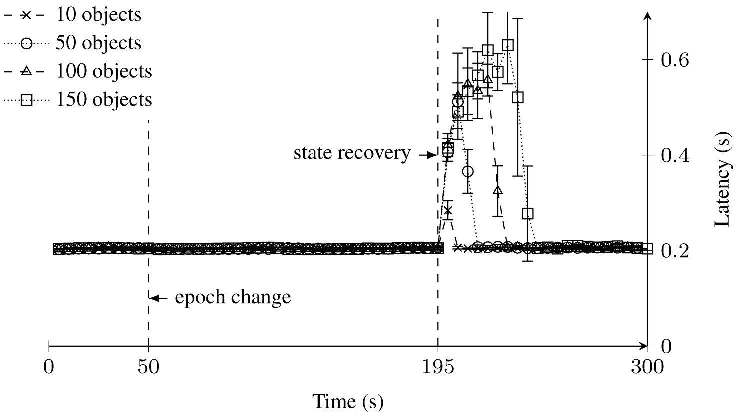

Fig. 9. Latency under varying numbers of objects to recover for the storage-service prototype.

Latency for State Recovery. Figure 9 compares latency during recovery in the same experiments used to generate Figure 8. In these experiments, we see that latency increases to a maximum of about 600 ms until recovery completes—a time directly proportional to the number of objects that need to be recovered, as would be expected from the throughput measurements of Figure 8. Like the decrease in throughput, this increase in latency is due to replicas spending CPU time processing packets for recovery instead of processing queries from clients.

5. DISCUSSION

Replication improves reliability but can be expensive and, therefore, is justifiable only for services that require high resilience to server failure. Proactive obfuscation adds to this expense but transforms a fault-tolerant service into an attack-tolerant service. Not all services require this additional degree of resilience; we show in this paper what implementing this resilience by proactive obfuscation might cost, and those costs are far from prohibitive. Our firewall prototype exhibits 92% of the throughput of a replicated implementation without proactive obfuscation, and the latency is increased by only several milliseconds. And our storage-service prototype experiences no decrease in throughput or increase in latency at the replicas.

Two significant costs in our prototypes can actually be reduced. First, our implementation of proactive obfuscation was done in user space—moving mechanisms into the kernel would avoid the cost of transferring packets across the kernel-user boundary. Second, the cost of digital signatures for authenticating replies from individual replicas in Reply Synthesis (for both prototypes) and Input Coordination (for the firewall prototype) could be significantly reduced by using MACs. However, the use of MACs requires replicas to share keys with clients and with each other, which adds to the refresh and recovery costs already present in our prototypes.

The attack tolerance of a service employing proactive obfuscation depends crucially on what obfuscator(s) are being used. Our investigations, by design, are independent of this choice. That said, our Obfuscation Independence (2.1) and Bounded Adversary (2.2) do provide a basis for examining and comparing obfuscation techniques. It is an open problem which obfuscation techniques satisfy these requirements.

On the one hand, several papers have examined weaknesses in known obfuscation techniques. Shacham et al. [2004] show that obfuscated executables are easily compromised if they are generated by obfuscators not using sufficient entropy, and the return-oriented programming attacks by Shacham [2007] can be executed even in the face of unbreakable instruction-set randomization. Furthermore, Sovarel et al. [2005] show how to defeat a weak form of RISE [Barrantes et al. 2003] instruction-set randomization; Weiss and Barrantes [2006] present more general attacks that work on RISE even without the weakening.

On the other hand, many obfuscated systems are not vulnerable to these published attacks. For instance, the attacks by Shacham et al. [2004] depend on weak randomization, and are not effective in 64-bit architectures. And while the work by Weiss and Barrantes shows how to defeat some implementations of instruction-set randomization, it also provides potential counter-measures. Pucella and Schneider [2006] analyze the effectiveness of obfuscation in general as a defense; they show that in theory it can be reduced to a form of dynamic type checking, which bodes well for the general approach. They also analyze obfuscation for a particular C-like language and give an upper bound on how good particular techniques might be.

