TOWARDS FAULT-TOLERANT AND SECURE ON-LINE SERVICES

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

Lidong Zhou

May 2001

TOWARDS FAULT-TOLERANT AND SECURE ON-LINE SERVICES

Lidong Zhou, Ph.D.
Cornell University 2001

Integrating fault tolerance and security is crucial for building trustworthy on-line services. Such integration is studied in this dissertation through the design and implementation of COCA (Cornell On-line Certification Authority), a fault-tolerant and secure on-line certification authority. COCA maintains a service private key to sign the responses it sends to clients, and achieves availability using replicated servers that employ threshold cryptography and store shares of the service private key. Periodic share refreshing, coupled with periodic recovery of server states, defends against so-called mobile adversaries which move from one server to another. COCA is designed for a weak system model: no assumptions are made about server speed or message delay, and communications are assumed to employ links that are intermittent. The result is a service with reduced vulnerability to attacks because, by their nature, weaker assumptions are more difficult for adversaries to invalidate. COCA further employs an array of defense mechanisms specific to denial of service attacks. COCA runs both on a local area network and on the Internet. Performance measurements of COCA under simulated denial of service attacks demonstrate the effectiveness of COCA's defenses.

Biographical Sketch

Lidong Zhou was born in 1971. He received his B.S. degree in Computer Science from Fudan University (Shanghai, China) in 1993. After two years as a Masters student at Fudan University, he entered the Ph.D. program at Cornell University with a major in Computer Science and a minor in Electrical Engineering. In 1998, he received an M.S. degree in Computer Science from Cornell University.

Dedicated to the memory of my mother

Acknowledgements

I would like to express my gratitude to my advisors, Professors Fred B. Schneider, Robbert van Renesse, and Zygmunt J. Haas, who provided invaluable guidance throughout my years at Cornell. It has been a great honor working with them. Special thanks to Fred for his unrelenting endeavor in shaping me into a computer scientist. His persistent pursuit of perfection and his deep insights into various subjects have always been an inspiration to me.

My deepest gratitude goes to my parents for their love, for their encouragement, for their support, for their sacrifice, and for their faith in me. They have been the origin of my strength and will always be. I would also like to thank my wife for all the love she has given me, for her understanding, and for putting up with me and standing by me during all these years.

My thanks also go to Yaron Minsky, Xiaoming Liu, Nikolay Mateev, Chris Hawblitzel, Fred Smith, and Úlfar Erlingsson for their friendship and for the enlightening discussions we had on various research topics. I am also in debt to Professors Nick Trefethen, Ken Birman, Sam Toueg, and Greg Morrisett for their help and advice, especially during my early years at Cornell.

I am grateful to Dag Johansen, David Kotz, and Keith Marzullo for loaning hardware that enabled my experiments on the Internet. Mike Reiter, Andrew Myers, Miguel Castro, Stuart Stubblebine, Christian Cachin, Li Gong, Catherine Meadows, Ueli Maurer, and Yacov Yacobi provided helpful feedback and comments on drafts of the papers that lead to this dissertation.

This work is supported in part by ARPA/RADC grant F30602-96-1-0317, AFOSR grant F49620-00-1-0198, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Material Command USAF under agreement number F30602-99-1-0533, National Science Foundation Grant 9703470, and a grant from Intel Corporation. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A growing reliance on on-line services demands that they be *trustworthy*—they must render service despite component failures as well as attacks by adversaries. Therefore, a trustworthy on-line service must be both fault-tolerant and secure.

Fault tolerance and security started as two separate research fields, leading to related but different nomenclatures, methodologies, and technologies. In this dissertation, we take a step towards unifying the two fields. In particular, we investigate how to marry fault tolerance and security by undertaking the design and implementation of a fault-tolerant and secure on-line certification authority.

1.1 Achieving Fault Tolerance

A service is *fault-tolerant* if it continues to operate correctly, despite occurrences of certain failures. *Failure models* characterize classes of failures that might be tolerated. Here are some common failure models for processors:[1]

[1] Other failure models have also been proposed, see [51] for a more complete list.

Byzantine Failures: A faulty processor deviates arbitrarily from its specified protocols [86, 67]. This class is also called *arbitrary failures* or *malicious failures*.

Crash Failures: A compromised server might halt prematurely. Before it halts, it behaves correctly [50].

Fail-Stop Failures: A processor fails by crashing, and the failure can be detected by other processors [96].

Services designed for a specific failure model will work provided the environment in which that service is deployed conforms to that model. But if failures occur that are not characterized by the failure model, then the service might fail.

Byzantine failures are the most severe and, not surprisingly, the most difficult to tolerate. A service that tolerates Byzantine failures is desirable because no assumptions are being made about the behavior of faulty components—all failures are thus admitted by the model. Crash failures and fail-stop failures are often referred to as *benign failures.* While benign failures are easier to tolerate, these failure models often inadequately characterize failures that could and do happen in reality, leading to failures of services that have been designed for such models.

Fault tolerance typically requires replication [97]. A fault-tolerant service is implemented by replicating *servers* that are assumed to fail independently. The probability that a set of these servers all fail is thus the product of the probability that each element of the set fails. As the cardinality of the set increases, this probability becomes lower, eventually becoming negligible.

Protocols coordinate the servers comprising a fault-tolerant service. These protocols are affected by assumptions related to server speed, message-delivery delay, and server-clock drift. Two distinct system models have been used to characterize the assumptions:

Synchronous System Model: There are known bounds on server speed and message delay. Every server has a local clock that has bounded drift with respect to real time.

Asynchronous System Model: There is no bound on server speed or message delay, nor is there a bound on local clock drift.

A service built for the synchronous system model could fail if the defining assumptions of that model become invalid. In contrast, the asynchronous system model entails no assumptions, so there is nothing to be invalidated. However, the design of fault-tolerance services that work in an asynchronous system model is often difficult and sometimes impossible [34]. And, no real-time guarantees can be made for services implemented under the asynchronous system model.

The replicated state machine approach [66, 98] is a general way of coordinating replicated servers. A service is defined as a state machine, and a fault-tolerant version of the service is obtained by replicating that state machine. Clients submit *requests* to the service for processing. The protocols that coordinate the servers ensure that all non-faulty servers process the same requests in the same order, thereby maintaining identical sequences of states. With a sufficient number of non-faulty servers, responses from faulty servers are masked by voting on outputs produced by the individual servers for each given request.

The generality of the replicated state machine approach comes with a cost. Its protocols must implement the following strong guarantees:

- **Agreement:** All non-faulty servers receive the same requests.

- **Order:** All non-faulty servers process requests in the same order.

Agreement and Order are unnecessarily strong for some applications; simpler and more efficient solutions could instead be used. Here is an example that does not require such strong guarantees.

Shared Variable Service: A service stores the value for some shared variable. A copy of the value, together with a version number of that copy, is replicated on servers that implement the service. The service supports both update and query operations:

- Clients update the variable by providing a new value and version number.
- Clients query the service for the current value and version number associated with the variable.

Clients expect that query returns the value with the largest version number over all that have ever been written by update.

Although such a service could be implemented using the state machine approach, a cheaper approach called a *quorum system* [45, 105, 41] suffices. In a quorum system, servers are organized into sets called *quorums*, where every pair of quorums intersect.[2] And, instead of requiring that all non-faulty servers receive and process the same requests (as in the state-machine approach), a quorum system requires only that, for each request, some quorum of servers receives and processes the request.

Assuming that there are in total $2t + 1$ servers and that crash failures by at most t must be tolerated, a quorum system $\mathcal{Q}$, where each quorum in $\mathcal{Q}$ consists

[2]Depending on the failure model and the type of services to be implemented, different kinds of quorum systems with different intersection properties have been proposed (e.g., Byzantine quorum systems [69] for tolerating Byzantine failures).

of exactly $t + 1$ servers, implements the Shared Variable Service, as follows. (Note that every two quorums in $\mathcal{Q}$ intersect because $(t + 1) + (t + 1) > 2t + 1$ holds):

update Implementation:

1. The client sends to all servers an update request that contains a value for the variable and a proposed version number.[3]
2. Each server, upon receiving an update request, updates its copy of the variable with the new value and version number if and only if the proposed version number is larger than the one that the server currently stores. The server then sends an acknowledgment back to the client. The acknowledgment signifies that the value stored by that server has a version number at least as large as that in the request.
3. Request execution is considered complete when the client receives acknowledgments from a quorum of servers.

query Implementation:

1. The client sends to all servers a query request.
2. Each server, upon receiving a query request, sends to the client the value and version number the server currently stores.

[3]How an appropriate version number is picked depends on the application. In cases where only one client is allowed to update the value, the client might use the value of a local counter as the version number; that counter is increased after each update request. When multiple clients can update the value, a client might first query the service for the current version number of the value and then pick a new version number that is larger than the current version number. Client identifiers could be attached as the lower part of the version number so that no two clients pick the same version number, even for concurrent updates.

3. Request execution is considered complete when the client receives responses from a quorum of servers and has picked the value with the largest version number among those received.

The above implementation of the Shared Variable Service ensures that a query always returns the value associated with the largest version number among those that have appeared in update requests:[4] For any query request, let Q be the quorum of servers from which the client gets responses for the request, and let Q' be the quorum that has processed the update request preceding the query and containing the largest version number. Because every pair of quorums in $\mathcal{Q}$ intersect, there exists a server that belongs to both Q and Q'. This server must have responded to the query request with the value of the largest version number; this value is then chosen by the client as the response for the query request.

1.2 Enforcing Security

Security is concerned with protecting a service against adversaries who launch *attacks*. *Adversary models*—analogous to the failure models of fault tolerance—have been proposed to characterize the power of adversaries. Two such models[5] are:

[4] To simplify the discussion, no concurrent requests are being considered here. The semantics of the Shared Variable Service becomes considerably more complicated when concurrent requests are possible. How to define and implement semantics that takes concurrency into account is discussed in Chapter 3.

[5] Adversary models used in cryptography usually specify the computational power of adversaries. The models here do not. However, assumptions on computational power are implicit in the cryptographic schemes used in this dissertation. In particular, we assume that an adversary cannot break the cryptographic schemes used.

Passive Adversaries: A *passive adversary* learns information from a system component but is unable to change the behavior of that component.

Active Adversaries: An *active adversary* controls the behavior of a system component, in addition to learning information from that component. Active adversaries are also called *Byzantine adversaries*.

An adversary can attack not only servers, but also communication links.

The notions of Passive Adversaries and Active Adversaries apply to both servers and communication links. For servers, a passive adversary can steal information stored on a server, while an active adversary can also cause the server to deviate from the specified protocols in any way that adversary desires. For communication links, a passive adversary can eavesdrop on the communication links to gain information from messages being transmitted; an active adversary can also alter what is being carried on the link (e.g., inserting, deleting, replaying, reordering, and modifying messages).

A comparison between failure models and adversary models reveals a fundamental difference between fault tolerance and security: information disclosure. Of no concern to fault tolerance, *confidentiality*, which concerns what information may be disclosed to which entities, plays a crucial role in security. While replication is a means to implement fault tolerance, it only makes matters worse for confidentiality. Other methods are needed. We now survey those.

Confidentiality Through Encryption

Confidentiality can be implemented using encryption that transforms *cleartext* into *ciphertext*. Ciphertext conceals cleartext in a manner that depends on an *encryption* *algorithm* and a *key*; cleartext can be recovered from ciphertext by a *decryption algorithm*, again using a key.

There are two classes of encryption/decryption algorithms: *secret key cryptography* (also called *symmetric key cryptography*) and *public key cryptography* [29] (also called *asymmetric key cryptography*). In secret key cryptography, the same *secret key* is used with both the encryption and decryption algorithms. In public key cryptography, two different keys are used—a *private key* is used with the decryption algorithm, and a *public key* is used with the encryption algorithm. A private key cannot be inferred from the corresponding public key. Public keys can thus be broadly disseminated, but private keys should be kept confidential by their owners.

Public key cryptography offers functionality beyond the capabilities of secret key cryptography. For example, public key cryptography can be used to implement *digital signatures* [29, 74] which, analogous to signatures on paper documents, ensure the authenticity and integrity of the information being signed. A client A can generate a digital signature for a message by invoking a *signature generation algorithm* using A's private key k_A.[6] Any client B with knowledge of A's public key K_A can verify the integrity of the message by applying a *signature verification algorithm* to the signature. Because private key k_A used to generate the signature is known only to client A, no one except A can produce such a signature. Any client who knows K_A (presumed to be publicly available) can verify the signature and be convinced that the message was signed by A and has not been subsequently altered by others.

Use of cryptography requires proper key management. For secret key cryptography, two communicating clients must share the same secret key. Manually distribut-

[6] In practice, a one-way hash function [75] is first applied to a message, and this hash is what is signed. We ignore such implementation details here.

ing shared keys for every pair of clients in a large network is unrealistic. Instead, a trusted *key distribution center* (KDC) [82] is used. The KDC shares a secret key with every client it serves and, when requested, creates a new shared secret *session key* for use by a pair of clients. Note that, because secret keys shared with clients are stored by the KDC and because the KDC knows the session keys, compromise of the KDC is disruptive.

Public key cryptography would seem to eliminate the need for a trusted entity like the KDC because public keys, which need to be distributed, are not kept secret. But while confidentiality of public keys is not a concern, integrity is. Consider the case where A wants to encrypt a message readable only by B. If A somehow becomes convinced that K_C, the public key of an adversary C, is the public key of B and uses K_C for encryption, then the encrypted message could be decrypted by C (and not B). A is thus led to disclose information to an adversary.

A standard defense against such attacks is to deploy a public key infrastructure involving a trusted *certification authority* (CA), whose public key is known to all. The CA certifies a binding between a name and a public key (and possibly other attributes) by signing, using the CA's private key, a *certificate* [64] specifying this binding. CA-signed certificates can be verified by clients using the CA's public key. And, as long as the private key of the CA remains confidential, no one except the CA can sign certificates. By using a certificate that binds B to K_B to check B's public key, A cannot be misled by adversary C.

A binding between a name and a public key could become invalid, for example, when the corresponding private key is compromised or disclosed. Standard approaches to implementing certificate invalidation include attaching unique identifiers and expiration dates to certificates in conjunction with the CA periodically issuing certificate revocation lists (CRLs) that enumerate invalid, but not yet expired, certificates. But the gap between when the CA learns that a certificate has become invalid and when users of the certificate find out that fact constitutes a vulnerability, because an adversary can exploit the window during which an outdated binding is taken to be valid.[7]

One way to reduce such a vulnerability is to consult an on-line CA. Now, clients can report certificate invalidation immediately, and they can query the on-line CA about the validity of a certificate just before using that certificate. The main subject of this dissertation is such an on-line CA. Note that, like KDC, compromise of a CA could be devastating to the clients served by that CA.

Confidentiality Through Secret Sharing

Encryption does not completely solve the problem of implementing confidentiality. Encryption simply relocates the confidentiality requirement on the information being encrypted to a confidentiality requirement on the secret key or the private key used for encryption [99]. The relocation of the confidentiality requirement is accompanied by a relocation of trust. For example, when an unencrypted confidential message is sent over a communication link, trust is being placed on the communication link—the sender and the receiver trust that the message will not be disclosed while in transit on the link. If the message is encrypted before being transmitted, then trust is relocated from the communication link to servers storing the encryption key—a server is now trusted to keep the encryption key secret, and the communication link need no longer be trusted to keep the message confidential. Such a relocation of

[7] Other gaps (e.g., between when a private key is compromised and when the CA learns the fact) also constitute vulnerabilities.

trust makes sense when the probability that servers are compromised by a passive adversary is considered lower than the probability that a communication link is compromised by a passive adversary.

The use of replication for fault tolerance is based on the premise that the probability a set of servers all fail is considerably lower than that the probability a single server fails. Following this same philosophy, confidentiality can be implemented by distributing a secret to multiple servers in a way that the secret is disclosed only if some subset of servers are compromised. (The probability of this happening is presumed to be lower than that of one server being compromised).[8] This can be achieved by *secret sharing* [7, 101]. Instead of storing the secret at one server, an $(n, t+1)$ secret sharing scheme divides the secret into n *shares* in such a way that the secret can be reconstructed only from $t+1$ or more shares. If shares are distributed among the different servers, then confidentiality of the secret is preserved provided that the number of shares distributed to corrupted servers does not exceed t.

Secret sharing alone is insufficient to defend against *mobile adversaries* [85] which attack, compromise, and control a server for a limited period of time before moving to the next victim. A mobile adversary might not be able to compromise more than t servers within a short period of time, but it might be able to do so over a long period of time, thereby obtaining enough shares from servers it has compromised to reconstruct the secret.

