Proactive Obfuscation*

Tom Roeder Fred B. Schneider

{tmroeder,fbs}@cs.cornell.edu
Department of Computer Science
Cornell University

Abstract

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 were quantified by measuring the performance of these prototypes.

1 Introduction

Independence of replica failures is crucial when using replication to implement reliable distributed services. But replicas that use the same code share the same vulnerabilities and, therefore, do not fail independently when under attack. This paper 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. Thus, the chances are reduced that an adversary can compromise too many of the replicas that constitute a service.

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 [36] in OpenBSD [34]) and (ii) a distributed storage service. Each service uses a different approach to replica

*Supported in part by AFOSR grant F9550-06-0019, National Science Foundation Grants 0430161 and CCF-0424422 (TRUST), and Microsoft Corporation.

management—we chose approaches commonly used to build network services that require a high degree of resilience to server failure.[1]

Proactive obfuscation employs *program obfuscation* for automatically creating diverse executables during compilation, loading, or at run-time. Address reordering and stack padding [17, 5, 51], system call reordering [11], instruction set randomization [24, 3, 2], and heap randomization [4] are all examples. They each produce *obfuscated executables*, which are believed more likely to crash in response to certain classes of attacks rather than fall under the control of an adversary. For instance, success of a buffer overflow attack typically will depend on stack layout details, so replicas using differently obfuscated executables based on address reordering or stack padding are likely to crash instead of succumbing to adversary control.

Obfuscation techniques are becoming common in commercial operating systems. For example, Windows Vista, OpenBSD, and Linux employ obfuscation, either by default or in easily-installed modules. And it has recently been suggested [44] that obfuscation be used for computer monocultures in order to preserve the benefits of using the same software on clients while mitigating against a catastrophic response to a single attack vector.

We distinguish between two kinds of replica failures. A replica can be *crashed*; a crashed replica does not perform any actions until it reboots. Or, a replica can be *compromised* because it has failed or come under control of an adversary. In the fault-tolerance literature, this second kind of failure is often called *Byzantine*. But the definition of Byzantine failure presumes replica 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.

Servers running obfuscated executables might share fewer vulnerabilities, but this artificially-created independence erodes over time, because an adversary with access to an obfuscated executable can analyze the obfuscated code and customize an attack based on that analysis. So, eventually, all replicas will be compromised. Proactive obfuscation defends against this by introducing *epochs*; a server is rebooted in each epoch, and, therefore, n servers have been rebooted after n epochs have elapsed. 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 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

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

to subvert the service. And 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.

Proactive obfuscation seeks to improve the independence of the code at different replicas. Some approaches to replica management also support *data independence*, whereby different replicas store different states. Replicated systems that support data independence are less vulnerable to certain attacks. For example, some implementation flaws can be exercised only when a replica is in a given state—if replicas have data independence, then an attack that exploits such an implementation flaw will not necessarily succeed at all replicas.

Neither replication nor proactive obfuscation can enhance the confidentiality of data stored by replicas. For some applications, confidentiality can be enforced by storing data in encrypted form under a different key on each server. 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 denial of service (DoS) attacks, which decrease availability. 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 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.

In analogy with fault tolerance, we say that services resilient to attack exhibit *attack tolerance*. There is a cost to achieving attack tolerance by employing proactive obfuscation in conjunction with replication for fault tolerance. A contribution of this paper is to quantify that additional cost. We proceed as follows. Proactive obfuscation is presented in §2 along with mechanisms for its implementation. Then, §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 §4 along with a description and evaluation of a storage-service prototype. Finally, §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 key κ—and produces an *obfuscated program* $\hat{P}$ semantically equivalent to P. Key κ specifies how transformations are applied to produce $\hat{P}$ from P. We abstract from the details of the obfuscator by defining properties we expect it approximates:

> **(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 $t+1$ replicas is greater than the time needed to reobfuscate, reboot, and recover n replicas.

Obfuscation Independence (2.1) implies that different obfuscated executables exhibit some measure of independence. Therefore, a single attack is unlikely to compromise multiple replicas. Obfuscation techniques being advocated for systems today attempt to approximate Obfuscation Independence (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 harder 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 no more than t replicas are ever compromised.

It might seem that an adversary could invalidate Obfuscation Independence (2.1) and Bounded Adversary (2.2) by performing attacks on replicas in parallel. That is, the adversary sends 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 now 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 the parallelism does not really help.

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 at each replica and by the costs of executing mechanisms for coordinating the replicas and performing reboot and

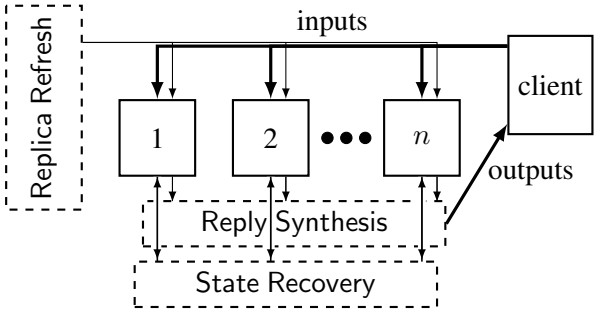

Figure 1: Implementing proactive obfuscation

recovery. Figure 1 depicts an implementation of a replicated service and identifies 3 mechanisms needed for supporting proactive obfuscation: Reply Synthesis, State Recovery, and Replica Refresh.

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 must also be *output similar*—a property defined separately 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 outputs from distinct replicas; replica management also could need authentication for doing inter-replica coordination. These authentication requirements are summarized as follows.

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

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 continue participating in the replicated service. Specifically, recovering replicas receive states from multiple replicas and convert 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, the replies used by g_δ must be *state similar*—a property defined separately for each approach to replica management and state synthesis function.

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:

> **(2.4) Replica Reboot.** Any replica, whether compromised or not, can be made to reboot by Replica Refresh.
>
> **(2.5) Executable Generation.** Executables used by recovering replicas are kept secret from other replicas and are generated by a correct host.

Replica Reboot (2.4) guarantees that no replica can be controlled indefinitely by the adversary. Executable Generation (2.5) ensures that replicas reboot using executables that have not been analyzed or modified by an adversary.

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 bound this number, we assume an upper bound on the amount of state at each replica and make the following assumptions about clock synchronization and message delays.

> **(2.6) Approximately Synchronized Clocks.** The difference between clocks on different correct hosts is bounded.
>
> **(2.7) Timely Links.** Messages sent on a link are either lost or are received in a bounded amount of time. There is a bound on the fraction of messages that are lost.

Approximately Synchronized Clocks (2.6) and Timely Links (2.7) imply that the system implements the *synchronous model* [29]. Together, they are used to guarantee a bound on the time involved in running State Recovery after a replica reboots. Epoch length must be chosen to exceed this bound so that replicas have enough time to recover before others reboot. The epoch length determines the *window of vulnerability* for the service: the interval of time in which a compromise of $t + 1$ replicas leads to the service being compromised.