Attack tolerance through proactive obfuscation requires that a system satisfy two basic properties: (1) the client service interface must be flexible enough to admit our authentication mechanisms for Reply Synthesis; (2) State Recovery and Replica Refresh must execute in a bounded amount of time. Property (2) suggests several characteristics that must be satisfied by systems implementing proactive obfuscation:

- The system must be built using a small number of machines, connected on a local switch.

—Reobfuscation must be quick enough to be employed at frequent intervals.

—Transfer of state between replicas must be possible in a bounded, timely fashion.

These conditions are easy to satisfy in services that have moderate-sized code base, state, and communication requirements. For example, proactive obfuscation could be sensible when implementing a high-security time service, the top levels of a hierarchical name service, or even a cryptographic key store.

Proactive obfuscation basically trades availability for integrity, since obfuscated replicas are likely to crash in response to an attack (because details the attack exploits are changed by obfuscations). This limits the rate at which adversaries can vet attacks on the obfuscated replicas. Adversary attempts at automated attack generation are thus blunted as a way to overcome the short windows of vulnerability that proactive obfuscation imposes. Attacks that cause crashes could lead to the crash threshold being exceeded, however, and this creates periods of system unavailability. We believe such periods are preferable to the status quo for today's replicated systems, where such attacks would compromise all replicas simultaneously and silently.

5.1 Denial of Service

Denial of service (DoS) is problematic for proactive obfuscation. First, DoS can violate underlying assumptions—in particular, Bounded-Rate Clocks (2.6), Synchronous Processors (2.7), and Timely Links (2.8)—needed for State Recovery. To see why, recall that replicas are rebooted in response to timeouts in the reboot clock. No information flows from the replicas to the reboot clock and, therefore, there is no way to delay reboots to accommodate slow recovery. Thus, recovering replicas must recover within a given amount of time. The alternative is to allow information flow from the replicas to a device that causes reboots. But any device receiving information is vulnerable to attack and compromise.

Other than for State Recovery, our prototypes use asynchronous protocols, like Byzantine Paxos and APSS, to implement the mechanisms used in supporting proactive obfuscation. This provides our system with some measure of resilience to attacks on availability, given the synchronicity constraint on State Recovery.[46]

Second, DoS can prevent rebooted replicas from finding enough running replicas for State Recovery to succeed. Consider an attack in the form of a client input that causes a replica to crash after writing this client input to disk. A crash results when this client input is later read for recovery. If too many replicas are corrupted in this manner, then State Recovery will no longer complete successfully, and replicas will not recover. In the end, the service will not be able to process input packets without intervention by a human operator to delete the problematic input.

[46] The strong requirements on RepeatRequest messages for Byzantine Paxos are only needed for State Recovery.

5.2 Related Research

Proactive obfuscation offers two orthogonal functions critical to building robust systems in the face of attack: proactively recovering state and proactively maintaining independence. Prior work has focused on the former but largely ignored the latter.

State Recovery. The goal of proactive state recovery for replicated systems is to put replicas in a known good state, whether or not corruption has occurred. Software rejuvenation [Huang et al. 1995; Vaidyanathan and Trivedi 2005] and Self-Cleansing Intrusion Tolerance (SCIT) [Huang et al. 2006b, 2006a; Arsenault et al. 2007] both implement replicated systems that periodically take replicas offline for this kind of state recovery. In both, replication masks individual replica unavailability, resulting in a system that achieves higher reliability in the face of crash failures or attacks that only corrupt the state of some replicas. Neither defends against attacks that exploit software bugs.

One of the versions of SCIT [Arsenault et al. 2007] employs a Controller that periodically takes replicas offline and proactively refreshes their code. This Controller does not take any input and can only send output to replicas that might be compromised, but, like our Controller, it takes input from replicas that have just rebooted and are in the process of recovery, since these replicas are correct by assumption (and have not taken any input from potentially compromised replicas). Our Controller not only refreshes replicas with different code and keys, but we developed a decentralized version of the Controller along new, complementary lines: in essence, the SCIT paper explains how to add fault-tolerance to the Controller, and we explain how to obtain the functionality of the Controller by trusting only a timer.