The defense here is *proactive secret sharing* [57, 54]. Proactive secret sharing

[8] The argument about probabilities in the context of fault tolerance hinges on the fundamental assumption that servers fail independently. This independence assumption might not be appropriate for server compromise, because a common vulnerability could lead to a single attack corrupting all servers. Therefore, it is crucial to create diversity among replicas, for example, by deploying different servers on different platforms.

allows servers periodically to generate new shares from old ones and to refresh old shares with new ones, without disclosing or changing the secret itself. The new shares are computed in such a way that old shares cannot be combined with new shares to reconstruct the secret. A mobile adversary is thus challenged to compromise more than t servers in the relatively short time between executions of the protocol that refreshes shares.

Finally, it might seem that implementing confidentiality would be of no concern to services that do not store confidential information. This is not necessarily so. As long as a service employs cryptography (e.g., for signing messages), confidentiality may come into play because the secret or the private keys used will need to remain confidential. In fact, confidentiality of any data maintained by the service can be reduced to maintaining the confidentiality of the secret or private key of the service—just encrypt the service-maintained data using that key. An important aspect of the service presented in this dissertation is how this service manages a piece of confidential data, namely, a service private key.

1.3 Marrying Fault Tolerance and Security

This dissertation describes the design and implementation of COCA (Cornell On-line Certification Authority), a fault-tolerant and secure on-line CA. But the significance of the effort goes beyond just building an on-line CA. COCA served as a vehicle to explore a general framework for integrating fault tolerance and security and for investigating issues that arise during the integration of these two essential elements of trustworthiness. We believe insights from COCA extend to other on-line services that are required to be highly trustworthy.

COCA uses replication and stores certificates on its servers. Certificates resemble variables in the Shared Variables Service. Like the implementation of update and query sketched in Section 1.1, COCA employs a quorum system rather than the replicated state machine approach. COCA supports Update and Query requests, where Update causes a new certificate specifying a binding to be signed and issued, and Query returns a currently stored certificate.

COCA maintains a *service private key* for signing certificates and for signing responses to clients. Secret sharing is used to generate shares for this service private key, with these shares distributed to COCA servers. But reconstructing the service private key on a server before each use would expose the service private key if that server is compromised, so, COCA employs *threshold cryptography* [27, 8, 28], whereby servers store shares of the service private key and perform cryptographic operations (e.g., generating a digital signature) using that service private key without ever materializing the key from the shares.

COCA is designed for a system model with weak assumptions, thereby exhibiting reduced vulnerability to attacks. Any assumption that a service relies on adds to the power of its adversaries, who can then disrupt the service by invalidating that assumption. For example, a service designed to work under the synchronous system model might be corrupted by denial of service attacks that delay message processing or message delivery long enough to violate the defining assumptions of the synchronous system model. COCA makes no timing assumptions; it is designed for the asynchronous system model, so it is less vulnerable to such denial of service attacks. Prior to our work, proactive secret sharing schemes existed only for some synchronous system model.

Besides weak assumptions, COCA servers employ various defense mechanisms to lower the impact of denial of service attacks. Vulnerability to denial of service attacks often arises when the cost to process a request (or message) outweighs the cost for an adversary to make the request (or send the message), as previously noted, for example, in [58, 72, 73]. A defense against denial of service attacks can thus be based on eliminating the imbalance. This philosophy led us to instantiate in COCA the following classic defense mechanisms for combating denial of service attacks:

1. Processing only those requests that satisfy authorization checks.

2. Grouping requests into classes and multiplexing resources so that demands from one class cannot impact processing of requests from another [46, 109, 76, 77].

3. Caching results of expensive cryptographic operations [84, 14].

Although resource-clogging denial of service attacks are not completely ruled out by COCA's defenses, this dissertation sheds light on the effectiveness of the classic defenses by presenting performance measurements for COCA under simulated denial of service attacks. And our data demonstrate that launching a successful attack against COCA is harder with these mechanisms in place.

For a long-running service like COCA, servers might get compromised and then recovered. Ideally, *server recovery* will occur right after compromise is detected. However, compromise of a server is not always detected immediately or even ever. Therefore, periodic server recovery, even when no compromise is detected, is prudent. Such recovery is called *proactive recovery* [85].

Proactive recovery effectively reduces the *window of vulnerability*—the time period during which an adversary has to complete its attack in order to disrupt the service. Proactive recovery divides the lifetime of a service into a series of intervals, where the interval between any two consecutive executions of the proactive recovery protocol constitutes a window of vulnerability. Any incomplete attack launched by an adversary before execution of the proactive recovery protocol is mooted by proactive recovery; the adversary is forced to re-initiate its attack. Without proactive recovery, the window of vulnerability for a service would be the entire lifetime of that service, offering adversaries ample opportunities to disrupt a service by compromising first one server and then another.

A prototype of COCA has been implemented. This dissertation not only describes the design and implementation of that prototype, but also documents our experiences with running it on a local area network and on the Internet (with servers at the University of Tromsø (Norway), Cornell, Dartmouth, and U.C. San Diego).

1.4 Road Map of this Dissertation

This dissertation is organized as follows. Chapter 2 describes our proactive secret sharing protocol for the asynchronous system model. Correctness requirements are given for proactive secret sharing protocols in the asynchronous system model and a step-by-step development of the protocol itself is presented.

Chapter 3 presents the design details of COCA. A specification for the service offered by COCA is described, together with the protocols that realize this specification. The presentation emphasizes the technical challenges and how we addressed them.

Chapter 4 gives implementation details for COCA, as well as describing performance evaluation experiments for COCA deployments on a local area network and on the Internet. Performance both in the normal circumstances and under certain simulated denial of service attacks is presented and analyzed.

Finally, Chapter 5 contains a general framework, derived from the design of

COCA, for building a fault-tolerant and secure on-line service. General approaches that facilitate the implementation of this framework are discussed.

Chapter 2

Asynchronous Proactive Secret Sharing

Achieving availability and confidentiality of a secret (such as a key) is challenging when failures and attacks are possible: A secret stored on a single server becomes unavailable if that server crashes and can be disclosed if that server becomes compromised. Replication of a secret on multiple servers increases availability but also increases the chances of disclosure.

A standard technique for resolving tensions between availability and secrecy is to use secret sharing. Instead of storing a secret on all servers, a set of shares are generated from the secret. We call such a set of shares a *sharing* of that secret. The shares are then distributed to servers. Proactive secret sharing (PSS), by performing periodic *share refreshing*, defends the confidentiality of the secret against mobile adversaries. During share refreshing, servers create a new and independent sharing of the same secret and replace old shares with new ones. Old shares are then deleted from servers, so that these old shares are not exposed to an adversary if these servers are compromised in the future. Because new shares cannot be combined with old shares to reconstruct the secret, an adversary is forced to obtain enough shares during the short period between two consecutive executions of share refreshing.

All PSS protocols appearing in the literature to date assume a synchronous system model. Any assumption constitutes a vulnerability, and the assumptions of the synchronous system model, which could be invalidated by denial of service attacks, is no exception. This chapter introduces a PSS protocol that does not require a synchronous system model; we refer to this protocol as an *asynchronous proactive secret sharing* (APSS) protocol.

Instead of generating a single new sharing from a single old sharing, our APSS protocol generates up to n new sharings from up to n old sharings. This somewhat relaxed requirement avoids the need to solve an agreement problem in the implementation of APSS—this is significant because agreement is known to be difficult in the asynchronous system model we are assuming. Each of the different new sharings has a unique label. When carrying out an operation using a sharing, servers use the label to indicate which sharing is being used.

Our APSS protocol can also be used to construct proactive protocols for threshold cryptography. Because our APSS protocol is constructed using abstract modules rather than specific cryptographic algorithms, these abstract modules can be instantiated with concrete implementations from threshold cryptography, leading to proactive threshold cryptography schemes. As we show in Chapter 3, a proactive threshold cryptography scheme derived from our APSS protocol has been incorporated into COCA to defend against mobile adversaries.

We describe the APSS protocol as a series of refinements, starting from a simple protocol that works in a benign environment. Each refinement step enables the protocol to tolerate more powerful adversaries. The presentation thus makes clear why each piece of the protocol is needed and how the various defenses are integrated into the protocol.

The chapter is organized as follows. Section 2.1 specifies the system model and the correctness requirements for an APSS protocol. Section 2.2 describes the cryptographic building blocks used in our protocol. Derivation of our APSS protocol is presented in Section 2.3, followed by a discussion of related work in Section 2.4. Finally, concluding remarks are presented in Section 2.5.

2.1 System Model and Correctness Requirements

Consider a service comprising n servers connected through a network. The service maintains a secret s; shares of that secret are distributed among servers. Each server has a public/private key pair and knows the public keys of all other servers.

We intend the protocol for use in an environment like the Internet. The protocol must tolerate failures and defend against malicious attacks that target servers and communication links, as follows.

Active Server-Adversaries: Servers are either *correct* or *compromised*. We assume a compromised server might stop executing, deviate arbitrarily from its specified protocols (i.e., Byzantine failure), and/or might disclose or change information stored locally. We also assume

- At most t of the n servers are ever compromised during each protocol-defined window of vulnerability, where $3t + 1 \leq n$ holds.
- Various cryptographic schemes (e.g., public key cryptography) the service employs are secure.

Active Link-Adversaries: We assume an adversary can launch eavesdropping, message insertion, corruption, deletion, reordering, and replay attacks, provided the following fair link assumption is not violated: A *fair link* is a communication channel that does not necessarily deliver all messages sent. But if a process sends infinitely many messages to a single destination then infinitely many of those messages are correctly delivered.

Although making even weaker assumptions about communication networks might be attractive, without making some assumption, comparable to our fair link assumption, an adversary could prevent servers from communicating with each other or with clients. No protocol could be expected to work there.

Asynchronous System: we assume the asynchronous system model. There is no bound on message delivery delay, server speed, or local clock drift.

As in prior work on PSS [57], we assume that all keys are stored in a tamper-proof cryptography co-processor and that all operations involving these keys are performed by this co-processor. Thus, for any server, the confidentiality of that server's private key, as well as the integrity of any public keys used by that server, cannot be violated, even if that server is compromised. Under this assumption, after server recovery, which, in the context of this chapter, involves a server rebooting from a clean copy of the system and having all its shares refreshed by an execution of proactive secret sharing, all Trojan horses left behind by the adversary are excised.

To eliminate the assumption about storing keys in tamper-proof cryptography co-processors, servers might refresh all their keys during server recovery. But then there must be a way for a server p to inform others what its new public key is. Server p could use its old private key to sign the message that notifies others about its new public key. However, if that server's private key is disclosed, then an adversary could mislead other servers about p's new public key. Alternatively, servers could rely on trusted administrators to propagate new public keys among servers through secure out-of-band communication channels. One way to establish such a secure communication channel is to have each administrator maintain a public/private key pair, with the public key known to all other administrators (and all servers) and the private key used to sign the notification message for the new public key of that server. Because the private key is only used to sign the notification messages, it is reasonable to assume that the private key stays off-line most of time and is immune to any on-line attacks (such as attacks by mobile adversaries). Other approaches to refreshing keys have been presented in [10].

2.1.1 Defining the Window of Vulnerability

The duration of a window of vulnerability cannot be characterized in terms of real time here due to the assumption of an asynchronous system, so the window of vulnerability for this service is defined in terms of events related to periodic executions of the APSS protocol. In theory, using protocol events to delimit a window of vulnerability affords attackers leverage. Denial of service attacks that slow servers and/or increase message delivery delays expand the real-time duration for the window of vulnerability, creating a longer period during which adversaries can try to compromise more than t servers. But in practice, we expect assumptions about timing can be made for those portions of the system that have not been compromised.[1] Given such information about server execution speeds and message-delivery delays,

[1] A server that violates these stronger execution timing assumptions might be considered compromised, for example.

real-time bounds on the window of vulnerability can be computed.

To define the window of vulnerability for our protocol precisely, the following notions are needed. An execution of proactive secret sharing where servers generate new shares from old ones is called a *run*. To distinguish new shares from old ones and to distinguish different runs, *version numbers* are assigned to shares and runs as follows (a sharing is assigned the same version number as its shares):

- Servers initially have shares of version number 0.
- If a run is executed with shares of version number v, then this run and the resulting new shares are assigned version number $v + 1$.

A server initiates[2] a run periodically (e.g., at 8 AM every morning) based on its local clock[3] or when instructed by its administrator (e.g., in response to a detected compromise). A server p might also participate in a run initiated by another server. To avoid denial of service attacks that cause frequent invocation of proactive secret sharing, a server will refuse to participate in the APSS protocols unless enough time has elapsed on its clock since the protocols last executed.

Run v *starts* when a correct server initiates or participates in run v. Run v *terminates locally* on a (correct) server when that server deletes (i) shares it stores with version number $v - 1$ and (ii) any secret information generated directly from these shares. Run v *terminates* when this run terminates locally on all servers that

[2] Here, we also assume that every server is rebooted from clean code when initiating a run or participating in a run.

[3] We require that local clocks on correct servers advance at a reasonable rate but not that local clocks are synchronized. Consequently, servers could initiate the same run at different times, although in practice differences among local clocks are often small. Note that our use of local clocks here do not violate our Asynchronous System assumption.

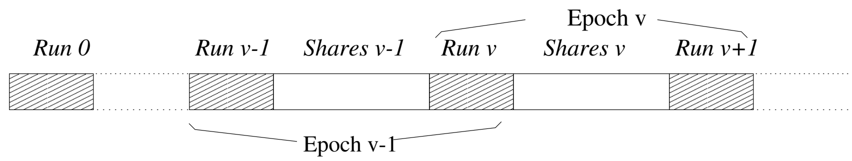

Figure 2.1: Epoch.

have remained correct since run v started. Every server p maintains *current version number* cvn_p, which reflects the version number of the last run that has terminated locally on server p, as follows:

- Initially, cvn_p is set to 0.
- When run v terminates locally on server p, cvn_p is advanced to v.

Define *epoch* v to be the interval from the start of run v to the termination of run $v+1$. Because new sharings are independent from old sharings, an adversary must obtain enough shares that have the same version number in order to reconstruct the secret. To collect shares of version number v, an adversary must compromise servers during epoch v—these shares are created in run v and deleted in run $v+1$ by servers that remain correct during epoch v. Every epoch thus constitutes a window of vulnerability. Due to Active Server-Adversaries, there are at most t servers compromised in every epoch. Figure 2.1 illustrates how epochs relate to different runs and different versions of shares.

A server is regarded as *correct in an epoch* v if and only if this server remains correct throughout this epoch. Similarly, a server is regarded as *correct in a run* v if and only if this server remains correct throughout this run. Servers that are correct either in epoch $v-1$ or in epoch v are, by definition, correct in run v, because run v belongs to both epoch $v-1$ and epoch v.

2.1.2 APSS Correctness Requirements

A service that maintains a secret s, whose shares are distributed to servers comprising the service and are refreshed by a proactive secret sharing protocol, must satisfy the following correctness requirements:

APSS Secrecy: Secret s remains unknown to adversaries.

APSS Availability: Correct servers together have sufficient shares of secret s to reconstruct s.

APSS Progress: Every run v eventually terminates, so all correct servers in run v eventually delete shares they store with version number $v-1$ and delete any secret information generated directly from these old shares.

2.2 Cryptographic Building Blocks

Our protocol uses secret sharing, verifiable secret sharing, and share refreshing. Any implementation of these abstractions could be used to construct a concrete instance of the APSS protocol described in this chapter. The functionality of the three abstractions is discussed in this section.

2.2.1 Secret Sharing

A secret sharing scheme defines two operations: split and reconstruct. The split operation generates a random set of shares from s. These shares constitute a sharing of s; reconstruct constructs secret s from certain subsets of these shares.

An *access structure* [55] is a collection of sets of shares, such that a set of shares can be used to construct the secret if and only if this set belongs to the collection.

Any collection of sets of shares defines an access structure, provided the following condition is satisfied: if a set is in the access structure, then so are all its supersets. Although our APSS protocol could easily be extended to secret sharing with any access structure, for clarity, the discussion in this chapter is with respect to $(n, t+1)$ *threshold* secret sharing, where split generates n shares, and the secret can be reconstructed from all shares in any set whose cardinality is more than the threshold t. Therefore, if each of the n shares is distributed to a different one of the n servers, then the following properties hold:

Threshold Availability: Secret s can be reconstructed by more than t correct servers.

Threshold Confidentiality: It is infeasible for up to t servers to reconstruct s.

Usually, shares are single values. We call such a secret sharing a *standard secret sharing*. There can be many sharings of a secret s, and labels can be used to distinguish these sharings: $\overline{s}^{:\Lambda}$ is used to denote a sharing of secret s labeled Λ, and $[\overline{s}^{:\Lambda}]_i$ denotes the ith share of that sharing. How labels are constructed varies; the construction will be given for each version of the protocol we present. Label Λ is omitted in cases where there is no confusion.

2.2.2 Combinatorial Secret Sharing

Shares of a secret sharing can also be sets of single values. We call a sharing with sets as shares a *combinatorial secret sharing*. To avoid confusion, we use *share sets* to denote shares of a combinatorial secret sharing and use "shares" only for the values comprising a standard secret sharing.