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 lossy networks labeled *input network* and *output network*, respectively, in Figure 2. Reply Synthesis is performed by clients. State Recovery is performed by replicas using a lossy network, labeled *internal service network*, that satisfies Timely Links (2.7). Replica

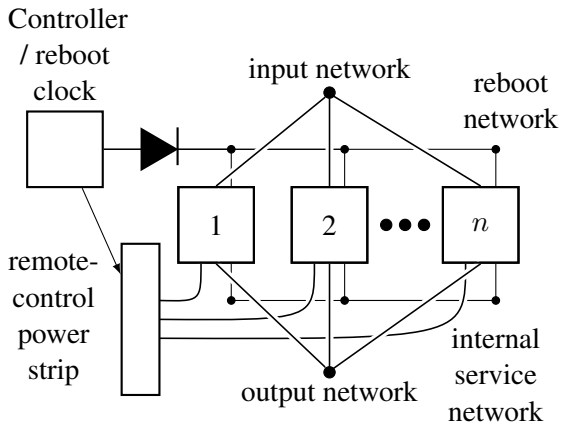

Figure 2: The prototype architecture

Refresh is implemented either by a host (called the *Controller* and assumed to be correct) or by decentralized protocols.

If we design the Controller so it never attempts to receive messages, then the Controller cannot be affected in any way by hosts in its environment. Because it cannot be affected by other hosts, the Controller cannot be attacked. The Controller can still send messages on the *reboot network* connected to all replicas; so, this network is used to provide boot code to replicas. 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.

Whether using the Controller or decentralized protocols, Replica Reboot (2.4) is implemented by a *reboot clock* that consists of a timer for each replica. The reboot clock uses a remote-control power strip in order to toggle power to individual replicas when the timer goes off for that replica. Replicas are rebooted in order mod n, one per epoch.

Epoch change can be signaled to replicas either by messages from the Controller or by timeouts. In either case, for any epoch change, the elapsed time between the first correct replica changing epochs and the last correct replica changing epochs is bounded, due to Approximately Synchronized Clocks (2.6) and Timely Links (2.7). 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 because the reboot clock takes no input and cannot send messages to replicas.

2.2.1 Reply Synthesis

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

In the first, each replica has its own private key, and clients authenticate digitally-signed individual responses from replicas; a replica changes private keys only 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[2] on its first packet after reboot.[3] 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 has a public key that is known to all clients and replicas, and the corresponding private key is shared (using secret sharing [46]) by the replicas. Each replica is given a *share* of the private key and uses this share to compute *partial signatures* [13] for messages. The secret sharing is refreshed on each epoch change, in an operation called *share refresh*, but 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 above. Recovering replicas acquire their shares by a *share recovery* protocol.

Each replica 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 [53] 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.

In fact, an optimization of the second Reply Synthesis implementation is possible, in which a replica—not the client—assembles a signature from partial signatures received from other replicas. This replica then sends the assembled signature with its output to the client. This optimization requires replicas to send partial signatures to each other, which increases inter-replica communication for each output, hence increases latency. But the optimization reduces the changes required in client

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

[3]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.

code that was designed to communicate with non-replicated services.

2.2.2 State Recovery

Normally, each replica gets to the current state by receiving and processing inputs from clients. This, however, is not often possible after reboot, because the inputs that led to the current state might not be available. Reboots occur periodically for proactive obfuscation and also occur due to crashes.

To facilitate recovery after a crash, each replica writes its state to non-volatile media after processing each input; a replica recovering from a crash (but not a reboot for proactive obfuscation) reads this state back as the last step of recovery. This allows a replica to acquire state without sending or receiving any messages. So, replica crashes resemble periods of replica unavailability.[4]

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—recall that one goal of proactive obfuscation is to evict the adversary from a replica. The obvious alternative to using local state is to obtain state from other correct replicas by executing a *recovery protocol*. However, obfuscation may mean that replicas participating in a recovery protocol 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 abstract representation that is the same for all replicas.

Replicas implement a recovery protocol for State Recovery using a generalization of a protocol by Schneider [42]. Before executing State Recovery, a recovering replica i establishes authenticated channels with all replicas it communicates with. The recovery protocol then proceeds as follows:

1. Replica i starts recording input packets and packets received from other replicas.

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

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

[4]This method of handling crashes only works for transient errors and for attacks that cause replicas to crash without writing state to disk. The period of unavailability begins just before receipt of the offending input. See §5 for a discussion of crashes caused as part of DoS attacks.

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

To be useful, 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. Timely Links (2.7) and Approximately Synchronized Clocks (2.6), along with the assumption that the state at each replica is finite, guarantees that steps 2 and 3 complete in a bounded amount of time.

But we must also guarantee that step 4 completes in a bounded amount of time. That is, replicas must be able replay and process recorded packets while continuing to receive and record packets, and this processing must terminate in a bounded amount of time. Recorded packets must therefore be processed more quickly than packets are received. This means there will be 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 a bounded amount of time.[5]

2.2.3 Replica Refresh

Replica Refresh involves 3 distinct functions: (i) reboot and epoch change notification, (ii) executable reobfuscation, 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.

Centralized Controller Solution. A centralized implementation of 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 a 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.

Executable Reobfuscation. The Controller itself obfuscates and compiles executables of the operating system and application source code. By assumption, this

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

guarantees that executables are generated by a correct host, as required by Executable Generation (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, we require that the reboot network implement a separate confidential channel from the Controller to each replica. Replicas may not send packets on these channels.

> **(2.8) 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 (2.8) would be 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 [8] modern switches allow such control of individual ports by a third party like the Controller.

Instead of using TCP to communicate with a host providing an executable (since TCP-based methods like PXE boot [23] 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.[6]

Key Distribution. The Controller performs key distribution to implement authenticated channels by generating a new public/private RSA [40] key pair for each recovering replica i and 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.[7] Reboot Channels (2.8) 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.

[6] 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 send and receive messages with the Controller. This boot procedure requires modifying Reboot Channels (2.8) as noted above to require the necessary control over which replicas can output to the reboot network.

[7] The Controller could include in this executable new public keys for replicas to be rebooted later. These keys are not included in order to deprive adversaries access to the keys as long as possible.

Decentralized Protocols Solution. The centralized Controller provides a simple way to implement Replica Refresh but is a single point of failure. Decentralized schemes tend to be more expensive but can avoid the single point of failure of centralized schemes.

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 notification can be achieved, however, by using timeouts, as discussed at the beginning of §2.2.

Executable Reobfuscation. Replicas can each generate their own obfuscated executables in order to satisfy Executable Generation (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:

> **(2.9) Read-Only Boot Code.** The semantics of boot code on replicas cannot be modified by an adversary.

This assumption can be discharged if two conditions hold: (i) the BIOS is not modifiable[8], 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.[9]

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, but Read-Only Boot Code (2.9) implies the code on the CD-ROM cannot change the BIOS in order to accomplish this switch.