Microreboot [Candea et al. 2004] separates state from code and restarts application components to recover from failures. Components can be restarted without rebooting servers, so these restarts can be performed quickly. And the separation of state and code allows restarted components to recover state transparently and quickly. This work does not address the problem of handling compromise caused by exploitable software bugs but could be used in conjunction with proactive obfuscation to increase replica fault-tolerance.

In systems that tolerate compromised servers, proactive state recovery becomes more complex, since replicas in these systems use distributed protocols to manage state. BFT-PR [Castro and Liskov 2005] adds proactive state recovery to Practical Byzantine Fault-Tolerance [Castro and Liskov 1999]. Proactive state recovery here is analogous to key refresh in proactive secret sharing [Herzberg et al. 1995] (PSS) protocols; it is a means of defending against replica compromise by limiting the window of vulnerability for attacks on replica state, just as the window of vulnerability for compromised keys is limited by PSS. However, BFT-PR never changes the code used by its replicas; in fact, its state recovery mechanism depends on replica code being unmodified. This assumption makes it impossible to apply proactive obfuscation to BFT-PR without redesigning the state recovery algorithm. So, the bulk of State Recovery and

Replica Refresh (the two mechanisms we propose for implementing proactive obfuscation) would not function in the design of BFT-PR.

Further, because a secure cryptographic co-processor is being assumed, public keys in BFT-PR are never changed (though symmetric keys established using these public keys are proactively refreshed). Our Replica Refresh provides a better defense against repeat attacks, since attacks that compromise a replica in BFT-PR can compromise this replica again after it has recovered. However, the experiments of Section 3.2.2.3 and Section 4.2.2.2 show that supporting these aspects of Replica Refresh involves a non-trivial cost at epoch change and recovery, thereby increasing the time available for adversaries to compromise $t + 1$ replicas. The knowledge of these costs and benefits we give now allows a system designer to choose appropriate mechanisms for a given application.

A series of papers by Sousa et al. [2006, 2007, 2008] concern intrusion resilience in replicated systems. The work in Sousa et al. [2007] implements a replicated (but stateless) firewall in the architectural hybrid model. This firewall uses synchronous components to perform all of the voting, signing, and leader-election work of the protocol. But the problems of Byzantine fault tolerance are avoided by assuming these components only can experience benign failures. For comparison, our firewall employs a larger number of replicas but tolerates Byzantine failures and achieves approximately 5 times better throughput.

Sousa et al. [2008] also independently begin to consider the problem of refreshing replica code over time, instantiating a framework first proposed by Schneider and Zhou [2004]. No performance results are given for the prototype discussed in Sousa et al. [2008], and the prototype does not completely instantiate the approach, because Replica Refresh and State Recovery run in software on a Xen VMM [Barham et al. 2003] without any code refresh of the VMM over time. So, vulnerabilities in Xen can be exploited. By contrast, all code in our system is obfuscated, and this obfuscation is changed at some replica for each execution of Replica Refresh—in short, we have formulated the general approach and reduced it to practice.

Independence. Replica failure independence has been studied extensively in connection with fault tolerance. In the N-version programming [Avizienis 1985] approach to building fault-tolerant systems, replica implementations were programmed independently as a way to achieve independence. The DEDIX (the DEsign DIversity eXperiment) N-version system [Avizienis 1985] consists of diverse replicas and a supervisor program that runs these diverse replicas in parallel and performs Input Coordination and Reply Synthesis; it can be implemented either using a single server or in a distributed fashion. But even running independently-designed replicas does not prevent an adversary from learning the different vulnerabilities of these replicas and compromising them one by one, over time.

Recent work on N-variant systems [Cox et al. 2006] uses multiple different copies of a program and votes on their output. The diverse variants of the program are generated using obfuscators, but all are run by a trusted monitor (a potential high-leverage target of attack) that computes the output from answers given by these different copies. The monitor compares responses from variants and deems a variant compromised if it produced a response that differs from the other variants. However, variants are never reobfuscated, so variants that are compromised can be compromised again if restarted automatically. And if variants are not restarted automatically, then intervention by a human operator is necessary.