Combinatorial secret sharing and standard secret sharing are closely related. For a combinatorial secret sharing, the union of all its share sets constitutes a standard sharing of the same secret—if the secret can be reconstructed from a set of share sets in the combinatorial secret sharing, then that secret can certainly be reconstructed from all shares in the union of these share sets; if a set of share sets is insufficient to reconstruct the secret, then so are all shares in the union of these share sets. And, a combinatorial secret sharing can also be constructed using sets comprising shares of a standard secret sharing. The following describes how the scheme presented in [55] can be used to construct share sets, one for each server, of an $(n, t+1)$ combinatorial secret sharing from a standard secret sharing.

Construction of $(n, t+1)$ Combinatorial Secret Sharing.

1. Create $l = \binom{n}{t}$ different sets $P_1, ..., P_l$ of servers, such that each set contains exactly t servers. These sets of servers represent the worst-case *failure scenarios*—sets of servers that could all fail under the assumption that at most t servers are compromised.[4]
2. Create a sharing $\{s_1, \ldots, s_l\}$ using an (l, l) standard secret sharing scheme. Associate share s_i with failure scenario P_i.
3. Include secret share s_i in S_p, the share set for a server p, if and only if p is not in corresponding failure scenario P_i. That is, for any server p, share set S_p equals $\{s_i \mid 1 \leq i \leq l \wedge p \notin P_i\}$. Note that, by not assigning s_i to any server in a failure scenario P_i, we ensure that servers in P_i do not together have all l shares to reconstruct the secret.

[4]Implicitly, our discussion is with respect to an epoch (i.e., a window of vulnerability). This is because, as shown in Figure 2.1, the lifetime of a sharing is an epoch.

Also, for any server p, construct an *index set* $I_p = \{i \mid 1 \leq i \leq l \ \wedge p \notin P_i\}$. Obviously, we have $I_p = \{i \mid s_i \in S_p\}$ and $S_p = \{s_i \mid i \in I_p\}$. The index sets provide a sharing-independent description of the share-set construction.

The construction just described satisfies Threshold Confidentiality. For any set of t servers, there is a failure scenario P_i comprising exactly these t servers. By construction, share s_i, which is associated with P_i, is not assigned to any server in P_i. Because $\{s_1, \ldots, s_l\}$ constitute an (l, l) secret sharing, without knowing s_i, servers in P_i cannot reconstruct s.

Threshold Availability is also satisfied. Given a set P of more than t servers, it suffices to show that any share s_i $(1 \leq i \leq l)$ is in the share set for some server in P. Because P_i consists of only t servers, and $|P| > t$ holds, there exists a server p such that $p \in (P - P_i)$ holds. According to our construction, because $p \notin P_i$, share s_i is in the share set for server p.

In summary, given a sharing $\{s_1, s_2, \ldots, s_l\}$, the constructed share sets satisfy the following conditions:

A1. For any set P of servers, where $|P| \geq t + 1$, the following holds:

$$(\bigcup_{p \in P} S_p) = \{s_1, s_2, \ldots, s_l\}.$$

A2. For any set P of servers, where $|P| \leq t$, the following holds:

$$(\bigcup_{p \in P} S_p) \subset \{s_1, s_2, \ldots, s_l\}.$$

These conditions in turn establish that the construction results in an $(n, t + 1)$ combinatorial secret sharing.

Server (p)	*Share set* (S_p)	*Index set* (I_p)
p_1	$\{s_2, s_3, s_4\}$	$\{2, 3, 4\}$
p_2	$\{s_1, s_3, s_4\}$	$\{1, 3, 4\}$
p_3	$\{s_1, s_2, s_4\}$	$\{1, 2, 4\}$
p_4	$\{s_1, s_2, s_3\}$	$\{1, 2, 3\}$

Figure 2.2: An Example of Combinatorial Secret Sharing.

Given any (l, l) standard secret sharing labeled Λ for a secret, the construction described earlier in this section creates the corresponding share sets that constitute an $(n, t+1)$ combinatorial secret sharing for the same secret. We use S_p^Λ to denote the resulting share set for server p.

Figure 2.2 illustrates a $(4, 2)$ (i.e., $n = 4$ and $t = 1$) combinatorial secret sharing based on a $(\binom{4}{1}, \binom{4}{1}) = (4, 4)$ standard secret sharing $\{s_1, s_2, s_3, s_4\}$. The share set for each server p_i consists of all shares except s_i. The index sets are also shown. Threshold Confidentiality holds because no share set for a single server contains all the 4 shares; Threshold Availability holds because the union of the share sets for any two servers does contain all 4 shares.

Our construction of an $(n, t+1)$ combinatorial secret sharing might expand exponentially the total number l $(= \binom{n}{t})$ of shares and the size $|S_p|$ $(= \binom{n}{t} - t)$ of share sets with respect to t. Such exponential expansion is not a major concern because, for practical applications, t is typically small (e.g., 1 or 2).

Use of a combinatorial secret sharing in place of a standard secret sharing offers the following benefits.

- Using a combinatorial secret sharing scheme enables simple and efficient recovery of share sets. Such recovery might be necessary because some servers never receive the shares in their share sets or because these servers were compromised. As illustrated in Figure 2.2, each share appears in multiple share sets and is thus replicated (e.g., s_1 appears on servers p_2, p_3, and p_4). Therefore, a server can obtain shares for its share set by asking other (correct) servers for these shares.[5]

- A combinatorial secret sharing is constructed from an $(n, t+1)$ standard secret sharing, where $n = t+1$ holds. Henceforth, this subset of standard secret sharing is referred to as (l, l) standard secret sharing. Schemes designed specifically for (l, l) standard secret sharings are usually simpler than those for the more general $(n, t+1)$ standard secret sharings (i.e., where n may or may not be equal to $t+1$). For example, Shamir's $(n, t+1)$ secret sharing scheme [101] requires polynomial calculation and interpolation, whereas an (l, l) scheme could perform modular additions and subtractions only.

- As shown in [55], a combinatorial secret sharing scheme can implement any access structure, including non-threshold structures. Such more general access structures might be desirable in cases where servers are not equally vulnerable (thus making a threshold structure inappropriate).

While a share set for a server describes what shares a server should have, the following notion captures what shares a server actually stores. A server p is said to *hold* a share s_i if and only if p stores that share. Recall that servers delete old shares after new shares are generated; a server no longer holds a share after that share is deleted.

Given an $(n, t+1)$ combinatorial secret sharing built upon an (l, l) standard secret sharing Λ of version number v, sharing Λ is *established* if and only if at least $t+1$ correct servers in epoch v each holds all shares in its share set of this sharing. For an

[5] For a server q, a correct server p only sends shares that are in $S_p \cap S_q$ and does so in a way (e.g., through encryption) that only q can retrieve these shares.

established sharing Λ, due to property A1 of combinatorial secret sharings, correct servers jointly hold all l shares of sharing Λ. Therefore, the following properties hold:

E1. If a sharing is established, then correct servers in the epoch together have enough shares to reconstruct the secret.

E2. If a sharing is established, then, for any share of this sharing, there exists a correct server that holds this share.

2.2.3 Verifiable Secret Sharing

Verifiable secret sharing [24] provides a means for servers to check whether a set of shares constitute a sharing of a secret, so that erroneous shares from compromised servers can be detected and discarded. We here adopt Feldman's style of verifiable secret sharing [33].[6]

In this section, we present verifiable secret sharing for (l, l) standard secret sharing schemes. With such verifiable secret sharing, verifying share sets in an $(n, t+1)$ combinatorial secret sharing is accomplished by verifying each share in the share sets—the given verifiable secret sharing for the underlying (l, l) standard secret sharing is used for such verification.

Verifiable secret sharing introduces a function oneWay, which maps confidential information (e.g., secrets and shares) from a domain $\mathcal{D}$ into a new domain $\mathcal{R}$. We call $vc.s = \mathsf{oneWay}(s)$ the *validity check* of s, $\overline{vc}.\overline{s} = (\mathsf{oneWay}([\overline{s}]_1), \ldots, \mathsf{oneWay}([\overline{s}]_l))$

[6] Any homomorphic non-interactive verifiable secret sharing scheme [6] (e.g., Pedersen's scheme in [87]) will work in our protocol. See [57] for a presentation and comparison of proactive secret sharing schemes built on Feldman's verifiable secret sharing and ones on Pedersen's.

$$\begin{tikzcd}
s \arrow[rr, shift left, "\mathsf{split}"] \arrow[dd, "\mathsf{oneWay}"'] & & \overline{s} \arrow[ll, shift left, "\mathsf{reconstruct}"] \arrow[dd, "\mathsf{oneWay}"] & \mathcal{D} \\
& & & \\
vc.s & & \overline{vc}.\overline{s} \arrow[ll, "\mathsf{vcConstr}"'] & \mathcal{R}
\end{tikzcd}$$

Figure 2.3: Secret Sharing and Verifiable Secret Sharing.

the *validity check* of sharing $\overline{s}$, and $vc.[\overline{s}]_i = \mathsf{oneWay}([\overline{s}]_i)$ the *validity check* of share $[\overline{s}]_i$.

Function oneWay has the property that the function itself is easy to compute, but its inverse is infeasible to compute. Therefore, disclosure of validity checks does not expose the corresponding secrets or shares. Function oneWay also has the following homomorphic property: there exists a function $\mathsf{vcConstr} : \mathcal{R}^l \rightarrow \mathcal{R}$, such that the following holds:

$$\mathsf{vcConstr}(\mathsf{oneWay}([\overline{s}]_1), \ldots, \mathsf{oneWay}([\overline{s}]_l)) = \mathsf{oneWay}(\mathsf{reconstruct}([\overline{s}]_1, \ldots, [\overline{s}]_l))$$

Figure 2.3 illustrates the relationships between operations split, $\mathsf{reconstruct}$, oneWay, and $\mathsf{vcConstr}$. Given a secret s, its validity check $vc.s$ is generated using oneWay (i.e., $vc.s := \mathsf{oneWay}(s)$). A sharing $\overline{s}$ is generated from s using split (i.e., $\overline{s} := \mathsf{split}(s)$). The validity check $\overline{vc}.\overline{s}$ for $\overline{s}$ is generated using oneWay (i.e., $\overline{vc}.\overline{s} := \mathsf{oneWay}(\overline{s})$).[7] Due to homomorphism, $\mathsf{vcConstr}(\overline{vc}.\overline{s}) = vc.s$ holds.

Using verifiable secret sharing, a set of shares $\{s_1, \ldots, s_l\}$ constitutes a sharing of s provided the following three conditions hold:

(1) $\mathsf{oneWay}(s) = y$,

[7] We use $\mathsf{oneWay}(\overline{s})$ as an abbreviation for $(\mathsf{oneWay}([\overline{s}]_1), \ldots, \mathsf{oneWay}([\overline{s}]_l))$.

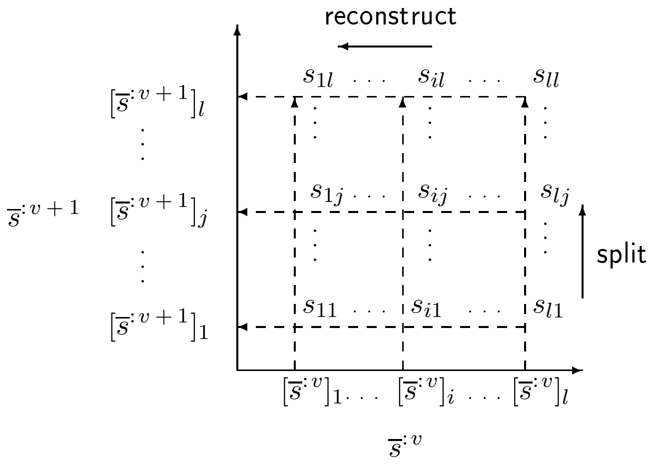

Figure 2.4: Share Refreshing.

(2) $\mathsf{oneWay}(s_i) = y_i$ for all $1 \leq i \leq l$, and

(3) $\mathsf{vcConstr}(y_1, \ldots, y_l) = y$.

Condition (1) ensures that y is the validity check for secret s, condition (2) guarantees that $\{y_1, \ldots, y_l\}$ is the validity check for $\overline{s}$, and condition (3) ensures that y can be constructed from $\{y_1, \ldots, y_l\}$ using vcConstr. Usually, y is given to servers during initialization and stored in ROM (Read Only Memory). Verification of condition (1) is then unnecessary. Any server can check condition (3) given y and $\{y_1, \ldots, y_l\}$, but only servers that store a certain share can verify condition (2) for that share.

2.2.4 Share Refreshing

Share refreshing for a combinatorial secret sharing can be achieved using share refreshing for the underlying (l, l) standard secret sharing—every new share set is constructed using the new shares generated by share refreshing for that (l, l) standard secret sharing.

Figure 2.4 depicts how one might generate a new (l, l) standard secret sharing

$\overline{s}^{:v+1}$ from an old one $\overline{s}^{:v}$: For every old share $[\overline{s}^{:v}]_i$, a server employs split to generate $\{s_{i1}, \ldots, s_{il}\}$. (We use s_{ij} as an abbreviation for $[\overline{[\overline{s}^{:v}]_i}]_j$.) We call each s_{ij} a *subshare* and call $\{s_{i1}, \ldots, s_{il}\}$ a *subsharing* of $[\overline{s}^{:v}]_i$. Each subsharing corresponds to a column in Figure 2.4. Here, we assume that exactly one subsharing is generated from each old share.

Each subshare s_{ij} is then propagated to every server q satisfying $j \in I_q$. Every server q collects subshares $s_{1j}, \ldots, s_{lj}$ (i.e., the jth row in Figure 2.4) for every $j \in I_q$ and generates new share $[\overline{s}^{:v+1}]_j := \mathsf{reconstruct}(s_{1j}, \ldots, s_{lj})$. A new sharing $\overline{s}^{:v+1}$ is so generated, and its shares have been distributed to servers based on I_q for each server q. Servers then delete the old shares and delete the subshares generated from these old shares.[8]

Note that, by definition, a subsharing generated from a share $[\overline{s}^{:v}]_i$ is a sharing of $[\overline{s}^{:v}]_i$. We distinguish sharings (of the secret) and subsharings (of a share) to indicate the different roles they play in the APSS protocol. Put in terms of the notation for sharings, we would have:

- $\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}$: a subsharing, labeled λ, generated from share $[\overline{s}^{:\Lambda}]_i$
- $[\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}]_j$: the jth subshare of subsharing $\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}$

As a convention, we use upper case Λ (and Λ', Λ_i, ...) as labels of sharings and use lower case λ (and λ', λ_i, ...) as labels of subsharings. A share set for a server p constructed from subsharing λ is referred to as S_p^{λ}.

[8] Here, we present one rather general way of achieving share refreshing. Other schemes might not fully conform to this abstract description. For example, for the scheme presented in [57], subsharings are generated as a sharing of secret 0, and each new share is generated not only from a set of subshares but also from the corresponding old share. Our APSS protocol can be easily adapted to accommodate such schemes.

A notable difference between sharings and subsharings is that the lifetime of a sharing is within an epoch (since each subsharing is generated in one run and deleted in the next run), whereas the lifetime of a subsharing is within a run (since each subsharing is generated and deleted in the same run). Consequently, a subsharing generated in run v from shares of version number $v-1$ is *established* if at least $t+1$ correct servers in run v each holds all subshares in its share set of this subsharing. Properties E1 and E2 apply to established subsharings, although in this case the notion of correct servers must be defined with respect to the current run rather than the current epoch.

2.3 Derivation of the APSS Protocol

The APSS protocol is presented as a series of refinements, starting with a relatively simple protocol that requires strong assumptions. These assumptions are then relaxed, with new mechanism added to preserve correctness under the new assumptions.

The protocol is presented for a single run v. Epoch $v-1$ is referred to as the *previous epoch* and run $v-1$ as the *previous run*, while epoch v is referred to as the *current epoch* and run v as the *current run*. Similarly, version $v-1$ shares are called *old shares*, while version v shares are called *new shares*. Version numbers are omitted in cases where it is the current epoch and the current run that are being referred to. Unless noted otherwise, correct servers refer to correct servers in the current run.

2.3.1 A First APSS Protocol

For the first version of the protocol, the assumptions of Active Server-Adversaries and Active Link-Adversaries are strengthened to

Passive Server-Adversaries with Crash Failures: A compromised server can experience crash failures only and can disclose any information stored locally to adversaries. Up to t servers may be compromised in an epoch.

Passive Link-Adversaries: Links are reliable; that is, a message sent always reaches its intended recipient. An adversary can only eavesdrop on links.

The protocol we now present extends share refreshing of Section 2.2.4 to work under this set of assumptions. Multiple servers might hold the same share (e.g., in Figure 2.2, servers p_2, p_3, and p_4 all hold share s_1). And because split is non-deterministic, these servers could each generate a different subsharing from this share. Consequently, having exactly one subsharing generated from each old share, as assumed in share refreshing of Section 2.2.4, is hard to achieve here:

- Predetermining which server to generate a subsharing for each share does not work, because the selected servers could be compromised.

