PROACTIVE OBFUSCATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Thomas Michael Roeder

February 2010

PROACTIVE OBFUSCATION

Thomas Michael Roeder, Ph.D.

Cornell University 2010

Proactive obfuscation is a new method for creating server replicas that are likely to have fewer shared vulnerabilities. It employs semantics-preserving code transformations to generate diverse executables from a single codebase and periodically restarts servers with these fresh images. Periodic restarts help bound the number of compromised replicas that a service ever concurrently runs, and therefore proactive obfuscation makes an adversary's job more difficult. 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.

Authentication is a large cost in proactive obfuscation. And some protocols used in the prototypes require transferable signed messages (for example, using digital signatures), which can be slow to compute. To reduce authentication costs, we introduce *multi-verifier signatures*, a new family of signature schemes that generalizes traditional digital signatures to a secret-key setting. Some of our multi-verifier signature schemes are faster to compute than digital signatures. Moreover, just like digital signatures, these signatures are both transferable and secure under arbitrary (unbounded) adaptive chosen-message attacks. Practical constructions of digital signature schemes rely on either strong number-theoretic assumptions or are proven secure only in the random oracle model. In contrast, we exhibit practical constructions of multi-verifier signature schemes that are provably secure in the plain model assuming the existence of pseudorandom functions and without assuming access to random oracles.

BIOGRAPHICAL SKETCH

Thomas Michael Roeder was born in Boston, Massachusetts in 1978; his family moved to Richmond, British Columbia, Canada when Thomas was young. His father taught him to program in Pascal on the family's Macintosh 512 at age 11, and he learned Basic and Hypercard not long thereafter. But Thomas showed only occasional interest in computers and programming, preferring music, languages, literature, and mathematics.

Thomas was accepted into the Transition program at University Hill Secondary School in Vancouver, which accelerated him through grades 11 and 12 in one year. He entered the University of British Columbia (UBC) at age 16 and started a program in Combined Honours Mathematics and Physics, intending to become a theoretical physicist. In his second year in the program, however, he took an introductory course in Relativity and Quantum Mechanics and was given an account on the Physics Department UNIX machines. There, he became enthralled by the idea of getting computers to do his work for him. He switched his major to Combined Honours Mathematics and Computer Science the next year.

Thomas graduated from UBC in May 2001, shortly after marrying Marcy Hutchinson, a former Transition program classmate. After a year of working for Silicon Chalk, an educational-software startup founded by UBC professors, Thomas came to Cornell University in August 2002 to pursue a Ph.D. in Computer Science.

To Marcy, John, Daniel, and Abigail

ACKNOWLEDGMENTS

Thanks first and foremost to Fred Schneider, my advisor, who taught me more than I thought there was to learn about the art of writing scientific prose. The clarity of his writing and the depth of his thought contributed greatly to my understanding of Computer Science. And he mentored me adroitly through the prosaic issues of graduate student life, from changing advisors to getting a job.

I am grateful to Rafael Pass for teaching me most of what I know about cryptography, through a couple of courses and several seminars. Rafael was a creative sounding board for my signature work and once came up with a clever short proof to avoid having to read my tedious long one. He was a great source of knowledge for writing papers in cryptography and a valuable check on my proofs.

The other members of my committee, Andrew Myers, Robbert van Renesse, and Draga Zec, all provided helpful insights and questions in the course of my research. I appreciate their work in reading and commenting on this dissertation.

Many other professors contributed to my learning at Cornell. I am grateful to those in the Linguistics department who taught me in classes and seminars, especially Abby Cohn, Amanda Miller, and Draga Zec.

I am also grateful for the funding that allowed me to continue my work through many years. The research presented in this dissertation was supported, in part, by AFOSR grant F9550-06-0019, National Science Foundation Grants 0430161 and CCF-0424422 (TRUST), and Microsoft Corporation. I was fortunate to find employment with Microsoft, first as an intern with Úlfar Erlingsson in Summer 2003 and Galen Hunt in Summer 2004, and now as a full-time employee, working on cryptography and security with Brian LaMacchia and the

Crypto Tools group.

Thanks to all my friends and officemates, especially Michael Clarkson, Scott Condie, Todd Humphreys, Filip Radlinski, Kevin Walsh, and Kamen Yotov, who provided good conversation, a friendly ear for ideas, and silence, when needed.

I was glad to find occasional moments at Cornell to do other things than research or taking care of my family. Yukiko Katagiri, Larry Bieri, and others in the Cornell Aikido Club introduced me to Aikido while I was here. And Mike Hammer and Kevin Feeney, along with others in the Cornell Amateur Radio Club, deserve great thanks for helping to awaken my latent interest in radio communications.

Finally, special thanks go to my parents, John and Kim Roeder, and Marcy's father, Norm Hutchinson, as well as other members of our immediate family: Scott and Janaia Roeder, Geoff Roeder and Erin Spencer, Chuck and Carla Hutchinson, AJ Hutchinson, Leo Hutchinson, and Sam Hutchinson. They all frequently cared for my children so I could focus on research and writing. The time they spent was finite, but their help was immeasurable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 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 coding vulnerabilities and, therefore, do not fail independently when under attack. One contribution of this dissertation is to introduce *proactive obfuscation*, a new method of restoring some measure of independence, whereby each replica is periodically restarted using an executable generated by semantics-preserving *program obfuscation* techniques. These techniques create fresh executables automatically through transformations applied during compilation, loading, or at run-time. Thus, the opportunities are reduced for an adversary to compromise too many of the replicas that constitute a service.

A program obfuscation technique relies on an *obfuscator*, which takes two inputs—a program P and a secret key κ—and produces an *obfuscated executable* semantically equivalent to P.[1] Key κ specifies how transformations are applied to produce the obfuscated executable from P. Each obfuscation technique defends against attacks that exploit implementation details; obfuscated executables are believed more likely to crash in response to certain classes of attacks than to fall under the control of an adversary. For example, 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. One goal of obfuscation is to provide *code independence*: an adversary that success-

[1]The term "obfuscation" has a different meaning in the cryptographic literature (e.g., see Barak et al. [4]). There, the goal of obfuscation is to prevent adversaries from learning anything about a program that they could not learn from the input and output behavior of the program. By contrast, program obfuscation techniques used in this dissertation generate multiple executables from a single original program; these executables should fail independently under some attacks.

fully attacks a replica and learns the key used there for obfuscation gains no advantage in attacking another replica.

Some approaches to replica management also support *data independence*: different replicas store different states. Data independence reduces the vulnerability of replicas to attacks that depend on a replica's state. For example, some implementation flaws can be exercised only when a replica is in a given state—if replicas can have different state, then an attack that exploits such an implementation flaw will not necessarily succeed at all replicas. So, proactive obfuscation can be seen as adopting the ideas of data independence, but for attacks that depend on a replica's implementation details.

Obfuscation techniques include address reordering and stack padding [36, 11, 79], system call reordering [26], instruction set randomization [47, 6, 5], and heap randomization [10]. The details of a given obfuscation technique condition the susceptibility of its obfuscated executables to particular attacks. However, proactive obfuscation itself does not depend on the details of a particular obfuscation technique. It merely depends on using an obfuscation technique that satisfies certain properties, and these properties are detailed below.

Many systems, including Windows Vista, Windows 7, OpenBSD, and Linux, employ obfuscation, either by default or in easily-installed modules. And it has recently been suggested [69] that obfuscation be used for computer monocultures in order to preserve the benefits of deploying the same software on clients while mitigating against a catastrophic response to a single attack vector; the independence provided by obfuscation makes simultaneous failure unlikely.

In addition to independence, building a reliable distributed service often requires implementing authentication—the ability of a server to determine the original source of a message received by that server. *Authenticated channels* are

an abstraction of one simple kind of authentication. A server v that receives a message m on an authenticated channel knows which server sent m, but v is not guaranteed to be able to convince other servers of the origin of m. Thus, authenticated channels provide authentication only between pairs of servers.

Many methods exist to implement authenticated channels. One conceptually simple method is to connect each pair of servers by a separate communication channel. However, this method is prohibitively expensive in all but the smallest systems. Practical authenticated channel implementations use cryptographic algorithms that produce *tags* for messages; these tags are checked by the receiver of a message to authenticate the source.

Transferable authentication generalizes authenticated channels. A server v' that receives a message m and a tag τ generated by a server v using a transferable authentication scheme will know that m was sent by v and can forward m and τ to any other server v'' and know that v'' will also know that m was sent by v. Unfortunately, existing transferable authentication schemes, like *digital signatures* [31], are expensive to compute. A second contribution of this dissertation is to introduce *multi-verifier signatures* (MVS), which can be used to implement transferable authentication and can be computed faster than digital signatures in some contexts.

The rest of this chapter motivates and provides an overview of proactive obfuscation and multi-verifier signatures. Section 1.1 describes our failure model and reviews common approaches to replica management in light of this model. In Section 1.2, we outline proactive obfuscation and its use in conjunction with replica management. Section 1.3 presents an overview of authentication schemes, and Section 1.4 describes our multi-verifier signature schemes. Section 1.5 then describes the structure of the dissertation.

1.1 Fault Tolerance and Attack Tolerance

Replication provides a way to implement fault-tolerant distributed services. In analogy with fault tolerance, we say that services with resilience to attack exhibit *attack tolerance*. A replica can be *crashed*, meaning it does not perform any actions until it reboots.[2] Or, a replica can be *compromised*, because it has malfunctioned or come under control of an adversary. In the fault-tolerance literature, this second kind of failure is often called *Byzantine*. But common usage in the literature presumes Byzantine failures are independent. So, to emphasize that attacks may cause correlated failures, we instead use the term "compromised". A replica that is not crashed or compromised is *correct*. Besides replicas, clients of a distributed service may also be crashed, compromised, or correct.

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

- When the compromise threshold is not exceeded, any replies the system sends to the clients are correct.
- When neither the compromise threshold nor the crash threshold is exceeded, the system produces correct replies to client requests

Since proactive obfuscation converts attacks that would compromise a replica into crashes, all replicas might crash simultaneously. During such crashes, the system will not respond to clients. But our implementation of proactive obfuscation provides a way for crashed replicas to recover without interacting with other replicas. So, these crashes can be treated as periods of unavailability.

[2]Some failure models for distributed systems count a replica as faulty for the entire execution of a protocol if the replica crashes at any time during the execution. Our model does not take this view of failures, since all replicas eventually crash and reboot in our system, both for proactive obfuscation and when under attack.

We assume that adversaries do not have physical access to the replicas, since an adversary with physical access could compromise all replicas simultaneously by disconnecting the servers and replacing them with hardware the adversary controls. Proactive obfuscation only addresses attacks sent to replicas in messages they receive.

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

Proactive obfuscation works in conjunction with various different approaches to replica management. Each replica management approach defines protocols used by replicas to handle client requests as well as protocols to handle crashed and compromised replicas. There are three common approaches to replica management.

The *primary/backup* approach [2] employs a single replica, called the *primary*, to handle all client requests sent to the service. The service also maintains a set of *backup* replicas to which the primary sends either state or the sequence of requests the primary handles. Each backup sends a confirmation to the primary when it has received and acted on a message received from the primary; upon receiving confirmations from enough backups, the primary sends a response to the client.

Backups monitor the primary. When a backup detects that the primary has crashed, some backup takes over for the crashed primary. The state at this

backup must therefore be sufficiently current for continued processing of requests. And the protocol used by the primary ensures that backups all normally store the same state.

Although the primary/backup approach can tolerate replica crashes, it cannot handle compromised replicas. For instance, a compromised primary could undetectably insert arbitrary requests or send incorrect responses to clients. Because an adversary only needs to compromise a single replica (the primary) to take control of the replicated system, code independence, as would be provided by obfuscation, is not all that useful here. For this reason, we did not implement proactive obfuscation for the primary/backup approach.

The *state machine approach* [48, 68] provides a way to build a reliable distributed service that implements the same interface as a program running on a single trustworthy host. Using it, a program is described as a *state machine*, which consists of *state variables* and deterministic[3] *commands* that modify state variables and may produce output. Correct replicas in the state machine approach perform the same actions and have the same state, so replicas in the state machine approach have no data independence.

There are protocols for the state machine approach that can tolerate compromised replicas by having many replicas concurrently process each client request and then coordinate their responses to the client. Replicas in the state machine approach must implement the same interface, since they process the same sequence of requests, but the replicas are not required to use the same implementation. This means that different replicas can be obfuscated differently, and we will see that there are advantages to doing so.

Quorum systems [76, 37, 41] are yet a third approach to replica management.

[3]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.

Here, replicas store *objects* consisting of *state*; an object supports *operations* to modify its state. A quorum system is defined by a collection $\mathcal{Q}$ of *quorums*, each a subset of replicas that satisfies an *intersection property*. The intersection property guarantees that any two quorums have sufficient overlap to allow clients to read the latest state written to objects.

Replicas in a quorum system do not usually all store the same state, because clients of a quorum system interact with only a quorum and, therefore, not all replicas execute the same sequence of operations or store the same objects. This means that quorum systems support data independence.

1.2 Proactive Obfuscation

A simple way to improve attack tolerance on a replicated system would be to obfuscate each replica differently, since servers running the different executables ought to share fewer vulnerabilities. However, this form of independence erodes over time, because an adversary with access to an obfuscated executable can analyze the obfuscated code and customize an attack. Eventually, the attacker will have attacks for each replica.

Proactive obfuscation defends against this by introducing *epochs*; one server is rebooted in each epoch, so that all n servers are 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. So, using proactive obfuscation with epoch length Δ seconds implies that an adversary is forced to compromise more than t replicas in $n\Delta$ seconds in order to subvert the service. 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 lowers the chances that all replicas can be compromised by a single attack. This helps availability and integrity properties of a service.[4] But neither replication nor proactive obfuscation enhances the confidentiality of data stored by the servers, because confidentiality can often be violated by the disclosure of a secret stored by a single replica; integrity and availability properties normally can only be violated by failures of multiple 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.

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 resources at the replicas. This second strategy includes DoS attacks that cause replicas to crash frequently 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 necessarily amenable to formal verification.[5]

[4]For this to hold, we assume that obfuscation itself does not compromise a replica.

[5]In principle, this property of C can make it difficult to perform obfuscation. But unlike most formal methods, obfuscation does not usually require annotation or any significant code analysis, so it is often easier to perform.

The practicality of an approach like proactive obfuscation depends on the cost of implementing the mechanisms to make it work. An additional contribution of this dissertation is to design these mechanisms and quantify their cost for two approaches to replica management. To investigate the costs, we prototyped two services: (i) a distributed firewall (based on the `pf` packet filter [60] in OpenBSD [57]) and (ii) a distributed storage service. One service uses the state machine approach and the other uses quorum systems. Moreover, by building working prototypes, we have decreased the chances of overlooking assumptions in our design or in the applicability of proactive obfuscation.

1.3 Authentication

Cryptographic authentication schemes are inspired by authentication schemes used in real life. For instance, a physical signature on a document is often used to prove that a given individual signed the document. But physical signatures are relatively easy to forge and can be copied and pasted from one document to another, since they do not depend on the document being signed. Moreover, a single instance of a physical signature is of little use, since there is no way to show that this signature was made by a given individual. For this reason, banks normally keep a copy of a depositor's signature to compare against future signatures that authenticate withdrawal requests. And, in general, physical signatures require *setup*: infrastructure and actions that bind signatures to individuals.

Analogous to physical signatures, a tag generated by a digital signature scheme for a given message m using a given *key* proves to any receiver that this signature was generated for m by a given server. The tag must be difficult for an adversary to forge for a given message, even given tags on other mes-

sages of its choice. Servers in a digital signature scheme take one of two roles: *signer* and *verifier*, where the verifier receiving a tag τ has some way of deciding which signer knows the key(s) used to generate τ. Here, too, there is a setup: the actions taken to create this binding between servers and keys.

Keys shared between a group of servers are called *secret* keys. Since each server in the group has the same key, authentication using secret keys is called *symmetric* authentication (and symmetric authentication algorithms are called Message Authentication Codes (MACs) [7]).

Secret-key setup involves generating and distributing keys among groups of servers; normally, keys are shared between pairs of servers, so a set of n servers must generate and share a total of $\binom{n}{2}$ keys. In symmetric authentication, the binding between a key and servers is known only to servers that know the key.

For example, if key k is shared by servers v and v', then when v receives a message m and a tag τ generated by a MAC using key k, server v knows that only v or v' could have generated τ for m. If v knows that it did not generate τ, then v can deduce that v' generated τ. But if three servers v, v', and v'' all share k, then when v receives m and τ that it did not generate, it knows only that one of v' or v'' generated τ.

MACs do not provide *transferability*—the property that tags *accepted* by one host and forwarded to another host will be accepted there too. Transferability is essential in many applications of digital signature schemes (e.g., in distributed systems [21, 81, 54]).

Asymmetric authentication (called *one-way* authentication when first proposed by Diffie and Hellman [31] and now usually identified with digital signatures) associates a server v with a *public* key that can be used to authenticate messages from v. Each public key has an associated *private* key known only to the server

and used by that server to generate tags.

Public-key setup normally involves a well-known trustworthy repository that provides a binding from public keys to servers. This requirement is stronger than what is required for secret-key setup, but it does provide transferability. That is, if server v' receives message m and tag τ from server v and checks τ using v's public key, then v' knows that any other server v'' will also check m and τ with v's public key and come to the same conclusion about τ. Since each server only has one key, a set of n servers only needs to set up n keys—considerably fewer keys than for symmetric authentication.

1.4 Multi-Verifier Signatures

Digital signatures are relatively expensive to generate. Moreover, practical digital signature schemes rely on either strong number-theoretic assumptions [28, 13] or are proven secure only in the random oracle model [64]. In contrast, MACs are orders of magnitude faster and can be based on pseudorandom functions. MACs, however, rely on secret-key setup.

A natural question is whether the secret-key setup used for MACs can be leveraged to get efficient, yet provably secure, digital signature schemes. We answer this question in the affirmative in this dissertation by introducing multi-verifier signatures, which generalize digital signatures to a secret-key setting with a signer and multiple verifiers, each using different keys. We provide two efficient MVS constructions. They are based only on the existence of pseudorandom functions and do not assume random oracles:

- *Atomic Signatures* requires the signer to solve a system of linear equations. As far as we know, Atomic Signatures constitutes the first practical and

provably secure signature scheme based only on symmetric-key primitives.

- *Chain Signatures* provides *λ-limited transferability*, the property that signatures can be transferred at least $\lambda - 1$ times and still be accepted by receivers. Although λ-limited transferability is weaker than transferability, it suffices in many settings (e.g., see systems [21, 54, 81, 61], where each message is forwarded only a fixed number of times). Furthermore, for values of λ used in practical protocols, Chain Signatures outperforms the fastest implementations of digital signature schemes (even though these digital signature schemes are only secure in the random oracle model [64]).

Our MVS constructions require an unusual secret-key setup—pairwise shared keys distributed in such a way that the signer does not know which key corresponds to which verifier. This additional secrecy prevents the signer from creating signatures that would be accepted by some hosts but not others. The required secret-key setup is easily implemented, for example, in an operating system (OS) or small distributed service. Processes in an OS already trust the OS, so the OS can distribute shared keys when a process is created. Similarly, a small distributed service (e.g., [21, 54, 81]) that is managed by a single administrator can distribute keys before the service begins executing. In these practical settings, MVS schemes provide a speed advantage over common digital signature schemes.

Our multi-verifier signature schemes require each server to keep n bits of state—one per server. Bit $\beta_{v'}^{v}$ is stored at server v' and tracks whether v' believes server v to be compromised. Server v' decides that v is compromised if v' receives a special return value $\perp$ from the multi-verifier signature scheme while

verifying a given message m and tag τ from v.[6] Return value $\perp$ signifies that, with all but negligible probability, τ could only have been generated for m by a compromised signer. Keeping state to track whether or not other servers are compromised is a natural function in a distributed system, since servers normally do not interact with other servers that they deem to be compromised.

1.5 Dissertation Structure

Proactive obfuscation is presented in Chapter 2, along with mechanisms for its implementation. Then, Chapter 3 gives an overview of the state machine approach to replica management, extends it with proactive obfuscation, and describes and evaluates a firewall prototype. Quorum systems are presented in Chapter 4, along with an extension with proactive obfuscation and a description and evaluation of a storage-service prototype. The theory of multi-verifier signatures is described in Chapter 5, and our two constructions are presented and evaluated in Chapter 6. Finally, Chapter 7 offers some conclusions.

[6] Replicas could also keep a list of compromised signers; this list will require $O(t)$ bits instead of n, so the list should be used when $t = o(n)$.

CHAPTER 2

PROACTIVE OBFUSCATION

Proactive obfuscation uses an obfuscator to produce obfuscated executables. We abstract from the details of the obfuscator by defining two properties we require it to implement.

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

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

Obfuscation Independence (2.1) implies that differently 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.

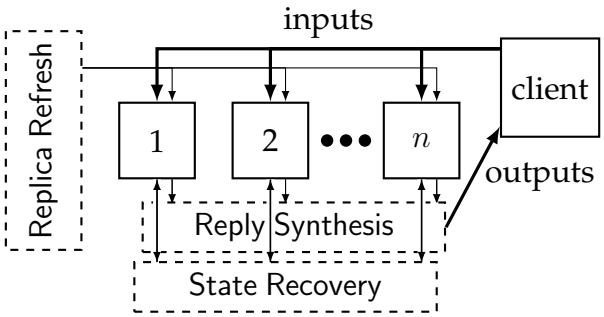

Figure 2.1: Implementing proactive obfuscation

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 the attacker. Furthermore, these crashes do not help the adversary violate Bounded Adversary (2.2), since they slow down both recovery and attacks by the same amount.

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 recovery. Figure 2.1 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 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) Synchronous Processors. Differences in the rate of instruction execution on different correct hosts are bounded.

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

Approximately Synchronized Clocks (2.6), Synchronous Processors (2.7), and Timely Links (2.8) imply that the system implements the *synchronous model* [52]. Together, they are used to guarantee a bound on the time involved in running Replica Refresh and subsequent State Recovery. 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:

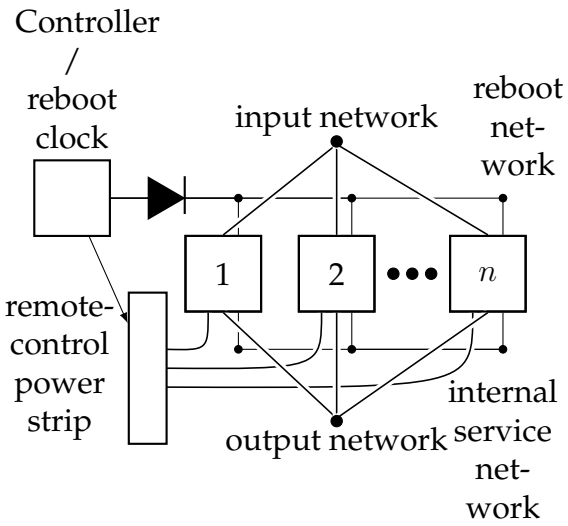

Figure 2.2: The prototype architecture

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 2.2 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 2.2. Reply Synthesis is performed by clients. State Recovery is performed by replicas using a lossy network, labeled *internal service network*, that satisfies Timely Links (2.8). Replica 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 2.2 on the line from the Controller depicts the constraint that the Controller never receives messages on the reboot network.[1]

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 (2.8). Epochs are labeled with monotonically increasing *epoch numbers* that are incremented at each epoch change. For epoch changes with our decentralized protocols, we use timeouts 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

[1]There are circumstances in which the Controller can communicate with hosts that are known not to be compromised. For example, see the centralized Controller implementation in Section 2.2.3.

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 [72]) by the replicas. Each replica is given a *share* of the private key and uses this share to compute *partial signatures* [29] 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 [82] 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,

[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.

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 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, especially those caused by attacks.

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 lo-

[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 Chapter 7 for a discussion of crashes caused as part of DoS attacks.

cally stored state for State Recovery, 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.

Before executing State Recovery, a recovering replica i establishes authenticated channels to all replicas it communicates with. The recovery protocol then proceeds as follows:

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

Notice that step 4 might cause the recovering replica to see messages corresponding to earlier states. So, the replication protocol must guarantee that such messages will not cause this replica to assume an incorrect state; most replica management protocols provide this guarantee.

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. To guarantee this property, we assume that the state at each replica is bounded. Furthermore, the implementation of the state recovery request for each approach to replica management must ensure that the operations it requires complete in a bounded amount of time.

One problem that might keep steps 2 and 3 from completing in a bounded amount of time is that other replicas processing the state recovery request might crash an unbounded number of times due to attacks by an adversary. This could slow down State Recovery by an arbitrary factor. To solve this problem, a replica that has recovered its state after a crash does not perform any operations. Instead, it listens for for a bounded amount of time to see if there are any state recovery requests from a recovering replica.[5] If there are no such messages, then the replica restarts execution of Input Coordination and Reply Synthesis and continues as before. Otherwise, it processes the State Recovery messages from the recovering replica before handling messages from clients or other replicas.

The goal of this crash-recovery protocol is to make sure that replicas crash and recover at most once during each period of State Recovery. Then, Approximately Synchronized Clocks (2.6), Synchronous Processors (2.7), and Timely Links (2.8) together imply that steps 2 and 3 complete in a bounded amount of time.

However, to achieve this goal, we must also ensure that replicas that recover from a crash initially only process messages from the recovering replica, since a message from a compromised replica might make this replica crash again. This

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

can be achieved as long as there is a way for replicas recovering from a crash to sort messages and process only those coming from a recovering replica. It must be the case that this method for sorting replica messages cannot cause the replica to crash again.[6]

Finally, we must 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.[7]

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 that uses a Controller for Replica Refresh can be quite efficient. For instance, a centralized implementation can provide epoch-change notification directly to replicas, can

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

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

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 guarantees that executables are generated by a correct host, as required by Executable Generation (2.5). Executables are transferred to recovering replicas through 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.

(2.9) 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, except as determined by the Controller, 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 (2.9) 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 [20] modern switches allow such control of individual ports by a third party like the Controller.

The Controller can perform bidirectional communication with a recovering replica that has not yet taken any input from the other replicas, since our assumptions implie that this replica cannot have been compromised yet. So, the Controller employs PXE boot [45] to load an image on a recovering replica. This works because Reboot Channels (2.9) implies that the Controller can temporarily modify the network to allow the rebooting replica bidirectional communication with the Controller.

Key Distribution. The Controller performs key distribution to implement authenticated channels by generating a new public/private RSA [64] 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.[8] Reboot Channels (2.9) 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.

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.

[8]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.

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 section 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:

(2.10) 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[9], 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.[10]

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 (2.10) implies the code on the CD-ROM cannot change the BIOS in order to accomplish this switch.