A related thread of research by Pool et al. [2007] considers ways to relax replica output similarity for multi-core systems. Their main idea is to provide programmers with a barrier construct, called *determinism hints*, similar to memory barriers; at a hint, their system performs actions to guarantee equivalent outputs from the replicas. Relaxed determinism could be used to complement proactive obfuscation by broadening the definition of output similarity for some replicated systems.

Similarly, TightLip [Yumerefendi et al. 2007] creates sandboxed replicas, called *doppelgangers*, of processes in an operating system. The goal of TightLip is to detect leaks of sensitive data—doppelgangers are spawned when a process tries to read sensitive data; each is given data identical to the original process except sensitive data is replaced by fabricated data that is not sensitive. The original and the doppelganger run concurrently; if their outputs are identical, then, with high probability, the output of the original does not depend on the sensitive data and can be output. TightLip shares with our work the goal of using multiple replicas of a program to implement security properties, but TightLip is concerned with confidentiality, and our work is concerned with integrity.

It is rare to find multiple independent implementations of exactly the same service, due to the cost of building each. BASE [Rodrigues et al. 2001] addresses this by providing an abstraction layer to unify differences across implementations of different but related interfaces or functionality. However, replicas in BASE are limited to pre-existing implementations. And these replicas can be compromised immediately upon recovery if they have been compromised before, since code is not changed during recovery.

The idea that replicas exhibiting some measure of independence could be generated by running an obfuscator multiple times with different secret keys on a single program was first published by Forrest et al. [1997]. They discuss several general techniques for obfuscation—from adding, deleting, and reordering code to memory and linking randomization. They implement a stack-reordering obfuscation and show how it disrupts buffer-overflow attacks. Many obfuscation techniques have followed: Bhatkar et al. [2003], Xu et al. [2003], Kc et al. [2003] Barrantes et al. [2005, 2003] Berger and Zorn [2006], and Cadar et al. [2008]; our work is independent of the details of obfuscation, so we do not present these techniques in detail here.

6. SUMMARY

Proactive obfuscation offers a new general framework for adding attack tolerance to the fault tolerance that replicated systems can provide. We described a general architecture for proactive obfuscation, analyzed its inherent costs, and showed instantiations for two very different approaches to replica management. And we established that the additional cost of adding proactive obfuscation to these replicated systems is reasonably small: the state-machine prototype firewall saw 92% of the throughput and 140% of the latency of the replicated version, whereas the quorum-based storage-service prototype had exactly the same throughput and latency as the corresponding replicated version. We also quantified the costs of decentralization for supporting our proactive obfuscation mechanisms. Finally, we described how to use an obfuscator to build a real distributed system, and gave a new basis (Obfuscation Independence (2.1) and Bounded Adversary (2.2)) for evaluating obfuscation technology in the future.

Our results help characterize when proactive obfuscation is a viable technique for restoring replica independence. Alternatives are also possible. Castro et al. [2009] propose to insert checks directly in code or binaries to detect similar attacks to what today's obfuscators are intended to handle. It is an open problem as to whether these reference-monitoring techniques provide the same defenses as proactive obfuscation and what the costs are. Our work is intended to help understand the trade-offs.

APPENDIX

PROOF OF SME HIGHEST STATE RECOVERY

To prove that the state recovery request protocol of Section 3.1 satisfies SME Highest State Recovery (3.1), we prove a lemma about Byzantine Paxos. The proof uses the following fact about Byzantine Paxos: for a replica in Byzantine Paxos to change to a state with sequence number s, it must receive messages from $2t + r + 1$ replicas that are in a state with sequence number $s - 1$.

We assume that at most r replicas can be in a state with a lower sequence number than the one they had when they sent any messages that determined the state with the highest sequence number held by any correct replica. This property is not difficult to guarantee: as long as State Recovery operates correctly each time, any replicas that lose their state by rebooting will be replaced by replicas that have a state with at least as high a sequence number (by SME Highest State Recovery (3.1)). So, this property can be proved for the first instance of State Recovery, then extended inductively to all later instances.