- Having backups that take over when the selected servers are compromised does not work either, because servers might not be able to detect that a server is compromised—there is no way to distinguish a server that has crashed from a server that is correct but slow, due to Asynchronous System.

Our first protocol avoids this problem by allowing multiple different subsharings to be generated from any single old share by different servers holding that share.

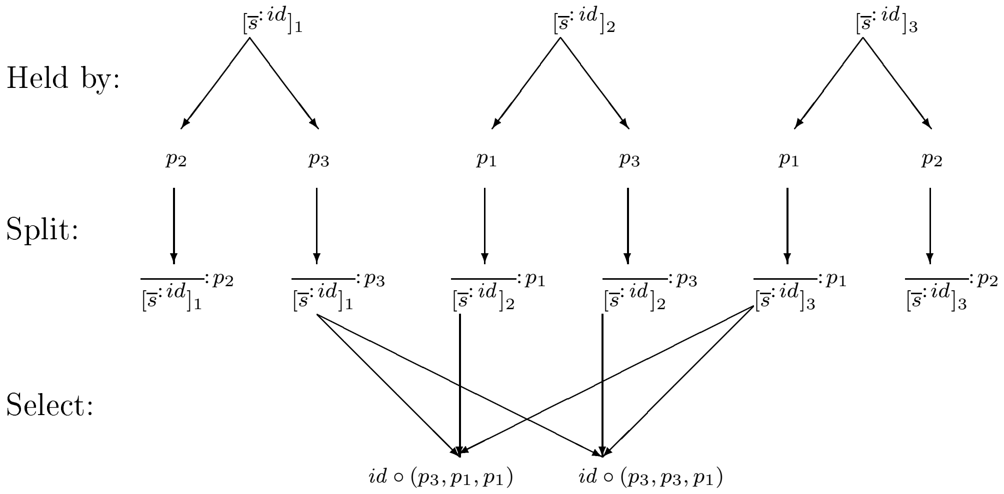

Figure 2.5: Candidate Sets of Subsharings.

Now, to generate a new sharing, it suffices to choose one subsharing generated from every share of a sharing, as shown in Figure 2.4. For a given sharing, we call a set of subsharings a *candidate set of subsharings* if that set consists of exactly one subsharing generated from each share of that sharing. A candidate set of subsharings

$$\overline{(\overline{s}^{:\Lambda})}^{:\lambda_1, \lambda_2, \ldots, \lambda_l} \triangleq \{\overline{[\overline{s}^{:\Lambda}]_1}^{:\lambda_1}, \overline{[\overline{s}^{:\Lambda}]_2}^{:\lambda_2}, \ldots, \overline{[\overline{s}^{:\Lambda}]_l}^{:\lambda_l}\},$$

is labeled $\lambda_1, \lambda_2, \ldots, \lambda_l$.

With multiple subsharings being generated from the same old share, there could be many different candidate sets of subsharings, which in turn leads to many different new sharings. This is illustrated in Figure 2.5. Starting with sharing Λ, servers p_1, p_2, and p_3 can generate 6 subsharings from the shares they hold. From these subsharings, there could be 8 candidate sets of subsharings; two of them are shown in Figure 2.5: $\overline{(\overline{s}^{:\Lambda})}^{:\lambda_2, \lambda_3, \lambda_5}$ and $\overline{(\overline{s}^{:\Lambda})}^{:\lambda_2, \lambda_4, \lambda_5}$.

To help decide on a single candidate set of subsharings, this first version of the APSS protocol postulates a centralized coordinator p, which chooses a candidate set of subsharings that servers use to construct new shares. So, for this first APSS protocol, we assume

Correct Coordinator: Coordinator p is always correct.

Although it is usually a server that also acts as the coordinator, we regard the coordinator as an independent process on that server in order to distinguish the different role played by the coordinator. Details of the protocol are presented in Figure 2.6. Because (only) one new sharing is generated in each run, version numbers are sufficient to serve as labels of sharings; a subsharing generated from $[\overline{s}^{:\Lambda}]_i$ by a server q is labeled $\Lambda \circ i \circ q$—server q is included in the label because different subsharings can be generated from the same share by different servers.

An outline of the protocol is given as follows:

1. The coordinator initiates a run by sending an init message to all servers.

2. Upon receiving the init message, servers generate subsharings from the old shares they hold and propagate these subshares to other servers in establish messages.

   Only subshares needed by a server to construct the new shares in its share set are propagated to that server. Subshares being propagated are encrypted, so that only the intended recipient can retrieve the subshares. (Recall, servers know each other's public keys. Therefore, they can authenticate each other and establish shared secret session keys for this encryption.)

   Servers then send the coordinator the labels of the subsharings that have been generated and propagated.

3. Servers generate new shares from subsharings selected by the coordinator.

1. Coordinator p sends, in an init message, the label Λ of the sharing to be refreshed to every server. (Here, Λ is $v-1$.)

2. Subshare generation and propagation.

   Each server q, upon receiving from coordinator p an init message containing a label Λ, performs the following steps:

   2.1. Generate $\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda} := \mathsf{split}([\overline{s}^{:\Lambda}]_i)$ for each share $[\overline{s}^{:\Lambda}]_i$ that q holds, where λ is $\Lambda \circ i \circ q$.

   2.2. For each subsharing λ that is generated, send, in an establish message, all subshares in S_r^{λ}, encrypted, to every server r.

   2.3. Send coordinator p, in a contribute message, the labels of all subsharings that q has generated and propagated in steps 2.1 and 2.2.

3. Share generation.

   3.1. Coordinator p awaits enough contribute messages, chooses a candidate set of subsharings with label "$\lambda_1, \lambda_2, \ldots, \lambda_l$" from those contained in these contribute messages, and sends this choice in a compute message to all servers.

   3.2. Each server r, upon receiving from coordinator p a compute message containing a choice "$\lambda_1, \lambda_2, \ldots, \lambda_l$", performs the following steps:

   3.2.1. Await establish messages until all encrypted subshares in $\bigcup_{1 \le i \le l}(S_r^{\lambda_i}) = \{[\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda_i}]_j \mid (1 \le i \le l) \wedge (j \in I_r)\}$ have been received and decrypted.

   3.2.2. Compute $[\overline{s}^{:\Lambda'}]_j := \mathsf{reconstruct}([\overline{[\overline{s}^{:\Lambda}]_1}^{:\lambda_1}]_j, [\overline{[\overline{s}^{:\Lambda}]_2}^{:\lambda_2}]_j, \ldots, [\overline{[\overline{s}^{:\Lambda}]_l}^{:\lambda_l}]_j)$ for each $j \in I_r$, where Λ' is v.

   3.2.3. Send a computed message notifying p that r holds all the shares in its share set of sharing Λ'.

   3.3. Coordinator p awaits $2t+1$ computed messages and then sends each server r a finished message, containing Λ', notifying r that a new sharing has been established.

4. Each server r, upon receiving from coordinator p a finished message containing the label of a sharing whose version number is v, deletes all its old shares and subshares, and updates cvn_r to v if cvn_r is less than v.

Figure 2.6: The First APSS Protocol.

3.1. The coordinator picks a candidate set of subsharings from subsharings that have been generated and propagated. The coordinator then informs all other servers about the decision.

3.2. Upon receiving the decision from the coordinator, servers wait until they have received the selected subshares they need. Then, servers construct new shares in their share sets from these subshares and send the coordinator a confirmation that they have constructed the shares in their share sets of this new sharing.

3.3. Upon receiving such confirmations from $2t + 1$ servers, the coordinator sends a finished message to all servers.

4. Upon receiving the finished message, servers delete the old shares and subshares, as well as updating their current version numbers.

Now we show that this protocol satisfies the correctness requirements in Section 2.1.

Lemma 2.1 *The protocol in Figure 2.6 satisfies APSS Secrecy, assuming Passive Server-Adversaries with Crash Failures, Passive Link-Adversaries, and Correct Coordinator.*

Proof Sketch: To prove APSS Secrecy, it suffices to show that an adversary cannot get all l shares of any sharing by compromising at most t servers in one epoch. Consider any sharing $\overline{s}^{:\Lambda}$, where Λ is the version number v. Let P be the set of servers that are ever compromised in epoch v. Because at most t servers are compromised in an epoch, $|P| \leq t$ holds. Therefore, due to A2, there exists a share $[\overline{s}^{:\Lambda}]_i$ that is not in share set S_p^Λ for any $p \in P$. It suffices to show that an adversary can never learn $[\overline{s}^{:\Lambda}]_i$.

There are three ways an adversary might learn $[\overline{s}^{:\Lambda}]_i$:

(1) collect all the l subshares of the subsharing from which $[\overline{s}^{:\Lambda}]_i$ is computed (in step 3.2.2 of run v),

(2) compromise a server when that server holds $[\overline{s}^{:\Lambda}]_i$, or

(3) collect all the l subshares of a subsharing generated from $[\overline{s}^{:\Lambda}]_i$ (in step 2.1 of run $v+1$).

Because correct servers generate subsharings randomly (using split), different subsharings of $[\overline{s}^{:\Lambda}]_i$ are independent. Therefore, an adversary is unable to learn $[\overline{s}^{:\Lambda}]_i$ by combining partial information from multiple subsharings. For example, an adversary is unable to learn $[\overline{s}^{:\Lambda}]_i$ from fewer than l subshares of the subsharing described in (1) and fewer than l subshares of a subsharing generated from $[\overline{s}^{:\Lambda}]_i$ (as described in (3)).

For attack (1), let $\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}$ be the subsharing from which $[\overline{s}^{:\Lambda}]_i$ is actually computed. Again, due to A2, there exists a subshare $[\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}]_j$ that is not in the share set S_p^λ for any $p \in P$. According to the protocol, $[\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}]_j$ must be generated by a correct server in epoch v and will only be distributed to some correct servers in epoch v. Subshare $[\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}]_j$ will not be exposed during transmission either, because all subshares being transmitted are encrypted. Therefore, an adversary cannot obtain this subshare in epoch v. Furthermore, an adversary cannot obtain this subshare outside of epoch v, because this subshare is both generated and deleted in epoch v (in run $v-1$, to be more precise) by some correct servers of epoch v. The same argument also applies to attack (3).

Finally, attack (2) would not work for the following reasons:

(i) An adversary is unable to obtain share $[\overline{s}^{:\Lambda}]_i$ directly from a server during epoch v because no compromised server in epoch v has this share stored.

(ii) An adversary is unable to obtain $[\overline{s}^{:\Lambda}]_i$ directly from a server before or after epoch v because the share is both constructed and deleted during epoch v. ■

Lemma 2.2 *The protocol in Figure 2.6 satisfies both APSS Availability and APSS Progress, assuming Passive Server-Adversaries with Crash Failures, Passive Link-Adversaries, and Correct Coordinator.*

Proof Sketch: The proof that the protocol satisfies APSS Availability and APSS Progress is by induction. Assume that an old sharing $\overline{s}^{:\Lambda}$, where Λ is version number $v-1$, is established when run v starts. It suffices to show that this run eventually terminates, that old shares are deleted only after a new sharing has been established, and that a new sharing of version number v is established both upon termination of this run and at the start of the next run.

Because sharing $\overline{s}^{:\Lambda}$ is established, due to E2, for every share $[\overline{s}^{:\Lambda}]_i$, there exists at least one correct server p_i in the previous epoch (and hence correct in this run) that holds this share. This correct server p_i will generate a subsharing $\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}$ from $[\overline{s}^{:\Lambda}]_i$ in step 2.1 of the protocol in Figure 2.6, propagate the subshares in step 2.2, and include λ in a contribute message to the coordinator. Therefore, the coordinator eventually receives enough contribute messages to choose a candidate set of subsharings from those whose labels are contained in these contribute messages.

Without loss of generality, assume that coordinator p chooses a candidate set of subsharings $\overline{(\overline{s}^{:\Lambda})}^{:\lambda_1, \lambda_2, \ldots, \lambda_l}$. Due to Passive Link-Adversaries, which assumes that links are reliable, every correct server r in this epoch will eventually get all the subshares in $\bigcup_{1 \leq i \leq l}(S_r^{\lambda_i})$. Using these subshares, r generates the set of new shares $S_r^{\Lambda'} = \{[\overline{s}^{\cdot\Lambda'}]_j \mid j \in I_r\}$ in step 3.2.2. There are at least $2t+1$ correct servers. Therefore, the coordinator will eventually get computed messages from $2t+1$ servers. Among these $2t+1$ servers that respond, given at most t compromised servers in one epoch, we conclude that at least $t+1$ servers must be correct in the current epoch. A new sharing $\overline{s}^{\cdot\Lambda'}$ is thus established, by definition. Because correct servers in the current epoch do not delete shares of Λ' until the next run, sharing Λ' remains established at the beginning of the next run.

Upon receiving $2t+1$ computed messages, the coordinator sends a finished message to all servers. Due to reliable links, every correct server in this run eventually receives the finished message and deletes its old shares and subshares. Consequently, this run eventually terminates. APSS Availability remains true because a new sharing has been established when old shares are deleted. ■

Note that this protocol works only if $n \geq 3t+1$ holds. This is because, to preserve APSS Availability, a coordinator sends a finished message, which causes servers to delete old shares and subshares, only after the coordinator knows that a new sharing has been established—$t+1$ correct servers in the current epoch must hold the shares in their share sets of this sharing. Because the coordinator does not know which servers are correct in the current epoch and there are up to t compromised servers, it must get confirmations (i.e., computed messages) from $(t+1)+t = 2t+1$ servers (instead of $t+1$ servers) in order to be convinced that a new sharing has been established. Again, because there are up to t compromised servers, $(2t+1)+t = 3t+1$ servers are needed in order to ensure that the coordinator can always get $2t+1$ computed messages in an asynchronous system: If there were only $3t$ servers, then the following two cases are problematic.

(1) There are t compromised servers, who refuse to respond to the coordinator.

The coordinator is thus unable to get more than $3t - t = 2t$ responses.

(2) There are t compromised servers, who do respond to the coordinator before correct servers do. If the coordinator only waits for up to $2t$ responses, then only t of the responses are from correct servers—this does not guarantee that a new sharing has been established.

2.3.2 APSS with Multiple Coordinators

The APSS protocol in Figure 2.6 is based on Correct Coordinator, which is an unacceptably strong assumption. This section describes how to derive a protocol that does not rely on this assumption.

Without Correct Coordinator, to have a single new sharing generated requires all correct servers to agree on which candidate set of subsharings to use for that new sharing. This would, in turn, require solving the agreement problem in an asynchronous system, which is known to be difficult. Our APSS protocol avoids the need for such an agreement by allowing multiple sharings—the protocol employs multiple coordinators. Here, coordinator p and server p are regarded as two separate processes on the same host p. Coordinator p is correct if and only if server p is correct. And, communication between coordinator p and server p is always reliable with no transmission delay.

By letting each server act as a coordinator,[9] at least one coordinator will be correct in each run, ensuring that at least one new sharing will be established. But having multiple coordinators could cause up to n new sharings to be generated in

[9] Because only up to t servers may be compromised, having $t+1$ servers function as coordinators suffices to ensure that a correct coordinator is always present. Here, we assume that every server acts as a coordinator to simplify the presentation of the protocol.

a run and to coexist—different coordinators might select different candidate sets of subsharings, leading to different new sharings.

With multiple coordinators, multiple instances of the protocol in Figure 2.6, each with a different coordinator, could be invoked. Each instance is called a *thread*. A thread *starts* when its coordinator initiates the protocol for this thread; a thread *terminates* when its coordinator terminates the protocol for this thread.

To accommodate multiple threads, certain changes to the protocol of Figure 2.6 are needed. The first change concerns labels of sharings and subsharings. Version numbers alone are no longer sufficient as labels of sharings, since there are now multiple sharings with the same version number. Instead, the label of a sharing will now be built from the version number and the name of the coordinator of the thread creating that sharing. The label of a subsharing generated from a share $[\overline{s}^{:\Lambda}]_i$ by a server q will remain $\Lambda \circ i \circ q$, although Λ, the label of the sharing, has changed to containing both a version number and the name of a coordinator. Servers can now use labels to specify which new sharing to use when reconstructing the secret.

The second change to the APSS protocol of the previous section concerns the termination of threads. For the protocol in Figure 2.6, which assumes a single coordinator that is always correct, the coordinator terminates its thread when it finishes executing step 3.3 of Figure 2.6. With multiple threads in a run, coordinator p will terminate its thread whenever it knows that any coordinator has terminated a thread in the same run (even if p has not finished executing all the steps in its own thread). This is because, when a coordinator terminates its thread, a new sharing must have been established, and there is no need for other threads to continue. For a coordinator to know about the termination of other threads, finished messages are propagated not only from coordinators to servers (as done in step 3.3 of Figure 2.6), but also from servers to coordinators.[10] Therefore, in this new protocol, the following rules govern the propagation of finished messages and the termination of threads:

- A coordinator terminates its thread

  (1) when the coordinator obtains $2t + 1$ computed messages in step 3.3 of Figure 2.6 or

  (2) when the coordinator receives from a server a finished message indicating that a (different) coordinator has terminated its thread.