[9]This, in turn, can be implemented using a secure co-processor like the Trusted Platform Module [77].

[10]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.

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 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 [18]. 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.[11]

To prevent more than one key from being signed per epoch, replicas use Byzantine Paxos [51], 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.[12]

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

[11] It might seem unnecessary to use APSS, an asynchronous protocol, since Replica Refresh must complete in a bounded amount of time and operates under synchronous assumptions. But using asynchronous protocols under synchronous assumptions provides the maximum practical defense if these synchronous assumptions are violated.

[12] 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.

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 section 2.2 mostly involve incurring higher CPU costs for increased decentralization. Under high load, these CPU costs divert a replica's resources away from 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 par-

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

Throughput can be improved in both of our Reply Synthesis implementations by batching output—instead of signing each output packet, replicas opportunistically produce a single signature for a batch of output packets up to a maximum batch size, called the *batching factor*. 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 net-

work, so this has little effect on performance. However, decentralized Executable Reobfuscation significantly increases the window of vulnerability: 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 [54]. And in the CODEX implementation of APSS, share refresh requires significant CPU resources, 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.

2.4 Related Work

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 [44, 78] and Self-Cleansing Intrusion Tolerance [43] 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.

Microreboot [14] 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 [22] adds proactive state recovery to Practical Byzantine Fault-Tolerance [21]. Proactive state recovery here is analogous to key refresh in proactive secret sharing [42] (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 repli-

cas 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, experiments (see section 3.2.2.3 and section 4.2.2.2 in the following chapters) 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 [3] 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 [3] consists of diverse replicas and a supervisor program that runs these diverse replicas in parallel and performs coordination of inputs, as well as 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 [27] 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 dif-

ferent copies. The monitor compares responses and deems a variant compromised if this variant 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 [80] 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 [65] 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. [36]. 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 [11, 59, 79] 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 [47, 6, 5] 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 [10] 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 this chapter.

CHAPTER 3

STATE MACHINE REPLICA MANAGEMENT

In the state machine approach with state machine m, a *state machine ensemble* SME(m) consists of n servers that each implement the same interface as m and that 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 have the same state and, therefore, the state machine approach does not have data independence.

Coordination of client commands to servers is not one of the mechanisms identified in Figure 2.1. 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* [50, 33] 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 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, or reordered before the agreement algorithm is run. However, the networks over which clients communicate with the service provide only a best-effort delivery guarantee, so it is reasonable to assume that clients would already employ mechanisms 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 section 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 Byzan-

tine Paxos while tolerating t compromised replicas is known to be $3t + 1$ [21]. 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 [49] 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 causes 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* [34]: 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.[1]

[1]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*.

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.[2]

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.[3]

A replica might fall behind in the execution of Byzantine Paxos. Such replicas need a way to obtain messages they 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

[2]The existence of a 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.

[3]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.

number. Upon receiving a RepeatRequest, a replica resends the requested message if it has a copy.[4]

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.[5] 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

[4]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.

[5]Note that requiring the replies to be output similar means that $t+1$ replies suffice. Of course, the client might need to receive up to $2t+1$ replies in total before it receives $t+1$ replies that are output similar.

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

1. The recovering replica contacts each replica with a state recovery request.
2. Each replica that receives this request performs two actions:
    a. This replica sends its current state to the recovering replica, along with a diff that allows the recovering replica to get the previous state and the messages that convinced the replica to change to this state.
    b. This replica also sends to the recovering replica all the messages that it receives for the agreement protocol over the next T seconds.
3. After $T + \eta$ seconds, the recovering replica uses the states, diffs, and messages it has received to recover.

For this algorithm to work, replicas must keep not only their current state, but also a diff to get to their previous state (if any), along with all messages that they have received and verified for the current round of agreement.[6]

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

[6]An optimization is for replicas to reply immediately with their replica state the first time they receive a state recovery request from a recovering replica, instead of running the protocol. If a recovering replica i does not receive $t + 1$ identical replica states from these responses, then i can send a second request; upon receiving the second request, replicas run the above protocol. Our firewall prototype implements this optimization, and the system has never executed a second request, because recovering replicas always got $t + 1$ identical replica states on their first request in the experiments we ran. And this optimized version guarantees that all replicas still acquire the current minimum state at the time the state recovery protocol starts.

be the highest sequence number held by a correct replica at time T seconds after the beginning of the state recovery protocol.

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

See Appendix A for a proof of this property. SME Highest State Recovery (3.2) implies SME State Recovery (3.1), since s is guaranteed to be at least the current minimum state at the time the state recovery protocol starts. And this protocol succeeds at recovering a state for the recovering replica in a bounded amount of time even when Byzantine Paxos operates asynchronously, as long as replicas can always marshal representations of their state in a bounded amount of time.[7]

The time needed to execute this protocol is bounded, given Timely Links (2.8), Synchronous Processors (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 section 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 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

[7]For this protocol to work, it must also be the case that a recovering replica receiving messages from the other replicas for its state recovery request cannot be made to crash by these messages.

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.

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 through 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 through RepeatRequest messages (this time is bounded, given Timely Links (2.8), Synchronous Processors (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 through 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

Table 3.1: The versions of our firewall prototype

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	dist
Reboot Clock	Byz Paxos	indvdl auth	dist	dist	dist
Threshold Client	Byz Paxos	thresh crypto	dist	dist	dist

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 section 2.2.2 with state recovery request and state synthesis as described in section 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 3.1.

3.2.1 Experimental Setup

Our implementations are written in C using OpenSSL [58]; we also use OpenSSL for key generation. We take $t = 1$ and $n = 6$; all hosts are 3 GHz Pentium 4 machines with 1 GB of RAM running OpenBSD 4.0. We can justify setting $t = 1$ provided Bounded Adversary (2.2) is satisfied; 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, mechanisms 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 [55]. The pseudo-device allows programs running in user space to take and replace packets on the network stack, similar to Linux netfilter [56].

The `pf` code provides a pseudo-device called `pfsync` [62] 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 [26] 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 [35] 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.[8] 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

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

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

Throughput. To quantify how throughput is 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 without averse effect. We consider two prototype versions that differ only in their implementation of the Input Coordination mechanism: the Replicated version and the Centralized version. Both use RSA signatures for Reply Synthesis.

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 3.1 shows throughput for each of the versions.[9] 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

[9]Discussion of the Threshold Client curve appears below in section 3.2.2.2.

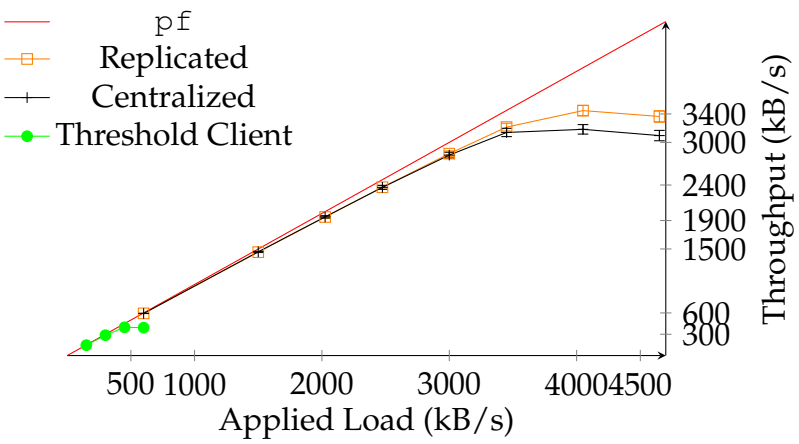

Figure 3.1: Overall throughput for the firewall prototype

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 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 3.1, but with one replica crashed. While the replica was crashed, throughput in the Centralized version drops to 1133 $\pm$ 10 kB/s.[10] 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 3.1, latency was measured at 39 $\pm$ 3 ms for the Centralized version, whereas latency in the Replicated version was 28 $\pm$ 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.[11] 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 ex-

[10] 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 $\pm$ 18 kB/s.

[11] 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.

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

3.2.2.2 Reply Synthesis

Throughput. Throughput for different Reply Synthesis implementations is already given in Figure 3.1, because in an experiment where no Replica Refresh occurs, any differences between the Centralized and Threshold Client versions 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 3.1 confirms the prediction of section 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 from 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 section 2.2.3 to create and distribute a certificate for this key.

Figure 3.2 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

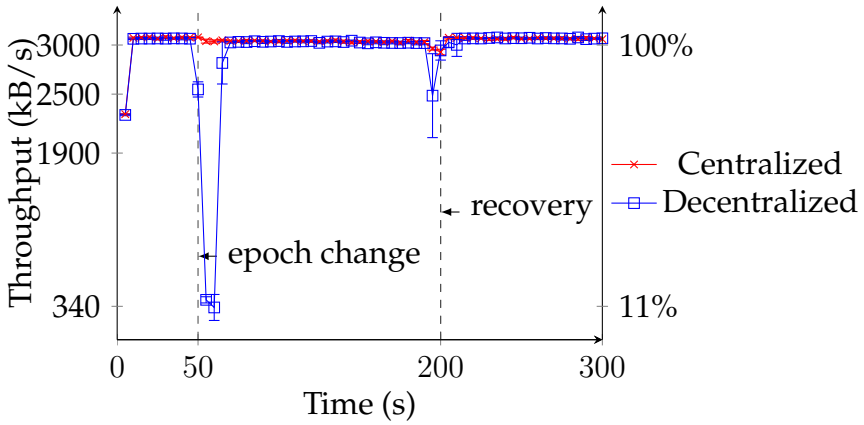

Figure 3.2: Two key distribution methods for the firewall prototype at an applied load of 3300 kB/s

perform leader adjustment to take the rebooting replica into account, so there is no throughput drop.

At the point marked "recovery" in Figure 3.2, 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.

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 3.2 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.

CHAPTER 4

QUORUM SYSTEM REPLICA MANAGEMENT

Recall from Chapter 1 that a quorum system stores objects consisting of object state; operations modify this object state. The replicas themselves store object states but do not necessarily perform the operations. And a quorum system Q consists of quorums, each a set of replicas. An intersection property guarantees some minimum overlap between any pair of quorums.

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 [53] for the semantics of concurrent operations:

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

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

On abort, 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 [53] are defined by a pair $(\mathcal{Q}, \mathcal{B})$; collection $\mathcal{Q}$ of replica sets defines a set of quorums, 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* [53]; 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 [53]:

(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 overlaps in 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. Correct clients then would choose the object state created by the compromised replicas. And this would prevent object states written by correct clients from ever being received. 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.[1]

[1] 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.

Of course, this defense only preserves the semantics of quorum systems for objects written by correct clients. An object that is written by a compromised client might not satisfy these properties. For instance, a compromised client might not write all a new object state to a complete quorum. So, one correct client reading the object might get one version while a second correct client might get a different version.

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

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. [74]). 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 ($n = 3t + 2r + 1$) chosen above 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 an *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.[2]

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

[2]Other implementations [53] 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 $t + 1$.

message with object ID o in the reply from j.[3] 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.

(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 section 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

[3] 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.

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 (2.8), Synchronous Processors (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 Synchronous Processors (2.7) and Approximately Synchronized Clocks (2.6).

Table 4.1: The versions of our storage-service prototype

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

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 use the State Recovery implementation from section 2.2.2 with the state recovery request as described in section 4.1. Table 4.1 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 section 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 section 4.2.2. Replicas do not currently write to disk for crash recovery.

4.2.2 Performance Measurements

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

4.2.2.1 Reply Synthesis

We performed experiments to quantify the performance of the individual authentication implementation for Reply Synthesis. In our experiments, replicas start with 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 4.1 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

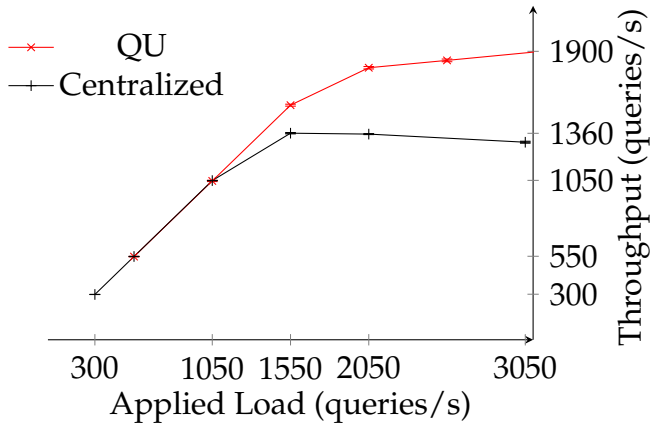

Figure 4.1: Overall throughput for the storage-service prototype

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. This implementation of the single server does not generate extra signatures for the objects it returns, since all objects are already signed. And clients check object signatures in both cases.

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 ± 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 ± 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 4.2 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 4.2—it also reaches a peak at a maximum batching

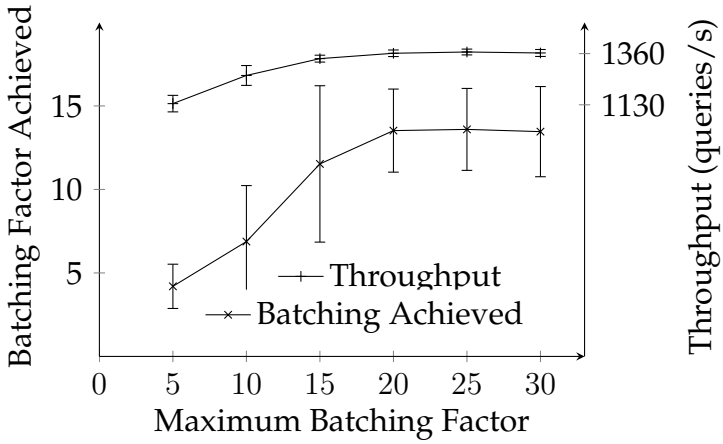

Figure 4.2: Batching factor and throughput for the storage-service prototype under saturation

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 slow-down 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 vulnerability.

Throughput. Query throughput measurements given in Figure 4.3 confirm the results of section 3.2.2.3, which compares centralized and decentralized key

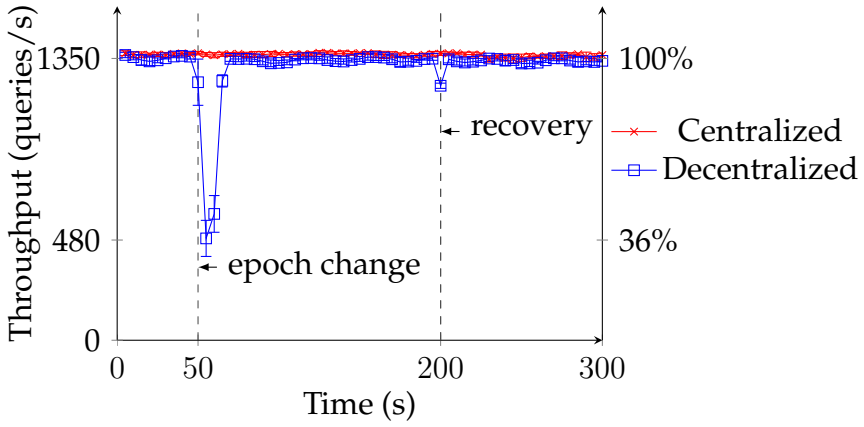

Figure 4.3: Two key distribution methods for the storage-service prototype at an applied load of 1400 queries/second

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 both State Recovery and share recovery.

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 per-

forming cryptographic operations for agreement. We believe that kernel-level packet handling operations performed by the storage service mesh better with the high CPU utilization of APSS than the cryptographic operations performed by the firewall.

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

Latency. The same experiments as for Figure 4.3 lead to a similar graph for latency. Latency in the Decentralized case for the storage-service prototype increases to 646 ± 89 ms during share refresh, as opposed to 205 ± 3 ms for the same interval in the Centralized case. This latency is lower than what was seen in section 3.2.2.3 for the firewall prototype during share refresh. As for throughput, we believe this is due to the kernel-level packet handling operations of the storage-service prototype competing better with the high CPU costs of APSS in user space. Latency also increases slightly during share recovery. The Decentralized version has a latency of 233 ± 3 ms during share recovery, whereas the Centralized version has a latency of 203 ± 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 affects throughput and latency. The state of the

storage 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 section 3.2.2.4. And, therefore, the amount of state held by replicas in the firewall prototype under saturation is held by replicas in the storage-service prototype when there are 115 object states.

However, we would typically expect a storage service to have many more than 115 object states. To confirm the analysis of section 2.3.2 without using too many object states, our storage-service prototype uses a simple implementation of the state recovery request that does not batch object states for recovery, but instead uses one round of communication for each object state from each replica. The cost of recovery can be reduced by batching object states and preventing identical object states from being sent multiple times. But this does not change the linear relationship between recovery time and the number of object states.

Throughput. Figure 4.4 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 4.4 shows that throughput drops during State Recovery to about 470 queries/s for time proportional to the number of objects being recovered—there is a linear relationship: each object adds about 260 ms to the recovery time. This reduction in throughput is due to the CPU time replicas spend sending and

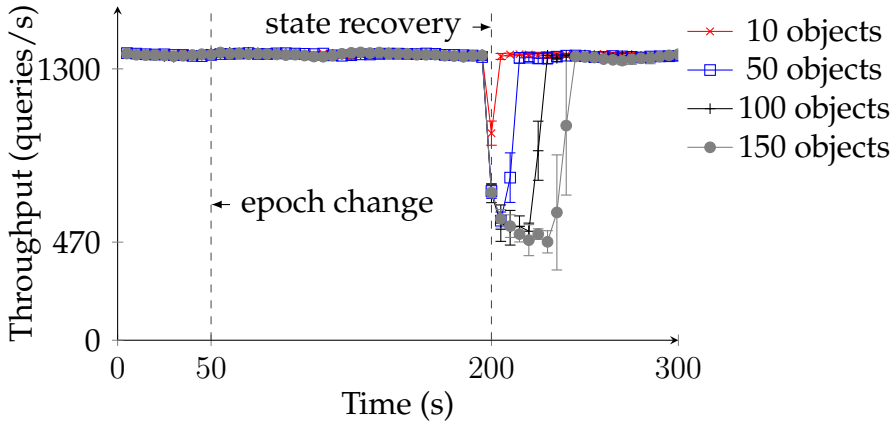

Figure 4.4: Throughput during recovery for the storage-service prototype

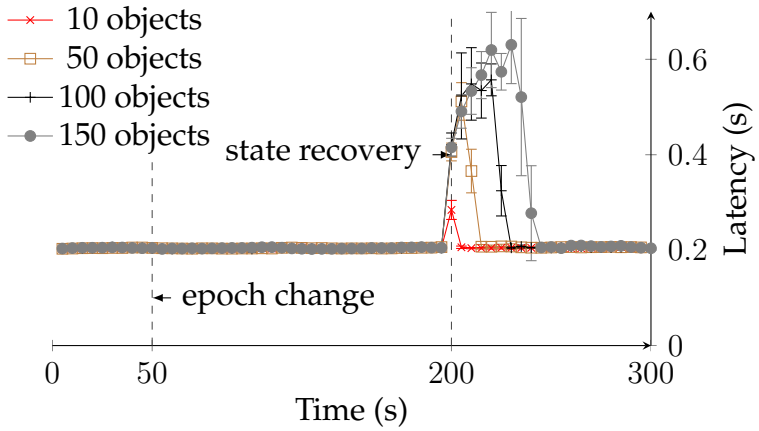

Figure 4.5: Latency during recovery for the storage-service prototype

receiving recovery messages (instead of processing inputs from the client).

Latency. Figure 4.5 compares latency during recovery in the same experiments used to generate Figure 4.4. 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.

CHAPTER 5

MULTI-VERIFIER SIGNATURES

Our constructions of MVS schemes rely on MACs, which take a message m and a key k as input and output a *tag* that can be used to authenticate m given k. The traditional model of MAC security [9] requires that it be hard for an adversary $\mathcal{A}$ to generate a message m and tag τ that will be accepted by a receiver, even if $\mathcal{A}$ has access to a MAC oracle (as long as $\mathcal{A}$ has not already requested that m be signed by the oracle). We write $\mathcal{A}_r$ for an adversary $\mathcal{A}$ using randomness r.

We require the MAC to satisfy a stronger property (similar to Bellare, Goldreich, and Mityagin [8]), called *Chosen Tag Attack (CTA) Unforgeability*, which additionally gives adversaries access to a verification oracle $\mathsf{VF}(m, \tau, k)$ such that

$$\mathsf{VF}(m, \tau, k) = \begin{cases} \infty & \text{if } \mathsf{MAC}(m, k) = \tau \\ 0 & \text{otherwise.} \end{cases}$$

If $\mathsf{VF}(m, \tau, k) = \infty$ holds, then the tag is accepted by a receiver.

Let $\mathsf{MACreq}(\mathcal{A}, m, k, r)$ be a predicate that is true if an adversary $\mathcal{A}_r$ requested m from its MAC oracle using key k in a given execution. Then, for a given security parameter d, we can define the following property for Probabilistic Polynomial-Time (PPT) adversaries.

(5.1) CTA Unforgeability. For every non-uniform PPT adversary $\mathcal{A}$, there exists a negligible function ϵ such that

$$\begin{aligned} \Pr[k \leftarrow \mathsf{Gen}(1^d); r \leftarrow \{0,1\}^\infty; \\ m, \tau \leftarrow \mathcal{A}_r^{\mathsf{MAC}(\cdot,k),\mathsf{VF}(\cdot,\cdot,k)}(1^d) : \\ \neg\mathsf{MACreq}(\mathcal{A}, m, k, r) \wedge \mathsf{VF}(m, \tau, k) = \infty] \leq \epsilon(d). \end{aligned}$$

Pseudorandom functions (see [38] for an overview) can be used to construct MACs satisfying CTA Unforgeability (5.1).

5.1 Defining Multi-Verifier Signatures

We define an MVS scheme to be a triple $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ of algorithms that depend on a set I of n *verifiers*; each verifier may use a different key when calling Ver. And each server executes a verifier with a different key. These algorithms operate as follows, where argument d is a security parameter and b specifies a given number of bits.

- $\mathsf{Gen}(1^d, 1^n)$ is a PPT algorithm that outputs a vector $\boldsymbol{k}$ of keys for the signer and a vector $\boldsymbol{K}_j$ of keys for each verifier $j \in I$. Key k in $\boldsymbol{k}$ or $\boldsymbol{K}_j$ (for any $j \in I$) is an element of $\{0,1\}^b$.
- $\mathsf{Sign}(m, \boldsymbol{k})$ takes $m \in \{0,1\}^b$ and produces[1] a tag τ.
- For each $j \in I$, $\mathsf{Ver}(m, \tau, \boldsymbol{K}_j)$ produces a value that is either ∞, 0, an element of $\mathbb{N}_{>0}$, or $\perp$.[2] If $\mathsf{Ver}(m, \tau, \boldsymbol{K}_j)$ returns ∞, then we say that τ is *accepted* by j, and if $\mathsf{Ver}(m, \tau, \boldsymbol{K}_j)$ returns 0, then we say that τ is not accepted by j. A return value $\lambda \in \mathbb{N}_{>0}$ means that τ is accepted by j, and also that there is a lower bound $\lambda - 1$ on the number of times this tag can be transferred and still cause verifiers at other correct receivers to return a value in $\mathbb{N}_{>0}$. If $\mathsf{Ver}(m, \tau, \boldsymbol{K}_j)$ returns $\perp$, then j sets a state variable v_j to $\perp$; verifier j thereafter believes the signer to be compromised.

Note that the above description implies Ver is a stateful algorithm: whenever Ver produces $\perp$ for a message purporting to come from signer i, it decides that i is compromised and remembers this fact. Thereafter, Ver will only return $\perp$ for message and tag pairs purporting to come from i.

[1] For simplicity of exposition, we define MVS schemes for b-bit messages only. These definitions are easily extended to arbitrary-length messages. Our constructions are described for both fixed-length and arbitrary-length messages.

[2] Assume that ∞ satisfies only the following properties: $\infty - \infty = 0$, $\forall a \in \mathbb{N} : \infty > a$, and $\forall a \in \mathbb{N} : |\infty - a| = |a - \infty| = \infty$.

Let λ be an element of $\mathbb{N}^{\infty} \triangleq \mathbb{N}_{>0} \cup \{\infty\}$, and let $\mathsf{req}(\mathcal{A}, m, \boldsymbol{k}, r)$ be a predicate that is true if an adversary $\mathcal{A}_r$ requested m from its signing oracle using keys $\boldsymbol{k}$ in a given execution. MVS schemes satisfy four properties: λ-Completeness, Unforgeability, Non-Accusability, and Transferability.

λ-Completeness (5.2) stipulates that, for any $\boldsymbol{K}_j$ and any message m, a tag τ generated by $\mathsf{Sign}(m, \boldsymbol{k})$ causes $\mathsf{Ver}(m, \tau, \boldsymbol{K}_j)$ to return a value that is greater than or equal to λ. Note that ∞-Completeness thus requires $\mathsf{Ver}(m, \tau, \boldsymbol{K}_j)$ to return ∞; the value ∞ means that a message can be transferred an arbitrary number of times, just like digital signatures. Formally, we can state this property as follows.

(5.2) **λ-Completeness.** There exists a negligible function ϵ such that for any message m and any $j \in I$,

$$\Pr[(\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i \in I}) \leftarrow \mathsf{Gen}(1^d, 1^n) : \mathsf{Ver}(m, \mathsf{Sign}(m, \boldsymbol{k}), \boldsymbol{K}_j) < \lambda] \leq \epsilon(d, n).$$

As with digital signature schemes, Unforgeability (5.3) requires that no adversary $\mathcal{A}$ be able to generate a message m and tag τ that will be accepted by any correct verifier if $\mathcal{A}$ has not previously requested a tag for m from its signing oracle, even if $\mathcal{A}$ might have compromised any set I' of verifiers. Formally, we can state this property as follows.

(5.3) **Unforgeability.** For every non-uniform PPT adversary $\mathcal{A}$, there exists a negligible function ϵ such that for any choice of $I' \subseteq I$,

$$\begin{aligned}\Pr[&(\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i \in I}) \leftarrow \mathsf{Gen}(1^d, 1^n); r \leftarrow \{0,1\}^{\infty};\\ &(m, \tau) \leftarrow \mathcal{A}_r^{\mathsf{Sign}(\cdot, \boldsymbol{k}), \{\mathsf{Ver}(\cdot, \cdot, \boldsymbol{K}_i)\}_{i \in I - I'}}(1^d, 1^n, \{\boldsymbol{K}_i\}_{i \in I'}) :\\ &\neg \mathsf{req}(\mathcal{A}, m, \boldsymbol{k}, r) \wedge (\exists j \in I - I' \ : \ \mathsf{Ver}(m, \tau, \boldsymbol{K}_j) \in \mathbb{N}^{\infty})] \leq \epsilon(d, n).\end{aligned}$$

Non-Accusability (5.4) requires that no adversary $\mathcal{A}$ be able to generate a message and tag pair that causes any correct verifier to return $\perp$ if the signer is not compromised, even if $\mathcal{A}$ can request signatures on arbitrary messages and has compromised an arbitrary subset of verifiers. Formally, we can state this property as follows.