Lemma 1. *At any time ω in the execution of Byzantine Paxos, there is a state with sequence number s such that the $t + 1$ correct replicas with the highest numbered state have states with sequence numbers either s or $s - 1$.*

Proof. Consider the state with highest sequence number s held by any correct replica. This replica entered this state upon receiving messages from $2t + r + 1$ replicas that were in a state with sequence number $s - 1$. At most t of those replicas are compromised at time ω, and at most r could be in states with lower sequence numbers (by having rebooted after sending the message and not yet recovered). So, there are at least $2t + r + 1 - t - r = t + 1$ correct replicas that are in states with sequence numbers either s or $s - 1$. And this means that all of the $t+1$ correct replicas with the highest sequence numbers are in states with sequence numbers s or $s-1$ (otherwise, the $t+1$ replicas in states with sequence number s or $s-1$ would have higher sequence numbers). □

States sent by replicas in response to the state recovery request contain the current and immediately previous state. So, Lemma 1 implies that a recovering replica receiving all the states for correct replicas at a given point in time ω will receive $t+1$ copies of the state with sequence number $s-1$, where s is the highest sequence number at a correct replica at time ω. Note that it is possible for the recovering replica to wait to hear from all correct replicas, since the network satisfies Timely Links (2.8). Furthermore, the recovering replica also receives the messages that caused the correct replica in the state with sequence number s to change to this state. So, the recovering replica will reach the state with sequence number s, the highest sequence number at time ω.

It remains to establish that a recovering replica receives all the state information held by correct replicas at time T seconds after the first replica is contacted. All replicas are contacted by time T, and all correct replicas then send their state at the time they are contacted, as well as all state changes and all messages for the agreement protocol for the next T seconds. Since all replicas have been contacted by time T, the recovering replica will receive all the information for all states of correct replicas through time T. Since η is the bound on the marshaling and transmission time, the recovering replica will receive this information by time $T + \eta$.

We have thus showed that a recovering replica has received, by time $T + \eta$ seconds after the beginning of the state recovery request protocol, enough information to recover to the state with the highest sequence number held by a correct replica at time T seconds after the start of the state recovery request protocol. This proves that the state recovery request protocol of Section 3.1 satisfies SME Highest State Recovery (3.1).

ACKNOWLEDGMENTS

The initial work on proactive obfuscation was begun jointly with Greg Roth. We thank Michael Clarkson, Paul England, Brian LaMacchia, John Manferdelli, Andrew Myers, Rafael Pass, Robbert van Renesse, and Kevin Walsh, as well as attendees of talks at Cornell University, Microsoft Research Silicon Valley, and Microsoft Corporation for discussions about this work. We also thank Andrew Myers and Robbert van Renesse for comments on a draft of this article. And comments from the TOCS referees led us to some important related work as well as prompting a more careful analysis of our assumptions.

REFERENCES

Arsenault, D., Sood, A., and Huang, Y. 2007. Secure, resilient computing clusters: Self-cleansing intrusion tolerance with hardware enforced security (SCIT/HES). In *Proceedings of the 2nd International Conference on Availability, Reliability and Security*. IEEE Computer Society Press, Los Alamitos, CA, 343–350.

Avizienis, A. 1985. The N-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng. 11*, 12, 1491–1501.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T. L., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. ACM, New York, 164–177.

Barrantes, E. G., Ackley, D. H., Forrest, S., Palmer, T. S., Stefanović, D., and Zovi, D. D. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*. ACM, New York, 281–289.

Barrantes, E. G., Ackley, D. H., Forrest, S., and Stefanović, D. 2005. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur. 8*, 1, 3–40.

Berger, E. D. and Zorn, B. 2006. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 158–168.

Bhatkar, S., DuVarney, D. C., and Sekar, R. 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*. USENIX, Berkeley, CA, 105–120.