  The coordinator propagates finished messages to all servers when terminating its thread.

- A server carries out step 4 of Figure 2.6 when it receives from a coordinator a finished message indicating that a coordinator has terminated its thread.

  After completing step 4 of Figure 2.6, a server always forwards the finished message to any coordinator who requests the participation of this server in a thread of the same run.

Lemma 2.3 *The protocol described above preserves APSS Secrecy, APSS Availability, and APSS Progress, assuming Passive Server-Adversaries with Crash Failures and Passive Link-Adversaries.*

Proof Sketch: From the proof for Lemma 2.1, for any sharing Λ, there exists a share $[\overline{s}^{:\Lambda}]_i$ that cannot be obtained by an adversary either directly or through subshares of that share, even if the adversary knows all the information on servers that are

[10] There are other ways of propagating finished messages; for example, a coordinator can propagate finished messages to other coordinators. Our choice is both simple and easily adapted to cases where links are only fair, as shown in Section 2.3.3.

compromised. With multiple coordinators, more subsharings and shares could be generated. We show that, even with more subsharings and shares generated, an adversary cannot learn share $[\overline{s}^{:\Lambda}]_i$.

- Although more subsharings could be generated, these subsharings (even for the same share) are independent because servers generate them randomly. Therefore, learning a share from subshares still requires that the adversary obtain an entire subsharing of that share, which has been proved impossible in the proof for Lemma 2.1 (Cases (1) and (3) in that proof).

- Although more shares could be generated, an adversary only has access to shares generated on compromised servers because correct servers do not disclose their shares. The shares generated on compromised servers do not add anything new to the knowledge of an adversary because the adversary is already assumed to know the subshares (on compromised servers) used to generate these shares.

To prove APSS Progress, it suffices to show that any correct coordinator p terminates its thread.[11] We consider two cases:

- If, during the execution of the thread with p as the coordinator, p receives a finished message, then p terminates its thread (Case (2) of thread termination described earlier).

- If no finished message is ever received by p, then no correct server has yet

[11] It might seem sufficient to have one correct coordinator terminate its thread, because that coordinator would propagate finished messages to servers upon termination of its thread. However, it is desirable (and necessary in practice) to have all coordinators terminate their threads, so that eventually no messages related to this run are sent.

received a finished message or deleted its old shares and subshares when participating in the thread with p being the coordinator. (Otherwise, that server would send back a finished message in response to a message from p, and p would eventually receive that message due to reliable links.) The proof for Lemma 2.2 shows that, in this case, p is also able to terminate its thread (Case (1) of thread termination described earlier).

The protocol also ensures APSS Availability, because old shares are deleted only when a finished message is constructed or received. Using the proof for Lemma 2.2, a new sharing must have been established when such a finished message is created. ■

2.3.3 Defending Against Active Link-Adversaries

We now relax Passive Link-Adversaries and obtain a protocol that defends against Active Link-Adversaries. Under the new assumption, an adversary could now insert, modify, delete, reorder, or replay messages. The previous protocol is vulnerable to these attacks.

Standard defenses can deal with some of these attacks. To tolerate message modification, senders sign each message they send, and receivers use those signatures to detect and discard compromised messages. To combat replay attacks, each message includes information (e.g., the version number of the run, the coordinator of the thread, and the type of the message) that distinguishes that message from other messages. This way, messages for different runs or for different threads of the same run can be distinguished and processed accordingly.

But none of these defenses is effective against message deletion. And, progress of the protocol might depend on the delivery of messages being deleted by an adversary.

For example, if establish messages sent to a server q in step 2.2 of Figure 2.6 are deleted, then server q would not be able to proceed in step 3.2.2 because it would never receive the needed subshares.

Because Active Link-Adversaries assumes fair links, retransmission can be employed to defend against such attacks. In fact, reliable links can be approximated by using fair links as follows: Given are a correct sender A and a correct receiver B.

1. A keeps sending a message m to B.
2. B sends back an acknowledgment $ack(m)$ to A whenever B receives m.
3. A stops sending the message when A receives $ack(m)$ from B.

It is easy to see that A eventually receives $ack(m)$ from B. A retransmits message m infinitely often if it does not receive $ack(m)$ from B. Because links are fair, B will receive m infinitely often and thus will send $ack(m)$ infinitely often to A. Invoking the fair link assumption again on $ack(m)$, server A eventually gets $ack(m)$ from B.

In our protocol, a coordinator always sends messages to a group comprising all servers. Responses from the servers to the coordinator can be regarded as acknowledgments. Therefore, the approximation of reliable links is extended to multicasts using the following group_send primitive.

A group_send(p, m, ack, d) by a server p works as follows: For any server q, server p constructs a message $m_{p \to q}$ from m and repeatedly sends message $m_{p \to q}$ to q. Every server q, after receiving $m_{p \to q}$ from p and after processing that message, sends back message $ack(m_{p \to q})$ to p. Server p terminates this group_send(p, m, ack, d) when it receives acknowledgments from d different servers. Note that, because recipients might be compromised, the initiator of a group_send might not receive acknowledgments from all servers. Moreover, even a correct server might not always be in a

- p, q, r: servers
- $\langle m \rangle_p$: message m signed by a server p using p's private key
- $[p \longrightarrow q : \; m]$: message m is sent from server p to server q
- $[\forall q.\; p \longrightarrow q : \; m_q]$: message m_q is sent from server p to server q for every server q
- $\mathcal{E}(m)$: m is encrypted in a way that only the intended recipient can decrypt m.

Figure 2.7: Notation Used in Protocol Presentation.

state to process the received message or to send back an acknowledgment. For example, a correct server might not have the shares of an established sharing in order to generate and propagate subsharings from these shares (as would be required in step 2 of Figure 2.6). Consequently, group_send(p, m, ack, d) is guaranteed to terminate if and only if d is chosen to be at most the number of correct servers that are able to send back acknowledgments.

A group_send(p, m, ack, d), upon termination, does not guarantee that d correct servers have received and processed message m. This is because there are at most t compromised servers, and t out of the d servers that have responded might be compromised when or after sending the acknowledgments.

We now revisit each message transmission in the previous protocol and show how group_send is used to get the desired effect. Some segments of the new protocol will also be presented; the notation used in presenting these segments is summarized in Figure 2.7.

Recall that a finished message might be sent to a coordinator or a server in response to any message m from that coordinator or that server.[12] This occurs

[12]Besides being sent between coordinators and servers, finished messages can also be sent from one server to another server informing the recipient that a new sharing has been established. Such

when the current run has already terminated locally on the server that receives m. (No bogus finished message could be sent because we are assuming Passive Server-Adversaries.) Therefore, a finished message could act as an acknowledgment for a group_send; the group_send terminates upon receiving such a finished message—there is no need for the sender to wait for more responses.[13] This possibility is included with all the group_sends employed in our protocol (and is not repeated in each of the discussions below).

Step 1 of Figure 2.6 is replaced by a coordinator p initiating

$$\mathsf{group\_send}(p, \mathsf{init}, \mathsf{contribute}, t+1).$$

Thus, contribute messages sent by servers in step 2.3 serve as acknowledgments to the init messages sent in step 1.

For subshare propagation, step 2.2 of Figure 2.6 is replaced by server q carrying out the group_send described in Figure 2.8, where establish messages are the messages q repeatedly send to all servers, and servers send back established messages as acknowledgments. Note that established messages were not needed in the protocol of Figure 2.6, because establish messages were always delivered due to the assumption of reliable links.

The protocol in Figure 2.8 for subshare propagation does not ensure that all correct servers receive subshares of the subsharing being propagated (as would be achieved by step 2.2 when links are reliable). A correct server r that does not get

---

propagation of finished messages is especially useful when links are fair but not necessarily reliable.

[13] As an exception, finished messages sent in step 3.3 of Figure 2.6 from coordinators to servers cannot be treated as acknowledgments for group_sends. However, as we shall show in the correctness proofs, a coordinator p, when propagating a finished message, only needs to ensure that the corresponding server p receives the message. This is satisfied because the communication between coordinator p and server p is reliable.

P-1. To propagate subshares of a subsharing labeled λ, server q employs $\mathsf{group\_send}(q, \mathsf{establish}, \mathsf{established}, 2t+1)$, where

$$\mathsf{establish}_{q \to r} = \langle \mathsf{establish}, v, \lambda, q, r, \mathcal{E}(S_r^\lambda) \rangle_q$$

is the message q repeatedly sends to each server r, and established messages, as shown in step P-2, serve as acknowledgments to the establish messages.

P-2. Each server r, upon receiving an establish message from q in a format shown in step P-1, stores the subshares and sends back an acknowledgment in an established message to q.

$$[r \longrightarrow q : \langle \mathsf{established}, v, \lambda, r, q \rangle_r]$$

Figure 2.8: Subshare Propagation Using group_send.

certain subshares might be unable to proceed in step 3.2.1 of Figure 2.6. If that is the case, in step 3.2.1, server r identifies subsharings whose subshares are missing and recover every such subsharing λ using the group_send described in Figure 2.9, where recover messages are the messages r repeatedly send to all servers, and servers send back recovered messages as acknowledgments. In the recovered message, only subshares that belong to share sets of both the sender and the receiver are included, and these subshares are encrypted, so that only the intended receiver can retrieve the subshares. Note that neither recover nor recovered messages showed up in Figure 2.6 because subshares were always delivered to the intended recipients when links were reliable, eliminating any need for subshare recovery.

Finally, to send a compute message in step 3.1, a coordinator p employs

$$\mathsf{group\_send}(p, \mathsf{compute}, \mathsf{computed}, 2t+1).$$

Here, the computed messages sent in step 3.3 of Figure 2.6 serve as acknowledgments to the compute messages sent in step 3.1.

With these changes, we now show that the new protocol preserves the APSS correctness requirements.

R-1. To recover subshares of a subsharing labeled λ, server r employs $\mathsf{group\_send}(r, \mathsf{recover}, \mathsf{recovered}, t+1)$, where

$$\mathsf{recover}_{r \rightarrow q} = \langle \mathsf{recover}, v, \lambda, r, q \rangle_r$$

is the message r repeatedly sends to each server q, and recovered messages, as shown in step R-2, serve as acknowledgments to the recover messages.

R-2. Each server q, upon receiving a recover message from q for subshares of subsharing labeled λ, checks whether it has the requested subshares. If so, q sends to r a recovered message.

$$[q \longrightarrow r : \langle \mathsf{recovered}, v, \lambda, q, \mathcal{E}(S_q^\lambda \cap S_r^\lambda), r \rangle_q]$$

Figure 2.9: Subshare Recovery Using group_send.

Lemma 2.4 *The protocol described above satisfies APSS Secrecy, assuming Passive Server-Adversaries with Crash Failures and Active Link-Adversaries.*

Proof Sketch: Even under the new attacks admitted by Active Link-Adversaries, correct servers always generate subsharings randomly regardless of what messages they receive; send subshares, encrypted, to other servers based on share sets; and never disclose any shares stored locally. Therefore, the proof of APSS Secrecy for Lemma 2.3 applies. ■

Lemma 2.5 *The protocol described above satisfies APSS Availability, assuming Passive Server-Adversaries with Crash Failures and Active Link-Adversaries.*

Proof Sketch: Every correct server in this protocol deletes its old shares only after it receives a finished message. The proof of APSS Availability for Lemma 2.3 shows that, when a finished message is constructed, a new sharing must have been established. Therefore, APSS Availability holds. ■

To prove APSS Progress, we first prove the following lemmas.

Lemma 2.6 *If the current run terminates locally on a correct server q, then the current run eventually terminates.*

Proof Sketch: Consider any correct coordinator r. We show that coordinator r eventually terminates its thread. If coordinator r never terminates its thread, then coordinator r will keep sending messages to all servers (including server q), and server q will send a finished message to coordinator r as the acknowledgment (because the current run has terminated locally on server q). Because links are assumed to be fair, p eventually receives the finished message from server q and terminates its thread. Therefore, every correct coordinator r eventually terminates its thread.

Upon terminating its thread, coordinator r sends a finished message to all servers, including server r. Because the communication between coordinator r and server r is reliable, server r receives the finished message, and deletes its old shares and subshares. So, the current run terminates locally on server r for any correct server r. By definition, since the current run terminates on all correct servers, the current run eventually terminates. ■

Due to Lemma 2.6, if any correct server has deleted its old shares and subshares, then the current run is guaranteed to terminate. Consequently, the following lemmas focus on the case where no correct servers have deleted old shares and subshares—we thus ignore the case where finished messages are sent and received as acknowledgments.

Lemma 2.7 *Given a correct server q, the protocol in Figure 2.8 always terminates. Upon termination, the subsharing being propagated is established.*

Proof Sketch: The protocol in Figure 2.8 is guaranteed to terminate, because there are at least $2t + 1$ correct servers, and every correct server always sends an $\mathsf{established}$ message as an acknowledgment upon receiving an $\mathsf{establish}$ message.

The $2t + 1$ servers from which q receives $\mathsf{established}$ messages might not all be correct in this run. Even so, $2t+1-t$ $(= t+1)$ of these $2t+1$ servers must be correct in this run. Therefore, upon termination, $t+1$ correct servers will have received the subshares in their share sets of the subsharing, thereby ensuring that the subsharing is established. ■

Lemma 2.8 *Given a correct coordinator p, the* $\mathsf{group\_send}(p, \mathsf{init}, \mathsf{contribute},\ t + 1)$ *used in step 1 of the protocol always terminates. Upon termination, p can choose a candidate set of subsharings that have been established based on the* $\mathsf{contribute}$ *messages that p has received.*

Proof Sketch: In the init message, coordinator p must have selected an old sharing $\overline{s}^{:\Lambda}$ that has been established (note that a server knows the label of an established sharing because the label has been propagated in a $\mathsf{finished}$ message generated in the previous run). By definition, $t+1$ servers that are correct in the previous epoch (and hence in this run) each holds all shares in its share sets of sharing $\overline{s}^{:\Lambda}$. These servers will generate subsharings from these shares and propagate these subsharings (using the protocol in Figure 2.8). Due to Lemma 2.7, such subshare propagation by a correct server always terminates. These $t+1$ servers will then return a $\mathsf{contribute}$ message to p. Thus, this $\mathsf{group\_send}$ is guaranteed to terminate, at which point coordinator p must have received $\mathsf{contribute}$ messages from $t + 1$ servers.

Due to E1, these $t + 1$ servers together must hold all shares of sharing $\overline{s}^{:\Lambda}$. At least one subsharing is thus generated from each of these shares, and the label of that subsharing must be included in a $\mathsf{contribute}$ message that p has received. Therefore, upon termination of this group_send, there exists a candidate set of subsharings whose labels appear in the contribute messages p has received. ■

Lemma 2.9 *Given a correct server r, the protocol in Figure 2.9 always terminates. Upon termination, r must have received all the subshares in its share set for the subsharing being recovered.*

Proof Sketch: A correct server r only recovers subshares of a subsharing in a candidate set of subsharings selected by a coordinator p. And, coordinator p selects only subsharings whose labels are enclosed in a contribute message. A server includes the label of a subsharing in a contribute message only after the termination of the group_send (Figure 2.8) for propagating this subsharing. Due to Lemma 2.7, the subsharing being recovered must have been established.

Let λ be the label of the subsharing to be recovered in this group_send. Because subsharing λ is established, there exists $t+1$ correct servers that together hold all subshares of subsharing λ. These correct servers will be able to send recovered messages to r, ensuring the termination of this group_send for subshare recovery.

Let P be the set of servers from which q has received recovered messages. Because the group_send used in Figure 2.9 terminates when r receives $t+1$ acknowledgments, $|P| = t+1$ holds. Server r must have received all subshares in $\bigcup_{q \in P}(S_q^\lambda \cap S_r^\lambda) = (\bigcup_{q \in P}(S_q^\lambda)) \cap S_r^\lambda$. Due to A1 in Section 2.2.1, $\bigcup_{q \in P}(S_q^\lambda)$ contains all subshares in the subsharing, thus, server r will hold all subshares in its share set S_r^λ. ■

Lemma 2.10 *The* group_send$(p, \mathsf{compute}, \mathsf{computed}, 2t+1)$ *that is used in step 3.1 by a correct coordinator p always terminates. Upon termination, a new sharing must have been established.*

Proof Sketch: Due to Lemmas 2.7 and 2.9, all correct servers in this epoch eventually receive (either through subshare propagation or through subshare recovery) the subshares they need to construct new shares in step 3.2.2 of Figure 2.6. Therefore, every correct server in this epoch is able to respond with a computed message in this group_send, thereby ensuring termination of this group_send. As in the case for subshare propagation (see the proof for Lemma 2.7), this group_send guarantees that the new sharing generated from the selected candidate set of subsharings is established when the group_send terminates. ■

Finally, we prove APSS Progress.