(5.4) Non-Accusability. For every non-uniform PPT adversary $\mathcal{A}$, there exists a negligible function ϵ such that for any choice of $I' \subseteq I$,

$$\begin{aligned}\Pr[(\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I}) \leftarrow \mathsf{Gen}(1^d, 1^n);\\ (m, \tau) \leftarrow \mathcal{A}^{\mathsf{Sign}(\cdot,\boldsymbol{k}),\{\mathsf{Ver}(\cdot,\cdot,\boldsymbol{K}_i)\}_{i\in I-I'}}(1^d, 1^n, \{\boldsymbol{K}_i\}_{i\in I'}) :\\ (\exists j \in I - I' \ : \ \mathsf{Ver}(m, \tau, \boldsymbol{K}_j) = \perp)] \leq \epsilon(d, n).\end{aligned}$$

Transferability (5.5) requires that even a compromised signer is unable to create a tag on which any pair of correct verifiers return values that differ by more than 1; this property implies that if verification of a message m and tag τ at any correct verifier returns v, then m and τ can be transferred between correct verifiers at least $v - 1$ times. A tag that does not satisfy Transferability (5.5) is said to be *split*.[3]

Our notion of Transferability (5.5) is slightly different than the notion of Transferability guaranteed by standard digital signatures. In particular, a verifier j that returns a value in $\mathbb{N}^\infty$ for a message m and tag τ is not guaranteed that any non-compromised verifier j' to which it passes m and τ will also return a value in $\mathbb{N}^\infty$; the other possibility is that j' returns $\perp$—which means that j' has acquired proof that the signer is compromised.

[3]Verification algorithms for MVS schemes allow some verifiers to return $\perp$ and conclude nothing about the tag. Each tag divides the correct verifiers into two groups: those that accept the tag and those that find the signer compromised. This specification is reminiscent of Crusader's Agreement [32], where the correct receivers are divided into those that agree on a single value and those that know the sender to be compromised.

Formally, we can state this property as follows.

(5.5) Transferability. For every non-uniform PPT adversary $\mathcal{A}$, there exists a negligible function ϵ such that for any choice of $I' \subseteq I$,

$$\begin{aligned}
&\Pr[(\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I}) \leftarrow \mathsf{Gen}(1^d, 1^n);\\
&\quad (m, \tau) \leftarrow \mathcal{A}^{\{\mathsf{Ver}(\cdot,\cdot,\boldsymbol{K}_i)\}_{i\in I-I'}}(1^d, 1^n, \boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I'}) :\\
&\quad (\exists j, j' \in I - I' \;:\; \mathsf{Ver}(m, \tau, \boldsymbol{K}_j) \neq \perp \wedge \mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'}) \neq \perp\\
&\qquad\qquad \wedge \left|\mathsf{Ver}(m, \tau, \boldsymbol{K}_j) - \mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'})\right| > 1)] \leq \epsilon(d, n).
\end{aligned}$$

An MVS scheme that satisfies λ-Completeness (5.2) for $\lambda \in \mathbb{N}_{>0}$ is called a *λ-MVS* scheme; tags generated by this scheme can be transferred at least $\lambda - 1$ times. An MVS scheme that satisfies ∞-Completeness is called an *∞-MVS* scheme; tags generated by this scheme can be transferred an unlimited number of times. A stronger version of Transferability (5.5), called *Perfect Transferability*, would require that $\epsilon(d, n) = 0$ in the definition of Transferability (5.5); Appendix 5.2 shows that ∞-MVS schemes satisfying Perfect Transferability are essentially traditional digital signature schemes.[4]

To gain confidence in our definitions, Appendix D provides a definition of MVS schemes based on an ideal signature functionality (similar to Canetti's definition of signatures [16] in the Universal Composability (UC) framework) and shows that a UC-secure implementation of this ideal functionality is equivalent to our game-based definition.

[4]Boneh, Durfee, and Franklin [12] show a related lower bound: digital signatures are required for short collusion-resistant multicast MAC constructions.

Strong Unforgeability

Similar to digital signature schemes, an MVS scheme can satisfy *Strong Unforgeability*, which requires that if $\mathcal{A}$ did not receive m and τ from its signing oracle, then m and τ will not cause any verifier to return a value greater than 0 (leading this verifier to accept the pair) or return $\perp$ (leading this verifier to conclude that the signer is compromised). Define predicate $\mathsf{recv}(\mathcal{A}, (m, \tau), \boldsymbol{k}, r)$ to be true if $\mathcal{A}$, using randomness r, received m and τ from the signing oracle using keys $\boldsymbol{k}$. Formally, the property is written as follows:

(5.6) Strong Unforgeability. For every non-uniform PPT adversary $\mathcal{A}$, there exists a negligible function ϵ such that for any choice of $I' \subseteq I$,

$$\begin{aligned}&\Pr[(\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I}) \leftarrow \mathsf{Gen}(1^d, 1^n); r \leftarrow \{0,1\}^\infty;\\ &\quad (m,\tau) \leftarrow \mathcal{A}_r^{\mathsf{Sign}(\cdot,\boldsymbol{k}),\{\mathsf{Ver}(\cdot,\cdot,\boldsymbol{K}_i)\}_{i\in I-I'}}(1^d, 1^n, \{\boldsymbol{K}_i\}_{i\in I'}) :\\ &\quad \neg\mathsf{recv}(\mathcal{A}, (m,\tau), \boldsymbol{k}, r) \wedge (\exists j \in I - I' \ : \ \mathsf{Ver}(m, \tau, \boldsymbol{K}_j) \neq 0)] \leq \epsilon(d, n).\end{aligned}$$

Note that Strong Unforgeability (5.6) implies both Unforgeability (5.3) and Non-Accusability (5.4). An MVS scheme that satisfies Strong Unforgeability (5.6) in addition to λ-Completeness (5.2) and Transferability (5.5) is called a *Strong λ-MVS* scheme.

5.2 Impossibility of Avoiding Split Tags

Transferability (5.5) asserts that, except with negligible probability, even compromised signers cannot find split tags. But for traditional digital signatures, it is impossible to have split tags, since all verifiers use the same function to check tags. A natural question is whether such perfect transferability can be achieved in our MVS setting.

We can prove that an ∞-MVS scheme must contain a digital signature scheme to avoid split tags perfectly. Start with an ∞-MVS scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ that satisfies ∞-Completeness (5.2), Unforgeability (5.3), and Non-Accusability (5.4). Now strengthen Transferability (5.5) to the following.

(5.7) Perfect Transferability. For any PPT $\mathcal{A}$ and for any choice of $I' \subseteq I$,

$$\begin{aligned}
&\Pr[(\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I}) \leftarrow \mathsf{Gen}(1^d, 1^n);\\
&\quad (m, \tau) \leftarrow A^{\{\mathsf{Ver}(\cdot,\cdot,\boldsymbol{K}_i)\}_{i\in I-I'}}(1^d, 1^n, \boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I'}) :\\
&\quad (\exists j, j' \in I - I' \ : \ \mathsf{Ver}(m, \tau, \boldsymbol{K}_j) \neq \perp \wedge \mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'}) \neq \perp\\
&\qquad\qquad \wedge \left|\mathsf{Ver}(m, \tau, \boldsymbol{K}_j) - \mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'})\right| > 1)] = 0
\end{aligned}$$

Note that the only difference between the definition of Transferability (5.5) (see page 80) and Perfect Transferability (5.7) is that $\epsilon(d, n)$ is set to 0 in the definition of Perfect Transferability (5.7).

Perfect Transferability (5.7) implies that there are no split tags under any choice of keys, because if a split tag existed, then an adversary that guessed message and tag pairs at random would have some non-zero probability of choosing it. Also note that we can remove the restriction on verifiers j and j' being non-compromised: if there is any pair of verifiers for which a split tag can be created, then an adversary can choose not to compromise those verifiers and guess the message and split tag with a non-zero probability. We can thus rewrite Perfect Transferability (5.7) as follows:

$$\begin{aligned}
&\forall d, n, \forall \boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I} \in \mathrm{Range}(\mathsf{Gen}(1^d, 1^n), \forall m, \tau, \forall j, j' \in I \ :\\
&\quad (\mathsf{Ver}(m, \tau, \boldsymbol{K}_j) \neq \perp \wedge \mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'}) \neq \perp) \implies\\
&\quad \left|\mathsf{Ver}(m, \tau, \boldsymbol{K}_j) - \mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'})\right| \leq 1
\end{aligned}$$

To simplify the statement of Perfect Transferability (5.7) further, we must consider a restricted class of ∞-MVS schemes.

Given any ∞-MVS scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$, we can define a new ∞-MVS scheme $\Sigma' = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver}')$, in which the verifier only returns 0, ∞, and $\perp$. To do so, $\mathsf{Ver}'(m, \tau, \boldsymbol{K}_j)$ calls $\mathsf{Ver}(m, \tau, \boldsymbol{K}_j)$ and gets a reply v. Then Ver' returns v if v is one of 0, ∞, or $\perp$. Otherwise, Ver' returns 0. Note that Ver' satisfies ∞-Completeness (5.2) if Ver does, since ∞-Completeness for Σ guarantees that Ver always returns ∞ on message and tag pairs generated by Sign. And it is trivial to see that Σ' satisfies each of Unforgeability (5.3), Non-Accusability (5.4), Strong Unforgeability (5.6), Perfect Transferability (5.7), and Transferability (5.5) if Σ does. We call Σ' a *normalized* ∞-MVS scheme from Σ.

Since verifiers that do not return $\perp$ in a normalized ∞-MVS scheme either return 0 or ∞, and $|\infty - 0| > 1$ holds, Perfect Transferability (5.7) can be rewritten for normalized ∞-MVS schemes as follows:

$$\begin{aligned}
&\forall d, n, \forall \boldsymbol{k}, \{\boldsymbol{K}_i\}_{i \in I} \in \mathrm{Range}(\mathsf{Gen}(1^d, 1^n), \forall m, \tau, \forall j, j' \in I \ : \\
&\quad (\mathsf{Ver}(m, \tau, \boldsymbol{K}_j) \neq \perp \wedge \mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'}) \neq \perp) \implies \\
&\quad\quad \mathsf{Ver}(m, \tau, \boldsymbol{K}_j) = \mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'})
\end{aligned}$$

Perfect Transferability (5.7) interacts with Unforgeability (5.3), since verifiers in a normalized ∞-MVS scheme that satisfies Perfect Transferability (5.7) can all simulate the actions of other verifiers perfectly; a tag that violates Unforgeability (5.3) must do so for all verifiers at once.

For digital signature schemes, the property corresponding to Unforgeability (5.3) is Chosen Message Attack (CMA) security [39]. A digital signature scheme is secure under CMA if no adversary $\mathcal{A}$ can produce a message m and tag τ that causes the verification algorithm to return ∞, even if $\mathcal{A}$ can see tags for messages of its choice. Naturally, as in Unforgeability (5.3), the adversary cannot return a message it requested from its signing oracle.

The following theorem says that any normalized ∞-MVS scheme that sat-

isfies Perfect Transferability (5.7) instead of Transferability (5.5) is effectively a digital signature scheme secure under CMA [39].[5] The intuition behind the theorem is that Perfect Transferability (5.7) effectively makes all verifiers use the same algorithm: access to one verifier allows perfect simulation of the actions of any other verifier.

Given a normalized ∞-MVS scheme $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ that satisfies ∞-Completeness (5.2), Unforgeability (5.3), Non-Accusability (5.4), and Perfect Transferability (5.7), we can define a digital signature scheme $\Sigma_j^D = (\mathsf{Gen}^D, \mathsf{Sign}^D, \mathsf{Ver}^D)$ as follows for any j in I. Let $\mathsf{Gen}^D(1^d)$ call $\mathsf{Gen}(1^d, 1^n)$ to produce public key $K = \boldsymbol{K}_j$ and secret key $k = \boldsymbol{k}$. Then $\mathsf{Sign}^D(\cdot, k)$ just calls $\mathsf{Sign}(\cdot, \boldsymbol{k})$. And $\mathsf{Ver}^D(\cdot, \cdot, K)$ calls $\mathsf{Ver}(\cdot, \cdot, \boldsymbol{K}_j)$, getting response v. Ver^D returns v if v is 0 or ∞, and Ver^D returns 0 if v is $\perp$, since all verifiers use the same verification function in Σ_j^D, hence never disagree about its return value.

Theorem 1. *If* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ *is a normalized* ∞*-MVS scheme satisfying* ∞*-Completeness (5.2), Unforgeability (5.3), Non-Accusability (5.4), and Perfect Transferability (5.7), then, for any* $j \in I$, Σ_j^D *is a digital signature secure under CMA.*

Proof. By the contrapositive. Suppose that we are given a non-uniform PPT adversary $\mathcal{A}$ that violates CMA security of Σ_j^D with non-negligible probability ϵ.

We will construct a PPT $\mathcal{B}$ that violates Unforgeability (5.3) of the MVS scheme for any $I' \subseteq I$ such that $j \in I'$ and $\exists j' \in I - I'$, as follows. $\mathcal{B}$ is given $1^d, 1^n$, $\{\boldsymbol{K}_i\}_{i \in I'}$, and oracle access to $\mathsf{Sign}(\cdot, \boldsymbol{k}) = \mathsf{Sign}^D(\cdot, k)$ and $\{\mathsf{Ver}(\cdot, \cdot, \boldsymbol{K}_i)\}_{i \in I - I'}$, so in particular, $\mathcal{B}$ knows $\boldsymbol{K}_j$. $\mathcal{B}$ proceeds as follows:

- $\mathcal{B}$ calls $A(1^d, \boldsymbol{K}_j)$

[5] Note that there is a normalized ∞-MVS scheme for every ∞-MVS scheme.

- When $\mathcal{A}$ requests $\mathsf{Sign}^D(m, k)$, $\mathcal{B}$ calls the signing oracle $\mathsf{Sign}(m, \boldsymbol{k})$ and returns its response.
- When $\mathcal{A}$ calls $\mathsf{Ver}^D(m, \tau, K)$, then $\mathcal{A}$ evaluates and returns $\mathsf{Ver}(m, \tau, \boldsymbol{K}_j)$.

- When $\mathcal{A}$ returns m and τ, $\mathcal{B}$ returns m and τ.

When $\mathcal{A}$ succeeds, m and τ satisfy $\mathsf{Ver}(m, \tau, \boldsymbol{K}_j) = \mathsf{Ver}^D(m, \tau, K) = \infty$. In this case, Perfect Transferability (5.7) and Non-Accusability (5.4) together imply that $\mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'}) = \infty$ with all but negligible probability: Non-Accusability (5.4) implies that $\mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'}) \neq \perp$ with all but negligible probability, and Perfect Transferability (5.7) (in its rewritten form for normalized ∞-MVS schemes) states that in this case, $\mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'}) = \mathsf{Ver}(m, \tau, \boldsymbol{K}_j) = \infty$ always holds. This violates Unforgeability (5.3). $\mathcal{B}$ has never requested m from its signing oracle, because $\mathcal{A}$ is required by assumption never to request m of its signing oracle. $\mathcal{B}$ succeeds with the same non-negligible probability ϵ as $\mathcal{A}$.

Thus, $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver}(\cdot, \cdot, \boldsymbol{K}_j))$ is a digital signature secure under CMA if $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ is a normalized ∞-MVS scheme that satisfies ∞-Completeness (5.2), Unforgeability (5.3), Non-Accusability (5.4), and Perfect Transferability (5.7). □

5.3 Implementing Agreement using Multi-Verifier Signatures

Multi-verifier signatures cannot be used directly in the place of digital signatures in algorithms for implementing Byzantine agreement, because signature verification on forwarded signed messages can sometimes return $\perp$. Call such a signed message a $\perp$ *message*. A replica that receives a $\perp$ message learns that the signer is compromised but comes to no conclusion about whether or not the message is correctly signed. To implement agreement using a particular proto-

col P with multi-verifier signatures replacing digital signatures in the execution of P, replica i takes the following actions when it receives a $\perp$ message.

1. Send a *compromise claim* message to all other replicas claiming that the signer is compromised.
2. Stop handling all messages for protocol P and continue sending compromise claim messages about the signer until it receives confirmation from each replica j that j received the claim or until the compromised signer is rebooted. This means that i acts like a crashed replica with respect to P.

Replica j believes replica j' to be compromised once j has received $t + 1$ compromise claim messages about j', since then j has received a compromise claim message about j' from at least one correct replica. We assume that there is a way for a replica that is detected as compromised to be rebooted and made correct.

Note that this protocol implies that all correct replicas will eventually believe a compromised signer to be compromised once $t + 1$ replicas have received a $\perp$ message from a given signer, as long as the network guarantees that enough messages will eventually be received. So, an implementation of Byzantine agreement can assume that no more than t replicas ever receive $\perp$ messages from each compromised replica. That is, at most t^2 replicas will stop before some compromised replica is discovered. So multi-verifier signatures can be used to replace digital signatures by allowing for t^2 extra replicas to crash.

Also, note that if more than t^2 replicas receive $\perp$ messages, then some compromised replica will eventually be detected as compromised. Once a compromised replica is detected, it can be rebooted and replaced, and the replicas that had stopped can start participating in the protocol again. But, during the time that more than t^2 replicas have stopped, the protocol might not be able to make progress.

This method for extending a protocol to use multi-verifier signatures can be applied, for instance, to Byzantine Paxos. If t replicas can be compromised and f replicas can crash, then Byzantine Paxos can be implemented with $3t + 2f + 1$ replicas [49]. So, Byzantine Paxos can be implemented with $3t + 2t^2 + 1$ replicas using multi-verifier signatures.

CHAPTER 6

MULTI-VERIFIER SIGNATURE CONSTRUCTIONS

The definitions of Chapter 5 allow each verifier to use different keys. So, multi-verifier signature schemes can employ a secret-key setup like the one used for MACs. But two questions remain: Do any constructions of multi-verifier signatures satisfy these definitions? And are such constructions faster to generate and check than digital signatures?

To demonstrate the existence and practicality of constructions of multi-verifier signatures, we present Atomic Signatures and Chain Signatures. We then compare the performance of these two schemes to fast implementations of digital signatures.

6.1 Atomic Signatures

Atomic Signatures is a Strong ∞-MVS scheme in which a signer computes a tag for a message m by using MACs to generate and solve a system of linear equations that depend on m. Unlike MACs and digital signature schemes, generation algorithm Gen^{AS} for Atomic Signatures distributes disjoint, equal-sized sets of random keys to each verifier; a verifier shares each key in each set with the signer, but the signer does not know which keys it shares with which verifier.

More precisely, for n verifiers and security parameter d, generation algorithm $\mathsf{Gen}^{AS}(1^d, 1^n)$ for Atomic Signatures produces dn keys $k_1, k_2, \ldots, k_{dn}$, where, for each j such that $1 \leq j \leq dn$, $k_j \in \{0,1\}^b$ is a key for a MAC. Gen^{AS} also generates dn random vectors $\boldsymbol{z}_j = \langle z_{j,1}, z_{j,2}, \ldots, z_{j,dn} \rangle$ (one vector for each key k_j), where for each j and i such that $1 \leq j \leq dn$ and $1 \leq i \leq n$, each entry $z_{j,i}$ is an element of $\{0,1\}^b$. Gen^{AS} then sets $\boldsymbol{k}$ to be a vector of key pairs $\langle (k_1, \boldsymbol{z}_1), (k_2, \boldsymbol{z}_2), \ldots, (k_{dn}, \boldsymbol{z}_{dn}) \rangle$. And Gen^{AS} creates n vectors $\boldsymbol{K}_1, \boldsymbol{K}_2, \ldots, \boldsymbol{K}_n$

that each contain a unique, randomly chosen set of d key pairs from $\boldsymbol{k}$ such that $\boldsymbol{K}_i$ and $\boldsymbol{K}_j$ are disjoint for each distinct pair i and j of indices. Set $\boldsymbol{k}$ is called the *signing key pairs* and the n vectors $\boldsymbol{K}_1, \boldsymbol{K}_2, \ldots, \boldsymbol{K}_n$ are called the *verifying key pairs*. We say that a verifier j *owns* each key k in $\boldsymbol{K}_j$.

Signing algorithm $\mathsf{Sign}^{AS}(m, \boldsymbol{k})$ takes message m and signing key pairs $\boldsymbol{k}$ as input[1] and outputs a vector $\boldsymbol{A}^d(m) = A_1, A_2, \ldots, A_{dn}$ of subtags, obtained by solving the following equation, where b-bit strings are treated as elements in the finite field GF(2^b),

$$\begin{pmatrix} \mathsf{MAC}(m, k_1) \\ \mathsf{MAC}(m, k_2) \\ \vdots \\ \mathsf{MAC}(m, k_{dn}) \end{pmatrix} = \begin{pmatrix} z_{1,1} & z_{1,2} & \cdots & z_{1,dn} \\ z_{2,1} & z_{2,2} & \cdots & z_{2,dn} \\ \vdots & \vdots & \vdots & \vdots \\ z_{dn,1} & z_{dn,2} & \cdots & z_{dn,dn} \end{pmatrix} \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_{dn} \end{pmatrix} \tag{6.1}$$

Recall that solving equation (6.1) corresponds to solving the dn instances ($1 \leq j \leq dn$) of the following equation simultaneously:

$$\mathsf{MAC}(m, k_j) = \sum_{t=1}^{dn} z_{j,t} A_t. \tag{6.2}$$

Define predicate $\mathsf{roweq}(m, \boldsymbol{A}, (k_j, \boldsymbol{z}_j))$ to be true exactly when equation (6.2) holds for m, $\boldsymbol{A}$, and $(k_j, \boldsymbol{z}_j)$.

Verification algorithm $\mathsf{Ver}^{AS}(\cdot, \cdot, \boldsymbol{K}_j)$ keeps state v_j to record that it has been called with a message and tag that indicate the signer is compromised. Algorithm $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j)$ operates as follows:

- If v_j is set to $\perp$, then $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j)$ returns $\perp$.
- Otherwise, if $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ holds for all d pairs $(k, \boldsymbol{z})$ in $\boldsymbol{K}_j$, then algorithm $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j)$ returns ∞.

[1] Atomic Signatures as described can sign arbitrary-length messages as long as the MAC can generate tags for arbitrary-length messages.

- If no $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ holds for any $(k, \boldsymbol{z})$ in $\boldsymbol{K}_j$, then $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j)$ returns 0.
- Otherwise, there is some key pair $(k, \boldsymbol{z}) \in \boldsymbol{K}_j$ for which $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ holds, and a different key pair $(k', \boldsymbol{z}') \in \boldsymbol{K}_j$ for which $\mathsf{roweq}(m, \tau, (k', \boldsymbol{z}'))$ does not hold. So, $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j)$ returns $\perp$ and sets v_j to $\perp$.

This verification scheme is based on the idea that it is hard for an adversary that does not know a key $(k, \boldsymbol{z})$ in $\boldsymbol{k}$ to produce a message m and tag τ that will cause $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ to hold. The intuition behind the proof is that the MAC is hard to forge.

This scheme also relies on it being hard for an adversary that has not compromised the signer to produce a message and tag pair that causes any non-compromised verifier to return $\perp$. This property must hold even if the adversary can see a polynomial number of messages and tags for messages of its choice. The intuition behind the proof of this property is that it is hard to find new solutions to under-determined equations over variables with randomly-chosen, unknown coefficients (the pseudorandom MAC values and the random rows of the matrix).

However, it is easy for a compromised signer to cause one verifier to return $\perp$: the signer can create a new set of signing keys $\boldsymbol{k}'$ by replacing one of the keys in $\boldsymbol{k}$ with a random bit string. Then the signer uses $\boldsymbol{k}'$ to generate and solve equation (6.1). Some verifier j will then only find $d - 1$ satisfied instances of $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ and will return $\perp$. The only task that must be difficult for a compromised signer is creating a message and tag pair that cause one non-compromised verifier to return ∞ and another to return 0.

The verification and signing algorithms for Atomic Signatures as described above are expensive to compute: each verifier has $d + d^2n$ keys, and the signer

must solve a matrix equation at cost $O(d^3n^3)$ to generate a signature. Both of these costs can be reduced.

One way to reduce the number of keys is for each verifier j to share $2d$ keys with the signer instead of $d + d^2n$—instead of keys $(k, \boldsymbol{z})$, a verifier shares a pair (k^1, k^2), where keys k^1 and k^2 are elements of $\{0,1\}^b$. Key k^1 is used to compute MACs in the place of k in equation (6.1), and k^2 is used to generate matrix row elements $z_t = \mathsf{MAC}(t, k^2)$ for t in $\{1, 2, \ldots, dn\}$; each verifier then only needs $2d$ keys: 2 for each of its d rows. The proofs follow in the same way as with a randomly chosen matrix but with an extra hybrid step to go from random matrix elements to matrix elements generated by a pseudorandom function.[2]

The cost of generating a signature can be reduced further. Gen^{AS} can produce a prefactored matrix (using the LU factorization, for instance) for use by Sign^{AS} on the right-hand side of equation (6.1). Factoring is cost-effective here, because the matrix is independent of the message to be signed—factoring costs $O(d^3n^3)$ but only needs to be done once, and solutions to equation (6.1) can be found for a factored matrix in time $O(d^2n^2)$.

6.1.1 Properties of Atomic Signatures

Since a Strong ∞-MVS scheme must satisfy ∞-Completeness (5.2), Strong Unforgeability (5.6), and Transferability (5.5), we prove that Atomic Signatures is a Strong ∞-MVS by proving two lemmas. The first establishes that Atomic Signatures satisfies ∞-Completeness (5.2) and Strong Unforgeability (5.6); the second that it satisfies Transferability (5.5).

Lemma 2. *If the* MAC *is a pseudorandom function, then Atomic Signatures satisfies*

[2]The adversary in this case is given oracle access to the MAC functions, so it can compute signatures efficiently.

∞-Completeness (5.2) and Strong Unforgeability (5.6).

Proof. **∞-Completeness.** This follows from non-singularity of the matrix in equation (6.1), since a non-singular matrix allows a solution $\boldsymbol{A}^d(m)$ to be found for equation (6.1). Any solution satisfies ∞-Completeness (5.2) by definition, since $\mathsf{roweq}(m, \boldsymbol{A}^d(m), (k_j, \boldsymbol{z}_j))$ will hold for each row j. The probability of a random matrix of size $dn \times dn$ over a finite field of size 2^b being singular is known [23] to be $1 - \prod_{i=1}^{dn}(1 - 1/2^{bi})$, which is negligible when d and n are polynomial in b.