In our prototypes, we resolved this dilemma by employing a timer. 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 begins providing power to the server's CD-ROM

[8] This, in turn, can be implemented using a secure co-processor like the Trusted Platform Module [49].

[9] 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 for ease of exposition.

drive. The server boots, finds the CD-ROM (so boots from that device), executes, and writes its newly obfuscated executable to its hard drive. The timer 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.

Key Distribution. In the decentralized implementation for this function, a recovering replica itself generates a new public/private key pair. It must then establish and disseminate a certificate for this new public key. Key generation can be performed by 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, Halevi, and Herzberg [7]. 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 [28], 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. Note that if replicas are allowed to create partial signatures for any single key in each epoch, and only $t+1$ partial signatures are required for signature reassembly, then 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. But if Byzantine Paxos is used to decide which key to sign, then 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.[10]

This key distribution scheme 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 recovering correct 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 place, and i can alert a human operator. This operator can check and reboot compromised replicas. However, i cannot convince other replicas in the service.

2.3 Mechanism Performance

Assumptions invariably bring vulnerabilities. Yet implementations having fewer assumptions are typically more expensive. For instance, decentralized protocols for Replica Refresh 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 §2.2 mostly involve incurring higher CPU costs for increased decentralization. Under high load, these CPU costs divert a replica's resources away from input handling. 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

[10] 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.

batch size, called the *batching factor*. This 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 directly on how much state must be recovered. Large state transfers consume network bandwidth and CPU time, both at the sending and receiving replicas. So, when recovering replicas must recover a large state under high load, State Recovery leads to significant degradation of throughput and latency.

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 Executable Reobfuscation 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 Executable Reobfuscation 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 [31]. 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 contribution to performance is negligible.

3 State Machine Replica Management

The *state machine approach* [25, 43] provides a way to build a reliable distributed service that implements the same interface as a program running on a single trust-

worthy host. Using it, a program is described as a *state machine*, which consists of *state variables* and deterministic[11] *commands* that modify state variables and may produce output.

Given a state machine m, a *state machine ensemble* $\mathsf{SME}(m)$ 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 state and, therefore, the state machine approach does not create 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, 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* [27, 14] 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 their state machine. For instance, Byzantine Paxos requires each replica to store a monotonically increasing sequence number that labels 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.

The combination of the extended sequence number and state variables forms the *replica state*. 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,

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

or reordered before the agreement algorithm is run. In fact, 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` as a state machine and uses the techniques and mechanisms of §2. We chose `pf` as the basis of our prototype because it is a production-quality firewall used 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 $3t + 1$ [9]. This number does not take into account rebooting replicas. However, a rebooting replica does not exhibit arbitrary behavior—it simply resembles a crashed replica. Lamport [26] shows that tolerating f crashed and t compromised replicas in Byzantine Paxos requires $3t+2f+1$ total replicas. So, if k replicas might be rebooting simultaneously, then we can set $f = k$, and we conclude that only $3t + 2k + 1$ replicas are needed, which means that only 2 additional replicas must be added to tolerate each rebooting one. In our prototypes, $k = 1$ 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* [15]: the leader for sequence number j is replica j mod 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 a new leader can be made very short. Replicas set a timer for each sequence number i; on timeout, replicas expect replica $(i + 1)$ mod 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.[12]

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.[13]

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 packet has not been received, then a matching input packet is requested from the leader.[14]

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 send what we call RepeatRequest messages for a given type of message and extended sequence number. Upon receiving a RepeatRequest, a replica resends the requested message if it has a copy.[15]

[12]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 *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*.

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

[14]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.

[15]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.

Synthesis Functions. The output synthesis and state synthesis functions in our firewall prototype depend on having at most t replicas be 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 implementation of Reply Synthesis—in both, γ is set to $t + 1$. 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 they contain identical outputs and their partial signatures together reassemble to give a correct signature on this output. 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 replica state if it has received this replica state in state-similar replies from $t + 1$ distinct replicas.

State Recovery Request. State Recovery must guarantee that each recovering replica acquires some minimum state from which it can advance by executing commands. Define the *current minimum state* to be a replica state $\sigma_{(e,s)}$ such that:

- there is some correct replica with replica state $\sigma_{(e,s)}$, and
- if some correct replica has replica state $\sigma_{(e',s')}$, then $(e, s) \leq (e', s')$.

Since all replicas begin in the same initial state, and rebooting puts a replica in that initial state, we conclude that a current minimum state always exists.

Normally, the current minimum state obtained from executing State Recovery will differ from the initial state. But even so, that state might not suffice for a recovering replica to resume operation as part of the state machine ensemble. The recovery protocol must also satisfy the following property, which guarantees that

replicas can always recover at least the current minimum state at the time a recovery protocol starts.

> **(3.1) SME State Recovery.** If $\sigma_{(e,s)}$ is the current minimum state at the time a replica i starts the recovery protocol, then there is a time bound Δ and some (e', s') such that $(e, s) \leq (e', s')$ holds and i recovers $\sigma_{(e',s')}$ in Δ seconds.

The state recovery request used in State Recovery is implemented by having replicas propose a special command, `RecoveryRequest`, for agreement; this command contains the identity of the recovering replica. Upon choosing this command, a correct replica sends its current state to the rebooting replica. Replica states are guaranteed to be the same for correct replicas at the same extended sequence number, all correct replicas execute the `RecoveryRequest` at that same point, and there are more than $t + 1$ correct replicas, so the recovering replica is guaranteed to receive more than $t + 1$ identical replica states, which suffices for state synthesis, since $\delta = t + 1$.[16] Note that these replica states have a sequence number greater than the current minimum state at the time the recovery protocol starts.

The time needed to execute this protocol is bounded, given Timely Links (2.7) and Approximately Synchronized Clocks (2.6) along with the assumed bound on the amount of state stored by any correct replica. So, this recovery protocol satisfies SME State Recovery (3.1). But, as noted in §2.2.2, for State Recovery to be able to complete in a bounded amount of time, a recovering replica must also be able to replay its recorded packets and catch up with the other replicas in the system in a bounded amount of time.

The processing of replayed packets might require replicas to send RepeatRequest messages to 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 q, the number 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 SME State Recovery (3.1) guarantees that State Recovery completes in a bounded amount of time.

[16] An optimization is for replicas to reply immediately with their replica state the first time they receive a `RecoveryRequest` from a recovering replica, instead of running an agreement algorithm. If a recovering replica i does not receive $t+1$ identical replica states from these responses, then i can send a second `RecoveryRequest`; a leader for agreement chooses the second `RecoveryRequest` as a command using agreement as in the State Recovery protocol. Our firewall prototype implements this optimization, and the system has never executed a second `RecoveryRequest` and agreement, because recovering replicas always got $t+1$ identical replica states on their first `RecoveryRequest` in the experiments we ran.