Lemma 2.11 *The protocol described in this subsection satisfies APSS Progress, assuming Passive Server-Adversaries with Crash Failures and Active Link-Adversaries.*

Proof Sketch: This follows Lemmas 2.6, 2.8, and 2.10. ■

2.3.4 Defending Against Active Server-Adversaries

We finally relax Passive Server-Adversaries with Crash Failures to Active Server-Adversaries, thereby admitting Byzantine behavior of compromised servers. Compromised servers can now launch attacks in a variety of new ways. To motivate the necessary extensions to the protocol, a few attacks are described here. This is not a complete list of all possible attacks but, because we prove that our new protocol is correct under our assumptions, we can have confidence of defense against unlisted attacks in addition to the ones we explicitly discuss here. The detailed final protocol is shown in Appendix A.1.

False notifications. In one class of attacks, a compromised server (or coordinator) could send a false notification (i.e., a bogus contribute or finished message) claiming that (i) a subsharing or (ii) a sharing is established, even though the subsharing or sharing has not been established. The first type of false notification (using contribute messages) might invalidate APSS Progress: a server might wait forever during subshare recovery because no correct servers provide the needed subshares—the subshares might not have even been propagated to those correct servers. The second type of false notification (using finished messages) might invalidate APSS Availability, because servers delete shares and subshares upon receiving such a notification, but these shares and subshares are still needed.

To defend against such false notifications, servers use *self-verifying messages*.[14] A self-verifying message comprises:

- information the sender intends to convey and
- evidence enabling the receiver to verify—without trusting the sender—that the information being conveyed by the message is valid.

Here, the evidence consists of *endorsements* from a set of servers, with each endorsement being a message signed by a server.

Two types of self-verifying messages are used in our APSS protocol. The first (contribute messages) certifies that a certain subsharing has been established.[15] Although a subsharing is established as long as $t + 1$ correct servers each holds the

[14] Although under different names, the concept of self-verifying messages has been used by a number of researchers in connection with Byzantine fault-tolerance [62, 15, 1, 30].

[15] More precisely, the subsharing is established until correct servers delete the subshares, at which point a new sharing must have been established. Because whether a subsharing is established or not becomes uninteresting after a new sharing has been established, being imprecise here is harmless. The same reasoning can be applied to established sharings.

subshares in its share set of that subsharing, confirmations from $2t + 1$ servers are needed in order to be convinced that a subsharing has been established—up to t of these $2t+1$ servers could be compromised in a run. Therefore, the evidence consists of $2t + 1$ signed established messages (endorsements)—a correct server provides an endorsement for a subsharing only if that server holds all the subshares in its share set of that subsharing. Therefore, we have the following lemma:

Lemma 2.12 *If a valid self-verifying* contribute *message for a subsharing* λ *has been constructed, then subsharing* λ *must have been established (before any correct server delete subshares of that subsharing).*

Proof Sketch: A valid self-verifying contribute message for a subsharing λ has as evidence $2t + 1$ signed established messages. A correct server sends an established message for a subsharing only if that server is holding all the subshares in its share set of that subsharing. Because there are up to t compromised servers, at least $t+1$ of these $2t + 1$ senders must be correct. These $t + 1$ correct servers must have held all the subshares in their share sets of subsharing λ (before any correct server delete subshares of that subsharing). By definition, subsharing λ is established. ■

The second class of self-verifying messages (finished messages) is for notification that a certain sharing is established. Here, the evidence consists of $2t + 1$ signed computed messages (endorsements)—a correct server provides its endorsement for a sharing only if the server holds all the shares in its share set. There are at most t compromised servers in an epoch. Therefore, with $2t + 1$ endorsements, at least $t + 1$ endorsements are from correct servers in the current epoch. By definition, these endorsements guarantee that the sharing is established. Therefore, we have the following lemma:

Lemma 2.13 *If a valid self-verifying* finished *message for a sharing* Λ *has been constructed, then sharing* Λ *must have been established (before any correct server in the current epoch delete shares of that sharing).*

Proof Sketch: A valid self-verifying finished message for a sharing Λ has as evidence $2t+1$ signed computed messages. A correct server sends a computed message for a sharing only if that server is holding all the shares in its share set of that sharing. Because there are up to t compromised servers in an epoch, at least $t+1$ of these $2t+1$ senders must be correct in the current epoch. These $t+1$ correct servers must have held all the shares in their share sets of sharing Λ (before any correct server delete shares of that sharing). By definition, sharing Λ is established. ■

Erroneous shares and subshares. In another class of attacks, a compromised server sends erroneous subshares to other servers during subshare propagation, leading to an incorrect new sharing. Self-verifying messages do not help in this case because a server p's action (e.g., generating and propagating a subsharing) might depend on certain secret shares and subshares that p holds. These shares and subshares cannot be made public to all servers. Therefore, other servers cannot always verify whether p is following the protocol.

Servers use verifiable secret sharing to defend against this attack: Validity checks are incorporated into labels of subsharings and sharings in order to allow servers to verify the correctness of shares and subshares.

Use of verifiable secret sharing ensures that it is infeasible to find two different sharings (or subsharings) with the same validity check. Thus, the validity check is sufficient to identify a sharing. Besides the validity check, the version number is also needed to distinguish new sharings from old ones. Thus, a sharing $\overline{s}$ of version number v is labeled $v \circ \overline{vc}.\overline{s}$, and a subsharing $\overline{[s^{:\Lambda}]_i}$ is labeled $\Lambda \circ i \circ \overline{vc}.\overline{[s^{:\Lambda}]_i}$.

The following describes how validity checks are generated and used.

- For any subsharing, the server generating that subsharing (in step 2.1 of Figure 2.6) also generates the validity check for that subsharing using oneWay.

  The validity check is then sent as a part of the label in established or recovered messages (both are used to propagate subshares). A recipient of an establish message or an recovered message then verifies the subshares it receives using the validity check of the subsharing.

- For any sharing, because shares of a new sharing are constructed from subshares instead of directly from the secret, the validity check for this sharing is generated from validity checks of the subsharings (from which this new sharing is generated) using vcConstr (in step 3.2.2 of Figure 2.6). Correctness of the shares can then be verified against the validity check.

Conflicting messages. As the final attack we consider, a compromised coordinator sends different init messages (containing different choices of old sharings) or compute messages (containing different choices of candidate sets of subsharings) to instruct servers to generate many different subsharings or to generate different sharings from different candidate sets of subsharings. This attack could lead to unnecessary resource consumption (both CPU and storage) on servers; it can be regarded as a type of denial of service attack.

As countermeasure, a server always makes sure that init messages or compute messages from a coordinator p in a run are the same (i.e., a coordinator should never make conflicting decisions about which sharing to use or which candidate set of subsharings to use for constructing a new sharing). Every correct server will alert the administrator if conflicting init or compute messages are received from any given coordinator. Such an alert should be accompanied by evidence (in this case, two conflicting messages signed by the compromised coordinator) to prevent a compromised server from sending false alerts to the administrators.

Local detection of inconsistent messages from a coordinator is sufficient to prevent a compromised coordinator from getting more than one new sharing established in a run. To see why, assume otherwise—that is, assume a compromised coordinator causes two new sharings to be established. Because the participation of $2t+1$ servers are needed to have a new sharing established, this means that $(2t+1)+(2t+1)-(3t+1) = t+1$ servers must have participated in both. At least one of the $t+1$ servers is correct in this run. This contradicts the fact that a correct server always detects and reports inconsistencies among messages from the same coordinator.

The same technique is used to prevent a compromised server from having different subsharings for the same share established (by sending conflicting establish messages).

Theorem 2.1 *The APSS protocol presented in Appendix A.1 satisfies APSS Secrecy, APSS Availability, and APSS Progress under the system model described in Section 2.1.*

Proof Sketch: For APSS Secrecy, the proof of APSS Secrecy for Lemma 2.4 still applies for the following reasons:

- Although compromised servers can now exhibit Byzantine behavior, compromised servers cannot influence correct servers regarding the generation of subshares, nor can they cause correct servers to disclose subshares and shares that should not be disclosed—any correct server generates subshares randomly regardless of what compromised servers do; a correct server sends subshares, encrypted, to another server q only if these subshares are in q's share set; and no correct server discloses any shares that are stored locally.

- Any new sharing generated in a run is still independent of any old sharing, for the following reason.

  Consider a new sharing Λ' constructed from a candidate set of subsharings, where each of the subsharings is generated from a share of an old sharing Λ. There are at most t compromised servers in a run. Due to A2, there exists a share $[\overline{s}^{:\Lambda}]_i$ that is not in share set S_p^{Λ} for any compromised server p. No compromised server can generate a valid subsharing for that share. Because of the use of verifiable secret sharing, a compromised server cannot have a bogus subsharing (for $[\overline{s}^{:\Lambda}]_i$) established (i.e., getting $2t+1$ established messages for that subsharing). Therefore, the new sharing Λ' uses at least one subsharing generated by a correct server; that subsharing is thus generated randomly by that server. This ensures that the new sharing is independent of the old sharing Λ. Obviously, the new sharing is also independent of any other old sharing.

- Validity checks, although sent in plaintext, do not disclose corresponding shares and subshares.

To show APSS Availability, it suffices to show that a new sharing has been established when a correct server deletes old shares and subshares. According to the protocol, a correct server deletes old shares and subshares only when it obtains a valid self-verifying finished message which, according to Lemma 2.13, ensures that a new sharing must have been established.

To show APSS Progress, it suffices to show that any correct coordinator eventually terminates its thread. With the use of self-verifying messages and verifiable secret sharing, an erroneous message from a compromised server can be detected and discarded. This effectively constrains the Byzantine behavior of compromised servers and makes the proof for Lemma 2.11 in Section 2.3.3 applicable. In particular, due to Lemma 2.12, every subsharing referenced in a valid self-verifying contribute message is guaranteed to have been established. Also, the use of validity checks in the labels of subsharings and sharings prevents from happening any attacks that lead to inconsistent subsharings and sharings. With this property, a proof similar to the proof of APSS Progress for Lemma 2.11 can be applied here to prove APSS Progress. ■

2.4 Related Work

What distinguishes our APSS protocol from prior work [54, 57] is the weaker system model—our protocols assumes the asynchronous system model and fair links. Previous schemes assume the synchronous system model and reliable links. Such a strong model introduces vulnerabilities to denial of service attacks, although the stronger model does enable the prior work to tolerate more (any minority) compromised servers than does our APSS protocol (which can tolerate only fewer than 1/3 of servers being compromised).

Various proactive threshold cryptography schemes have also been proposed. In [53], proactive schemes for discrete-logarithm based public key cryptography were presented. In [37, 36, 89], proactive *RSA* schemes were proposed. As in [54, 57], all of these schemes are designed for the synchronous system model.

Our APSS protocol is presented with respect to a set of abstract operations (e.g., split, reconstruct, oneWay, and vcConstr). Therefore, any concrete implementation extracted from existing schemes (e.g., the ones proposed in [53, 89]) can be used in our APSS protocol, leading to corresponding proactive protocols that work in the asynchronous system model. In fact, our APSS protocol has been applied to the threshold RSA scheme proposed in [89]. The resulting proactive threshold RSA scheme is used in COCA [111], as described in Chapters 3 and 4, to combat mobile adversary attacks.

Proactive secret sharing and proactive threshold cryptography are specific instances of *proactive security*. The notion of proactive security was introduced by Ostrovsky and Yung in [85], where they studied how to enhance the security of certain secure multi-party protocols [108, 47, 5, 23]. This pioneering work leaves the door open for practical solutions to achieve proactive security, with its focus on general but impractical secure multi-party protocols.

Besides proactive secret sharing and proactive threshold cryptography, other proactive schemes have also been proposed. In [13], Canetti shows how to construct a proactive pseudo-random generator with application to secure sign-on. In [25], a proactive protocol for generating cryptographically secure pseudo-random numbers is presented. In [12], Canetti presents a proactive scheme for maintaining authenticated and secure links among a set of parties despite mobile adversaries. Naor, Pinkas, and Reingold [80] propose a distributed Key Distribution Center that is invulnerable to mobile adversaries, while Garay [39] studies Byzantine agreement with mobile adversaries taken into account. These proactive schemes are vastly different from our APSS protocol. A survey on proactive security can be found in [11].

In addition to theoretical research in proactive security, implementations of threshold cryptography and proactive secret sharing schemes for stronger system models are also reported in [2, 106, 31, 19]. A proactive scheme for threshold cryptography, again under the strong synchronous system model, has also been integrated into the e-vault data repository [56, 40] at IBM T.J. Watson Research Center.

2.5 Concluding Remarks

While protocols that work under weak assumptions exhibit reduced vulnerability to malicious attacks, designing such protocols is hard. The design of our APSS protocol reduces that difficulty by reformulating the problem into one with weaker requirements. Traditionally, a proactive secret sharing protocol creates a single new sharing from a single old sharing. Our APSS protocol instead creates multiple new sharings. The ultimate goals, APSS Secrecy and APSS Availability, remain the same. With such a requirement weakening, our APSS protocol need not solve the agreement problem among servers in the asynchronous system model.

While traditionally studied from the perspective of security, a PSS protocol can also be regarded as a protocol that tolerates Byzantine failures of a subset of the servers. And, the perspective from fault tolerance was invaluable in this work. The fundamental impetus for our use of a combinatorial secret sharing scheme is that it causes shares and subshares to be replicated on servers. Such replication makes applicable certain traditional fault-tolerance solutions, such as recovering states (shares and subshares) by querying correct servers, which would not be possible if different servers have non-overlapping shares and subshares. These fault tolerance solutions eliminate certain cryptographic operations in traditional PSS protocols, such as share recovery that involves rather complex multi-party computation, as described, for example, in [57]. Another notable use of fault-tolerant mechanisms is the employment of self-verifying messages, which have traditionally been used in connection with Byzantine fault-tolerance [67]. By combining self-verifying messages and verifiable secret sharing, our APSS protocol allows servers to detect and discard erroneous messages from other servers, therefore limiting the impact of compromised servers.

Chapter 3

COCA: A Secure Distributed On-line Certification Authority

In a public key infrastructure, a certificate specifies a binding between a name and a public key or other attributes. Over time, public keys and attributes might change—a private key might be compromised, leading to selection of a new public key, for example. The old binding and the certificate that specifies that binding then become *invalid*. A certification authority (CA) attests to the validity of bindings in certificates by digitally signing the certificates it issues and by providing a means for clients to check the validity of certificates. With an on-line CA, principals can check the validity of certificates just before using them. COCA (Cornell On-line Certification Authority), the subject of this chapter, is such an on-line CA.

COCA employs replication to achieve availability and employs proactive recovery with threshold cryptography for digitally signing certificates in a way that defends against mobile adversaries. COCA is designed under a set of qualitatively weak assumptions about communication links and execution timing, making it less vulnerable to malicious attacks that succeed by invalidating the underlying assumptions.

Various forms of defense against denial of service attacks are also incorporated into COCA.

This chapter is organized as follows. Section 3.1 discusses our assumptions about environments in which COCA can be deployed and describes the services COCA provides. Protocols to coordinate COCA servers are the subject of Section 3.2. Section 3.3 elaborates on the mechanisms COCA incorporates to defend against denial of service attacks, followed by a discussion of related work in Section 3.4. Section 3.5 contains some concluding remarks.

3.1 System Model and Services Supported

COCA is implemented by a set of servers, each running on a separate processor in a network. We adopt the same system model described in Section 2.1; that is, we assume Active Server-Adversaries, Active Link-Adversaries, and Asynchronous System.

These assumptions endow adversaries with considerable power. Adversaries can:

- attack servers, provided fewer than 1/3 of the servers are compromised within a given interval,
- launch eavesdropping, message insertion, corruption, deletion, reordering, and replay attacks, provided that the fair link assumption is not violated, and
- conduct denial of service attacks that delay messages or slow servers by arbitrary finite amounts.

The weak system model makes COCA applicable to environments other than the Internet. In fact, COCA is inspired by the work in [110], which articulates the need for an on-line CA to secure an ad hoc network, where mobile hosts communicate through wireless links and act as routers for other hosts. That paper also argues that the on-line CA must work in a weak system model because stronger assumptions might not hold in such a network. COCA can potentially be used as such an CA.

As in Chapter 2, system execution comprises a sequence of protocol-defined windows of vulnerability. Unless otherwise stated, the discussion in this chapter always refers to one window of vulnerability. And, a server is deemed correct in a window of vulnerability if and only if that server is not compromised throughout that period. A precise definition of this window of vulnerability for COCA will be presented in Section 3.1.2.