Strong Unforgeability. Lemmas 12–14 from Appendix B simplify the problem to the case where all but one verifier q is compromised, $\mathsf{MAC}(\cdot, k)$ for row j is replaced by a random function $v_j(\cdot)$, and adversaries are allowed no verification oracle queries.

For such an adversary to violate Strong Unforgeability (5.6), it must produce a message m and tag τ' such that it has never received (m, τ') from the signing oracle, and $\mathsf{Ver}^{AS}(m, \tau', \boldsymbol{K}_q) \neq 0$ holds; this case occurs exactly when predicate $\mathsf{roweq}(m, \tau', (k, \boldsymbol{z}))$ holds for at least one pair $(k, \boldsymbol{z})$ in $\boldsymbol{K}_q$. We consider two cases. In Case 1, m has been received from the signing oracle, but with a different tag τ. In Case 2, m has never been received from the signing oracle. Since the MAC for a given row i is replaced with a random function v_i, we write $(v_i, \boldsymbol{z}_i)$ in the place of key information $(k_i, \boldsymbol{z}_i)$.

Case 1. Consider an adversary that makes a series of signing oracle queries m_1, m_2, …, m_p, receives responses τ_1, τ_2, …, τ_p and finally outputs a message m_w (where $1 \leq w \leq p$) and tag τ' such that $\tau' \neq \tau_w$. Note that τ' must not equal τ_w because $\mathcal{A}$ cannot return a message and tag pair that it received from its signing oracle. We show that for any choices of m_1, m_2, …, m_p, τ_1, τ_2, …, τ_p, the probability that τ' is a valid tag for m_w is negligible.

We consider the case where the adversary violates Strong Unforgeability for a particular message m_w and produces a tag τ' satisfying the equation for m_w for a particular row i. We will then use the Union Bound to bound the probability for any row and any message. Let $(v, \boldsymbol{z})$ be the key information for row i. Recall that message m and tag τ induces an equation for key information $(v, \boldsymbol{z})$: $v(m) = \sum_{t=1}^{dn} z_t \tau_t$.

The probability that a tag τ' produced by an adversary satisfies the given equation is bounded by the following conditional probability for any choice of τ', τ_1, τ_2, ..., τ_p where $\tau' \neq \tau_w$.

$$\Pr_{v,\boldsymbol{z}}\left[v(m_w) = \sum_{t=1}^{dn} z_t \tau'_t \;\middle|\; \forall j \in \{1,\ldots,p\} \;:\; v(m_j) = \sum_{t=1}^{dn} z_t \tau_{j,t}\right]$$

This is the probability that a given tag τ' violates Strong Unforgeability using a given row with randomly chosen coefficients, constrained by the fact that τ_1 through τ_p satisfy the equation for this row. And the definition of conditional probability states that, for events A and B, $\Pr[A \mid B] = \Pr[A \wedge B]/\Pr[B]$. For the conditional probability above, event A is $v(m_w) = \sum_{t=1}^{dn} z_t \tau'_t$, and event B is $\forall j \in \{1,\ldots,p\} \;:\; v(m_j) = \sum_{t=1}^{dn} z_t \tau_{j,t}$. So, the probability can be split into two components: (i) $\Pr_{v,\boldsymbol{z}}[\forall j \in \{1,\ldots,p\} \;:\; v(m_j) = \sum_{t=1}^{dn} z_t \tau_{j,t}]$ and (ii) $\Pr_{v,\boldsymbol{z}}[v(m_w) = \sum_{t=1}^{dn} z_t \tau'_t \;\wedge\; \forall j \in \{1,\ldots,p\} \;:\; v(m_j) = \sum_{t=1}^{dn} z_t \tau_{j,t}]$. We will consider these two components separately. For each component, we can consider the values of τ' and τ_1, τ_2, ..., τ_p as fixed and the $(v, \boldsymbol{z})$ as variables.

Fixing any values for the dn variables z_1, z_2, ..., z_{dn} in component (i) induces a single solution (namely $v(m_i) = \sum_{t=1}^{dn} z_t \tau_{i,t}$) for the values of $v(m_i)$. This means that there are 2^{bdn} ways to choose the variables to satisfy the equations. And there are, a priori, $2^{b(dn+p)}$ ways to choose the values for $dn + p$ variables. Since the probability is over the random choice of variables, this probability can be computed as the quotient: $2^{bdn}/2^{b(dn+p)} = 1/2^{-bp}$.

In component (ii), each way of setting $dn-1$ of the variables leads to a unique solution for the remaining $p+1$ variables. To see why, recall that vectors τ' and τ_w must differ. So, they must differ in at least one of their dn entries; suppose, without loss of generality, that they differ in position a. If we set the $dn-1$ variables z_1, z_2, ..., z_{a-1}, z_{a+1}, ..., z_{dn} to any value, then we are left with p equations of the form $v(m_i) = \tau_{i,a} z_a + c_i$ for $i \in \{1, \ldots, p\}$ and constants c_i and one equation $v(m_w) = \tau'_a z_a + c'$ for some constant c'. Subtracting the equation $v(m_w) = \tau_{w,a} z_a + c_w$ from $v(m_w) = \tau'_a z_a + c'$ eliminates $v(m_w)$ and leaves an equation that uniquely determines the value of z_a, since $\tau'_a - \tau_{w,a} \neq 0$. This value for z_a then uniquely determines the values of $v(m_1)$, $v(m_2)$, ..., $v(m_p)$, by definition. So, there are $2^{b(dn-1)}$ solutions in total. And, as before, there are $2^{b(dn+p)}$ ways to choose values for these variables. So, the probability is $2^{b(dn-1)}/2^{b(dn+p)} = 2^{-b(p+1)}$.

We can use the definition of conditional probability and the two component probabilities to compute the bound on an adversary violating Strong Unforgeability using a particular equation and a particular message: $2^{-b(p+1)}/2^{-bp} = 2^{-b}$. The Union Bound over the d equations and p possible messages then gives a general bound of $pd/2^b$, which is negligible.

Case 2. Now, suppose that m was never received from the signing oracle, and consider any pair m and τ generated by adversary $\mathcal{A}$. Since $\mathcal{A}$ never received m from the signing oracle, no function of the values $v_i(m)$ for row i has been seen by $\mathcal{A}$. So, the output of $\mathcal{A}$ is independent of $v_i(m)$. Fix a key $(v_j, \boldsymbol{z}_j)$ in the keys owned by verifier q and suppose, without loss of generality, that the $\boldsymbol{z}_j$ are all known to the adversary. For a given m, there is one choice of the value of $v_j(m)$ such that $v_j(m) = \sum_{t=1}^{dn} z_{j,t}\tau_t$ holds and there are a total of 2^b ways to choose the value of $v_j(m)$.

Since the choice of τ is independent of $v_j(m)$, and v_j is a random function, the probability of τ satisfying the given row equation is $1/2^b$. The Union Bound then gives the probability of (m, τ) violating Strong Unforgeability (5.6) for any of the d row equations to be $d/2^b$, which is negligible.

Then, Lemmas 12–14 give a polynomial increase in each bound to get back to the general case. But a polynomial increase of a negligible function still leaves it negligible. So, Atomic Signatures satisfies Strong Unforgeability (5.6). □

Proving that Atomic Signatures satisfies Transferability (5.5) is more challenging. A compromised signer able to generate a split tag must have knowledge about which verifier owns which keys. So, we devise a game over the signing keys, called *Idealized Random Keys*, by which we show that no adversary gets enough information to divide the signing keys into two disjoint sets, where each set contains all the keys owned by some verifier. Then, given an adversary that can violate Transferability (5.5), we produce a new adversary that can divide the signing keys into two such disjoint sets. This shows that the security of Idealized Random Keys reduces to Transferability (5.5) of Atomic Signatures. And we also show that Idealized Random Keys is secure, which means that Transferability (5.5) holds for Atomic Signatures.

6.1.2 Idealized Random Keys

Idealized Random Keys is a game between a *requester* and a set I of n *checkers* that each own a set of keys. The requester can only perform *ownership queries*: asking checkers about the ownership of keys. Checkers return $\perp$ in response to ownership queries about keys they own. The important property of Idealized Random Keys, called *Non-Separability*, is that no adversary $\mathcal{A}$ can use ownership queries to separate the set of keys into two disjoint subsets, each of which

contains all the keys owned by some checker that has not already returned $\perp$.

Formally, Idealized Random Keys consists of a pair $(\mathsf{Gen}^{IR}, \mathsf{Check}^{IR})$ of algorithms that operate as follows:

- $\mathsf{Gen}^{IR}(1^d, 1^n)$ generates a vector $\boldsymbol{k}$ of dn keys uniformly at random and partitions them into n disjoint sets $\boldsymbol{K}_1$, $\boldsymbol{K}_2$, ..., $\boldsymbol{K}_n$ of size d, each set owned by one checker $j \in I$.
- Check algorithm $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ keeps state s_j to record whether the requester has ever made an ownership query to j for an element of $\boldsymbol{K}_j$. When requester i makes an ownership query on a given key $k \in \boldsymbol{k}$, check algorithm $\mathsf{Check}^{IR}(k, \boldsymbol{K}_j)$ operates as follows.
  - if $k \in \boldsymbol{K}_j$ or $s_j = \perp$, then $\mathsf{Check}^{IR}(k, \boldsymbol{K}_j)$ returns $\perp$ and sets s_j to $\perp$.
  - Otherwise, $\mathsf{Check}^{IR}(k, \boldsymbol{K}_j)$ returns 1.

This behavior corresponds to information adversaries can glean from attacks on Atomic Signatures. For instance, a compromised signer j can create a tag τ for any message m by solving equation (6.1) using correct keys for all rows but one. Then j sends m and τ to a verifier j'. If j' owns the keys for this row, then j' will return $\perp$. This corresponds to a requester asking checker j' about a key in $\boldsymbol{K}_{j'}$. But if j' does not own the keys for this row, then j' will return ∞; so, j learns that j' does not own the keys for this row. This corresponds to a requester asking checker j' about a key it does not own.

Non-Separability can be written formally as follows:

(6.1) Non-Separability. For all adversaries $\mathcal{A}$, there is a negligible function ϵ

such that for any choice of $I' \subseteq I$,

$$\begin{aligned}
&\Pr[(\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I}) \leftarrow \mathsf{Gen}^{IR}(1^d, 1^n);\\
&\quad (K, K') \leftarrow \mathcal{A}^{\{\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_i)\}_{i\in I-I'}}(1^d, 1^n, \boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I'}) :\\
&\quad (\exists j, j' \in I - I' \ : \ s_j \neq \perp \wedge s_{j'} \neq \perp\\
&\qquad\qquad \wedge \boldsymbol{K}_j \subseteq K \wedge \boldsymbol{K}_{j'} \subseteq K' \wedge K \cap K' = \emptyset)] \leq \epsilon(d, n).
\end{aligned}$$

To show that Non-Separability holds for Idealized Random Keys, we first consider the case $n = 2$, then provide a reduction from $n = 2$ to general n. These proofs will also show how to choose d for a given n so that Idealized Random Keys satisfies Non-Separability (6.1) with a given probability $\epsilon_0 > 0$,

Lemma 3. *Idealized Random Keys for 2 checkers satisfies Non-Separability (6.1) with probability* $\binom{2d}{d}^{-1}$.

Proof. Consider a passive adversary $\mathcal{B}$ that never makes any ownership queries to its Check^{IR} oracles but produces sets K and K' that violate Non-Separability (6.1). Keys are distributed randomly, so $\mathcal{B}$ has no information about which keys correspond to which checkers. Therefore, $\mathcal{B}$'s output is independent of the distribution of keys to checkers. There are $\binom{2d}{d}$ choices for K and K', only one of which violates Non-Separability (6.1). So, $\mathcal{B}$'s probability of outputting this K and K' is $\binom{2d}{d}^{-1}$.

Now suppose that some adversary $\mathcal{A}$, potentially making ownership queries to its Check^{IR} oracles, violates Non-Separability (6.1) with probability ϵ_0. When $\mathcal{A}$ succeeds, no ownership queries to its oracles can have returned $\perp$, since there are only two checkers, j and j', and neither s_j nor $s_{j'}$ can be $\perp$ for Non-Separability (6.1) to be violated. So, $\mathcal{A}$'s ownership queries must always have returned 1. This means that $\mathcal{B}$ can simulate these ownership queries by always returning 1. Whenever $\mathcal{B}$'s simulation would be incorrect (because Check^{IR}

should have returned $\perp$), $\mathcal{A}$ would have received $\perp$ and failed to violate Non-Separability (6.1). So, $\mathcal{B}$ succeeds at least as often as $\mathcal{A}$. Then, $\mathcal{A}$'s success probability is also no better than $\binom{2d}{d}^{-1}$, since this is an upper bound on any passive adversary $\mathcal{B}$. □

Theorem 4. *Idealized Random Keys for n checkers satisfies Non-Separability (6.1) with probability $\binom{n}{2}/\binom{2d}{d}$.*

Proof. We proceed by contradiction. Suppose adversary $\mathcal{A}$ violates Non-Separability (6.1) for n checkers for some $I' \subseteq I$ with probability greater than $\binom{n}{2}/\binom{2d}{d}$. We construct $\mathcal{B}$ that violates Non-Separability (6.1) for Idealized Random Keys with 2 checkers with probability greater than $\binom{2d}{d}^{-1}$, which contradicts Lemma 3.

$\mathcal{B}$ is given $\boldsymbol{k}$ as well as Check^{IR} oracles for its two checkers and proceeds to construct keys for $\mathcal{A}$ and simulate $\mathcal{A}$'s oracles:

1. $\mathcal{B}$ calls $\mathsf{Gen}^{IR}(1^d, 1^n)$ to get $\boldsymbol{k}'$ and $\{\boldsymbol{K}_i\}_{i \in I}$.
2. $\mathcal{B}$ then chooses j and j' uniformly at random from $I - I'$ and forms $\boldsymbol{k}'' = (\boldsymbol{k}' - (\boldsymbol{K}_j \cup \boldsymbol{K}_{j'})) \cup \boldsymbol{k}$. This replaces the keys for j and j' in $\boldsymbol{k}'$ with the keys in $\boldsymbol{k}$. $\mathcal{B}$ assigns one of its Check^{IR} oracles to j and the other to j'.
3. $\mathcal{B}$ calls $\mathcal{A}$ with 1^d, 1^n, $\boldsymbol{k}''$, and $\{\boldsymbol{K}_i\}_{i \in I'}$ and simulates $\mathcal{A}$'s ownership queries as follows:
   - if $i \neq j$ and $i \neq j'$, then $\mathcal{B}$ has all the keys for $\boldsymbol{K}_i$, so $\mathcal{B}$ can emulate exactly the execution of $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_i)$, including keeping state.
   - If $i = j$ or $i = j'$, then $\mathcal{B}$ forwards the ownership query on to $\mathcal{B}$'s oracle for i.
4. When $\mathcal{A}$ returns K and K', $\mathcal{B}$ returns K and K'.

If $\mathcal{A}$ succeeds, then there is some pair i, $i' \in I - I'$ such that $\boldsymbol{K}_i \subseteq K$ and $\boldsymbol{K}_{i'} \subseteq K'$ and $K \cap K' = \emptyset$. Since j and j' were chosen uniformly at random, the probability that the unordered pair (i, i') is the same as the pair (j, j') is $\binom{n}{2}^{-1}$, so $\mathcal{B}$ succeeds with probability greater than $\frac{\binom{n}{2}/\binom{2d}{d}}{\binom{n}{2}} = \binom{2d}{d}^{-1}$. This contradicts Lemma 3.

So, contrary to the initial assumption, $\mathcal{A}$ must only be able to succeed with probability less than or equal to $\binom{n}{2}/\binom{2d}{d}$. □

Theorem 4 provides a way to determine the value of d for a given choice of probability ϵ_0 of a compromised requester violating Non-Separability (6.1) for n verifiers. Theorem 4 implies that $\epsilon_0 < \binom{n}{2}/\binom{2d}{d}$. Since $\binom{2d}{d} \geq 2^d$ for $d \geq 0$, this can be simplified to $\epsilon_0 < \binom{n}{2}/2^d$. Thus, it suffices to set d to $O\left(\log\left(\frac{n^2}{\epsilon_0}\right)\right)$.

6.1.3 Transferability of Atomic Signatures

Idealized Random Keys provides a framework for proving that Atomic Signatures satisfies Transferability (5.5). We prove this in the form of a reduction, showing that Non-Separability (6.1) of Idealized Random Keys implies Transferability (5.5) of Atomic Signatures. Since Theorem 4 shows that Non-Separability (6.1) holds for Idealized Random Keys, it then follows that Transferability (5.5) holds for Atomic Signatures.

Lemma 5. *If Idealized Random Keys satisfies Non-Separability (6.1), then Atomic Signatures satisfies Transferability (5.5).*

Proof. We prove the contrapositive by constructing an adversary $\mathcal{B}$ that violates Non-Separability (6.1) of Idealized Random Keys using an adversary $\mathcal{A}$ that violates Transferability (5.5) of Atomic Signatures. $\mathcal{B}$ is given keys $\boldsymbol{k}$ for Idealized Random Keys.

For each key $k \in \boldsymbol{k}$, $\mathcal{B}$ generates random values $\boldsymbol{z} = \langle z_1, z_2, \ldots, z_{dn} \rangle$ to construct a set of keys to pass to $\mathcal{A}$ for Atomic Signatures. $\mathcal{B}$ will respond to signing and verification oracle queries for $\mathcal{A}$; $\mathcal{B}$ knows signing keys $\boldsymbol{k}$ but does not know which correct verifiers own which keys. Answering signing queries is easy for $\mathcal{B}$, since no knowledge of key ownership is required—$\mathcal{B}$ simply uses all signing keys to generate a signature.

For verification queries on a message m and tag τ, $\mathcal{B}$ knows the signing keys, so $\mathcal{B}$ can check to see if $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ holds for each $(k, \boldsymbol{z}) \in \boldsymbol{k}$. If $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ holds for all $(k, \boldsymbol{z}) \in \boldsymbol{k}$, then $\mathcal{B}$ can return ∞; and if no instance of $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ holds for any $(k, \boldsymbol{z}) \in \boldsymbol{k}$, then $\mathcal{B}$ can return 0.

But if some instances hold and others do not, then $\mathcal{B}$ does not know what to return for a given verification query to verifier j, since $\mathcal{B}$ does not know which instances use keys in $\boldsymbol{K}_j$. One way to solve this problem would be for $\mathcal{B}$ to make ownership queries to Check^{IR} for keys for some instances—knowing which verifiers own which keys for more keys gives $\mathcal{B}$ a higher probability of answering correctly. But if $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ returns $\perp$ on any such ownership query when $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j)$ would not have returned $\perp$, then s_j gets set to $\perp$, and v_j is not set to $\perp$ in Atomic Signatures. In this case, $\mathcal{B}$ might not be able to violate Non-Separability (6.1) using the message and tag that $\mathcal{A}$ returns to violate Transferability (5.5); for instance, $\mathcal{A}$ might return a message and tag that violate Transferability (5.5) for j and some other verifier. And such a j could not be used to violate Non-Separability (6.1), since $s_j = \perp$ would hold. So, $\mathcal{B}$ needs to ensure that if an ownership query to $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ causes $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ to return $\perp$ in the course of simulating a call to $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j)$, then $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j)$ would also have returned $\perp$.

To gain more information about the values returned by Ver^{AS}, $\mathcal{B}$ proceeds as

follows: $\mathcal{B}$ randomly generates n additional *known key pairs* $(k_1^*, \boldsymbol{z}_1^*)$, $(k_2^*, \boldsymbol{z}_2^*)$, ..., $(k_n^*, \boldsymbol{z}_n^*)$, and assigns (k_j^*, z_j^*) to verifier j for $1 \leq j \leq n$. $\mathcal{B}$ adds these keys to $\boldsymbol{k}$, creating $\boldsymbol{k}'$, and creates n state variables s_1, s_2, ..., s_n, initializing each to 1. So, $\mathcal{B}$ will use $\mathcal{A}$ on security parameter $d+1$ to violate Non-Separability (6.1) on security parameter d.[3]

1. $\mathcal{B}$ calls $A(1^{d+1}, 1^n, \boldsymbol{k}', \{\boldsymbol{K}_i\}_{i \in I'})$ and answers $\mathcal{A}$'s queries $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j)$ as follows:

   (a) *Initialization.* Set S^0 and S^1 to $\emptyset$.

   (b) *Check s_j.* If $s_j = \perp$, then return $\perp$.

   (c) *Key Discovery.* For each key pair $(k, \boldsymbol{z})$ in $\boldsymbol{k}$, if $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ holds, then add k to set S^1; otherwise add k to set S^0.

   (d) *Check Opposite Keys.* If $\mathsf{roweq}(m, \tau, (k_j^*, \boldsymbol{z}_j^*))$ holds, then iterate over each key k in S^0. Otherwise iterate over keys in S^1. Use oracle access to call $\mathsf{Check}^{IR}(k, \boldsymbol{K}_j)$ on each such key k, and return $\perp$ if $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ ever returns $\perp$. Set s_j to $\perp$ when returning $\perp$.

   (e) If $\mathsf{roweq}(m, \tau, (k_j^*, \boldsymbol{z}_j^*))$ holds, then return ∞. Otherwise, return 0.

2. When $\mathcal{A}$ returns m and τ, run Key Discovery as before to get S^0 and S^1. Return $K = S^1$ and $K' = S^0$.

It remains to show that this algorithm correctly simulates the operation of $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j)$. There are only three possible return values from the verification oracle $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j)$: 0, ∞, and $\perp$. We consider each case in turn:

- $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j) = 0$. In this case, the definition of Ver^{AS} states that, for all key pairs $(k, \boldsymbol{z})$ in $\boldsymbol{K}_j$, $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ will not hold. So, all of j's keys

[3]The security parameter is now $d+1$ instead of d, because there are now $dn + n = (d+1)n$ keys.

will be placed in S^0. Further, $\mathsf{roweq}(m, \tau, (k_j^*, \boldsymbol{z}_j^*))$ will also fail to hold, so Check Opposite Keys will iterate over S^1. Thus, none of j's keys will be passed to $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$, so $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ will not return $\perp$, which means the simulation will not return $\perp$. Since $\mathsf{roweq}(m, \tau, (k_j^*, \boldsymbol{z}_j^*))$ does not hold, the simulation returns 0, as required.

- $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j) = \infty$. In this case, the definition of Ver^{AS} states that, for all key pairs $(k, \boldsymbol{z})$ in $\boldsymbol{K}_j$, $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ holds. So, all of j's keys will be placed in S^1. Further, $\mathsf{roweq}(m, \tau, (k_j^*, \boldsymbol{z}_j^*))$ must hold, so Check Opposite Keys will iterate over S^0. Thus, none of j's keys will be passed to $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$, so $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ will not return $\perp$, which means that the simulation will not return $\perp$. Since $\mathsf{roweq}(m, \tau, (k_j^*, \boldsymbol{z}_j^*))$ holds for all $(k, \boldsymbol{z}) \in \boldsymbol{K}_j$, the simulation returns ∞, as required.

- $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j) = \perp$. The definition of Ver^{AS} provides two cases in which a verifier might return $\perp$.

  First, it might be that $v_j \neq \perp$. In this case, the definition of Ver^{AS} states that there must be some key pairs $(k_r, \boldsymbol{z}_r)$ and $(k_{r'}, \boldsymbol{z}_{r'})$ in $\boldsymbol{K}_j$ such that $\mathsf{roweq}(m, \tau, (k_r, \boldsymbol{z}_r))$ holds and $\mathsf{roweq}(m, \tau, (k_{r'}, \boldsymbol{z}_{r'}))$ does not hold. So, key pair $(k_r, \boldsymbol{z}_r)$ is put in S^1, and key pair $(k_{r'}, \boldsymbol{z}_{r'})$ is put in S^0.

  If $\mathsf{roweq}(m, \tau, (k_j^*, \boldsymbol{z}_j^*))$ holds, then Check Opposite Keys will iterate over S^0, which contains $k_{r'}$, so $\mathsf{Check}^{IR}(k_{r'}, \boldsymbol{K}_j)$ will be called and will return $\perp$. If not, then Check Opposite Keys will iterate over S^1, which contains k_r, so $\mathsf{Check}^{IR}(k_r, \boldsymbol{K}_j)$ will be called and will return $\perp$. Either way, the simulation returns $\perp$, as required. And in both cases, the simulation sets s_j to $\perp$.

  Second, it might be that $v_j = \perp$. By definition of Ver^{AS}, this means that $\mathsf{Ver}^{AS}(\cdot, \cdot, \boldsymbol{K}_j)$ must have previously returned $\perp$. This means that s_j has

already been set to $\perp$, by induction. So, the simulation returns $\perp$ in the step *Check* s_j, as required.

Thus, $\mathcal{B}$ simulates $\mathcal{A}$'s oracle calls correctly. When $\mathcal{A}$ succeeds, the definition of Transferability (5.5) implies that there is some pair $j, j' \in I - I'$ such that the values of m and τ returned by $\mathcal{A}$ satisfy $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j) = \infty$ and $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_{j'}) = 0$. Therefore, $\mathsf{roweq}(m, \tau, (k, \boldsymbol{z}))$ must hold for every $(k, \boldsymbol{z}) \in K_j$ and for no $(k, \boldsymbol{z}) \in K_{j'}$, which means that $\boldsymbol{K}_j \subseteq S^1 = K$ and $\boldsymbol{K}_{j'} \subseteq S^0 = K'$, as required. And $K \cap K' = \emptyset$ holds by definition. Values s_j and $s_{j'}$ are not set to $\perp$, since $\mathsf{Ver}^{AS}(\cdot, \cdot, \boldsymbol{K}_j)$ and $\mathsf{Ver}^{AS}(\cdot, \cdot, \boldsymbol{K}_{j'})$ never returned $\perp$. $\mathcal{B}$ succeeds with the same non-negligible probability as $\mathcal{A}$.

□

Note that the constructed adversary $\mathcal{B}$ in the proof of Lemma 5 needs the known key pairs to simulate the operation of the verification function. But known key pairs are not needed in the construction itself. This difference in the use of known keys leads to the difference in security parameters between $\mathcal{B}$ and $\mathcal{A}$.

The following theorem uses the previous results to show that Atomic Signatures is a Strong ∞-MVS scheme.