For RepeatRequest messages to guarantee that packet replay completes in a bounded amount of time, 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 time needed to learn a command for an extended sequence number via RepeatRequest messages (this time is bounded, given Timely Links (2.7) and Approximately Synchronized Clocks (2.6)). In this case, there is a bound b 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 various different implementations for our firewall prototype. We consider:

- Input Coordination performed either by a variant of Byzantine Paxos that does not support proactive obfuscation or by a variant of Byzantine Paxos that does.
- 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 §2.2.2 with state recovery request and state synthesis as described in §3.1. The *Replicated* version provides Byzantine Paxos without accounting for rebooting replicas: it does not perform proactive obfuscation or any form of proactive recovery. The result is 5 different versions of our firewall prototype, listed in Table 1.

3.2.1 Experimental Setup

Our implementations are written in C using OpenSSL [37]; 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 $t = 1$

Version name	Input Coordination	Reply Synthesis	Replica Refresh epoch	reobf	key
Replicated	Byz Paxos	indvdl auth	none	none	none
Centralized	Byz Paxos	indvdl auth	cntrl	cntrl	cntrl
Decentralized	Byz Paxos	indvdl auth	cntrl	cntrl	dcntrl
Reboot Clock	Byz Paxos	indvdl auth	dcntrl	dcntrl	dcntrl
Threshold Client	Byz Paxos	thresh crypto	dcntrl	dcntrl	dcntrl

Table 1: The versions of our firewall prototype

provided Bounded Adversary (2.2) is satisfied; 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 Replicated version only needs $3t + 1$ hosts 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 [32]. The pseudo-device allows programs running in user space to take and replace packets on the network stack, similar to Linux netfilter [33].

The `pf` code provides a pseudo-device called `pfsync` [35] 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 [11] 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 [16] inserts and checks a random value after the return value of functions to protect against stack-smashing attacks. However, any obfuscation method that can be applied during compilation, linking, or loading could be used in our prototype.

Recall, our interest is not in the details of the obfuscation but rather in the details of the mechanisms needed to deploy obfuscation in an effective way.

The time that must elapse between reboots bounds the window of vulnerability for cryptographic keys used by each replica. This allows replicas in our prototype to use 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 window of vulnerability in which secrecy of the key matters—one replica is rebooted each several minutes, so each key is refreshed on the order of once per half hour.

We also use 512-bit RSA keys for the Replicated version even though it does not perform proactive recovery and, therefore, should be using 1024-bit keys. However, using 512-bit keys for the Replicated version allows direct performance comparisons with the Centralized version, since the two versions then differ only in their numbers of replicas.

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.[17] We set the batching factor to 43, because this value provided the highest performance in our experiments.

Recall that commands for agreement are 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 currently 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

To evaluate our different mechanism implementations for proactive obfuscation, 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.

3.2.2.1 Input Coordination

To quantify how throughput and latency are affected by the Input Coordination implementation, we performed experiments in which there are no compromised, crashed, or rebooting replicas, so Replica Refresh and State Recovery can be disabled with averse effect. We consider two prototype versions that differ only in

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

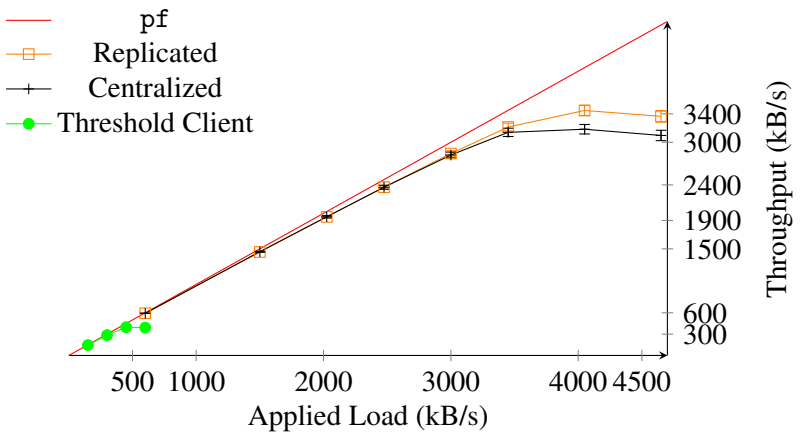

Figure 3: Overall throughput for the firewall prototype

their implementation of the Input Coordination mechanism: the Replicated version and the Centralized version. Both use RSA signatures for Reply Synthesis.

Throughput. 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.[18] The curve labeled `pf` represents 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 Replicated version reaches a maximum throughput of about 3450 kB/s. So, the Centralized version reaches about 92% of the throughput of the Replicated version under high load. This suggests that the cost of adding proactive obfuscation to an already-replicated system is not excessive.

The Replicated 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 can be reduced significantly. The other major cost, which cannot be reduced in our implementation, arises from copying packets between the kernel and mechanism implementations running in user space. Multiple system calls to our pseudo-device

[18] Discussion of the Threshold Client curve appears below in §3.2.2.2.

are performed for each packet received by the kernel. Reducing this cost requires implementing the mechanisms for proactive obfuscation in the kernel.

Throughput decreases for both the Replicated and the Centralized versions after saturation. This decrease occurs because the higher applied load means that replicas spend more time dropping packets. And packets in the firewall prototype are copied into user space and deleted from the kernel before being handled. So, dropped packets still consume non-trivial CPU resources.

The choice of batching factor and the choice of timeout for leader recovery 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.[19] 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 this version) 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. Latency in the firewall prototype is also affected by the choice of Input Coordination implementation. In the same experiment as used to produce Figure 3, latency was measured at 39 ± 3 ms for the Centralized version, whereas latency in the Replicated version was 28 ± 6 ms under the same circumstances. This difference is due to replicas in the Replicated version needing to handle fewer replies from replicas per message round in the execution of Input Coordination.

Unlike throughput, however, latency is not affected by the batching factor, since latency depends only on the time needed to execute the agreement algorithm.[20] And batching is opportunistic, so replicas do not wait to fill batches. This also keeps batching from increasing latency.

To understand the latency when one replica is crashed, we ran a different experiment where the outside client sent 1500-byte packets, but with one replica crashed. With a crashed replica, latency increases to 342 ± 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 pro-

[19] 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.

[20] 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.

cess 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. Throughput for different Reply Synthesis implementations is already given in Figure 3, because 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 §2.3.1: the Threshold Client version exhibits significantly lower throughput, due to the high CPU costs of the calculations required for generating partial signatures using the threshold cryptosystem used 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. Latency for the Threshold Client version (measured in the same experiment as for throughput) is 413 $\pm$ 38 ms as compared with 39 $\pm$ 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 of 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 (containing 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 in our experiments.

A Controller can perform reobfuscation for one replica while another replica is rebooting, so reobfuscation and reboot 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 affects the Reboot Clock version: it has the same performance as the Decentralized version, except for a longer window of vulnerability caused by the extra time needed for CD-ROM-based executable generation.