3.1.1 Operations Implemented by COCA

COCA supports one operation (Update) to create, update, and invalidate bindings; a second operation (Query) retrieves certificates specifying those bindings. A client invokes an operation by issuing a *request* and then awaiting a *response*. COCA expects each request to contain a nonce. Responses from COCA are digitally signed with a COCA service key and include the client's request, hence the nonce[1], thereby enabling a client to check whether a given response was produced by COCA for that client's request.

A request is considered *accepted* by COCA once any correct COCA server receives the request or participates in processing the request[2]; and a request is con-

[1] In the current implementation, requests contain sequence numbers which, along with the client's name, form unique numbers. Therefore, the text of the request itself can serve as the nonce.

[2] The exact time when a request is accepted can be determined only after the window of vulnerability ends, because before then it is unknown whether a COCA server has remained correct

sidered *completed* once some correct server has constructed the response. It might, at first, seem more natural to deem a request "completed" once the client receives a response. But such a definition would make a client action (receipt of a response) necessary for a request to be considered completed, and implementing COCA's

Request Completion: Every request accepted is eventually completed.

guarantee then becomes problematic in the absence of assumptions about clients. But a correct client that makes a request will eventually receive a response from COCA.

Certificates stored by COCA are X.509 [17] compliant. It will be convenient here to regard each certificate ζ simply as a digitally signed attestation that specifies a binding between some name *cid* and some public key or other attributes *pubK*. In addition, each certificate ζ also contains a unique serial number $\sigma(\zeta)$ assigned by COCA, and the following semantics of COCA's Update and Query give meaning to the natural ordering on these serial numbers—namely, that a certificate for *cid* invalidates certificates for *cid* having lower serial numbers.

Update: Given a certificate ζ for a name *cid* and given a new binding $pubK'$ for *cid*, an Update request returns an acknowledgment after COCA has created a new certificate ζ' for *cid* such that ζ' binds $pubK'$ to *cid* and $\sigma(\zeta) < \sigma(\zeta')$ holds.

Query: Given a name *cid*, a Query request $\mathcal{Q}$ returns a certificate ζ for *cid* such that:

throughout the entire window of vulnerability. This uncertainty does not constitute a problem, because the exact time that a request is accepted is uninteresting. What matters is that accepted requests get processed and that requests that are not yet accepted do not impact the state of the service.

(i) ζ was created by some Update request that was accepted before $\mathcal{Q}$ completed.

(ii) For any certificate ζ' for name *cid* created by an Update request that completed before $\mathcal{Q}$ was accepted, $\sigma(\zeta') \leq \sigma(\zeta)$ holds.

By assuming an initial default binding for every possible name, the operation to create a first binding for a given name can be implemented by Query (to retrieve the certificate for the default binding) followed by Update. And an operation to revoke a certificate for *cid* is easily built from Update by specifying a new binding for *cid*.

Update creates and invalidates certificates, so it should probably be restricted to certain clients. Consequently, COCA allows an authorization policy to be defined for Update. In principle, a CA could always process a Query, because Query does not affect any binding. In practice, that policy would create a vulnerability to denial of service attacks, so COCA adopts a more conservative approach discussed in Section 3.3.

The semantics of Update associates larger serial numbers with newer certificates and, in the absence of concurrent execution, a Query for *cid* returns the certificate whose serial number is the largest of all certificates for *cid*. Certificate serial numbers are actually consistent only with a *service-centric* causality relation: the transitive closure of relation $\rightarrow$, where $\zeta \rightarrow \zeta'$ holds if and only if ζ' is created by an Update having ζ as input. Two Update requests $\mathcal{U}$ and $\mathcal{U}'$ submitted, for example, by the same client, serially, and where both input the same certificate, are not ordered by the $\rightarrow$ relation. Thus, the semantics of Update allows $\mathcal{U}$ to create a certificate $\zeta, \mathcal{U}'$ to create a certificate ζ', and $\sigma(\zeta') < \sigma(\zeta)$ to hold—consistent with the service-centric causality relation but the opposite of what is required for serial numbers consistent with Lamport's more-useful potential causality relation [66] (because execution of

$\mathcal{U}$ is potentially causal for execution of $\mathcal{U}'$).

COCA is forced to employ the service-centric causality relation because COCA has no way to obtain information it can trust about causality involving operations it does not itself implement. Clients would have to provide COCA with that information, and compromised clients might provide bogus information. By using service-centric causality, COCA and its clients are not hostage to information about causality furnished by compromised clients.

Update and Query are not indivisible and (as will become apparent in Section 3.2) are not easily made so: COCA's Update involves separate actions for the invalidation and for the creation of certificates. In implementing Update, we contemplated either possible ordering for these actions: Execute invalidation first, and there is a period when no certificate is valid; execute invalidation last, and there is a period when multiple certificates are valid.

Since we wanted Query to return a certificate, having periods with no valid certificate for a given name would have meant synchronizing Query with concurrent Update requests. We rejected this because the synchronization creates an execution-time cost and introduces a vulnerability to denial of service attacks—repeated requests by an attacker for one operation could now block requests for another operation. Our solution is to have Update create the new certificate before invalidating the old one, but it too is not without unpleasant consequences. Both of the following cannot now hold.

(i) A certificate for *cid* is valid if and only if it is the certificate for *cid* with largest serial number.

(ii) Query always returns a valid certificate.

And COCA clients therefore live with a semantics for Query that is more complicated than one might have hoped for.

3.1.2 Bounding the Window of Vulnerability

COCA is designed to operate provided no more than t servers are compromised within a protocol-defined window of vulnerability. The duration of this window of vulnerability is defined in terms of events marking the beginning and completion of proactive recovery protocols, which are execute periodically, as follows: An execution of proactive recovery begins when a correct server executes the protocols for proactive recovery; the execution terminates when servers that remain correct from the beginning of the execution all finish executing these protocols. A window of vulnerability is then defined to be the period of time from the beginning of an execution of proactive recovery to the termination of the next execution of proactive recovery. Thus, every execution of the proactive recovery protocols is part of two successive windows of vulnerability.

Each execution of proactive recovery reconstitutes the state of each COCA server (which might have been corrupted during the previous window of vulnerability) and obsoletes keys and shares an adversary might have obtained by compromising servers. Execution of proactive recovery for COCA involves more than a run of proactive secret sharing that refreshes shares of a secret maintained by the service. The following shows the components of proactive recovery.

Limiting the Utility of Compromised Keys

Server Keys. Each COCA server maintains a private/public key pair, with the public key known to all COCA servers. These public keys allow servers to authenticate the senders of messages they exchange with other servers.

Public keys of COCA servers are not given to COCA clients so that clients need not be informed of changed server keys—attractive in a system with a large number of clients and where a proactive recovery protocol periodically refreshes server keys. But without knowledge of server keys, clients cannot easily determine the COCA server that sent a message. This, in turn, precludes voting or other schemes in which a client synthesizes or counts responses from individual COCA servers to obtain COCA's response.

Server keys are refreshed during proactive recovery to ensure that, after proactive recovery, no adversary could impersonate a server using an old server private key. How to refresh these server keys has been discussed in Section 2.1 (the last paragraph before Section 2.1.1).

Service Key. There is one service private/public key pair. It is used for signing responses and certificates. All clients and servers know the service public key.

The service private key is held by no COCA server, for obvious reasons. Instead, different shares of the key are stored on each of the servers, and threshold cryptography is used to construct signatures on responses and certificates. To sign a message:

(1) each COCA server generates a *partial signature* from the message and that server's share of the service private key;[3]

(2) some COCA server combines these partial signatures and obtains the signed

[3]COCA employs a combinatorial secret sharing, which has been described in Chapter 2. A share of a server is thus a set of shares, and a partial signature has to be generated using every share in that set. And, $t+1$ sets of these partial signatures, each generated by a server, are used for constructing a signature.

message.[4]

With $(n, t+1)$ threshold cryptography, $t+1$ or more partial signatures are needed in order to generate a signature. An adversary must therefore compromise $t+1$ servers in order to forge COCA signatures.

During proactive recovery, these shares are refreshed using the proactive secret sharing protocol described in Chapter 2, instantiated with modules from the threshold cryptography scheme used.

Server State Recovery

In addition to generating new server keys and new shares of the service key, COCA also periodically refreshes the states of its servers. This is done as part of proactive recovery. The state of a COCA server consists of a set of certificates. In theory, this state could be refreshed by performing a Query request for each name that could appear in a certificate. But the cost of that becomes prohibitive when many certificates are being stored by COCA. So instead, during proactive recovery, a list with the name and serial number for every valid certificate stored by each server is sent to every other. Upon receiving this list, a server retrieves any certificates that appear to be missing. Certificates stored by COCA servers are signed (by COCA)—a certificate retrieved from another server can thus be checked to make sure it is not bogus. The certificate serial numbers enable servers to determine which of their

[4] One might think partial signatures could be combined by clients (instead of COCA servers) to obtain signed messages, but that introduces a vulnerability to denial of service attacks. Lacking COCA server public keys, clients do not have a way to authenticate the origins of messages conveying the partial signatures. Therefore, a client could be bombarded with bogus partial signatures, and only by actually trying to combine these fragments—an expensive enterprise—could the bona fide partial signatures be identified.

certificates have been invalidated (because a certificate for that same name but with a larger serial number exists).

There is one non-obvious point of interaction involving the protocols used to refresh server keys and service key shares. To satisfy Request Completion (of Section 3.1.1), an accepted request that has not been completed when a window of vulnerability ends must become an accepted request in the next window of vulnerability. Such a request can be regarded as part of the state that needs to be propagated to other servers during server recovery. More specifically, a correct server, when executing the proactive recovery protocol, resubmits to all servers any request that is then in progress and makes sure that it receives acknowledgments from at least $t+1$ servers. This ensure that some server that is correct in this next window of vulnerability will receive that request. (Recall that this execution of proactive recovery also belongs to the next window of vulnerability.) Thus, by definition, in-progress accepted requests in the previous window of vulnerability remain accepted in the next one. To avoid new requests delaying the completion of an execution of proactive recovery (a potential way of launching denial of service attacks), servers could choose to ignore all messages except those used for proactive recovery during execution of the proactive recovery.[5]

In practice, windows of vulnerability tend to be long (*viz.* days) relative to the time (seconds) required for processing a Query or Update request. It is thus extremely unlikely that a request restarted in a subsequent window of vulnerability would not be completed before proactive recovery is again commenced.

[5] Ignoring (a finite number of) messages is allowed by Active Link-Adversaries—for example, a client cannot distinguish between an ignored request and one that never reached COCA. It is not a problem in practice either, because the execution time for the proactive recovery protocol should be short.

3.2 Protocols

In COCA, every client request is processed by multiple servers and every certificate is replicated on multiple servers. The replication is managed as a dissemination Byzantine quorum system [69], which is feasible because we have assumed $3t+1 \leq n$ holds. In a dissemination Byzantine quorum system, servers are organized into quorums satisfying:[6]

Quorum Intersection: The intersection of any two quorums contains at least one correct server.

Quorum Availability: A quorum comprising only correct servers always exists.

And every client request is processed by all correct servers in some quorum.

Detailed protocols for Query and Update appear in Appendix A; in this section, we explain the main ideas. The technical challenges are:

- Because requests are processed by a quorum of servers but not necessarily by all correct COCA servers, different correct servers might process different Update requests. Consequently, different certificates for a given name *cid* are stored by correct servers. Certificate serial numbers provide a solution to the problem of determining which of those is the correct certificate.

- Because clients do not know COCA server public keys, a client making a request cannot authenticate messages from a COCA server and, therefore, cannot determine whether a quorum of servers has processed that request.

[6] Provided there are $3t+1$ servers and at most t of those servers may be compromised, the quorum system $\{Q : |Q| = 2t+1\}$ constitutes a dissemination Byzantine quorum system. For simplicity, we assume $n = 3t+1$ holds; the protocols are easily extended to cases where $n > 3t+1$ holds.

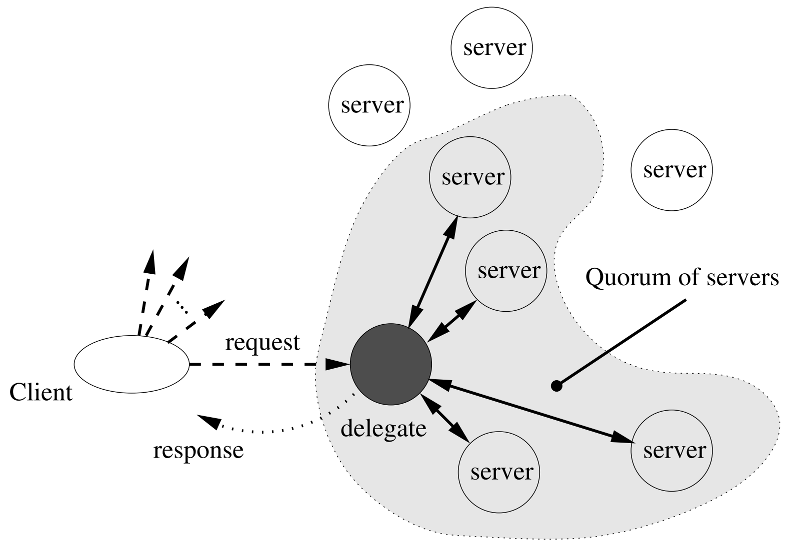

Figure 3.1: Overview of Client Request Processing.

The solution is for some COCA servers to become *delegates* for each request. A delegate presides over the processing of a client request and, being a COCA server, can authenticate server messages and assemble the needed partial signatures from other COCA servers. A client request is handled by $t+1$ delegates to ensure that at least one of these delegates is correct.

- Because communication is done using fair links, retransmission of messages may be necessary.

Figure 3.1 summarizes this high-level view of how COCA operates by depicting one of the $t+1$ delegates and the quorum of servers working with that delegate to handle a client request.

Protocol Details

Certificate Serial Numbers. The serial number $\sigma(\zeta)$ for a COCA certificate ζ is a pair $\langle v(\zeta), h(R_\zeta)\rangle$, where $v(\zeta)$ is a *version number* and $h(R_\zeta)$ is a collision-resistant hash of the Update request $\mathcal{R}_\zeta$ that led to creation of ζ. Version numbers encode the service-centric causality relation as follows.

- The first certificate created to specify a binding for a name *cid* is assigned version number 0.

- A certificate ζ' produced by an Update given certificate ζ is assigned version number $v(\zeta') = v(\zeta) + 1$.

Because different requests have different collision-resistant hashes, certificates created by different requests have different serial numbers. The usual lexicographic ordering on serial numbers yields the total ordering on serial numbers we seek—an ordering consistent with the transitive closure of the $\rightarrow$ relation.

Note that, even with serial numbers on certificates, the same new certificate will be created by COCA if an Update request is re-submitted. This is because the serial number of a certificate is entirely determined by the arguments in the request that creates the certificate. So, Update requests are idempotent, which proves useful for tolerating compromised COCA servers.

Determining a Response for Query. COCA Update requests are processed by correct servers in some quorum and not necessarily by all correct COCA servers. Consequently, a correct COCA server p can be ignorant of certificates having larger serial numbers than p stores for a name *cid*. Part (ii) in the specification for Query implies that all completed Update requests (hence, all certificates) are taken into account in determining the response to a Query request $\mathcal{Q}$. To satisfy this, a quorum of servers must be engaged in processing $\mathcal{Q}$. All servers are contacted and responses from a quorum of servers are expected. Each server in a quorum Q responds with the certificate (signed by COCA) having the largest serial number among all certificates (for cid) known to the server. The certificate ζ that has the largest serial number among the correctly signed certificates received in the responses from Q is the response to $\mathcal{Q}$.

This choice of ζ satisfies parts (i) and (ii) in the specification for Query. Part (i) stipulates that a certificate returned for Query is created by an accepted Update. This condition will be satisfied by ζ because a certificate is signed by COCA only after the Update request creating that certificate has been accepted. The $(n, t+1)$ threshold cryptography being employed for digital signatures requires cooperation (collusion) by more than t servers in order to sign a certificate. Given our assumption of at most t compromised servers, we conclude that there are not enough compromised servers to create bogus signed certificates. Therefore, when a certificate is signed, a correct server must have participated in processing the request that created the certificate; the request creating the certificate had to have been accepted. The signature on certificates also prevents a compromised server from submitting a bogus certificate with an arbitrarily large serial number during the processing of a Query request without being detected.

Part (ii) of the Query specification requires that, for any Update request $\mathcal{U}$ naming cid and completed before $\mathcal{Q}$ is accepted, $\sigma(\zeta') \leq \sigma(\zeta)$ must hold where ζ' is the certificate created by $\mathcal{U}$. This holds for the implementation outlined above due to Quorum Intersection, because some correct server p in Q must also be in the quorum that processed $\mathcal{U}$. Let certificate ζ_p be p's response for $\mathcal{Q}$. Because p always chooses the certificate for cid with the largest serial number, $\sigma(\zeta') \leq \sigma(\zeta_p)$ holds. Because ζ is the certificate that has the largest serial number among those from all servers in Q, $\sigma(\zeta_p) \leq \sigma(\zeta)$ holds. Therefore, $\sigma(\zeta') \leq \sigma(\zeta)$ holds.