Theorem 6. *If the* MAC *is a pseudorandom function, then Atomic Signatures is a Strong ∞-MVS scheme.*

Proof. Lemma 2 shows that Atomic Signatures satisfies ∞-Completeness (5.2) and Strong Unforgeability (5.6) if the MAC is a pseudorandom function. And Lemma 5 and Theorem 4 together imply that Atomic Signatures satisfies Transferability (5.5). So, Atomic Signatures is a Strong ∞-MVS scheme. □

The reduction in Lemma 5 adds one key per verifier to the set of keys used in Atomic Signatures. So, the value $d = O\left(\log\left(\frac{n^2}{\epsilon_0}\right)\right)$ computed in section 6.1.2 using the probability ϵ_0 of a compromised signer being able to create a split tag gives a value of d that is 1 lower than the value needed for Atomic Signatures. But this does not affect the asymptotic complexity of the scheme: Atomic Signatures can be computed by generating a vector of dn MACs in time $O(|m|dn)$ and solving the factored matrix equation in time $O(n^2d^2) = O\left(n^2\log^2\left(\frac{n^2}{\epsilon_0}\right)\right)$. So, the total asymptotic complexity of generating a tag is $O\left(|m|n\log(\frac{n^2}{\epsilon_0}) + n^2\log^2\left(\frac{n^2}{\epsilon_0}\right)\right)$.

6.2 Chain Signatures

Chain Signatures is a λ-MVS scheme that creates tags consisting of vectors of *subtags*. Each subtag contains the output of a MAC on the concatenation of previous subtags.

Key generation algorithm $\mathsf{Gen}^{CS}(1^d, 1^n)$ produces n *known keys* $\boldsymbol{k}_1 = k_1^*, k_2^*, \ldots, k_n^*$ and dn *unknown keys* $\boldsymbol{k}_2 = \langle k_1, k_2, \ldots, k_{dn}\rangle$, where each key is an element of $\{0,1\}^b$. $\mathsf{Gen}^{CS}(1^d, 1^n)$ then sets $\boldsymbol{k} = (\boldsymbol{k}_1, \boldsymbol{k}_2)$ and creates n vectors $\boldsymbol{K}_1, \boldsymbol{K}_2, \ldots, \boldsymbol{K}_n$. A vector $\boldsymbol{K}_j$ contains k_j^* as well as a set of d keys chosen uniformly at random from $\boldsymbol{k}_2$ such that vectors $\boldsymbol{K}_1, \boldsymbol{K}_2, \ldots, \boldsymbol{K}_n$ are disjoint.

$\mathsf{Sign}^{CS}(m, \lambda, \boldsymbol{k})$ produces[4] a vector $\boldsymbol{C}^{\lambda,d}(m)$ consisting of λ *sections*, each divided into two *components*: we call component 1 the *known-key* component and component 2 the *unknown-key* component. Component 1 contains n subtags, and component 2 contains dn subtags, so each section contains $(d+1)n$ subtags. We write $\boldsymbol{C}^{\lambda,d}(m)[r, c, j]$ for the jth subtag in the cth component of the

[4]The definition of MVS schemes does not allow signing algorithm $\mathsf{Sign}(\cdot, \boldsymbol{k})^{CS}$ to take three parameters. To avoid this difficulty when Chain Signatures is considered as a λ-MVS scheme for some $\lambda \in \mathbb{N}$, we define $\mathsf{Sign}^{CS}(m, \boldsymbol{k})$ to mean $\mathsf{Sign}^{CS}(m, \lambda, \boldsymbol{k})$.

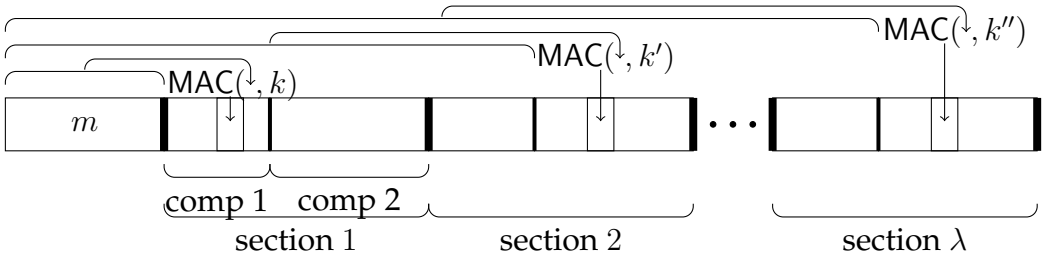

Figure 6.1: The structure of a signed message using Chain Signatures

rth section of the tag generated by Chain Signatures for m and λ.[5] We use the natural lexicographic ordering on triples (r, c, j) to index the subtags of Chain Signatures.[6] The value of a subtag is computed recursively as the MAC of the concatenation of m with the subtags in all previous components:

$$\boldsymbol{C}^{\lambda,d}(m)[r,c,j] \triangleq \mathsf{MAC}(m \underset{(t,t',t'')<(r,c,1)}{||} \boldsymbol{C}^{\lambda,d}(m)[t,t',t''], \boldsymbol{k}_c[j]) \tag{6.3}$$

Subtag j in component c of section r is said to be *supported* if the value of this subtag is identical to the MAC of the message and all previous components under key $\boldsymbol{k}_c[j]$. Figure 6.1 shows the structure of a signed message using Chain Signatures.

Verification algorithm $\mathsf{Ver}^{CS}(\cdot,\cdot,\boldsymbol{K}_j)$ keeps state v_j to record whether it has ever been called with a message and tag that indicate that the signer is compromised. When called with a message m and tag τ, the verification algorithm checks all the subtags of j in τ to see if there is a supported subtag that follows a non-supported subtag. If so, then verification concludes that the signer is compromised. And if not, then verification returns the value of the highest section in which it found a supported subtag.

More precisely, the verification algorithm works as follows, where λ' is the

[5]Note that we can compute the offset of $\boldsymbol{C}^{\lambda,d}(m)[r,c,j]$ in this tag as $(d+1)n(r-1)+(c-1)n+j-1$.

[6]Note that sections, components, and subtags indexed by our triples are numbered starting at 1 rather than the more customary value of 0.

value of the highest section in which verification found a supported subtag for j:

- If $v_j = \perp$, then return $\perp$.
- Otherwise, if each known-key component below c contains exactly one supported subtag for j, and each unknown-key component below c contains d supported subtags for j, then return λ'.
- If no component contains supported subtags for j, then return 0.
- Otherwise, some component below c contains a non-supported subtag for j, so return $\perp$ and set v_j to $\perp$.

The alternation of known-key and unknown-key components in tags generated by Chain Signatures is critical to the security of the algorithm. If a supported subtag t in one component follows a non-supported subtag t' in another component, then there must be some pair of adjacent components c and c' such that c' comes before c, there is a non-supported subtag in c', and there is a supported subtag in c. Since known-key and unknown-key components alternate, exactly one of c and c' must be a known-key component. So, whenever verification returns $\perp$, there is a known-key component and an adjacent unknown-key component that justify this return value.

A simpler—but wrong—version of Chain Signatures would not include any known-key components. This would give a compromised signer an easy way to violate Transferability (5.5). For example, suppose compromised signer i creates a tag τ in which all subtags in the first $\lambda - 1$ sections are supported. However, in section λ of tag τ, only one subtag is supported. The key for this subtag is owned by some verifier, say j. So, verification at j will return λ, and all other verifiers will return $\lambda - 1$. This reveals that j owns the key for this subtag. And

i can perform this attack on each key to learn its attribution; i can create split tags once it knows the attribution of enough keys.

This attack fails in Chain Signatures due to the known-key components. Suppose that compromised signer i creates a tag τ in which all subtags in the first $\lambda - 1$ sections are supported. The first component in section λ is a known-key component. So, if i makes one of the subtags in the known-key component supported and all other subtags in section λ non-supported, then j will return λ, and all other verifiers will return $\lambda - 1$. This reveals the attribution of j's known key. But the attribution of j's known key is known, so the adversary does not learn anything new about the keys.

Another variant on the same attack would be for the compromised signer to make all the subtags in the known-key component supported. But then, no matter which subtags are supported in the unknown-key component of section λ, all verifiers will return λ, since all find a supported subtag in this section.

A more complex variant of this attack would be to make some subtags in the known-key component supported and some non-supported. Also, at least one subtag in the unknown-key component of section λ must be made supported (otherwise, the adversary learns nothing, as discussed above). But now there is some probability that there is a verifier j' that has a non-supported tag in the known-key component and a supported tag in the unknown-key component. This verifier will return $\perp$ on τ. So, if the adversary requests verification of τ from a verifier j', and j' does not return $\perp$, then the adversary learns that j' does not own the key used for the subtag in the unknown-keys.

Attacks in which some verifiers might return $\perp$ yield information to the adversary, but they also risk revealing to verifiers that the signer is compromised. If too many verifiers learn that the signer is compromised, then Transferabil-

ity (5.5) cannot be violated, since violations m and τ depend on verifiers j and j' such that $\mathsf{Ver}(m, \tau, \boldsymbol{K}_j) \neq \perp$ and $\mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'}) \neq \perp$. Our reduction below from Idealized Random Keys shows how to choose a value of d such that, with high probability, compromised signers never learn enough information to create split tags without revealing themselves as compromised to too many verifiers.

Chain Signatures as described above is expensive to compute; generating a tag costs $O(dn\lambda(|m| + dn\lambda))$, since inputs to the MAC grow linearly in the length of the tag. Figure E.2 in Appendix E gives an algorithm that reduces the cost to $O(|m| + dn\lambda \log \lambda)$ using collision-resistant hash functions. Modified proofs of λ-Completeness (5.2), Unforgeability (5.3), and Non-Accusability (5.4) follow the description of this more efficient version.

A λ-MVS scheme must satisfy λ-Completeness (5.2), Unforgeability (5.3), Non-Accusability (5.4), and Transferability (5.5). We prove that Chain Signatures is a λ-MVS scheme for $\lambda \in \mathbb{N}_{>0}$ using two lemmas. The first shows that Chain Signatures satisfies λ-Completeness (5.2), Unforgeability (5.3), and Non-Accusability (5.4); the second reduces Non-Separability (6.1) of Ideal Random Keys to Transferability (5.5) of Chain Signatures. Then, Theorem 4, along with the second lemma, implies that Chain Signatures satisfies Transferability (5.5).

Note that Chain Signatures does not satisfy Strong Unforgeability (5.6). Any adversary that receives a tag τ for message m that causes verifier j to return $\lambda > 1$ can produce a new tag τ' that causes verifier j to return $\lambda - 1$. All the adversary needs to do is to remove the last section, since tags for earlier sections do not depend on the last section. The previous $\lambda - 1$ sections consist entirely of supported subtags, so verifier j will return $\lambda - 1$ for m and τ'. But the adversary never received τ' from its signing oracle, so m and τ' together violate Strong Unforgeability (5.6).

Lemma 7. *For any $\lambda \in \mathbb{N}_{>0}$, if the* MAC *satisfies CTA Unforgeability (5.1), then Chain Signatures satisfies λ-Completeness (5.2), Unforgeability (5.3), and Non-Accusability (5.4).*

Proof. **λ-Completeness.** This follows directly from the definition: all subtags are supported by construction, so $\mathsf{Ver}^{CS}(m, \mathsf{Sign}^{CS}(m, \lambda, \boldsymbol{k}), \boldsymbol{K}_j) = \lambda$.

Unforgeability. We prove the contrapositive. Suppose that adversary $\mathcal{A}$ violates Unforgeability (5.3) for some $I' \subseteq I$ with probability ϵ_0. We construct an adversary $\mathcal{B}$ that violates CTA Unforgeability (5.1) of the MAC (for some key k') with probability ϵ_0/n. $\mathcal{B}$ chooses a key k_t^* uniformly at random from the n known keys and generates a new instance of Chain Signatures by calling Gen^{CS} and replacing calls to $\mathsf{MAC}(\cdot, k_t^*)$ with (i) calls to $\mathcal{B}$'s MAC oracle when signing and (ii) $\mathcal{B}$'s verification oracle when verifying. When $\mathcal{A}$ succeeds, returning m and τ, the definition of Unforgeability (5.3) states that there is some $j \in I - I'$ for which $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j) > 0$. This means j must have (at least) a supported subtag in component 1 of section 1.

$\mathcal{B}$ returns m as its message and $\tau[1, 1, t]$ as its tag. With probability $1/n$, we have $t = j$, since t was chosen uniformly at random and independently of j. And $\mathsf{MAC}(m, k') = \tau[1, 1, t]$, because $\tau[1, 1, t] = \tau[1, 1, j]$ is the only subtag for j in component 1 of section 1, so it must be supported. The unique length of inputs to MACs for each component implies the only component for which $\mathcal{B}$ could have requested m from its MAC oracle is the very first. This request could only have been made if $\mathcal{A}$ requested m from its signing oracle, which does not occur by definition.

So, $\mathcal{B}$ never requested m from its MAC oracle, and $\mathcal{B}$ succeeds in violating CTA Unforgeability (5.1) with probability ϵ_0/n.

Non-Accusability. We prove the contrapositive. Suppose some adversary $\mathcal{A}$

violates Non-Accusability (5.4) for some $I' \subseteq I$ with probability ϵ_0. Similar to the proof of Unforgeability (5.3), we construct a $\mathcal{B}$ that violates CTA Unforgeability (5.1) of the MAC by building a new instance of Chain Signatures and calling $\mathcal{A}$. Instead of choosing a key at random from the known keys, however, $\mathcal{B}$ chooses a key k_t from the union of the known keys and the unknown keys. When $\mathcal{A}$ succeeds and returns m and τ, the definition of Non-Accusability (5.4) states that there must be some j in $I - I'$ such that $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$ returns $\perp$, which means that there is some supported subtag for j that takes as input a non-supported subtag for j in some component r.

With probability $1/((d+1)n)$, key k_t was used to compute this supported subtag, since t was chosen uniformly at random and independently of the choice of the non-supported subtag. In this case, $\mathcal{B}$ returns this supported subtag in component r as tag τ' and the message m concatenated with all components before component r as message m'. The length of m' means that it could only have been input to $\mathcal{B}$'s MAC oracle in component r. But since it contains a non-supported subtag, it never would have been input to a MAC in $\mathcal{B}$'s simulation of the signing oracle, since only concatenations of supported subtags are input to the MAC oracle in $\mathcal{B}$'s simulation, by construction. So, m' has never been requested from $\mathcal{B}$'s MAC oracle.

Thus, m' and τ' violate CTA Unforgeability (5.1) of the MAC with probability $\epsilon_0/((d+1)n)$, which is non-negligible. □

To prove that Chain Signatures satisfies Transferability (5.5), we reduce from Idealized Random Keys. This reduction relies on the following characterization of tags: for each verifier j, message m, and tag pair τ, there is a highest known-key component of τ containing a supported subtag; we call this known-key component $\mathsf{highSup}(m, \tau, j, \boldsymbol{k})$. And there is a lowest known-key component

containing a non-supported subtag for j; we call this known-key component $\mathsf{lowNonSup}(m, \tau, j, \boldsymbol{k})$. Verification returns $\perp$ when a supported subtag follows a non-supported subtag for a given verifier. As argued in section 6.2, at least one such pair of supported and non-supported subtags in this case always involves a known-key component. The following lemma shows that these special known-key components indicate when to return $\perp$.

Lemma 8. *For verifier j, if the state v_j is not $\perp$, then $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$ returns $\perp$ if and only if there is at least one of the two following kinds of subtags in τ:*

- *a non-supported subtag for j in a component below $\mathsf{highSup}(m, \tau, j, \boldsymbol{k})$*
- *a supported subtag for j in a component above $\mathsf{lowNonSup}(m, \tau, j, \boldsymbol{k})$*

Proof. The "if" direction trivially follows from the definition of Ver^{CS}: if a supported subtag for verifier j, message m, and tag τ follows a non-supported subtag for j, then $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$ returns $\perp$.

We prove the "only if" direction by the contrapositive. Let λ^{HS} be the value $\mathsf{highSup}(m, \tau, j, \boldsymbol{k})$, and let λ^{LN} be the value $\mathsf{lowNonSup}(m, \tau, j, \boldsymbol{k})$. Suppose that all non-supported subtags for j are in λ^{HS} or higher components—in fact, all non-supported tags for j must be in higher components, since there is only one subtag for j in λ^{HS}, and this subtag is supported. Suppose further that all supported tags for j are in λ^{LN} or lower components—by the same argument as for λ^{HS}, all supported subtags for j must actually be in lower components than λ^{LN}.

We will show that $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$ cannot return $\perp$. For the sake of contradiction, suppose that it does. Then, since v_j is not $\perp$, the definition of Ver^{CS} states that there is a pair of keys k_1 and k_2 associated with j and components r^N and r^S such that $r^N < r^S$ holds, the subtag generated with k_1 in component r^N is not supported, and the subtag generated with k_2 in component r^S is

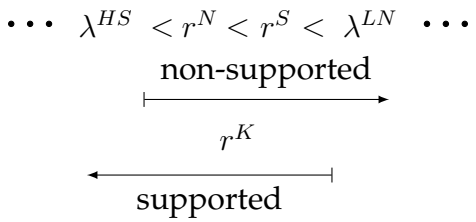

Figure 6.2: Components used in the proof of Lemma 8

supported. This happens because the verification algorithm returns $\perp$ only if a non-supported subtag occurs in a lower component than a supported subtag.

By the argument above, $r^N > \lambda^{HS}$ and $r^S < \lambda^{LN}$ both hold. So, $\lambda^{HS} < r^N < r^S < \lambda^{LN}$ holds; see Figure 8 for a depiction of these components. This means that there are at least two distinct components r^N and r^S between λ^{HS} and λ^{LN}, so one of the components between λ^{HS} and λ^{LN}, say r^K, must be a known-key component. Since $r^K < \lambda^{LN}$ holds, the definition of λ^{LN} requires that j's subtag in r^K be supported. But, since $r^K > \lambda^{HS}$ holds, the definition of λ^{HS} requires that j's subtag in r^K not be supported. This is a contradiction, since r^K only has one subtag for j, so $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$ cannot return $\perp$. $\square$

It may seem like a more natural construction for Chain Signatures would make the input to the MAC for each subtag be the concatenation of the message and all previous *subtags*, instead of the message and all subtags in previous *components*. But Lemma 8 no longer holds in this version of Chain Signatures. The problem is that a non-supported subtag for some verifier j could be followed by a supported subtag for j in the same unknown-key component. In this case, there is no contradiction in the proof of Lemma 8. This means that the highest known-key component with a supported subtag and the lowest known-key component with a non-supported subtag are not sufficient to simulate verification in the simpler version of Chain Signatures; Lemma 8 is critical in the reduction from Non-Separability (6.1) of Idealized Random Keys to Transfer-

ability (5.5) of Chain Signatures.

We now proceed to show the following lemma.

Lemma 9. *If Idealized Random Keys satisfies Non-Separability (6.1), then Chain Signatures satisfies Transferability (5.5).*

Proof. We prove the contrapositive: we construct an adversary $\mathcal{B}$ that violates Non-Separability (6.1) of Idealized Random Keys using an adversary $\mathcal{A}$ that violates Transferability (5.5) of Chain Signatures. The reduction for Transferability (5.5) of Atomic Signatures relies on known keys that are added for the proof. For Chain Signatures, however, known keys are already part of the construction, so they do not need to be added for the proof.

$\mathcal{B}$ is given $\boldsymbol{k}$ for Idealized Random Keys and generates n keys to serve as the known keys for Chain Signatures. $\mathcal{B}$ then forms $\boldsymbol{k}'$ consisting of $\boldsymbol{k}$ and these known keys to pass to $\mathcal{A}$.

Since $\mathcal{B}$ knows all the keys, $\mathcal{B}$ can check each subtag in each component to see if it is supported. $\mathcal{B}$ divides the keys into two sets for each component c based on whether or not the MAC using a key in c is supported; we call these sets *supported* and *non-supported*, respectively. $\mathcal{B}$ then uses the known keys to decide on which sets to call $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ to simulate a given call to $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$, as follows.

1. $\mathcal{B}$ finds $\mathsf{lowNonSup}(m, \tau, j, \boldsymbol{k})$ and calls $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ on all keys in all supported sets for higher components.
2. Similarly, $\mathcal{B}$ finds $\lambda = \mathsf{highSup}(m, \tau, j, \boldsymbol{k})$ and calls $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ on all keys in all non-supported sets for lower components.
3. If any of the calls to $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ return $\perp$, then $\mathcal{B}$ returns $\perp$,
4. Otherwise, $\mathcal{B}$ returns λ, since verification for j returns the value of the highest section that contains a supported subtag for j.

This strategy simulates $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$ perfectly. To see why, we consider the possible return values of $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$. When $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$ returns 0, there are no supported subtags for j, so the simulation will also return 0.

When $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$ returns $\lambda > 0$, there must be some supported subtag for j in section λ, and no supported subtags in higher sections. And since verification did not return $\perp$, all subtags for j in lower components must also be supported. Since the known-key subtag for j is the first subtag for j in section λ, it must also be supported; this means that the simulation will return λ.

By Lemma 8, the simulation is also correct when $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$ returns $\perp$. Note that Lemma 8 implies that $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ in $\mathcal{B}$'s simulation will only return $\perp$ when $\mathsf{Ver}^{CS}(\cdot, \cdot, \boldsymbol{K}_j)$ does, since the simulation calls $\mathsf{Check}^{IR}(\cdot, \boldsymbol{K}_j)$ only on keys for subtags that match the description in the hypothesis of Lemma 8.

When $\mathcal{A}$ succeeds, returning a message m and tag τ, the definition of Transferability (5.5) implies that there is some pair $j, j' \in I - I'$ and a section λ' such that $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j) = \lambda'$ and $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_{j'}) < \lambda' - 1$. $\mathcal{B}$ finds j, j', and λ' by simulating the verification function as before for each verifier. Then $\mathcal{B}$ returns the supported subtags in the unknown-key component of section $\lambda' - 1$ as K and the non-supported subtags in the unknown-key component of section $\lambda' - 1$ as K'. This strategy always succeeds, since a violation of Transferability (5.5) in section λ' for verifiers j and j' means that the subtags for j must be supported in the unknown-key component of section $\lambda' - 1$ and the subtags for j' must not be supported. Thus, $\mathcal{B}$ succeeds with the same probability as $\mathcal{A}$. □

Just as in the proof of Lemma 5 for Atomic Signatures, the adversary $\mathcal{B}$ in the proof of Lemma 9 depends critically on the known keys in its simulation of the verification oracle. But, unlike Atomic Signatures, the known keys are essential to the construction of Chain Signatures, as shown in Lemma 8 and section 6.2.

The following theorem uses the previous results to show that Chain Signatures is a λ-MVS scheme.

Theorem 10. *For any $\lambda \in \mathbb{N}_{>0}$, if the* MAC *satisfies CTA Unforgeability (5.1), then Chain Signatures is a λ-MVS scheme.*

Proof. Lemma 7 shows that Chain Signatures satisfies λ-Completeness (5.2), Unforgeability (5.3), and Non-Accusability (5.4) if the MAC satisfies CTA Unforgeability (5.1). Lemma 9 shows that Non-Separability (6.1) of Idealized Random Keys implies Transferability (5.5) of Chain Signatures. Since Theorem 4 shows that Non-Separability (6.1) of Idealized Random Keys holds, it follows that Chain Signatures satisfies Transferability (5.5). □

The reduction in Lemma 9 shows that the value of the security parameter d is set to $d+1$ in order for Chain Signatures to have the same security as Idealized Random Keys has for d. However, the asymptotic complexity of d is the same, so the value $d = O\left(\log\left(\frac{n^2}{\epsilon_0}\right)\right)$ computed in section 6.1.2 using the probability ϵ_0 of a compromised signer being able to create a split tag is the same for Chain Signatures as for Idealized Random Keys. This means that Chain Signatures can be computed in time $O(|m| + dn\lambda \log \lambda) = O\left(|m| + n\lambda \log\left(\frac{n^2}{\epsilon_0}\right) \log \lambda\right)$.

6.3 Performance

We implemented Atomic Signatures (AS) and Chain Signatures (CS) in C using OpenSSL 0.9.8e [58]. Using a hash function h, we compute a MAC for a message m and key k by setting $\mathsf{MAC}(m, k) = h(h(m)||k)$, as suggested by Canetti et al. [17] for cases where many MACs must be computed for the same message. In our implementation, h is SHA-1 [70].[7] All shared keys comprise 160

[7] Under the assumption that SHA-1 is pseudorandom, this MAC satisfies the properties required for our proof, according to Bellare et al. [8].

bits, and the output of the MAC is also 160 bits, so parameter $b = 160$.[8] We use all optimizations described in the body of the dissertation and the appendices: pseudorandom functions are used to generate a factored matrix for Atomic Signatures, and hashing is used as in the pseudo-code of Figure E.1 in Appendix E to reduce the running time of Chain Signatures. The probability ϵ_0 that a compromised signer will be able to create a split tag is set to 2^{-64}, except where otherwise stated. Parameter λ is considered up to $\lambda = 3$, since this is a common value for protocols used in implementing distributed services.

All tests were run on a 2.13GHz Pentium M over Gentoo Linux kernel 2.6.22-gentoo-r9. RSA and DSA measurements were made for OpenSSL by running the commands `openssl speed rsa` and `openssl speed dsa` on this system. Each value represents a mean over 1000 runs; the error gives the sample standard deviation around this mean.

The performance of signature algorithms depends on three factors: the execution time for generating and checking tags, the tag size, and the key infrastructure required. Figures 6.3 and 6.4 show the execution time for generating and checking Chain Signatures. In Figure 6.3, for $\lambda = 3$, Chain Signatures can generate tags faster than 1024-bit RSA for $n \leq 50$ and faster than 2048-bit RSA for all $n < 100$, which is more than sufficient for many applications. Figure 6.4 shows that checking Chain Signatures (for $\lambda = 3$ and $\epsilon = 2^{-64}$) is faster than 2048-bit RSA for $n < 75$. Higher probabilities of split tags may be acceptable in some contexts and lead to faster generation and checking of signatures.

Atomic Signatures costs $O(d^2n^2)$, so tags that use many random keys are

[8] Atomic Signatures requires that an adversary only be able to violate Strong Unforgeability (5.6) with a given probability ϵ_0'. The proof of Lemma 2 bounds ϵ_0' by $\mathrm{poly}(d, n)/2^b$, but the exact value of the polynomial factor $\mathrm{poly}(d, n)$ depends on Lemmas 12–14, which provide asymptotic, rather than concrete, bounds. So, we instead choose b to satisfy $\epsilon_0' < 1/2^b$, since the polynomial factor will make only a small difference in the choice of b for small values of n and d.