Key distribution for Replica Refresh involves generating, signing, and disseminating a new key for a recovering replica. In the decentralized implementation, replicas must also 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. 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 §2.2.3 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. 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 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 they perform leader adjustment to take the rebooting replica into account, so there is no throughput drop.

At the point marked "recovery" in Figure 4, the recovering replica runs the State Recovery mechanism in both versions. 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.

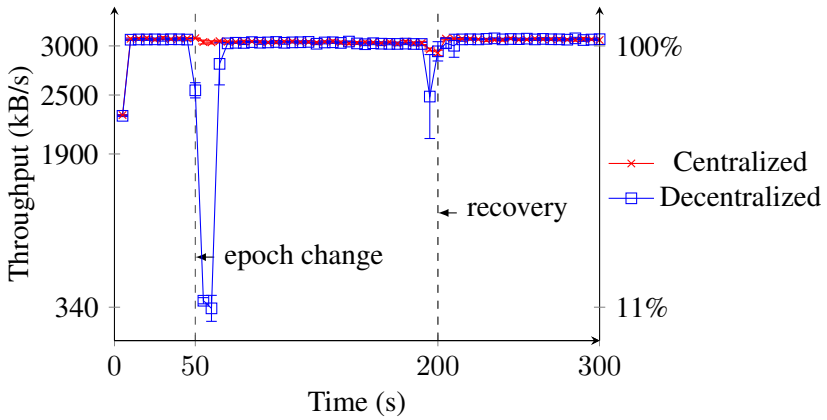

Figure 4: Two key distribution methods for the firewall prototype at an applied load of 3300 kB/s

Latency. Latency increases during share refresh. The same experiment as for measuring throughput shows that the Centralized version has a latency of 37 $\pm$ 2 ms after an epoch change, similar to its latency of 39 $\pm$ 3 ms when there are no failures. But latency in the Decentralized version goes up to 2138 $\pm$ 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 $\pm$ 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.

4 Quorum System Replica Management

A *quorum system* [48, 18, 19] stores *objects* containing *object state*; objects support *operations* to modify their 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 and computing an object state from a quorum of replicas, executing the operation using this object state, then writing the resulting object state back to a quorum of replicas. We follow a common choice [30] for the semantics of concurrent operations:

1. Reads that are not concurrent with any write generate the latest object state written, 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 a prior object state that is not guaranteed to be the latest object state written.

On abort, clients can retry the operation.

The object state stored by a replica is labeled by the client that wrote this object state; this label is a totally ordered, monotonically increasing *sequence number* and is kept as part of the object state. Replicas only store a new object state for an object o if it is labeled with a higher sequence number than the object state being 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 [30] 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, in any execution of an operation, only replicas in a 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* [30]; 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 has not 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 with $\mathcal{B}$ as a threshold fail-prone system for threshold t must satisfy the following properties [30]:

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

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

Threshold DQS Correctness (4.1) and Threshold DQS Availability (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 (4.2) 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. 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 is by allowing only one client per object o to write object state for o but allowing any client to read it.[21]

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 (4.2) 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:

(4.3) Proactive Threshold DQS Availability.

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

[21] 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) (as shown previously by Sousa et al. [47]). This follows 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. But Proactive Threshold DQS Availability (4.3) guarantees that a quorum $\hat{Q}$ of $2t + r + 1$ replicas is always available, where $\hat{Q}$ does not contain any rebooting replicas. By definition, an intersection between $\hat{Q}$ and any other quorum consists only of replicas that are not rebooting. So, an overlap of $t + 1$ replicas from $\hat{Q}$ 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 (4.1):

(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 also implemented a storage-service prototype using a dissemination quorum system over a threshold fail-prone system with threshold t and 1 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 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*, which means that 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 can apply 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.

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

For the individual authentication implementation of Reply Synthesis, 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 the object state in that set with the highest sequence number or 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 complete quorum returned confirmations (so, some replies must have been errors instead of confirmations). In both cases, γ is set to the quorum size.

The threshold cryptography implementation of Reply Synthesis is incompatible with dissemination quorum systems. 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 object state will have 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.[22]

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

[22]Other implementations [30] of Byzantine quorum systems 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 $t + 1$.

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 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. A recovering replica sends a state recovery request in the storage-service prototype to all replicas and waits until it has received replies from a quorum; upon receiving a state recovery request, a correct replica sends the object state for each object it has stored to the recovering replica. Since 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 inserts an unknown-object message with object ID o in the reply from j.[23] This makes the object IDs found in each reply set the same, and, therefore, makes the replies received by the recovering replica state similar.

A recovering replica must acquire enough state to replace any replica. To do so, it must acquire the current object state for each object at the time it recovers. We characterize this formally, defining s to be the *current maximum sequence number* for an object o as follows:

- the correct replicas in some quorum have object state for o with sequence number s, and
- if the correct replicas in any other quorum have object state for o with sequence number s', then $s' \leq s$.

The recovery protocol for quorum systems must then satisfy the following property, which guarantees, for each object, that replicas recover an object state with at least the current maximum sequence number at the time recovery starts.

[23] 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.

(4.5) QS State Recovery. For each object o with current maximum sequence number s at the time replica i starts the recovery protocol, there is a bound Δ such that i recovers an object state for o with sequence number s' such that $s' \geq s$ holds in Δ seconds.

The State Recovery protocol of §2.2.2, using the definition of the state recovery request above, satisfies QS State Recovery (4.5). To see why, notice that Proactive Threshold DQS Availability (4.3) guarantees that some quorum is available even when one replica is rebooting. 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 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. 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. Timely Links (2.7) and Approximately Synchronized Clocks (2.6), 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 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, by Approximately Synchronized Clocks (2.6).

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 the threshold cryptography Reply Synthesis mechanism is not applicable. All versions of our prototype

Version name	Replica Refresh		
	epoch	reobf	key
Replicated	none	none	none
Centralized	cntrl	cntrl	cntrl
Decentralized	cntrl	cntrl	dcntrl
Reboot Clock	dcntrl	dcntrl	dcntrl

Table 2: The versions of our storage-service prototype.

use the State Recovery implementation from §2.2.2 with the state recovery request as described in §4.1. Table 2 enumerates the salient characteristics of the three versions of the prototype we analyzed. We also present a *Replicated* version that does not perform any proactive recovery for replicas.