Cadar, C., Akritidis, P., Costa, M., Martin, J.-P., and Castro, M. 2008. Data randomization. Tech. rep., Microsoft Research. MSR-TR-2008-120.

Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. 2004. Microreboot—A technique for cheap recovery. In *Proceedings of the 16th Symposium on Operating Systems Design and Implementation*. USENIX, Berkeley, CA, 31–44.

Canetti, R., Halevi, S., and Herzberg, A. 1997. Maintaining authenticated communication in the presence of break-ins. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, 15–24.

Case, J., Fedor, M., Schoffstall, M., and Davin, J. 1990. A simple network management protocol. RFC 1157.

Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R. 2009. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. ACM, New York, 45–58.

Castro, M. and Liskov, B. 1999. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Berkeley, CA, 173–186.

Castro, M. and Liskov, B. 2005. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*, 4 (Nov.), 398–461.

Chew, M. and Song, D. 2002. Mitigating buffer overflows by operating system randomization. Tech. rep., School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Nguyen-Tuong, A., and Hiser, J. 2006. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*. USENIX, Berkeley, CA, 105–120.

Deng, J., Han, R., and Mishra, S. 2004. Intrusion tolerance and anti-traffic analysis strategies for wireless sensor networks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'04)*. IEEE Computer Society Press, Los Alamitos, CA, 637–650.

Desmedt, Y. and Frankel, Y. 1990. Threshold cryptosystems. In *Proceedings of the Advances in Cryptology (CRYPTO'90)*. Lecture Notes in Computer Science, vol. 435. Springer-Verlag, Berlin, Germany, 307–315.

Dwork, C., Lynch, N., and Stockmeyer, L. 1988. Consensus in the presence of partial synchrony. *J. ACM 35*, 2, 288–323.

Etoh, H. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp.

Forrest, S., Somayaji, A., and Ackley, D. H. 1997. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*. IEEE Computer Society Press, Los Alamitos, CA, 67–72.

Ghosh, A. K., Pendarakis, D., and Sanders, W. H. 2009. National cyber leap year summit 2009 co-chairs report (section 4). http://www.nitrd.gov/NCLYSummit.aspx.

Gifford, D. K. 1979. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating System Principles*. ACM, New York, 150–162.

Herlihy, M. 1986. A quorum-consensus replication method for abstract data types. *ACM Trans. Comput. Syst. 4*, 1, 32–53.

Herzberg, A., Jarecki, S., Krawczyk, H., and Yung, M. 1995. Proactive secret sharing or: How to cope with perpetual leakage. In *Proceedings of the Advances in Cryptology (CRYPTO'95)*. Lecture Notes in Computer Science, vol. 963. Springer-Verlag, Berlin, Germany, 339–352.

Huang, Y., Arsenault, D., and Sood, A. 2006a. Closing cluster attack windows through server redundancy and rotations. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid*. IEEE Computer Society Press, Los Alamitos, CA, 21.

Huang, Y., Arsenault, D., and Sood, A. 2006b. Incorruptible self-cleansing intrusion tolerance and its application to DNS security. *J. Netw. 1*, 5, 21–30.

Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. D. 1995. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*. IEEE Computer Society Press, Los Alamitos, CA, 381–390.

Intel Corporation. 1999. Preboot execution environment (PXE) specification. Version 2.1. http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf.

Kc, G. S., Keromytis, A. D., and Prevelakis, V. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*. ACM, New York, 272–280.

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7, 558–565.

Lamport, L. Shostak, R. and Pease, M. 1982. The Byzantine generals problem. *ACM Trans. Prog. Lang. Syst. 4*, 3 (July), 382–401.

Malkhi, D. and Reiter, M. 1998. Byzantine quorum systems. *Distrib. Comput. 11*, 4, 203–213.

Marsh, M., and Schneider, F. B. 2004. CODEX: A robust and secure secret distribution system. *IEEE Trans. Depend. Secure Comput. 1*, 1 (Jan.-Mar.), 34–47.