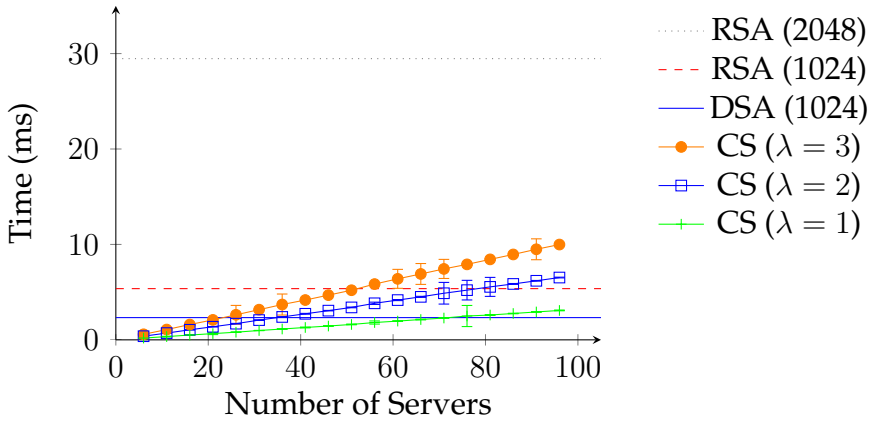

Figure 6.3: Execution time for generating Chain Signatures ($\epsilon = 2^{-64}$)

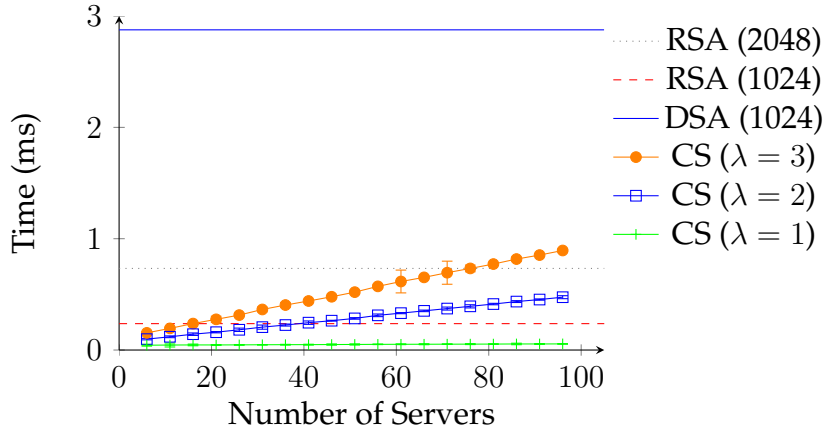

Figure 6.4: Execution time for checking Chain Signatures ($\epsilon = 2^{-64}$)

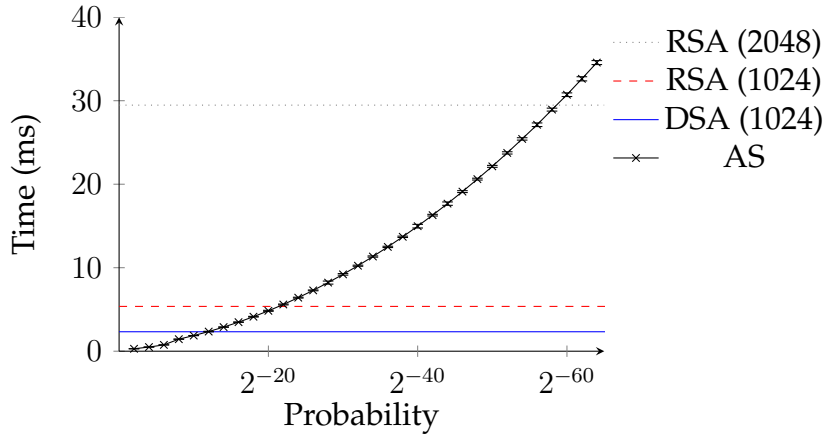

Figure 6.5: Execution time to generate Atomic Signatures for 6 verifiers and different probabilities of generating a split tag

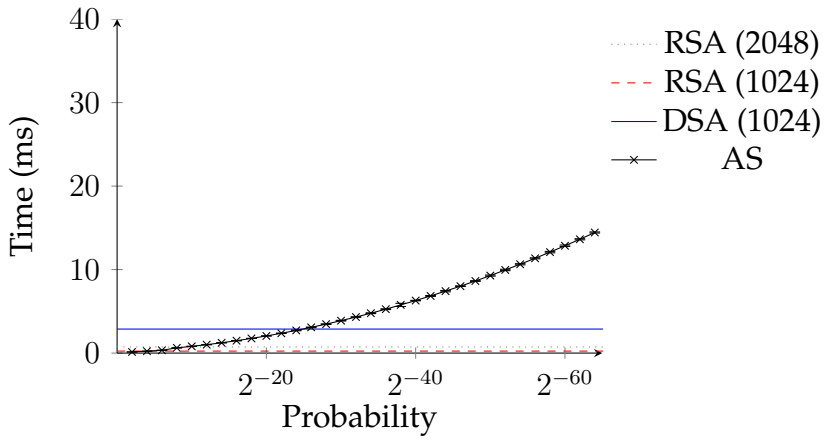

Figure 6.6: Execution time to check Atomic Signatures for 6 verifiers and different probabilities of generating a split tag

more expensive to generate; the efficiency depends on the probability that a signer can generate split tags. Figures 6.5 and 6.6 show how generating and checking times vary for 6 verifiers and different probabilities of creating a split tag. Atomic Signatures can generate tags for 6 verifiers faster than 2048-bit RSA for probabilities down to about 2^{-55}. But checking tags generated by Atomic Signatures is more expensive for probabilities below about 2^{-25}.

Even though execution time for generating and checking tags based on the schemes in this dissertation is sometimes lower than RSA and DSA, tag size for our signature algorithms is significantly larger. Chain Signatures and Atomic Signatures require significant space even for small n, since the size of the signature depends linearly on d. For instance, for 6 verifiers, $\epsilon = 2^{-64}$, and $\lambda = 3$, generating signatures takes about $581\mu s$, which is fast, but the size of a tag is 13680 bytes. These sizes are acceptable in circumstances where signature transfer time is negligible—for instance, between processes in operating systems, or across local-area networks using Gigabit Ethernet switches.

Key-management infrastructure costs for Chain Signatures and Atomic Signatures are also relatively high, since each verifier must store $O(dn)$ keys. For

instance, with $n = 4$ and $\epsilon = 2^{-64}$, each verifier must share $d = 36$ keys with the signer. And if $n = 36$ with the same value of ϵ, then d becomes 40. Rekeying requires that signer i not learn with which verifier it shares a given key. If the keys for a single verifier j were replaced without replacing keys for other verifiers, then i would learn which of its keys correspond to j. Even if keys for some subset of the verifiers were replaced, then signer i would gain some information about which keys correspond to which verifiers. Thus, all keys must be replaced simultaneously.

These performance results show that in some contexts, MVS schemes have comparable, and sometimes even better, performance than digital signature schemes. Unlike these schemes, however, MVS schemes are proven secure only assuming the existence of pseudorandom functions, whereas these digital signature schemes are only known to be secure in the heuristic random oracle model. The results of our experiments show that it is possible to have provable security and efficiency for signature schemes.

6.4 Related Work

Many authentication schemes use symmetric message authentication codes and try to achieve properties similar to digital signature schemes. But none is able to handle an unbounded number of adaptive queries. We succeed by using a unusual secret-key setup along with state kept by verifiers. Previous work achieves different properties.

λ-Limited Transferability. Chaum and Roijakkers [25] were the first to suggest constructing tags that could be transferred a finite number of times. Their scheme allows signed messages to be transferred only once. Pfitzmann and

Waidner [61] followed with a construction, called *pseudosignatures*, that is somewhat similar to Chain Signatures: it creates tags that can be transferred an arbitrary fixed number of times. Both the work of Chaum and Roijakkers and Pfitzmann and Waidner provide unconditional security.

Like MVS schemes, pseudosignatures depend on a secret-key setup; multiple keys are shared between the signer and each verifier, and the signer cannot attribute keys to verifiers. However, pseudosignature tag size is directly proportional to the number of queries an adversary can submit to a verification oracle. Even if pseudosignatures were implemented with computationally-secure MACs, they would only be able to tolerate a fixed number of verification queries.

Arbitrary Transferability with Unconditional Security. Many schemes have been proposed for tags that are both unconditionally secure and can be transferred an arbitrary number of times. For instance, recent work [40, 73, 66] generalizes Multi-Receiver Authentication (MRA) codes (invented by Desmedt et al. [30]) to unconditionally-secure polynomial codes that satisfy similar properties to Transferability. These constructions are called *MRA^3 codes*. MRA^3 codes constrain the number of signing and verification oracle queries as well as the number of possible signatures that a signer can create, since each signature leaks information.

Johansson [46] proposes a different unconditionally-secure authentication scheme; it is similar in form to Atomic Signatures: signers and verifiers each have secret keys that are used to solve a matrix equation. But unlike Atomic Signatures, each signature in Johansson's scheme provides a set of linear equations over the signer's secret keys, so keys must be refreshed after a fixed number of signatures.

Computational Security. Other schemes similar to MVS have been designed for particular protocols in the computational model. For instance, MACs are sometimes considered shared-key signatures, despite not satisfying Transferability. And in some fault-tolerant distributed systems (e.g., Practical Byzantine Fault Tolerance (PBFT) [21]), vectors of MACs are used to improve protocol speed over digital signatures.

Srikanth and Toueg [75] implement a scheme called *authenticated broadcast* that achieves transferable authentication using only MACs. Their solution relies on extra communication between the servers in a distributed system to provide transferability. Each round of an authenticated protocol is transformed into two rounds of communication between the servers; the problem of transferable authentication is solved by replicas providing online authentication of messages.

Authenticated broadcast achieves transferability, hence it has properties similar to multi-verifier signatures. But authenticated broadcast also has two important additional requirements. First, there is a bound on the number of compromised replicas; authenticated broadcast assumes that at most t replicas are compromised. By contrast, the correctness of multi-verifier signatures does not depend on the number of compromised verifiers. Second, authenticated broadcast requires extra communication between the replicas to verify a signature. In some contexts, there is already an assumed bound on the number of compromised replicas, and the load on the system is not so high as to preclude extra rounds of communication for signature verification. In such cases, authenticated broadcast might provide a more efficient solution than multi-verifier signatures.

In similar research, Aiyer et al. [1] present schemes in which servers use MACs to generate tags having similar properties to digital signatures. Unlike

digital signatures and MVS schemes, however, their construction relies on communication between clients and servers to produce and verify signed messages. And they also require that no more than $1/3$ of the servers in the system be compromised.

In a distributed setting where at most t signers may be compromised, Lamport [49] suggests (in a set of slides on Byzantine Paxos) collecting $(\lambda + 1)t + 1$ tags from different signers. A signer in the scheme creates λ vectors consisting of n subtags each, where each subtag of each vector contains a MAC of all the vectors before it, along with the message. This scheme does not provide adaptive security, since an adversary with oracle access to the signing functionality can create a split tag by the following procedure. The adversary requests a tag for m and receives τ. Then the adversary corrupts τ to τ' by overwriting some subtags with random strings and requests a tag for $m \mid\mid \tau'$, receiving a tag τ''. The tag $\tau' \mid\mid \tau''$ is split for m, since all subtags in τ'' are supported, but some subtags in τ' are not supported.

Canetti et al. [17] propose a multicast MAC scheme that is closely related to the schemes in this dissertation. In this scheme, a collection of keys is associated with each verifier; keys are chosen randomly from a large set. Signers create a tag for a message m by generating a MAC of m for each key they know. The algorithm distributes keys at random with probability $\frac{1}{t+1}$ if up to t verifiers may collude to try to forge tags. Keys in this protocol may thus be shared by more than one verifier.[9] Canetti et al. show that given $\epsilon > 0$, having $e(t+1)\log(\frac{1}{\epsilon})$ keys in total suffices to guarantee that tags can be forged only with probability less than ϵ. However, these tags do not satisfy Transferability (5.5), since an

[9]The key distribution algorithms for Atomic Signatures and Chain Signatures are closely related to the algorithm by Canetti et al., but each key in Atomic Signatures and Chain Signatures is only shared between a pair of servers, so compromised relays are forced to guess keys to forge tags for messages.

adversary can create a new tag from a correctly-signed tag by corrupting one subtag. This new tag will be accepted by some verifiers and not by others.

CHAPTER 7

CONCLUSIONS

Replication improves reliability but can be expensive—only services that require high resilience to server failure ought to employ replication. Proactive obfuscation adds to this expense but transforms a fault-tolerant service into an attack-tolerant one. Not all services require this additional degree of resilience, and we show in this dissertation 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 be further reduced. First, our mechanisms for proactive obfuscation execute 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 our individual authentication Reply Synthesis implementation using digital signatures, and thus it is not fundamentally different from the case we studied. The use of MACs does require replicas to set up shared keys, and this cost would be added to the refresh and recovery costs already present in our prototypes.

Multi-verifier signature schemes can also provide lower-cost authentication than traditional digital signature schemes while guaranteeing similar properties, as shown in this dissertation. Achieving these properties in Atomic Signatures and Chain Signatures requires implementing a specialized secret-key setup. The setup encodes an asymmetric relationship between the signer and verifiers, since verifiers know which keys are owned by which signers, but

signers do not know which keys are owned by which verifiers. Asymmetry in knowledge about keys is critical for achieving Transferability, both in digital signature schemes and multi-verifier signature schemes.

We did not employ multi-verifier signatures in our prototypes for three reasons.

1. The costs of using digital signatures in our prototypes are significantly reduced by using RSA keys of size 512 bits—this reduced key size is sensible, given the rapid key refresh rate under proactive obfuscation.

2. Rekeying multi-verifier signatures requires executing the specialized key distribution algorithm at each epoch change, which would incur significant extra overhead.

3. Using multi-verifier signatures in our scheme requires adding more replicas. Recall from section 3.1 that Byzantine Paxos requires $3t + 2r + 1$ replicas if t replicas might be compromised and r replicas might be rebooting. As explained in section 5.3, using multi-verifier signatures for Byzantine Paxos requires adding t^2 extra replicas that might crash. So, the total number of replicas required for Byzantine Paxos using multi-verifier signatures and proactive obfuscation is $3t+2(r+t^2)+1$. If only one replica may reboot at a time, then this reduces to $3t+2\times 1+2t^2+1 = 3t+2t^2+3$, which means running 8 replicas in the case $t = 1$. This is more costly than the 6 replicas already required for proactive obfuscation using digital signatures.

The attack tolerance of a service employing proactive obfuscation depends fundamentally on what obfuscator(s) are in use. Our work, by design, has been largely independent of this obfuscation choice. That said, Obfuscation Independence (2.1) and Bounded Adversary (2.2) 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. [71] shows that obfuscated executables are easily compromised if they are generated by obfuscators not using enough randomness. On the other hand, Pucella and Schneider [63] 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 trades availability for integrity. In particular, an obfuscated replica that is processing input that conveys an attack is likely to crash (because the attack is unlikely to be well matched to the obfuscations that were applied). However, 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 proactive obfuscation imposes.

Attacks on availability 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 (2.8). Synchronicity is needed for State Recovery. To see why, recall that replicas are rebooted based on timeouts in the reboot clock. So, 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, which gives rise to the strong assumptions on synchronicity 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 any device receiving

information from replicas becomes subject to attacks conveyed by these inputs.

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 constraints on Replica Refresh and State Recovery; even if these constraints fail to hold, our implementations of Byzantine Paxos and APSS will continue to operate correctly.

DoS attacks reduce availability and are not affected by proactive obfuscation. DoS attacks by clients overloading a resource must still be countered by blocking the offending requests or terminating their source(s). 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. These attacks might leverage 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 (but not immediately) causing i to crash. Suppose i writes its state to disk before crashing, including that packet or the effect of its execution. After i crashes and reboots, the state 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.

Secret-Based Defenses and Secret Erosion

Proactive obfuscation and multi-verifier signatures both involve secret-based methods of defending against attacks. Proactive obfuscation uses secrets to obfuscate replica code, increasing attack tolerance by making replica compromise more difficult. Multi-verifier signatures uses secrets to randomly permute the order of keys held by the signer, preventing even a compromised signer from producing signatures that violate Transferability (5.5).

Under attack, however, secrets can erode over time. For example, an adversary can eventually disentangle the obfuscation used by each replica in a distributed service and mount a successful attack. So, proactive obfuscation periodically reobfuscates code and reboots replicas to bound the time available for adversaries to perform reverse engineering on replicas and craft attacks. Similarly, a compromised signer in a system employing multi-verifier signatures can create tags that cause verifiers to reveal information about which verifiers know which keys. So, verifiers in multi-verifier signatures attempt to detect such attacks; on detection, they refuse further interaction by returning $\perp$ to the signer thereafter. This defense prevents compromised signers from ever acquiring enough information to violate Transferability (5.5). Therefore, both proactive obfuscation and multi-verifier signatures successfully prevent secret erosion.

APPENDIX A

PROOF OF SME HIGHEST STATE RECOVERY

To prove that the state recovery request protocol of Section 3.1 satisfies SME Highest State Recovery (3.2), we use a lemma about Byzantine Paxos. To prove this lemma, we will need the following fact about Byzantine Paxos: for a replica in Byzantine Paxos to change to a state with sequence number s, it must receive messages from $2t + r + 1$ replicas that are in a state with sequence number $s - 1$.

Also, we assume that no more than r replicas can be in a state with a lower sequence number than the one they had when they sent any messages that determined the state with the currently highest sequence number held by any correct replica. This property is not difficult to guarantee: as long as State Recovery operates correctly each time, any replicas that lose their state by rebooting will be replaced by replicas that have a state with at least as high a sequence number (by SME Highest State Recovery (3.2)). So, this property can be proved for the first instance of State Recovery, then extended inductively to all later instances.

Lemma 11. *At any point in time ω in the execution of Byzantine Paxos, there is a state with sequence number s such that the $t + 1$ correct replicas with the highest numbered state have states with sequence numbers either s or $s - 1$.*

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

$s - 1$ (otherwise, the $t + 1$ replicas in states with sequence number s or $s - 1$ would have higher sequence numbers).

□

States sent by replicas in response to the state recovery request contain the current state and a diff to get to the previous state. So, Lemma 11 implies that a recovering replica receiving all the states and diffs for correct replicas at a given point in time ω will receive $t + 1$ copies of the state with sequence number $s - 1$, where s is the highest sequence number at a correct replica at time ω. Note that the recovering replica can wait to hear from all correct replicas, since the network satisfies Timely Links (2.8). Furthermore, the recovering replica also receives the messages that caused the correct replica in the state with sequence number s to change to this state. So, the recovering replica will reach the state with sequence number s, the highest sequence number at time ω.

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

So, this shows that a recovering replica has received, by time $T + \eta$ seconds after the beginning of the state recovery request protocol, enough information to recover to the state with the highest sequence number held by a correct replica at time T seconds after the start of the state recovery request protocol. This proves

that the state recovery request protocol of Section 3.1 satisfies SME Highest State Recovery (3.2).

APPENDIX B

LEMMAS FOR STRONG UNFORGEABILITY OF ATOMIC SIGNATURES

We prove several lemmas that together simplify the proof of Strong Unforgeability (5.6) of Atomic Signatures to the case where all but one verifier is compromised, no verifier queries are allowed for an adversary, and $\mathsf{MAC}(\cdot, k)$ is replaced by a random function $v_k(\cdot)$.

Lemma 12. *If Atomic Signatures satisfies Strong Unforgeability (5.6) when all but one verifier is compromised, then Atomic Signatures satisfies Strong Unforgeability (5.6).*

Proof. Suppose $\mathcal{A}$ violates Strong Unforgeability (5.6) using an arbitrary set $I' \subseteq I$ of compromised verifiers. We construct an adversary $\mathcal{B}$ that violates Strong Unforgeability (5.6) when all but one verifier, say verifier j, is compromised. $\mathcal{B}$ is given the keys for all verifiers but j and is given oracle access to a verification oracle for j as well as a signing oracle. $\mathcal{B}$ maps its verifiers randomly to verifiers in the simulation for $\mathcal{A}$; this does not change the view of $\mathcal{A}$, since keys are chosen uniformly at random. $\mathcal{B}$ runs $\mathcal{A}$ and simulates its oracle queries as follows.

1. $\mathcal{B}$ calls $\mathcal{A}(1^d, 1^n, \{\boldsymbol{K}_i\}_{i \in I'})$.
2. When $\mathcal{A}$ makes a signing oracle query, $\mathcal{B}$ passes the query to its signing oracle and returns its response.
3. When $\mathcal{A}$ makes a verification oracle query for verifier j', $\mathcal{B}$ calls its verification oracle if $j' = j$, and otherwise uses its knowledge of the keys for j' to perform verification for j' and return the result.
4. When $\mathcal{A}$ returns m and τ, $\mathcal{B}$ checks that $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_j) \neq 0$. If so, then $\mathcal{B}$ returns m and τ, and otherwise, $\mathcal{B}$ aborts.

When $\mathcal{A}$ returns m, τ, there is some $j' \in I - I'$ such that $\mathsf{Ver}^{AS}(m, \tau, \boldsymbol{K}_{j'}) \neq 0$, and there a $1/n$ chance that $j = j'$, since the position of j in $I - I'$ was chosen independently of the view of $\mathcal{A}$. So, $\mathcal{B}$ succeeds with probability ϵ/n if $\mathcal{A}$ succeeds with probability ϵ. □

Lemma 13. *If Atomic Signatures satisfies Strong Unforgeability (5.6) when no verifier queries are allowed to an adversary and all but one verifier is compromised, then Atomic Signatures satisfies Strong Unforgeability (5.6) when all but one verifier is compromised.*

Proof. Given an adversary $\mathcal{A}$ that succeeds with non-negligible probability with polynomial bound $p(n)$ on its number of verification queries, we can construct a new adversary $\mathcal{B}$ that succeeds with non-negligible probability without using any verifier queries. Assume, without loss of generality, that $\mathcal{A}$ always makes a verifier query for the pair m and τ that it outputs.[1]

$\mathcal{B}$ simulates verifier queries from $\mathcal{A}$ for a message-tag pair m, τ as follows: if m has been requested already from the signing oracle, which returned τ, then return ∞. If m has not been requested from the signing oracle, or the signing oracle returned anything but τ, then return 0. $\mathcal{B}$ stores each verification query. When $\mathcal{A}$ outputs m and τ, $\mathcal{B}$ chooses one of the stored verification queries m', τ' uniformly at random, and outputs it.

$\mathcal{A}$ either uses more than one verification query (the final one) or it does not. If it does not, then $\mathcal{B}$ succeeds every time $\mathcal{A}$ does, since $\mathcal{B}$ always chooses the one query that $\mathcal{A}$ made. And this is the value that $\mathcal{A}$ returned.

If $\mathcal{A}$ uses more than one verification query, then either some verification queries (other than the last) that were not received from the signing oracle

[1] If this is not the case, then there is another adversary $\mathcal{D}$ that succeeds with the same probability as $\mathcal{A}$ but performs the extra query. $\mathcal{D}$ runs $\mathcal{A}$, and when $\mathcal{A}$ returns m and τ, $\mathcal{D}$ queries m and τ from the verification oracle before returning them.

should have returned a value other than 0, or all verification queries (other than the last) that were not received from the signing oracle should have returned 0. If all except the last should return 0, then $\mathcal{B}$ succeeds only when it returns the m and τ from the last query. $\mathcal{A}$ succeeds with non-negligible probability ϵ and makes at most $p(n)$ queries, so $\mathcal{B}$ succeeds, in this case, with probability greater than or equal to $\epsilon/p(n)$, which is non-negligible.

If some queries not received from the signing oracle should return a value other than 0, then there is no guarantee about the success probability of $\mathcal{A}$, since $\mathcal{B}$ no longer simulates all of the verification queries correctly. But there is still a maximum bound of $p(n)$ on the number of verification queries, and there is at least one query that caused $\mathcal{B}$ to fail to simulate the verification queries correctly. The definition of $\mathcal{B}$ guarantees that this query violates Strong Unforgeability (5.6), since the query was not received from the signing oracle. This means that the probability of $\mathcal{B}$ returning a query m and τ that violates Strong Unforgeability (5.6) is at least $1/p(n)$, which is non-negligible.

So, $\mathcal{B}$ always succeeds in violating Strong Unforgeability (5.6) with non-negligible probability. □

For simplicity in stating the next lemmas, call the version of Atomic Signatures in the hypothesis of Lemma 13 *verifier-free* Atomic Signatures, and call Atomic Signatures when no verifier queries are allowed, all but one verifier is compromised, and $\mathsf{MAC}(\cdot, k)$ is replaced by a random function $v_k(\cdot)$ *verifier-free random* Atomic Signatures.

Lemma 14. *If* MAC *is a pseudorandom function, then if verifier-free random Atomic Signatures satisfies Strong Unforgeability (5.6), then verifier-free Atomic Signatures satisfies Strong Unforgeability (5.6).*

Proof. Suppose, on the contrary, that verifier-free random Atomic Signatures

satisfies Strong Unforgeability (5.6), but verifier-free Atomic Signatures does not. This means that there is an adversary $\mathcal{A}$ that succeeds with non-negligible probability ϵ when interacting with oracles that use $\mathsf{MAC}(\cdot, k)$ to answer signing queries, and, by assumption, succeeds with only negligible probability ϵ' when interacting with oracles that use random functions $v_k(\cdot)$ to answer signing queries.

Now construct a sequence of hybrids as follows: H_i uses $v_{k_j}(\cdot)$ instead of $\mathsf{MAC}(\cdot, k_j)$ for keys k_j such that $1 \leq j \leq i$, then uses $\mathsf{MAC}(\cdot, k_j)$ for the remaining keys k_j such that $i+1 \leq j \leq dn$. Note that H_0 is verifier-free Atomic Signatures, and H_{dn} is verifier-free random Atomic Signatures.

A standard hybrid argument shows that there must be an i such that $\mathcal{A}$ succeeds with non-negligible probability on hybrid H_i and and succeeds with negligible probability on hybrid H_{i+1}. But then there exists a algorithm $\mathcal{D}$ that can distinguish a random function from MAC, since the only difference between hybrids H_i and H_{i+1} is that H_i uses MAC in its $i+1$st position, whereas H_{i+1} uses a random function in this position.

$\mathcal{D}$ sets up an instance of the hybrid Atomic Signatures scheme using its oracle (which is either a pseudorandom or a random function) in the $i+1$st position. Then $\mathcal{D}$ calls $\mathcal{A}$ on this instance and returns 1 if $\mathcal{A}$ succeeds and 0 if $\mathcal{A}$ fails. Since $\mathcal{A}$ succeeds with non-negligible probability when there is a pseudorandom function in the $i+1$st position and succeeds only with negligible probability when there is a random function in the $i+1$st position, $\mathcal{D}$ succeeds in distinguishing pseudorandom from random functions with non-negligible probability. This contradicts the hypothesis that MAC is a pseudorandom function. □

APPENDIX C

CORRECT SIGNERS

The constructions of MVS schemes in this paper allow the signer to be compromised. But there are places where it is reasonable to make stronger assumptions. For instance, when it is sound to assume that the signer is not compromised, we can simplify our constructions significantly. This assumption holds in some common contexts: for example, in operating systems, the OS itself is trusted by the processes and sometimes signs messages (e.g., capabilities) to processes.