4.2.1 Experimental Setup

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

We implemented the storage-service server and client in C using OpenSSL. Neither server nor client makes any 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—we describe our reasons for selecting this batching size in §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 our experiments, replicas start with

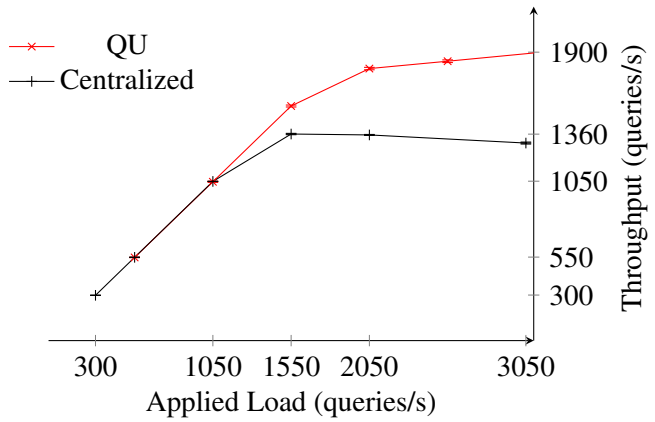

Figure 5: Overall throughput for the storage-service prototype

a set of 4-byte integer objects from the client. The client performs queries, but not updates, on randomly chosen objects at a specified rate. This workload allows us to characterize two things: an upper bound on the throughput of the service and a lower bound on its latency, since the objects are as small as possible, and queries do not incur CPU time on the client for signatures or on the server for verification (whereas updates would). Update operations by the client would increase the cryptographic load on the client and server, hence slow both down.

Figure 5 graphs throughput for these experiments—throughput of the Centralized version peaks at 1360 queries/second at an applied load of 1550 queries/second. After this point, throughput declines slightly as the applied load increases. This decline is due to the increasing cost of queries that must be dropped because they cannot be handled. The "QU" curve shows the performance of a single server and thus fixes the best possible case. The throughput of the server starts saturating at about 1800 queries/second, and increases slightly with higher applied loads. Throughput saturation 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. The kernel never accumulates a significant queue of packets in the experiments we ran, since it deletes packets once they have been copied. In particular, even packets that are dropped are first copied from the kernel to user space. So, high loads induce a significant CPU overhead in the replicas. In the single server version, packet queues are managed by the OpenBSD kernel, so dropping packets is significantly less costly—we expect to see the same effect of decreasing throughput in the QU plot, but at much higher

applied loads.

The Replicated version has exactly the same performance as 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 occurs in the Replicated version at exactly the same number of queries per second as in the Centralized version.

The latency of the storage-service prototype in these experiments is 21 $\pm$ 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 $\pm$ 3 ms. The 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 at applied loads above 1360 queries/second—the maximum load a replica can handle. Given this bound, we perform our experiments for Replica Refresh and State Recovery at an applied load of 1400 queries/second, and thus show behavior at saturation. We allow servers to reach a steady state in their processing of packets before starting our measurements. This ensures that replicas are not measured in their startup transients, where throughput and latency have not stabilized.

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.

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 only component of Replica Refresh that causes significant performance differences in throughput and latency between prototype versions is share refresh and share recovery. We do not show results for the Reboot Clock version, since the only difference between the Reboot Clock version and the Decentralized version is that the Reboot Clock version has a longer epoch length, hence longer window of

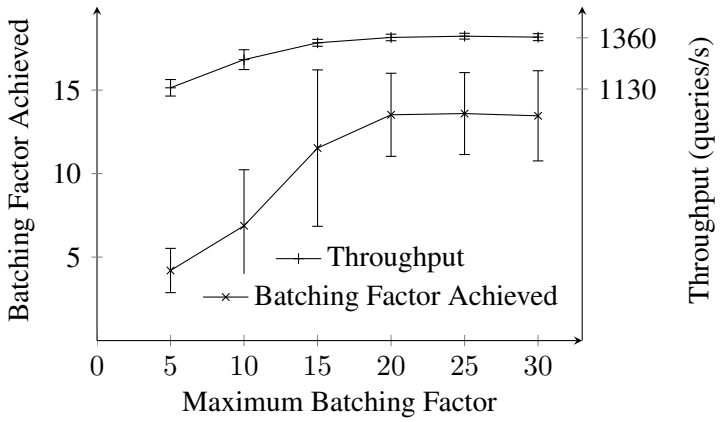

Figure 6: Batching factor and throughput for the storage-service prototype under saturation

vulnerability.

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

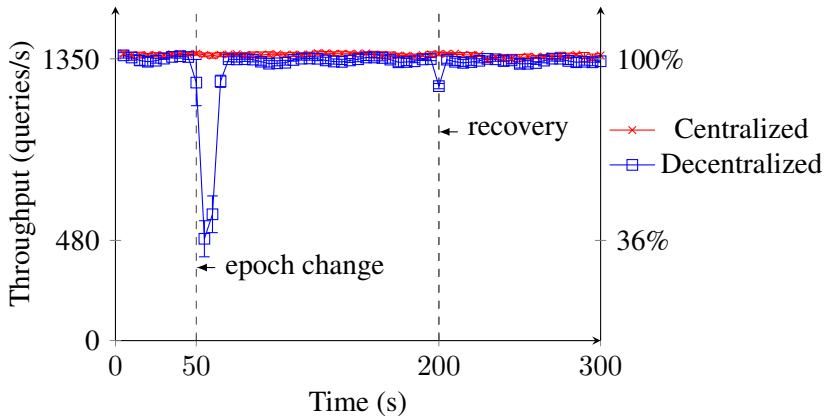

Figure 7: Two key distribution methods for the storage-service prototype at an applied load of 1400 queries/second

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. The same experiments as for Figure 7 lead to a similar graph for latency. Latency in the Decentralized case for the storage-service prototype increases to 646 $\pm$ 89 ms during share refresh, as opposed to 205 $\pm$ 3 ms for the same interval in the Centralized case. This latency is lower than what was seen in §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 $\pm$ 3 ms during share recovery, whereas the Centralized version has a latency of 203 $\pm$ 1 ms.

4.2.2.3 State Recovery

The number of object states that must be recovered after a reboot in the storage-service prototype significantly effects throughput and latency. The state of the stor-

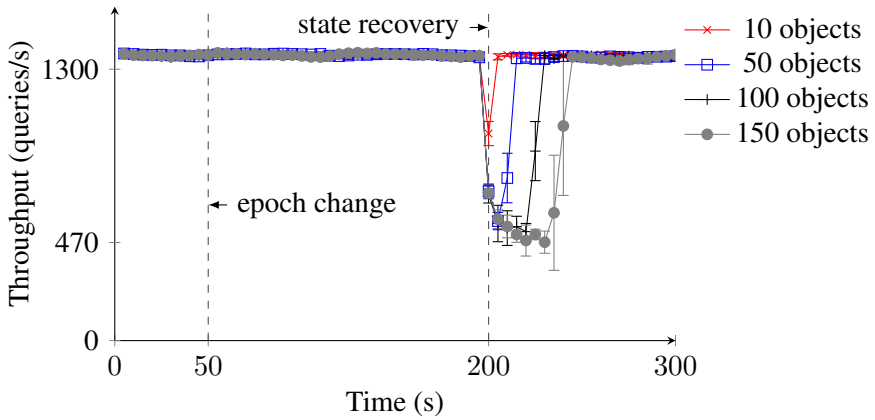

Figure 8: Throughput under varying numbers of objects to recover for the storage-service prototype

age service increases linearly in the number of object states stored (although each object state in the prototype only contains a 4-byte integer, each also has headers of length 24 bytes and a signature of length 64 bytes, so each object contains 92 bytes). So, a storage-service state with 115 object states has 10580 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 §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 115 object states.