Mogul, J. C. 1989. Simple and flexible datagram access controls for UNIX-based gateways. In *Proceedings of the Usenix Summer Technical Conference*. USENIX, Berkeley, CA, 203–222.

Netfilter. http://www.netfilter.org.

OpenBSD. http://www.openbsd.org.

OpenBSD. PF: Firewall redundancy with CARP and pfsync.
http://www.openbsd.org/faq/pf/carp.html.

OpenBSD. PF: The OpenBSD packet filter. http://www.openbsd.org/faq/pf.

OpenSSL. http://www.openssl.org.

Pool, J., Wong, I. S. K., and Lie, D. 2007. Relaxed determinism: Making redundant execution on multiprocessors practical. In *Proceedings of the 11th Workshop on Hot Topics on Operating Systems*. USENIX, Berkeley, CA.

Pucella, R. and Schneider, F. B. 2006. Independence from obfuscation: A semantic framework for diversity. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, Los Alamitos, CA, 230–241.

Rivest, R. L., Shamir, A., and Adelman, L. M. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commu. ACM 21*, 2, 120–126.

Rodrigues, R., Castro, M., and Liskov, B. 2001. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th Symposium on Operating Systems Principles*. ACM, New York, 15–28.

Schneider, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4, 299–319.

Schneider, F. B. and Zhou, L. 2004. Distributed trust: Supporting fault-tolerance and attack-tolerance. Tech. rep., Cornell Univeriesty.

Shacham, H. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, New York, 552–561.

Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*. ACM, New York, 298–307.

Shamir, A. 1979. How to share a secret. *Comm. ACM 22*, 11, 612–613.

Sousa, P. 2006. Proactive resilience. In *Proceedings of the 6th European Dependable Computing Conference Supplemental Volume*. IEEE Computer Society Press, Los Alamitos, CA, 27–32.

Sousa, P., Bessani, A., and Obelheiro, R. R. 2008. The FOREVER service for fault/intrusion removal. In *Proceedings of the 2nd Workshop on Recent Advances on Intrusion-Tolerant Systems*. ACM, New York, Article No. 5.

Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., and Verissimo, P. 2007. Resilient intrusion tolerance through proactive and reactive recovery. In *Proceedings of the 13th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'07)*. IEEE Computer Society Press, Los Alamitos, CA, 373–380.

Sousa, P., Neves, N. F., and Verissimo, P. 2006. Proactive resilience through architectural hybridization. In *Proceedings of the ACM Symposium on Applied Computing*. ACM, New York, 686–690.

Sovarel, A. N., Evans, D., and Paul, N. 2005. Where's the FEEB? The effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium*. USENIX, Berkeley, CA, USA, 145–160.

Thomas, R. H. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Datab. Syst. 4*, 2, 180–209.

Trusted Computing Group. http://www.trustedcomputinggroup.org.

Vaidyanathan, K. and Trivedi, K. S. 2005. A comprehensive model for software rejuvenation. *IEEE Trans. Depend. Secure Comput. 2*, 2, 124–137.

Verissimo, P. 2006. Travelling through wormholes: A new look at distributed systems models. *ACM SIGACT News 37*, 1, 66–81.

Weiss, Y. and Barrantes, E. G. 2006. Known/chosen key attacks against software instruction set randomization. In *Proceedings of the 22nd Annual Computer Security Applications Conference*. IEEE Computer Society Press, Los Alamitos, CA, 349–360.

Xu, J., Kalbarczyk, Z., and Iyer, R. K. 2003. Transparent runtime randomization for security. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, 260–269.

Yumerefendi, A. R., Mickle, B., and Cox, L. P. 2007. TightLip: Keeping applications from spilling the beans. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*. USENIX, Berkeley, CA.

Zero-Day Initiative. http://www.zerodayinitiative.com.

Zhou, L., Schneider, F. B., and van Renesse, R. 2005. APSS: Proactive secret sharing in asynchronous systems. *ACM Trans. Inform. Syst. Secur. 8*, 3, 259–286.

Received March 2009; revised February 2010; accepted April 2010