When the signer is not compromised, Transferability (5.5) can be weakened to the following:

(C.1) Weak Transferability. For every non-uniform PPT adversary $\mathcal{A}$, there exists a negligible function ϵ such that for any choice of $I' \subseteq I$,

$$\Pr[(\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I}) \leftarrow \mathsf{Gen}(1^d, 1^n);$$
$$(m, \tau) \leftarrow A^{\mathsf{Sign}(\cdot,\boldsymbol{k}),\{\mathsf{Ver}(\cdot,\cdot,\boldsymbol{K}_i)\}_{i\in I-I'}}(1^d, \{\boldsymbol{K}_i\}_{i\in I'}) :$$
$$(\exists j, j' \in I - I' \ : \ \left|\mathsf{Ver}(m, \tau, \boldsymbol{K}_j) - \mathsf{Ver}(m, \tau, \boldsymbol{K}_{j'})\right| > 1)] \leq \epsilon(d, n).$$

Weak Transferability (C.1) implies that even an adversary that controls an arbitrary subset of verifiers and has signing and verification oracles (for the other verifiers) cannot produce message and tag pair on which two correct verifiers will produce values that differ by more than one. Notice that the adversary in this case does not control the signer.

We call λ-MVS scheme that satisfies λ-Completeness (5.2), Unforgeability (5.3), and Weak Transferability (C.1) a *Weak λ-MVS* scheme.

C.1 Known-Key Atomic Signatures

Known-Key Atomic Signatures (KA) is a Weak ∞-MVS scheme based on Atomic Signatures, and it only uses one key for each verifier. KA follows exactly the algorithms for Atomic Signatures for the case $d = 1$. But this means that verifiers can never return $\perp$, since either their single instance of equation (6.2) is satisfied or it is not.

Theorem 15. *If the* MAC *is a pseudorandom function, then Known-Key Atomic Signatures is a Weak ∞-MVS scheme.*

Proof. **∞-Completeness.** Same reasons as Atomic Signatures.

Strong Unforgeability. This follows from exactly the same proof as for Atomic Signatures. The only difference is that the Union Bound does not include a factor of d, since each verifier only has 1 key rather than d.

Weak Transferability. We show that Strong Unforgeability (5.6) implies Weak Transferability (C.1). As we have already shown that Known-Key Atomic Signatures satisfies Strong Unforgeability, this implies that it also satisfies Weak Transferability.

We prove the contrapositive. Suppose that there is an adversary $\mathcal{A}$ that can violate Weak Transferability (C.1) with non-negligible probability ϵ. $\mathcal{A}$ uses access to a signing oracle and verification oracles to produce a message m and tag τ such that for some pair $j, j' \in I - I'$, it holds that $|\mathsf{Ver}^{KA}(m, \tau, \boldsymbol{K}_j) - \mathsf{Ver}^{KA}(m, \tau, \boldsymbol{K}_{j'})| > 1$. This means that one verifier must return ∞ and the other must return 0. Without loss of generality, assume that j returns ∞ and j' returns 0.

We now produce an adversary $\mathcal{B}$ that violates Strong Unforgeability (5.6). $\mathcal{B}$ is given the same signing and verification oracles as $\mathcal{A}$ and must produce a message and tag that it has never received from the signing oracle but causes

some verifier $j \in I - I'$ to produce a value that is not 0. $\mathcal{B}$ simply calls $\mathcal{A}$, simulates $\mathcal{A}$'s oracle calls by passing them to $\mathcal{B}$'s oracles, and returns the values of m and τ returned by $\mathcal{A}$.

Since verification for j returns ∞, the values m and τ will suffice to violate Strong Unforgeability (5.6) as long as m and τ were never received from $\mathcal{B}$'s signing oracle. But the same values of m and τ also cause $j' \in I - I'$ to return 0. And ∞-Completeness implies that no message and tag returned from the signing oracle ever causes a correct verifier to return 0. So, m and τ were not received from the signing oracle, and $\mathcal{B}$ succeeds in violating Strong Unforgeability (5.6). So, Strong Unforgeability (5.6) implies Weak Transferability (C.1).

And since we proved above that Known-Key Atomic Signatures satisfies Strong Unforgeability (5.6), it follows that Known-Key Atomic Signatures also satisfies Weak Transferability (C.1). □

C.2 Known-Key Chain Signatures

Known-Key Chain Signatures (KC) is a Weak λ-MVS scheme obtained by simplifying Chain Signatures—it does not use unknown-key components in tags. So, each verifier shares exactly one key with the signer. Its algorithms operate as follows

- $\mathsf{Gen}^{KC}(1^n)$ simply sets up pairwise shared keys. The signer is given a vector $\boldsymbol{k}$ of keys, and each verifier j is given $\boldsymbol{k}[j]$.
- $\mathsf{Sign}^{KC}(m, \lambda, \boldsymbol{k})$ performs exactly the same operations as in Chain Signatures, but only uses the known-key components. We index subtag j in section r by a pair (r, j) with the natural lexicographic ordering. This sub-

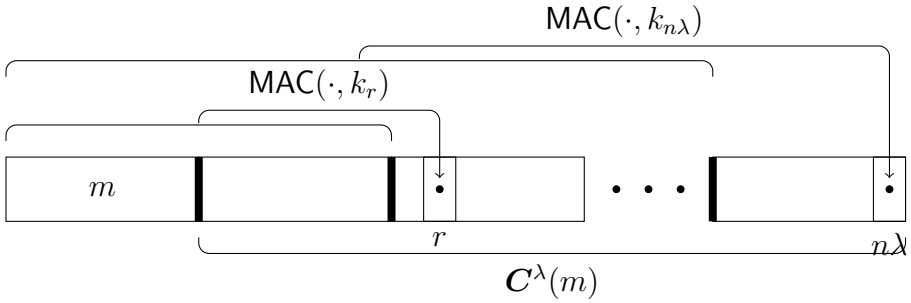

Figure C.1: The structure of Known-Key Chain Signatures

tag is computed for the tag $\boldsymbol{C}^{\lambda}(m)$ as follows.

$$\boldsymbol{C}^{\lambda}(m)[r,j] \triangleq \mathsf{MAC}(m \mid\mid \underset{(t,t')<(r,1)}{\mid\mid} \boldsymbol{C}^{\lambda,d}(m)[t,t'], \boldsymbol{k}[j]) \tag{C.1}$$

- $\mathsf{Ver}^{KC}(m, \tau, \boldsymbol{K}_j)$ finds the highest section λ for which j's subtag is supported, and returns λ. If there is no such section, then it returns 0. Verification never returns $\perp$, since the signer cannot be compromised.

Figure C.1 shows the structure of KC, where we write k_p for $\boldsymbol{k}[p \bmod n]$

Generating a signature requires $n\lambda$ steps: the tag contains $n\lambda$ subtags, and each step produces one subtag by computing the MAC of a vector that is of size at most $|m| + n\lambda$. Thus the total cost of generating a tag is $O(n\lambda(|m| + n\lambda))$. The total cost can be significantly reduced, as explained in Appendix E.

Theorem 16. *For any $\lambda \in \mathbb{N}_{>0}$, if the* MAC *satisfies CTA Unforgeability (5.1), then KC is a Weak λ-MVS scheme.*

Proof. **λ-Completeness.** This follows trivially from the definition of KC, just as for Chain Signatures.

Unforgeability. The proof is the same as for Chain Signatures when the adversary causes some verifier to return a value in $\mathbb{N}_{>0}$: an adversary $\mathcal{B}$ is constructed that violates CTA Unforgeability (5.1) of the MAC.

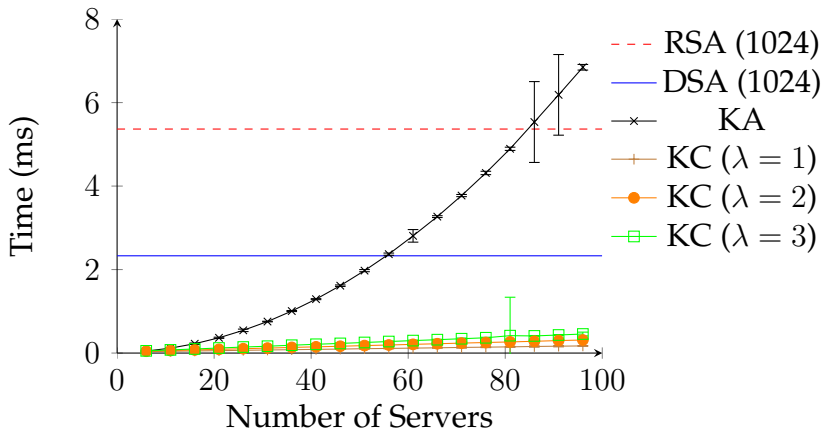

Figure C.2: Execution time for generating Known-Key Chain Signatures and Known-Key Atomic Signatures by correct signers

Weak Transferability. The proof is almost identical to the proof of Non-Accusability (5.4) of Chain Signatures: if an adversary violates Weak Transferability (C.1), then by definition, some subtag will be supported that takes as input a subtag that is not supported. The probability of success for our constructed adversary in this case, however, is $\frac{\epsilon}{n}$ rather than $\frac{\epsilon}{(d+1)n}$, since the constructed adversary that violates CTA Unforgeability (5.1) of the MAC guesses a key k_t from a set of size n rather than size $(d+1)n$. □

C.3 Performance

Our implementation of Known-Key Atomic Signatures and Known-Key Chain Signatures uses the same libraries and hash function as our implementation of Atomic Signatures and Chain Signatures.

Figures C.2 and C.3 show the execution time for generating and checking Known-Key Chain Signatures and Known-Key Atomic Signatures compared to RSA and DSA signatures when the signer is known not to be compromised.

In Figure C.2, Known-Key Atomic Signatures are faster to generate than

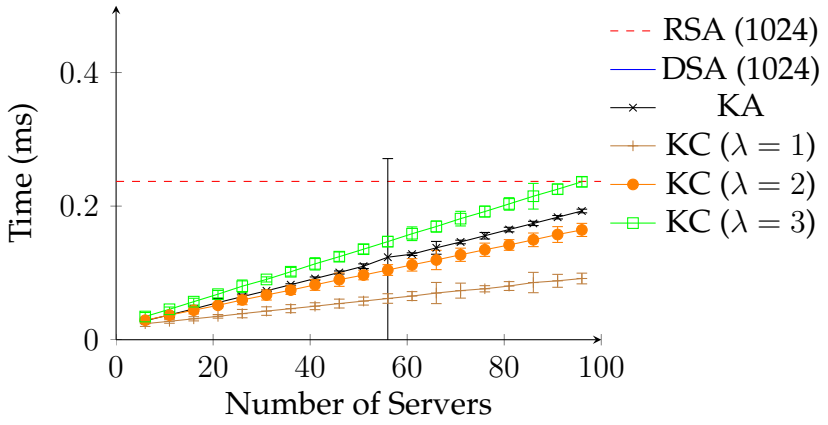

Figure C.3: Execution time for checking Known-Key Chain Signatures and Known-Key Atomic Signatures from correct signers

RSA signatures until the number of verifiers n is about 80. Further, Known-Key Chain Signatures for $\lambda = 1, 2,$ and 3 are much faster to generate than either DSA or RSA signatures for all $n < 100$. Thus, when the signer is not compromised, tags based on shared keys and that protect against compromised relays can be created much more quickly than tags based on public keys. The time required for checking Known-Key Atomic Signatures is shown in Figure C.3 to be faster than RSA for all n less than 100. The time required for checking Known-Key Chain Signatures is much smaller than RSA for all $n < 100$. DSA checking is not given on the graph, since it requires nearly 3ms, and this is off the scale.

Signature size is significantly smaller than for Atomic Signatures and Chain Signatures, as shown in Figure C.4.

The smaller size and faster signature generation time shown in the results above suggest that Known-Key Chain Signatures and Known-Key Atomic Signatures could be profitably applied in domains such as operating systems, where there is already a trusted third party (the OS) managing interactions between processes. For example, either scheme could be used for OS capabilities.

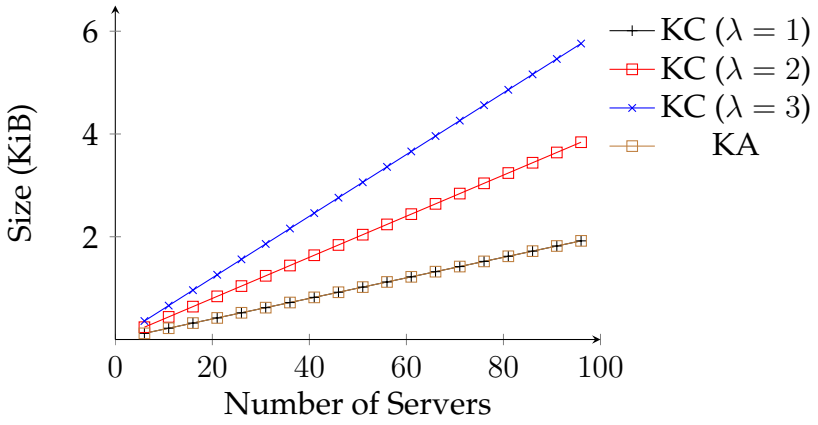

Figure C.4: Size of Known-Key Chain Signatures and Known-Key Atomic Signatures

APPENDIX D

UNIVERSALLY COMPOSABLE DEFINITIONS

To gain confidence in our game-based MVS definition, we present an ideal functionality $\mathcal{F}_{\mathrm{SIG}}^{C,\beta}$ (see Figure D.1) that generalizes the signature functionality $\mathcal{F}_{\mathrm{SIG}}$ [16] by allowing verification of signatures to return $\perp$, which signifies that the signer is compromised, and by allowing verification to return positive values other than ∞ and 0. For setup in our secret-key setting, we rely on a key distribution functionality $\mathcal{F}_{\mathrm{DC}}(\mathsf{Gen})$ that calls Gen for a given MVS scheme and passes the keys that Gen outputs to the appropriate server.[1] See Figure D.2 for a description of $\mathcal{F}_{\mathrm{DC}}(\mathsf{Gen})$. We call $\mathcal{F}_{\mathrm{DC}}(\mathsf{Gen})$ the Dining Cryptographer's Key Distribution functionality, because a signer receiving keys from verifiers does not know which keys are known to which verifiers, as in the Dining Cryptographer's problem [24].

$\mathcal{F}_{\mathrm{SIG}}^{C,\beta}$ follows the same structure as $\mathcal{F}_{\mathrm{SIG}}$ [16]. Key Generation requests signing and verification algorithms from an adversary; unlike $\mathcal{F}_{\mathrm{SIG}}$, the verification algorithms are not published, but rather are distributed to servers directly by $\mathcal{F}_{\mathrm{SIG}}^{C,\beta}$. This change is required, because the verification functions for MVS schemes are not *a priori* known to all verifiers. Signature Generation ensures that it is only called by the designated signer S. It also checks that s outputs a value that satisfies β-Completeness (5.2). Signature Verification for a given server V associated with verifier j returns $\perp$ if the state of verifier j is already $\perp$. It also ensures that any message-tag pair for which it returns a value satisfies Unforgeability (5.3), Non-Accusability (5.4), and Transferability (5.5) for every recorded verifier. If any of these conditions are not satisfied, then $\mathcal{F}_{\mathrm{SIG}}^{C,\beta}$ returns

[1] We do not describe how to define protocols that realize $\mathcal{F}_{\mathrm{DC}}(\mathsf{Gen})$, but it can be realized as a setup assumption in many common settings. For example, a trusted third party could implement $\mathcal{F}_{\mathrm{DC}}(\mathsf{Gen})$ for a small distributed service before the service began executing.

Functionality $\mathcal{F}_{\text{SIG}}^{C,\beta}$

Key Generation: Upon receiving a value $(\texttt{KeyGen}, sid, n)$ from some server S, verify that $sid = (S, sid')$ for some sid'. If not, then ignore it. Otherwise, pass $(\texttt{KeyGen}, sid, n)$ to the adversary. Upon receiving $(\texttt{Algorithms}, sid, s, \{v_i\}_{i=1}^n)$ from the adversary, where s is a PPT and each v_i is a deterministic PPT, pass $(\texttt{SigningAlgorithm}, sid, s)$ to S, and, for each verifier i, pass $(\texttt{VerificationAlgorithm}, sid, S, v_i)$ to i. Store $(\texttt{Signer}, sid, s)$, and for $i \in \{1, \ldots, n\}$, store $(\texttt{Verifier}, sid, v_i, i, 1)$; the last element of the $\texttt{Verifier}$ tuple is called the *state* of verifier v_i.

Signature Generation: Upon receiving $(\texttt{Sign}, sid, m)$ from S, verify that $sid = (S, sid')$ for some sid'. If not, then ignore it. Produce $\sigma \triangleq s(m)$. If $v_i(m, \sigma) \geq \beta$ for each recorded v_i, then output $(\texttt{Signature}, sid, m, \sigma)$ to S and record (sid, m, σ). Otherwise, output an error to S.

Signature Verification: Upon receiving $(\texttt{Verify}, sid, m, \sigma, v')$ from verifier j, if $v' \neq v_j$, then output $(\texttt{Verified}, sid, m, v'(m, \sigma))$ to j. Otherwise, if v_j has state $\perp$, then return $(\texttt{Verified}, sid, m, \perp)$.

If S is not compromised, and for some recorded, non-compromised verifier v it holds that $v(m, \sigma) \in \mathbb{N}^\infty$ and there is no entry (sid, m, σ), or it holds that $v(m, \sigma) = \perp$ and the state of v is not $\perp$, then output an error to V.

Further, if $\left|v_i(m, \sigma) - v_{i'}(m, \sigma)\right| > 1$ for any recorded pair of verifiers v_i, $v_{i'}$ such that neither v_i nor $v_{i'}$ has state $\perp$, $v_i(m, \sigma) \neq \perp$, and $v_{i'}(m, \sigma) \neq \perp$, then output an error to V.

Otherwise, output $(\texttt{Verified}, sid, m, v'(m, \sigma))$ to V. If $v_j(m, \sigma) = \perp$, then set the state of v_j to $\perp$.

Figure D.1: The generalized signature functionality $\mathcal{F}_{\text{SIG}}^{C,\beta}$.

Functionality $\mathcal{F}_{\mathrm{DC}}$(Gen).

Setup: Upon receiving of the first $(\texttt{Setup}, sid, n, d)$ from S such that $sid = (S, sid')$ for some sid', record (sid, n, d). Call $\mathsf{Gen}(1^d, 1^n)$ to get $\boldsymbol{K}_j$ for each server j and $\boldsymbol{k}$ for the signer.

Then, pass $(\texttt{Key}, sid, \boldsymbol{k})$ to S with $\boldsymbol{k}$ randomly permuted, and, for each server j, pass $(\texttt{Key}, sid, \boldsymbol{K}_j)$ to j.

Figure D.2: The dining cryptographers key distribution functionality $\mathcal{F}_{\mathrm{DC}}(\mathsf{Gen})$

an error.

We show that a UC-secure implementation of $\mathcal{F}_{\mathrm{SIG}}^{C,\beta}$ is equivalent to the game-based definition. More precisely, we consider a protocol $\pi(\Sigma)$ (a simple generalization of Canetti's protocol to realize $\mathcal{F}_{\mathrm{SIG}}$ [16]) that responds to `KeyGen`, `Sign`, and `Verify` by calling $\mathcal{F}_{\mathrm{DC}}(\mathsf{Gen})$, Sign and Ver, respectively, given a tuple of algorithms $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$. Note that this theorem implies that MVS schemes are secure under concurrent composition.

Theorem 17. *Let $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ and let λ be an element of $\mathbb{N}^\infty$. Then $\pi(\Sigma)$ securely realizes $\mathcal{F}_{\mathrm{SIG}}^{C,\lambda}$ in the $\mathcal{F}_{\mathrm{DC}}(\mathsf{Gen})$-hybrid model if and only if Σ is a λ-MVS scheme.*

Proof. This proof follows a similar proof by Canetti [16]. To prove the first direction of the implication, we assume that Σ violates one of λ-Completeness (5.2), Transferability (5.5), Non-Accusability (5.4), or Unforgeability (5.3) and show that $\pi(\Sigma)$ does not securely realize $\mathcal{F}_{\mathrm{SIG}}^{C,\lambda}$.

To do so, we construct an environment $\mathcal{Z}$ and an adversary $\mathcal{A}$ such that for all ideal process adversaries $\mathcal{S}$, environment $\mathcal{Z}$ can tell whether it is interacting with $\mathcal{A}$ and $\pi(\Sigma)$ or with $\mathcal{S}$ and the ideal process for $\mathcal{F}_{\mathrm{SIG}}^{C,\lambda}$. Environment $\mathcal{Z}$ does not communicate with $\mathcal{A}$, so we leave $\mathcal{A}$ unspecified. There are four cases, corresponding to the four ways in which Σ can fail to satisfy the λ-MVS definition.

1. Assume that Σ does not satisfy λ-Completeness (5.2). Thus there exists an

m such that there exists a j in I such that, for infinitely many values of d,

$$\Pr[\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I} \leftarrow \mathsf{Gen}(1^d, 1^n) : \mathsf{Ver}(m, \mathsf{Sign}(m, \boldsymbol{k}), \boldsymbol{K}_j) < \lambda] > \epsilon(d)$$

First $\mathcal{Z}$ writes 1 to its output tape. Then $\mathcal{Z}$ chooses party V (corresponding to verifier j) and S, sets $sid = (S, 0)$, and calls S with $(\texttt{KeyGen}, sid, n)$. $\mathcal{Z}$ sends $(\texttt{Sign}, sid, m, \lambda)$ and receives σ. $\mathcal{Z}$ then sends $(\texttt{Verify}, sid, m, \sigma)$ to V. With non-negligible probability, V returns a value that is less than λ. In this case, $\mathcal{Z}$ overwrites its output tape with a 0. Note that when this adversary runs against $\mathcal{F}_{\text{SIG}}^{C,\lambda}$, $\mathcal{Z}$ always returns 1, since λ-Completeness (5.2) is enforced in the ideal functionality (and, in case of a λ-Completeness (5.2) error, $\mathcal{F}_{\text{SIG}}^{C,\lambda}$ returns an error to S, so $\mathcal{Z}$ never receives a tag to send to V).

2. Assume that Σ does not satisfy Transferability (5.5). Then there exists a non-uniform PPT G such that there exists an $I' \subseteq I$ such that for infinitely many d,

$$\begin{aligned}
&\Pr[\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I} \leftarrow \mathsf{Gen}(1^d, 1^n); \\
&\quad (m, \sigma) \leftarrow G^{\{\mathsf{Ver}(\cdot,\cdot,\boldsymbol{K}_i)\}_{i\in I-I'}}(1^d, 1^n, \boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I'}) : \\
&\quad \exists j, j' \in I - I' \ : \ \mathsf{Ver}(m, \sigma, \boldsymbol{K}_j) \neq \perp \wedge \mathsf{Ver}(m, \sigma, \boldsymbol{K}_{j'}) \neq \perp \\
&\qquad\qquad \wedge \left|\mathsf{Ver}(m, \sigma, \boldsymbol{K}_j) - \mathsf{Ver}(m, \sigma, \boldsymbol{K}_{j'})\right| > 1] > \epsilon(d).
\end{aligned}$$

To distinguish between interactions, $\mathcal{Z}$ writes 1 to its output tape, runs a copy of G, and proceeds as before to send a $(\texttt{KeyGen}, sid, n)$ request to some $S \in I'$ to set up keys. $\mathcal{Z}$ compromises parties in I' and gets the signing key and associated verification keys to pass to G. $\mathcal{Z}$ then simulates request (m', σ') from G to verification oracle $\mathsf{Ver}(\cdot, \cdot, \boldsymbol{K}_j)$ of G by passing $(\texttt{Verify}, sid, m', \sigma')$ to V associated with j and returning its response to G. When G outputs a pair (m, σ), $\mathcal{Z}$ requests verification from each server in

$I - I'$ and checks their responses to see if Transferability (5.5) holds. If not, then $\mathcal{Z}$ outputs 0.

Note that when $\mathcal{Z}$ interacts with $\pi(\Sigma)$, G succeeds with non-negligible probability, so $\mathcal{Z}$ outputs 0. But, when interacting with the ideal process and $\mathcal{S}$, $\mathcal{Z}$ never outputs 0, since $\mathcal{F}_{\text{SIG}}^{C,\lambda}$ outputs an error for message and tag pairs that violate Transferability (5.5), so $\mathcal{Z}$ receives no responses to compare.

3. Assume that Σ does not satisfy Non-Accusability (5.4). Then there exists a non-uniform PPT G such that there exists an $I' \subseteq I$ such that, for infinitely many d,

$$\begin{aligned}\Pr[&\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i \in I} \leftarrow \mathsf{Gen}(1^d, 1^n); \\ &(m, \sigma) \leftarrow G^{\mathsf{Sign}(\cdot, \boldsymbol{k}), \{\mathsf{Ver}(\cdot, \cdot, \boldsymbol{K}_i)\}_{i \in I}}(1^d, 1^n, \{\boldsymbol{K}_i\}_{i \in I}) : \\ &\exists j \in I - I' \ : \ \mathsf{Ver}(m, \sigma, \boldsymbol{K}_j) = \perp] \geq \epsilon(d).\end{aligned}$$

$\mathcal{Z}$ proceeds as for Transferability (5.5) to write 1 to its output tape, run G, and simulate its oracles. Instead of compromising the signer, however, $\mathcal{Z}$ simulates G's oracle calls to sign message m' by requesting that signer S sign m'. Then, when G outputs (m, σ), $\mathcal{Z}$ sends (m, σ) to all verifiers and outputs 0 if any of them return $\perp$ (unless a verifier had already returned $\perp$ before. In this case, $\mathcal{Z}$ outputs 0 and halts). As for Transferability (5.5), $\mathcal{Z}$ outputs 0 with non-negligible probability when running over $\pi(\Sigma)$, since G succeeds with non-negligible probability, but always outputs 1 when running over $\mathcal{F}_{\text{SIG}}^{C,\lambda}$, since $\mathcal{F}_{\text{SIG}}^{C,\lambda}$ outputs an error when m and τ violates Non-Accusability (5.4), so no verifier ever returns $\perp$ to $\mathcal{Z}$.