However, we would typically expect a storage service to have many more than 115 object states. To confirm the analysis of §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. Of course, the cost of recovery can 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. Figure 8 shows the effect of State Recovery on throughput when different numbers of object states must be recovered. As before, in this experiment, 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.

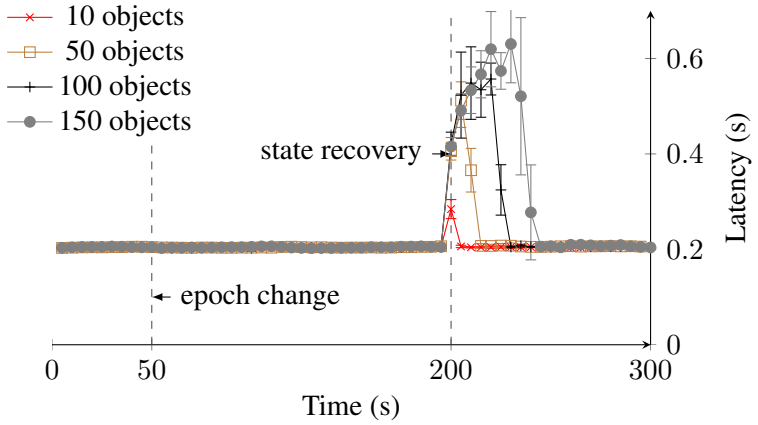

Figure 9: Latency under varying numbers of objects to recover for the storage-service prototype

Figure 8 shows that throughput 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).

Latency. 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 and stays there until recovery completes—a time directly proportional to the number of objects that need to be recovered, as would be expected from the throughput. Like the decrease in throughput, this increase in latency is due to the 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, only services that require high resilience to server failure ought to be replicated. 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, and we show in this paper what the additional costs are in implementing proactive obfuscation. The costs are far from prohibitive. For instance, our firewall prototype's

performance only differs from a replicated implementation without proactive obfuscation by exhibiting 92% of the throughput.

Moreover, two significant costs in our prototypes can actually be reduced. First, our implementation of proactive obfuscation was done in user space—moving these mechanisms to the kernel would avoid the cost of transferring packets across the kernel-user boundary. Second, the cost of digital signatures for individual authentication could be significantly reduced by using MACs. This is actually an optimization of the same individual authentication Reply Synthesis implementation using digital signatures, and thus not fundamentally different from the case we studied. However, the use of MACs also requires replicas to set up shared keys with clients and with each other, and this cost must be added to the refresh and recovery costs already present in our prototypes.

The attack tolerance of a service employing proactive obfuscation depends fundamentally on what obfuscator(s) are in use. Our work, by design, is 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, Shacham et al. [45] shows that obfuscated executables are easily compromised if they are generated by obfuscators not using enough randomness. On the other hand, Pucella and Schneider [39] analyze the effectiveness of obfuscation in general as a defense and show that it can be reduced to a form of dynamic type checking, which bodes well for the general approach. They also present a theoretical framework for obfuscation and analyze obfuscation for a particular C-like language. This gives an upper bound on how good particular techniques might be.

Proactive obfuscation basically trades availability for integrity. In particular, an obfuscated replica that is processing an attack is likely to crash (because the attack is unlikely to be well matched to the obfuscations that were applied). This also has the effect of limiting the rate at which adversaries can vet their attacks. And this, in turn, blunts adversary attempts at automated attack generation as a way to overcome the short windows of vulnerability that proactive obfuscation imposes.

Denial of service attacks can violate our assumptions about synchronicity, since we make strong assumptions about our servers and network communication in Approximately Synchronized Clocks (2.6) and Timely Links (2.7). This synchronicity is needed for State Recovery. To see why, recall that replicas are rebooted based on timeouts in the reboot clock. No information flows from the replicas to the reboot clock, and, therefore, there is no way to change the timing of reboots based on the time needed for recovery. Thus, recovering replicas must recover within a given amount of time, and, therefore, strong assumptions on synchronicity are needed to ensure that State Recovery completes in a timely manner. The alternative is to allow information to flow from the replicas to a device that causes reboots. We do not

use this implementation, since a device receiving information from replicas might be compromised. Other than for State Recovery, we use asynchronous protocols, like Byzantine Paxos and APSS, to implement the mechanisms for proactive obfuscation. This provides our system with the maximum resilience to attacks on availability, given the synchronicity constraint on State Recovery.[24]

DoS attacks reduce availability and are not blunted by proactive obfuscation. DoS attacks by clients overloading a resource must still be countered by blocking the offending requests or terminating their source(s). And for DoS attacks by servers overloading some resource, the usual defenses apply, such as per-server resource limits and elimination of resource sharing.

However, DoS attacks that cause replicas to crash can keep correct replicas from ever recovering without outside intervention if the attack leverages state written to disk and later read for recovery. Such an attack could work as follows. A replica i receives a packet that exercises a flaw in i, eventually causing i to crash. But i writes state to disk before crashing, including that packet or the effect of its execution. After i crashes and reboots, the state that i reads from disk during recovery might cause i to crash again. In this case, replica i will continue to reboot, read its state, and crash until it is rebooted for proactive obfuscation. If too many replicas have crashed in this manner, then State Recovery will no longer complete successfully, so replicas will not recover. And the service will not be able to process input packets. This attack must be resolved by intervention of a human operator.

Related Research

Proactive obfuscation provides two 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 or been detected. Software rejuvenation [22, 50] and Self-Cleansing Intrusion Tolerance [21] 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 as well as some attacks. However, neither defends against attacks that exploit software bugs.

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

Microreboot [6] 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 [10] adds proactive state recovery to Practical Byzantine Fault-Tolerance [9]. Proactive state recovery here is analogous to key refresh in proactive secret sharing [20] (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 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. Recovery in BFT-PR also relies on state written by replicas to disk—the BFT-PR implementation assumes implicitly that replicas will not crash or be compromised upon reading state written by a compromised replica. We do not make this assumption, since it rules out the possibility of denial of service attacks that cause replicas to crash on reading their state, as discussed above.

Further, public keys in BFT-PR are never changed (though symmetric keys established using these public keys are proactively refreshed), because a secure cryptographic co-processor is assumed. 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 §3.2.2.3 and §4.2.2.2 show that our implementations of these aspects of Replica Refresh incur a non-trivial cost at epoch change and recovery across different approaches to replica management, and this may increase the time available for adversaries to compromise $t + 1$ replicas. Knowledge of these costs and benefits allows a system designer to choose the appropriate mechanism for a given application.

Independence. Replica failure independence has been studied extensively in connection with fault tolerance. In the N-version programming [1] 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 [1] 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 [12] uses multiple different copies of a program to vote on 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 the 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.