4. Assume that Σ does not satisfy Unforgeability (5.3). Then there exists a non-uniform PPT G such that there exists an $I' \subseteq I$ such that, for infinitely

many d,

$$\Pr[\boldsymbol{k}, \{\boldsymbol{K}_i\}_{i\in I} \leftarrow \mathsf{Gen}(1^d, 1^n);$$
$$(m, \sigma) \leftarrow G_r^{\mathsf{Sign}(\cdot,\boldsymbol{k}),\{\mathsf{Ver}(\cdot,\cdot,\boldsymbol{K}_i)\}_{i\in I}}(1^d, 1^n, \{\boldsymbol{K}_i\}_{i\in I}) :$$
$$\neg\mathsf{req}(G, m, \boldsymbol{k}, r) \wedge \exists j \in I - I' \ : \ \mathsf{Ver}(m, \sigma, \boldsymbol{K}_j) \in \mathbb{N}^\infty] \geq \epsilon(d).$$

$\mathcal{Z}$ proceeds as for Transferability (5.5) to write 1 to its output tape, run G, and simulate its oracles. Instead of compromising the signer, however, $\mathcal{Z}$ simulates G's oracle calls to sign message m' by requesting that signer S sign m'. Then, when G outputs (m, σ), $\mathcal{Z}$ halts if m was already requested of the signing oracle. Otherwise, $\mathcal{Z}$ sends (m, σ) to all verifiers and outputs 0 if any of them return a value in $\mathbb{N}^\infty$. As for Transferability (5.5), $\mathcal{Z}$ outputs 0 with non-negligible probability when running over $\pi(\Sigma)$, since G succeeds with non-negligible probability, but always outputs 1 when running over $\mathcal{F}_{\text{SIG}}^{C,\lambda}$. And with non-negligible probability, $\mathcal{Z}$ never halts prematurely, since when G succeeds, m was not requested from the signing oracle (by definition).

For the reverse implication, consider security against the dummy adversary $\mathcal{D}$ that acts as a pass-through for $\mathcal{Z}$ (security against the dummy adversary is shown in [15] to be equivalent to security against arbitrary adversaries; the intuition in this case is that the environment can simulate any adversary). Assume that $\pi(\Sigma)$ does not securely realize $\mathcal{F}_{\text{SIG}}^{C,\lambda}$ in the $\mathcal{F}_{\text{DC}}(\mathsf{Gen})$-hybrid model. This means that there exists an environment $\mathcal{Z}$ that can distinguish interacting with $\mathcal{D}$ and $\pi(\Sigma)$ from interacting with any $\mathcal{S}$ and $\mathcal{F}_{\text{SIG}}^{C,\lambda}$ with non-negligible probability ϵ. Since $\mathcal{Z}$ can distinguish its interaction for any ideal-process adversary, we define a particular ideal-process adversary $\mathcal{S}$ that only generates keys for $\mathcal{F}_{\text{SIG}}^{C,\lambda}$ and responds to compromise requests from $\mathcal{Z}$. To generate keys, $\mathcal{S}$ calls

$\mathsf{Gen}(1^d, 1^n)$ to get $\boldsymbol{k}$ and $\{\boldsymbol{K}_i\}_{i \in I}$ and returns $\mathsf{Sign}(\cdot, \boldsymbol{k})$ and $\{\mathsf{Ver}(\cdot, \cdot, \boldsymbol{K}_i)\}_{i \in I}$ to $\mathcal{F}_{\text{SIG}}^{C,\lambda}$. $\mathcal{S}$ simply acts as a pass-through between $\mathcal{F}_{\text{SIG}}^{C,\lambda}$ and $\mathcal{Z}$ for compromise requests.

Now assume that Σ satisfies λ-Completeness (5.2), Transferability (5.5), and Non-Accusability (5.4). We will show that Σ does not satisfy Unforgeability (5.3) by constructing a forger G that runs $\mathcal{Z}$ and indistinguishably simulates its interaction with $\pi(\Sigma)$ and $\mathcal{D}$. We choose randomly a verifier j in I to be non-compromised, and we build a forger given this j. G is given a signing oracle, a verification oracle for $\mathsf{Ver}_{\boldsymbol{K}_j}$, and verification keys for all other $n-1$ verifiers. When $\mathcal{Z}$ requests a compromise of server i, G aborts if i is the signer or is j. Otherwise, G returns the keys and state of i. G replies to requests from $\mathcal{Z}$ as follows.

When $\mathcal{Z}$ requests $(\texttt{KeyGen}, sid, n)$, G sends to $\mathcal{Z}$ the keys for any servers compromised by $\mathcal{Z}$ (this simulates the output of $\mathcal{F}_{\text{DC}}(\mathsf{Gen})$ as seen by $\mathcal{Z}$). When $\mathcal{Z}$ requests $(\texttt{Sign}, sid, m)$, G returns the result of requesting a signature for m from its signing oracle. When $\mathcal{Z}$ requests $(\texttt{Verify}, sid, m, \sigma, v')$ for some server i, if v' is not v_i, then G simply returns a `Verified` response using v'. Otherwise, if $i \neq j$, then G uses its verification keys for i to compute $\mathsf{Ver}(m, \sigma, \boldsymbol{K}_i)$, and returns the result (keeping state as necessary for returning $\perp$). G also requests m from j's verification oracle, and returns m and σ if the value returned from the oracle is in N^∞. There is only a negligible probability of the oracle returning $\perp$, since Non-Accusability (5.4), so G aborts if the oracle returns $\perp$.

If $i = j$ in the request from $\mathcal{Z}$, then G requests that its verification oracle check (m, σ), and returns the result. If the result is greater than 0, then G returns (m, σ). Otherwise, G continues.

Note that the output of G and the output of $\mathcal{D}$ with $\pi(\Sigma)$ are indistinguish-

able to $\mathcal{Z}$, since G is effectively running a centralized version of $\pi(\Sigma)$, and $\mathcal{D}$ simply relays requests from $\mathcal{Z}$. Further, since $\mathcal{S}$ instantiates $\mathcal{F}_{\text{SIG}}^{C,\lambda}$ with the protocols in Σ, and these protocols satisfy λ-Completeness (5.2), Non-Accusability (5.4), and Transferability (5.5), the only way the output of $\pi(\Sigma)$ and $\mathcal{D}$ could differ from the output of $\mathcal{S}$ and $\mathcal{F}_{\text{SIG}}^{C,\lambda}$ is in the case where $\mathcal{Z}$ produces a message that has not been signed but causes some non-compromised verifier to return a value in $\mathbb{N}^{\infty}$.

So, $\mathcal{Z}$ must create an (m, σ) that violates Unforgeability (5.3) for some server j' with non-negligible probability. And we assume, without loss of generality, that $\mathcal{Z}$ always passes this forgery to the verification function. Since such a message will only allow $\mathcal{Z}$ to distinguish the ideal from the real case if the signer and j' are not compromised, a request from $\mathcal{Z}$ for such a pair (m, σ) must occur before $\mathcal{Z}$ requests to compromise the signer or verifier j'.

Finally, since j was chosen uniformly at random and independently of $\mathcal{Z}$, and $\mathcal{Z}$'s view is independent of which j was chosen, the probability that $j = j'$ is $\frac{1}{n}$, so the success probability of G is $\frac{\epsilon}{n}$, which is non-negligible if ϵ is non-negligible. □

Note that the same methods used in Canetti and Rabin [19] to generalize $\mathcal{F}_{\text{SIG}}$ to multiple instances of the signature functionality that all share the same state applies to $\mathcal{F}_{\text{SIG}}^{C,\lambda}$ essentially without modification, so the JUC (Universal Composability with Joint State) Theorem can be applied to $\mathcal{F}_{\text{SIG}}^{C,\lambda}$ as well. This means that MVS schemes can be composed securely even when the same key is used to sign more than one message and verifiers keep state between messages.

APPENDIX E

EFFICIENT CHAIN SIGNATURES

To make Chain Signatures and Known-Key Chain Signatures more efficient, we employ a different implementation that uses a family of collision-resistant hash functions to keep the size of the input to the MAC constant. The algorithm for generating tags using Known-Key Chain Signatures is presented in Figure E.1. There, h is chosen from H, a family of collision-resistant hash functions, operator $||$ is concatenation as before, and we define $x \,||\, y = x$ if $y = \texttt{NULL}$. To generate a tag, a signer follows the same algorithm as before, except that the input to the MAC in a given section is now the section number, along with the hash of the concatenation of two values: (1) the input to the previous section, and (2) the hash of the previous section.

To check a subtag in section p, a verifier must use each subtag in each section that precedes section p and build up $w_p(m)$, the input to the MACs in section p. Verifiers follow the algorithm in Figure E.1 to build up $w_p(m)$ and use it compute the MACs corresponding to the subtags they are checking.

To calculate the time needed to compute a tag, we assume that both the hash and the MAC execute in time linear in the length of their input. We also assume that both the hash and the MAC produce a constant-size output.

The loop over p in Figure E.1 has λ iterations, and each iteration involves a hash of a value of constant length, followed by a loop with n iterations and a hash computation over data of size $O(n)$. Since p has size $\log \lambda$ and $w_p(m)$ has constant size, the loop over p' takes time $O(n \log \lambda)$. There is also an initial cost of time $O(|m|)$ to compute $w_1(m)$. So, the total time to generate Known-Key Chain Signatures is $O(|m| + \lambda + n\lambda \log \lambda + n) = O(|m| + n\lambda \log \lambda)$.

This more efficient algorithm for Known-Key Chain Signatures is general-

$w_0(m) := m$;
$v_0(m) := \texttt{NULL}$;
for $p := 1$ to λ
    $w_p(m) := h(w_{p-1}(m) \mid\mid v_{p-1}(m))$;
    for $p' := 1$ to n
        $\boldsymbol{C}^{\lambda}(m)[p, p'] := \mathsf{MAC}(p \mid\mid w_p(m), k_p)$;
    $v_p(m) := h(\mathop{\Big\|}_{t=1}^{n} \boldsymbol{C}^{\lambda}(m)[p, t])$

Figure E.1: The hashing version of Known-Key Chain Signatures

$w'_0(m) := m$;
$v'_0(m) := \texttt{NULL}$
for $p := 1$ to λ
    $w_p(m) := h(w'_{p-1}(m) \mid\mid v'_{p-1}(m))$
    for $p' := 1$ to n
        $\boldsymbol{C}^{\lambda,d}(m)[p, 1, p'] := \mathsf{MAC}(2(p-1)||w_p(m), \boldsymbol{k}_0[p'])$
    $v_p := h(\mathop{\Big\|}_{t=1}^{n} \boldsymbol{C}^{\lambda,d}(m)[p, 1, t])$
    $w'_p(m) := h(w_p(m)||v_p(m))$
    for $p' := 1$ to dn
        $\boldsymbol{C}^{\lambda,d}(m)[p, 2, p'] := \mathsf{MAC}((2(p-1)+1)||w'_p(m), \boldsymbol{k}_1[p'])$
    $v'_p(m) := h(\mathop{\Big\|}_{t=1}^{dn} \boldsymbol{C}^{\lambda,d}(m)[p, 2, t])$

Figure E.2: The hashing version of Chain Signatures

ized to Chain Signatures in Figure E.2. Similar to Known-Key Chain Signatures, the input to the MAC for a given component is a number indexing the component, along with the hash of the concatenation of two values: (1) the input to the previous component, and (2) the hash of the previous component.

We can calculate the running time of the algorithm of Figure E.2 as follows. The loop over p has λ iterations. And each iteration has a hash over data of constant size, followed by a loop with n iterations (each performing a MAC of data of size $O(\log \lambda)$), and a hash of data of size $O(n)$. Then there is a hash of data of constant size, a loop with dn iterations (each performing a MAC of data of size $O(\log \lambda)$), and a hash of data of length $O(dn)$. And, as before, there is an

initial cost of $O(|m|)$ to compute $w_1(m)$. So, the time needed to compute Chain Signatures using the algorithm of Figure E.2 is $O(|m| + \lambda(n \log \lambda + n + dn \log \lambda + dn)) = O(|m| + dn\lambda \log \lambda)$.

To use the algorithms of Figures E.1 and E.2 in Known-Key Chain Signatures and Chain Signatures, we must modify the proofs of Unforgeability (5.3) and Non-Accusability (5.4) for Chain Signatures, and Weak Transferability (C.1) for Known-Key Chain Signatures, since arguments based on unique input sizes to the MACs of each section no longer work. Instead, the unique prefix p along with the length of $p \mid\mid w_p(m)$ in the computation $\mathsf{MAC}(p \mid\mid w_p(m), k_p)$ in the algorithm guarantees that the signing oracle would only have performed a given computation for a subtag in the pth section.

In these new versions of Chain Signatures and Known-Key Chain Signatures, we say that a subtag is *supported* if it is identical to the MAC of the hash value using w_p or w'_p defined recursively in Figures E.1 and E.2 over all previous components and the message. So, a verifier can determine if its subtags are supported by computing the hashes of previous components and the message and computing the MAC of this value.

Lemma 18. *If* MAC *satisfies CTA Unforgeability (5.1) and H is a family of collision-resistant hash functions, then Chain Signatures using the algorithm described in Figure E.2 satisfies λ-Completeness (5.2), Unforgeability (5.3), and Non-Accusability (5.4).*

Proof. **λ-Completeness.** As before, λ-Completeness follows by construction: signing and verification use the same algorithms to generate and check tags, so $\mathsf{Ver}^{CS}(m, \mathsf{Sign}^{CS}(m, \lambda, \boldsymbol{k}), \boldsymbol{K}_j) = \lambda$ for any λ and any choice of j.

Unforgeability. We prove the contrapositive. Suppose that adversary $\mathcal{A}$ violates Unforgeability (5.3) for some $I' \subseteq I$ with probability ϵ. We construct an adversary $\mathcal{B}$ that attempts to violates CTA Unforgeability (5.1) of the MAC (for

some key k') and collision-resistance of the family H. $\mathcal{B}$ is given MAC and VF oracles and is given oracle access to a hash function h chosen randomly from H.

$\mathcal{B}$ chooses a key k_t^* uniformly at random from the n known keys and generates a new instance of Chain Signatures by calling Gen^{CS} and replacing calls to $\mathsf{MAC}(\cdot, k_t^*)$ with calls to $\mathcal{B}$'s MAC oracle when signing and $\mathcal{B}$'s verification oracle when verifying. $\mathcal{B}$ uses h as its hash function in the execution of signing and verification. When $\mathcal{A}$ succeeds, returning m and τ, the definition of Unforgeability (5.3) states that there is some $j \in I - I'$ for which $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j) > 0)$. This means j must have (at least) a supported subtag in component 1 of section 1.

$\mathcal{B}$ returns $0||h(m)$ as its message and $\tau[1,1,t]$ as its tag. With probability $1/n$, it holds that $t = j$, since t was chosen uniformly at random and independently of j. And $\mathsf{MAC}(0||h(m), k') = \tau[1,1,t]$ in this case, because $\tau[1,1,t] = \tau[1,1,j]$ is the only subtag for j in component 1 of section 1, so it must be supported. The prefix 0 in the MAC computation guarantees that this MAC could only have been computed for the first component of the first section. There are two possible cases.

In the first case, $\mathcal{A}$ requested some $m' \neq m$ from its signing oracle such that $h(m) = h(m')$. Then $0||h(m) = 0||h(m')$, so this message was requested of the MAC oracle, and CTA Unforgeability (5.1) is not violated. But m' and m are a collision for the hash function, so $\mathcal{B}$ returns m and m' and violates collision resistance of H.

In the second case, $\mathcal{A}$ did not request any m' such that $h(m) = h(m')$, so $\mathcal{B}$ never requested $0||h(m)$ from its MAC oracle, since $\mathcal{A}$ never requested m, by assumption. So, $\mathcal{B}$ succeeds in violating CTA Unforgeability (5.1).

So, either $\mathcal{B}$ violates CTA Unforgeability (5.1) or returns a collision. And at

least one case must occur with non-negligible probability if $\mathcal{A}$ succeeds with non-negligible probability. So, $\mathcal{B}$ succeeds with non-negligible probability and Chain Signatures satisfies Unforgeability (5.3).

Non-Accusability. We prove the contrapositive. Suppose that some adversary $\mathcal{A}$ violates Non-Accusability (5.4) for some $I' \subseteq I$ with probability ϵ. Similar to the proof of Unforgeability (5.3), we construct a $\mathcal{B}$ that attempts to violates CTA Unforgeability (5.1) of the MAC and collision resistance of the family H by building a new instance of Chain Signatures and calling $\mathcal{A}$. Instead of choosing a key at random from the known keys, however, $\mathcal{B}$ chooses a key k_t from the union of the known keys and the unknown keys. When $\mathcal{A}$ succeeds and returns m and τ, the definition of Non-Accusability (5.4) states that there must be some j in $I - I'$ such that $\mathsf{Ver}^{CS}(m, \tau, \boldsymbol{K}_j)$ returns $\perp$, which means that there is some supported subtag for j in a component r that takes as input a non-supported subtag for j. Without loss of generality, let the component for the non-supported subtag for j immediately precede the component for the supported subtag for j. And let this be the lowest position in the tag at which a supported subtag in one component follows a non-supported subtag in the previous component.

There is a $1/((d+1)n)$ probability that k_t is the key used to compute this supported subtag, since t was chosen uniformly at random and independently of choice of the non-supported subtag. Suppose that the non-supported subtag for t is in component 2 of section $r-1$, followed by a supported subtag for t in component 1 of section r (the same argument applies for a non-supported subtag in component 1 of some section r followed by a supported subtag in component 2 of section r, but the indices differ accordingly). $\mathcal{B}$ returns as a message m' the input for the supported subtag in component 1 of section r: $2(r-1)||h(w'_{r-1}(m)||v''_{r-1}(m))$, where $w'_{r-1}(m)$ is the normal computed value

for component 2 of section $r-1$ in the signing algorithm of Figure E.2, and $v''_{r-1}(m) = h(\mathop{||}_{p=1}^{dn} \tau[r-1,2,p])$. This is the normal algorithm for computing the v' value in the pseudo-code, but v'' contains a non-supported subtag for j from τ. $\mathcal{B}$ returns as its tag τ' the corresponding supported subtag for j in component 1 of section r.

By the definition of Non-Accusability (5.4), when $\mathcal{A}$ succeeds, the MAC of the message returned by $\mathcal{B}$ is the value of the tag returned by $\mathcal{B}$. So, the only question is whether or not this message was requested from the MAC oracle already.

Since the message starts with $2(r-1)$, it could only be requested from the MAC oracle in an execution of the signing algorithm for component 1 of section r. There are two possible cases.

In the first case, $\mathcal{A}$ requested an m'' signed such that the execution of the signing algorithm leads to a hash collision with $v''_{r-1}(m)$ or $h(w'_{r-1}(m)||v''_{r-1}(m))$. $\mathcal{B}$ can find either collision by computing these values for all messages submitted to the signing oracle. This violates collision resistance of h.

In the second case, no such m'' was requested, so the message returned by $\mathcal{B}$ would never have been input to a MAC in $\mathcal{B}$'s simulation of the signing oracle. To see why, notice that $v''_{r-1}(m)$ is the output of a hash that takes as input a non-supported MAC. This cannot occur in the normal computation of the signing oracle. So, if no message was requested that leads to the same value of $v''_{r-1}(m)$ or $h(w'_{r-1}(m)||v''_{r-1}(m))$, then m' was never requested of the signing oracle. And this means that the message and tag returned by $\mathcal{B}$ violated CTA Unforgeability (5.1) of the MAC.

Thus, either m' and τ' violate CTA Unforgeability (5.1) of the MAC with non-negligible probability or a collision is found with non-negligible probability. So,

$\mathcal{B}$ succeeds with non-negligible probability, and Chain Signatures satisfies Non-Accusability (5.4). □

As before, Weak Transferability (C.1) of Known-Key Chain Signatures follows from a parallel argument to Non-Accusability (5.4) of Chain Signatures, but with a success probability of ϵ/n instead of $\epsilon/(d+1)n$.

BIBLIOGRAPHY

[1] A. S. Aiyer, L. Alvisi, R. A. Bazzi, and A. Clement. Matrix signatures: From MACs to digital signatures in distributed systems. In *Proceedings of the 22nd International Symposium on Distributed Computing*, volume 5218 of *Lecture Notes in Computer Science*, pages 16–31, Berlin, Germany, 2008. Springer.

[2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570. IEEE Computer Society, 1976.

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

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology—CRYPTO 2001, Proceedings of the 21st Annual International Cryptology Conference*, pages 1–18. Springer, August 2001.

[5] 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.

[6] 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.

[7] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology—CRYPTO 1996, Proceedings of the 16th Annual International Cryptology Conference*, volume LNCS 1109, pages 1–15. Springer, August 1996.

[8] M. Bellare, O. Goldreich, and A. Mityagin. The power of verification queries in message authentication and authenticated encryption. *Cryptology ePrint Archive*, 2004. Report 2004/309, 2004.

[9] M. Bellare, J. Kilian, and P. Rogaway. The security of cipher block chaining. In Y. Desmedt, editor, *Advances in Cryptology—CRYPTO 1994, Proceedings of the 14th Annual International Cryptology Conference*, volume 839 of *Lecture Notes in Computer Science*, pages 341–358, Berlin, Germany, 1994. Springer.

[10] 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.

[11] 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.

[12] D. Boneh, G. Durfee, and M. Franklin. Lower bounds for multicast message authentication. In B. Pfitzmann, editor, *Advances in Cryptology—EUROCRYPT 2001, Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*, volume 2045 of *Lecture Notes in Computer Science*, pages 437–452, Berlin, Germany, 2001. Springer.

[13] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

[14] 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.

[15] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science, FOCS 2001*, pages 136–145, Las Vegas, Nevada, USA, 14–17 October 2001. IEEE Computer Society.

[16] R. Canetti. Universally composable signature, certification, and authentication. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW)*, pages 219–235, Pacific Grove, CA, USA, 28–30 June 2004. IEEE Computer Society.

[17] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *IEEE INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 706–716, New York, NY, March 1999. IEEE.

[18] 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.

[19] R. Canetti and T. Rabin. Universal composition with joint state. In D. Boneh, editor, *Advances in Cryptology—CRYPTO 2003, Proceedings of the 23rd Annual International Cryptology Conference*, volume 2729 of *Lecture Notes in Computer Science*, pages 265–281, Berlin, Germany, 2003. Springer.

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

[21] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, LA, February 1999. USENIX Association, Co-sponsored by IEEE TOCS and ACM SIGOPS.

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

[23] L. S. Charlap, H. D. Rees, and D. P. Robbins. The asymptotic probability that a random biased matrix is invertible. *Discrete Mathematics*, 82(2):153–163, June 1990.

[24] D. Chaum. The dining cryptographers problem: unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[25] D. Chaum and S. Roijakkers. Unconditionally secure digital signatures. In A. Menezes and S. A. Vanstone, editors, *Advances in Cryptology—CRYPTO 1990, Proceedings of the 10th Annual International Cryptology Conference*, volume 537 of *Lecture Notes in Computer Science*, pages 206–214, Berlin, Germany, 1990. Springer.

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

[27] 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.

[28] R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In H. Krawczyk, editor,

Advances in Cryptology—CRYPTO 1998, Proceedings of the 18th Annual International Cryptology Conference, volume 1462 of *Lecture Notes in Computer Science*, pages 13–25, Berlin, Germany, 1998. Springer.

[29] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *Advances in Cryptology—CRYPTO 1990, Proceedings of the 10th Annual International Cryptology Conference*, volume 435 of *Lecture Notes in Computer Science*, pages 307–315, Berlin, Germany, 1990. Springer.

[30] Y. Desmedt, Y. Frankel, and M. Yung. Multi-receiver/multi-sender network security: efficient authenticated multicast/feedback. In *IEEE INFOCOM '92: Proceedings of the 11th Annual Joint Conference of the IEEE Computer and Communications Societies on One World through Communications*, pages 2045–2054, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[31] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[32] D. Dolev. The Byzantine Generals strike again. *Journal of Algorithms*, 3(1):14–30, March 1982.

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

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

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

[36] 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.

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

[38] O. Goldreich. *Foundations of Cryptography*, volume I. Cambridge University Press, 2001.

[39] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, April 1988.

[40] G. Hanaoka, J. Shikata, Y. Zheng, and H. Imai. Unconditionally secure digital signature schemes admitting transferability. In T. Okamoto, editor, *Advances in Cryptology—ASIACRYPT 2000, Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security*, volume 1976 of *Lecture Notes in Computer Science*, pages 130–142, Berlin, Germany, 2000. Springer.

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

[42] 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 1995, Proceedings of the 15th Annual International Cryptology Conference*, volume 963 of *Lecture Notes in Computer Science*, pages 339–352. Springer, 27–31 August 1995.

[43] 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.

[44] 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.

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

[46] T. Johansson. Further results on asymmetric authentication schemes. *Information and Computation*, 151(1–2):100–133, May 1999.

[47] 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.

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

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

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

[51] 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.

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

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

[54] 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.

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

[56] Netfilter. See http://www.netfilter.org.

[57] OpenBSD. See http://www.openbsd.org.

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

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

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

[61] B. Pfitzmann and M. Waidner. Unconditional Byzantine agreement for any number of faulty processors. In A. Finkel and M. Jantzen, editors, *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, volume 577 of *Lecture Notes in Computer Science*, pages 339–350, Berlin, Germany, 1992. Springer.

[62] PF: Firewall redundancy with CARP and pfsync, Accessed July 2008. See http://www.openbsd.org/faq/pf/carp.html.

[63] 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.

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

[65] 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.

[66] R. Safavi-Naini, L. McAven, and M. Yung. General group authentication codes and their relation to "Unconditionally-secure signatures". In F. Bao, R. H. Deng, and J. Zhou, editors, *Public Key Cryptography—PKC 2004, 7th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2947 of *Lecture Notes in Computer Science*, pages 231–247, Berlin, Germany, 2004. Springer.

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

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

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

[70] Fips 180-1. Secure hash standard. Federal Information Processing Standard (FIPS), Publication 180-1, National Institute of Standards and Technology, US Department of Commerce, Washington, D.C., April 1995.

[71] 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.

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

[73] J. Shikata, G. Hanaoka, Y. Zheng, and H. Imai. Security notions for unconditionally secure signature schemes. In L. R. Knudsen, editor, *Advances in Cryptology—EUROCRYPT 2002, Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*, volume 2332 of *Lecture Notes in Computer Science*, pages 434–449, Berlin, Germany, 2002. Springer.

[74] 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.

[75] T. K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

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

[77] Trusted computing group. See http://www.trustedcomputinggroup.org.

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

[79] 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.

[80] 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.

[81] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.

[82] 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.