Similarly, TightLip [52] creates sandboxed replicas, called *doppelgangers*, of processes in an operating system. The goal of TightLip is to detect leaks of data designated sensitive—doppelgangers are spawned when a process tries to read sensitive data and are given data identical to the original process except that sensitive data is replaced by fabricated data that is not sensitive. The original and the doppelganger are 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 achieve higher resilience to failure, but TightLip seeks only to detect data leaks rather than handling compromised replicas.

It is rare to find multiple independent implementations of exactly the same service, due to the cost of building each. BASE [41] addresses this by using different implementations and providing an abstraction layer to unify the differences and thereby facilitate communication and state recovery. 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 their code does not change 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. [17]. 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 since been developed.

Address obfuscation [5, 38, 51] permutes the code and data segments of a program as it is loaded into memory. Under this obfuscation, attacks that rely on the absolute or relative locations of code or data in memory are not likely to succeed,

since these locations are unknown—attacks might reveal some information about randomized locations, but code and data locations can be rerandomized each time an executable is loaded.

Instruction-set randomization [24, 3, 2] transforms the instruction encoding in a given instruction set—in one implementation, instructions are XOR'd with a random key before being stored. These instructions must be XOR'd with the same key to recover the original instruction stream. Therefore, injected instructions from an adversary are unlikely to decode into a useful attack. Similarly, interpreted languages can be randomized with low overhead by modifying the interpreter. But without specialized hardware, instruction-set randomization is expensive for code run natively on a processor, since each instruction must be translated before it is executed.

DieHard [4] performs randomization of the run-time heap; attacks that rely on heap locations are unlikely to succeed under such a transformation. DieHard can also run multiple replicas of an executable and require that all replicas produce the same output for that output to be taken as the output of the executable. This kind of replication prevents many kinds of compromised replicas from providing incorrect output, since attacks are unlikely to affect differently randomized heaps in the same way.

Implementing proactive obfuscation sometimes changes the independence properties exhibited by our underlying approaches to replica management. For example, quorum systems can provide a degree of data independence, since replicas do not necessarily all store the same object states. This is because clients of a quorum system interact only with a quorum rather than interacting with all replicas, so different replicas receive different client messages, hence store different object states. However, our storage-service prototype exhibits little data independence, because it employs a hub to receive input from clients and, therefore, all replicas tend to receive the same messages and store the same object state. In short, our prototypes trade quorum system data independence to gain greater resilience against parallel attacks on Obfuscation Independence (2.1) and Bounded Adversary (2.2), as discussed in §2.

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, 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 paper.

References

[1] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[2] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *ACM Conference on Computer and Communications Security*, pages 281–289, Washington, D.C., October 2003. ACM Press.

[3] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security*, 8(1):3–40, 2005.

[4] E. D. Berger and B. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168, Ottawa, Canada, June 2006.

[5] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, pages 105–120, Washington, D.C., August 2003.

[6] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot—a technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 31–44. USENIX, December 2004.

[7] R. Canetti, S. Halevi, and A. Herzberg. Maintaining authenticated communication in the presence of break-ins. In *ACM Symposium on Principles of Distributed Computing*, pages 15–24, Santa Barbara, California, August 1997.

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

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Symposium on Operating System Design and Implementation*, pages 173–186, New Orleans, Louisiana, February 1999.

[10] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2005.

[11] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical report, School of Computer Science, Carnegie Mellon University, 2002.

[12] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium*, pages 105–120, Vancouver, Canada, July 2006.

[13] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *Advances in Cryptology*, volume 435 of *Lecture Notes in Computer Science*, pages 307–315, Berlin, Germany, 1990. Springer-Verlag.

[14] D. Dolev and H. R. Strong. Polynomial algorithms for multiple processor agreement. In *ACM Symposium on Theory of Computing*, pages 401–407, 1982.

[15] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

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

[17] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72, Cape Cod, Massachusetts, May 1997.

[18] D. K. Gifford. Weighted voting for replicated data. In *Seventh Symposium on Operating Systems Principles*. ACM Press, 10-12 December 1979.

[19] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, 1986.

[20] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In D. Coppersmith, editor, *Advances in Cryptology—Crypto '95*, volume 963 of *Lecture Notes in Computer Science*, pages 339–352. Springer-Verlag, 27–31 August 1995.

[21] Y. Huang, D. Arsenault, and A. Sood. Closing cluster attack windows through server redundancy and rotations. In *Sixth IEEE International Symposium on*

Cluster Computing and the Grid, page 21. IEEE Computer Society, 16–19 May 2006.

[22] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 381–390. IEEE Computer Society, 27–30 June 1995.

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

[24] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *ACM Conference on Computer and Communications Security*, pages 272–280, Washington, D.C., October 2003. ACM Press.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[26] L. Lamport. Personal communication, 2006.

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

[28] B. W. Lampson. The ABCD's of Paxos. In *Twentieth Annual ACM Symposium on Principles of Distributed Computing*, page 13, Newport, RI, USA, 26–29 August 2001. ACM Press.

[29] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[30] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[31] M. Marsh and F. B. Schneider. CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47, January-March 2004.

[32] J. C. Mogul. Simple and flexible datagram access controls for UNIX-based gateways. In *USENIX Summer Technical Conference*, pages 203–222, Baltimore, Maryland, 1989.

[33] Netfilter Project. See http://www.netfilter.org.

[34] OpenBSD Project. See http://www.openbsd.org.

[35] OpenBSD Project. PF: Firewall redundancy with CARP and pfsync. See http://www.openbsd.org/faq/pf/carp.html.

[36] OpenBSD Project. PF: The OpenBSD packet filter. See http://www.openbsd.org/faq/pf.

[37] OpenSSL Project. See http://www.openssl.org.

[38] PaX. See http://pax.grsecurity.net.

[39] R. Pucella and F. B. Schneider. Independence from obfuscation: A semantic framework for diversity. In *IEEE Computer Security Foundations Workshop*, pages 230–241, Venice, Italy, July 2006. IEEE Computer Society.

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

[41] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Symposium on Operating Systems Principles*, pages 15–28, Banff, Canada, October 2001.

[42] F. B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, 1982.

[43] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[44] F. B. Schneider and K. P. Birman. The monoculture risk put into context. *IEEE Security and Privacy*, 7(1):14–17, 2009.

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

[46] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[47] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo. Resilient intrusion tolerance through proactive and reactive recovery. In *Pacific Rim International Symposium on Dependable Computing*, pages 373–380, Melbourne, Victoria, Australia, December 2007. IEEE Computer Society.

[48] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[49] Trusted Computing Group. See http://www.trustedcomputinggroup.org.

[50] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, 2(2):124–137, 2005.

[51] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *IEEE Symposium on Reliable Distributed Systems*, pages 260–269, Florence, Italy, October 2003.

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

[53] L. Zhou, F. B. Schneider, and R. van Renesse. APSS: Proactive secret sharing in asynchronous systems. *ACM Transactions on Information and System Security*, 8(3):259–286, 2005.