The Role of Delegates. After making a request $\mathcal{R}$, a client awaits notification that $\mathcal{R}$ has been processed. Every request is processed by all correct servers in some quorum; the client must be notified once that has occurred. Direct notification by servers in the quorum is not possible because clients do not know the public keys for COCA servers and, therefore, have no way to authenticate messages from those servers. So, instead, a COCA server is employed as a delegate to detect the completion of request processing and then to notify the client, as follows.

- To process a Query request $\mathcal{Q}$ for name *cid*, the delegate obtains certificates from a quorum of servers, picks the certificate ζ having the largest serial number, and uses the threshold signature protocol to produce a signed response containing ζ:
  1. Delegate forwards $\mathcal{Q}$ to all COCA servers.
  2. Delegate awaits certificates for *cid* from a quorum of COCA servers.
  3. Delegate picks the certificate ζ having the largest serial number of those received in step 2.
  4. Delegate invokes COCA's threshold signature protocol to sign a response containing ζ; that response is sent to the client.
- To process an Update request $\mathcal{U}$ for name *cid*, the delegate constructs the certificate ζ for the given new binding (using the threshold signature protocol to have COCA digitally sign it) and then sends ζ to all COCA servers. A server p replaces the certificate ζ_p^{cid} for *cid* that it stores with ζ if and only if the serial number in ζ is larger than the serial number in ζ_p^{cid}:
  1. Delegate constructs a new certificate ζ for *cid*, using the threshold signature protocol to sign the certificate.

2. Delegate sends ζ to every COCA server.

3. Every server, upon receipt, replaces the certificate for *cid* it had been storing if the serial number in ζ is larger. The server then sends an acknowledgment to the delegate.

4. Delegate awaits these acknowledgments from a quorum of COCA servers.

5. Delegate invokes COCA's threshold signature protocol to sign a response; that response is sent to the client.

Quorum Availability ensures that a quorum of servers are always available, so step 2 in Query and step 4 in Update are guaranteed to terminate. Since quorums contain $2t + 1$ servers, compromised servers cannot prevent a delegate from using $(n, t + 1)$ threshold cryptography in constructing the COCA signature for a certificate or a response. Thus, step 4 in Query and steps 1 and 5 in Update cannot be disrupted by compromised servers.

A compromised delegate might fail to complete the protocol just outlined for processing Query and Update requests. COCA ensures that such behavior does not disrupt the service by enlisting $t+1$ delegates (instead of just one) for each request. At least one of $t + 1$ delegates must be correct, and this delegate can be expected to follow the Query and Update protocols. So, we stipulate that a (correct) client making a request to COCA submits that request to $t+1$ COCA servers; each server then serves as a delegate for processing that request.[7]

With $t+1$ delegates, a client might receive multiple responses to each request, and each request might be processed repeatedly by some COCA servers. The duplicate responses are not difficult for clients to deal with—a response is discarded if it is

[7] An optimization discussed in Chapter 4 makes it possible for clients, in normal circumstances, to submit requests to only a single delegate.

received by a client not waiting for a request to be processed. That each request might be processed repeatedly by some COCA servers is not a problem either, because COCA's Query and Update implementations are idempotent.

But a compromised client might not submit its request to $t+1$ delegates, as is now required. We must ensure that Request Completion is not violated. The problem occurs if the delegates receiving that request $\mathcal{R}$ execute the first step of Query or Update processing and then halt. Correct COCA servers have now participated in the processing of $\mathcal{R}$, so (by definition) $\mathcal{R}$ is accepted. Yet no (correct) delegate is responsible for $\mathcal{R}$. Request $\mathcal{R}$ is never completed, and Request Completion is violated.

We must ensure that some correct COCA server becomes a delegate for each request that has been received by any correct COCA server. The solution is straightforward:

- Messages related to the processing of a client request $\mathcal{R}$ contain $\mathcal{R}$.
- Whenever a COCA server receives a message related to processing a client request $\mathcal{R}$, that server becomes a delegate for $\mathcal{R}$ if it is not already serving as one.

The existence of a correct delegate is now guaranteed for every request that is accepted.

Self-Verifying Messages. Compromised delegates could also attempt to produce an incorrect (but correctly signed) response to a client by sending erroneous messages to COCA servers. For example, in processing a Query request, a compromised delegate might construct a response containing a bogus or invalidated certificate and try to get other servers to sign that; in processing an Update request, a compromised delegate might create a fictitious binding and try to get other servers to sign that; or when processing an Update request, a compromised delegate might not disseminate the updated binding to a quorum (causing the response to a later Query to contain an invalidated certificate).

COCA's defense against erroneous messages from compromised servers is self-verifying messages, as introduced in Section 2.3.4. More specifically, in COCA, every message a delegate sends on behalf of a request contains a transcript of relevant messages previously sent and received in processing that request (including the original client request). Because messages contained in the transcript are signed by their senders, a compromised delegate cannot forge the part of the transcript contributed by correct servers. And, because the members of the quorum participating in the protocol are known to all, the receiver of such a self-verifying message can independently establish whether messages sent by a delegate are consistent with the protocol and the messages received.[8]

Returning to the erroneous message examples given earlier, here is how the self-verifying messages used in COCA prevent subversion of the service:

- Compromised delegates cannot cause COCA to sign a Query response containing a bogus or invalidated certificate, because messages instructing servers to sign such a response must contain signed messages from a quorum of servers, where these signed messages contain the certificates submitted by servers for this Query.

[8] In [49], Gong and Syverson introduce the notion of a *fail-stop protocol*, which is a protocol that halts in response to certain attacks. One class of attacks is thus transformed into another, more benign, class. Our self-verifying messages can be seen as an instance of this approach, transforming certain Byzantine failures to more-benign failures.

- Compromised delegates are prevented from creating a certificate that specifies a fictitious binding, because every message pertaining to an Update request must include the original client-signed request. COCA servers check that request before signing a new certificate.

- Compromised delegates that do not disseminate some new certificate to a quorum are foiled, because every subsequent message the delegate sends in processing this request must contain the signed responses from a quorum of servers attesting that they received the new certificate.

Communicating using Fair Links. Active Link-Adversaries assumes only fair links. As in Section 2.3.3, retransmissions are used to approximate reliable communication using fair links. More specifically, the protocols in COCA are structured as a series of multicasts[9], with information piggybacked on the acknowledgments. A client starts by doing a multicast to $t+1$ delegates; the signed response from a single delegate can be considered the acknowledgment part of that multicast. A delegate then interacts with COCA servers by performing multicasts and awaiting responses from servers. For the threshold signature protocol, $t+1$ correct responses suffice; for retrieving and for updating certificates, responses from a quorum of servers are needed. Thus, with at least $2t+1$ correct servers, COCA's multicasts always terminate due to Quorum Availability since a delegate is now guaranteed to receive enough acknowledgments at every step and, therefore, eventually that delegate will finish executing the protocol for a request.

[9] Each multicast is an instance of group_send introduced in Section 2.3.3.

3.3 Defense Against Denial Of Service Attacks

A large class of successful denial of service attacks work by exploiting an imbalance between the resources an attacker must expend to submit a request and the resources the service must expend to satisfy that request, as has been noted, for example, in [58, 72, 73]. If making a request is cheap but processing one is not, then attackers have a cost-effective way to disrupt a service—submit bogus requests to saturate server resources. A service, like COCA, where request processing involves expensive cryptographic operations and multiple rounds of communication is especially susceptible to such resource-clogging attacks.

COCA implements three classic defenses: request-processing authorization, resource management, and caching, as outlined in Section 1.3, to blunt resource-clogging denial of service attacks. The details for COCA's realizations of these defenses constitute the bulk of this section.

Note, however, that our Asynchrony System assumption is an important defense against denial of service attacks, too. An attacker stealing network bandwidth or cycles from processors that run COCA servers is not violating the assumption needed for COCA's protocols to work. Such a "weak assumptions" defense is not without a price, however. Implementing real-time service guarantees on request processing requires a system model with stronger assumptions than we are making. Consequently, COCA can guarantee only that requests it receives are processed eventually. Those who equate availability with real-time guarantees (e.g., [46, 109, 76, 77]) would not be satisfied by an eventuality guarantee.

Finally, COCA employs connectionless protocols for communication with clients and servers, so COCA is not susceptible to connection-depletion attacks such as the well-known TCP SYN flooding attack [100]. But the proactive secret sharing protocol in the current COCA implementation does use SSL (Secure Socket Layer) [38] and is, therefore, subject to certain denial of service attacks. This vulnerability is not inherent to the protocol and could be eliminated by restricting the rate of SSL connection requests, by reprogramming the proactive secret sharing protocol, or by adopting the mechanisms described in [58].

3.3.1 Request-Processing Authorization

Each message received by a COCA server must be signed by the sender. The server rejects messages that

- do not pass certain sanity checks,
- are not correctly signed, or
- are sent by clients or servers that, from messages received in the past, were deemed by this server to have been compromised.

An invalid self-verifying message, for example, causes the receiver r to judge the sender s compromised, and the request-processing authorization mechanism at r thereafter will reject messages signed by s (until instructed otherwise, perhaps because s has been repaired).

Verifying a signature is considerably cheaper than executing an Update or Query request (which involves threshold cryptography and multiple rounds of message exchange). But verifying a signature is not free, and an attacker might still attempt to flood COCA with requests that are not correctly signed. Should this vulnerability ever become a concern, we would add a still-cheaper authorization check that requests must pass before signature verification is attempted. Cookies [59, 84], hash chains [61], and puzzles [58] are examples of such checks.[10]

Of course, any server-based mechanism for authorization will consume some server resources and thus could itself become the target of a resource-clogging attack, albeit an attack that is more expensive to launch by virtue of the additional authorization mechanism. An ultimate solution is authorization mechanisms that also establish the origin of the request being checked, since fear of discovery and reprisal is an effective deterrent [84].

3.3.2 Resource Management

Because requests are signed, COCA servers are able to identify the client and/or server associated with each message received. And this enables each COCA server to limit the impact that any compromised client or server can have. In particular, each COCA server stores messages it receives in one of a set of *input queues* and employs some scheduler to service those queues. The queues and scheduler limit the fraction of a server's cycles that can be co-opted by an attacker.[11] Others have also advocated similar approaches [46, 109, 76, 77].

Our COCA prototype has a configurable number of input queues at each server. A round-robin scheduler services these queues. Client requests are stored on one or

[10] A related notion is *proofs of work* [32], which requires a client to attach with each request (or message) a proof that the client has consumed a certain amount of resource (often on a hard computational problem) for this request or message. This increases the resource consumption on the client side and limits the ability of a client to launch a successful denial of service attack.

[11] Clearly, this offers no defense against distributed denial of service attacks [94] in which an attacker, masquerading as many different clients, launches attacks from different locations. If the clients involved in such an attack can be detected, then their requests could be isolated using COCA's queues and scheduler. But solving the difficult problem—determining which clients are involved in such an attack—is not helped by this COCA mechanism.

more queues, and messages from each COCA server are stored on a separate queue associated with that server. Duplicates of an element already present on a queue are never added to that queue. Each server queue has sufficient capacity so replays of messages associated with a request currently being processed cannot cause the queue to overflow (since that would constitute a denial of service vulnerability).

In a production setting, we would expect to employ a more sophisticated scheduler and a rich method for partitioning client requests across multiple queues. Clients might be grouped into classes, with requests from clients in the same administrative domain stored together on a single queue.

3.3.3 Caching

Replays of legitimate requests are not rejected by COCA's authorization mechanism. Nor should they be, since the assumption of fair links forces clients to resend each request until enough acknowledgments are received. But attackers now have an inexpensive way to generate requests that will pass COCA's authorization mechanism, and COCA must somehow defend against such replay-based denial of service attacks.

There are actually two ways to redress an imbalance between the cost of making requests and the cost of satisfying them. One is to increase the cost of making a request, and that is what the signature checking in COCA's authorization mechanism does. A second is to decrease the cost of processing a request. COCA also embraces this latter alternative. Each COCA server caches responses to client requests and caches the results of expensive cryptographic operations for requests that are in progress, as also suggested in [84, 14]. Servers use these cached responses instead of recalculating them when processing replays.

The cache for client responses is managed differently than the cache for in-progress cryptographic results. We first discuss the client-response cache. Each COCA server cache has finite capacity, so all responses to clients cannot be cached indefinitely. If the server cache is to be effective against replays submitted by clients, we must minimize the chance of such replays causing cache misses (and concomitant costly computation by the server). The solution is to ensure that client replays are forced to exhibit a temporal locality consistent with the information being cached. In particular, by caching COCA's response for each client's most recent request,[12] by restricting clients to making one request at a time, and by having clients associate ascending sequence numbers with their requests, older requests not stored in the cache can be rejected as bogus by COCA's authorization mechanism.

Because requests are processed by a quorum of COCA servers—and not necessarily by all COCA servers—a given server's cache of client responses might not be current. Thus, a replay request signed by client c to some server s might have a sequence number that is larger than the sequence number for the last response cached at s for c. The larger sequence-numbered request would not be rejected by s and could not be satisfied from the cache—the request would have to be processed. But with quorums comprising $2t + 1$ of the $3t + 1$ COCA servers, at most t such replays can lead to computation by COCA servers. COCA's implementation further limits susceptibility to these attacks. Whenever a COCA server sends a response to a client, that response is also sent to all other COCA servers. Each server is thus quite likely to have cached the most recent response for every client request.

Clients are not the only source of replay-based denial of service attacks. Com-

[12]In a system with a million clients, this client cache would be roughly 5 gigabytes because approximately 5K bytes is needed to store a client's last request and COCA's response.

promised servers also could attempt such attacks. COCA's defense here too is a cache. Servers cache results from all expensive operations, such as computing validity checks of subshares for proactive secret sharing and computing partial signatures for in-progress requests. The cache at each server is sufficiently large to handle the maximum number of requests that all COCA servers could have in-progress at any time. A total of 60K bytes suffices for a cache to support one client request, when X.509 certificates do not exceed 1024 bytes (which seems reasonable given observed usage).

COCA limits the number of requests that can be in-progress at any time by having each delegate limit the number of requests it initiates. Of course, a compromised delegate would not respect such a bound. But recall that COCA servers are notified when responses are sent, so a server can estimate the number of concurrent requests that each server (delegate) has in progress. COCA servers can thus ignore messages from servers that initiate too many concurrent requests.

3.4 Related Work

Systems. A fault-tolerant authentication service [92] for supporting secure groups in the Horus system appears to be the first use of threshold cryptography along with replication for implementing a CA. That led to the design and implementation of Ω [93], a stand-alone general-purpose CA having more ambitious functionality, performance, and robustness goals. Unlike COCA, this early work was not intended to resist denial of service attacks or mobile adversaries. And, as discussed below, some vulnerability to denial of service attacks seems to be inherent. On the other hand, Ω does provide clients with key escrow operations, something that COCA does not currently support.[13]

Ω was built using middleware (called Rampart [90, 91]) that implements process groups in an asynchronous distributed system where compromised processors can exhibit arbitrary behavior. The Rampart middleware manages groups of replicas and removes non-responsive members from process groups to ensure the system does not stall due to compromised replicas. However, it is impossible to distinguish between slow and halted processors in an asynchronous system, so Rampart uses timeouts for identifying processors that might be compromised. A correct but slow server might thus be removed from a process group, which constitutes a denial of service vulnerability. In addition, because making group membership changes involves expensive protocols, an adversary can launch denial of service attacks against Rampart by instigating membership changes. Furthermore, neither Rampart nor Ω employs proactive recovery, so these systems are vulnerable to mobile adversaries.

An approach related to Rampart is embodied in the Byzantine Fault Tolerance work (BFT) discussed in [15]. BFT extends the state machine approach [66, 98] to tolerate arbitrary failures in an asynchronous system. State machines are more powerful than the dissemination Byzantine quorum systems used by COCA. The additional power is not needed for implementing COCA's Query and Update but would be needed if the specification of Update were changed to take a less service-centric view of causality than COCA now takes. BFT also is extremely fast because, wherever possible, it uses MACs (message authentication codes) instead of public key cryptography. This replacement would also boost COCA's performance, although executing some public key cryptographic operations is inevitable in COCA

[13] The same threshold decryption and blinding [20, 21, 22] that Ω uses for supporting this additional functionality would allow COCA to support these features too.

for signing certificates and responses to clients.

As with COCA, BFT employs proactive recovery [16]. Even though BFT replicas do not store shares of a service private key, these replicas do need to refresh their key pairs and shared secret keys to combat mobile adversaries—secure co-processors are assumed for this task. BFT takes denial of service attacks into account and employs defenses similar to the mechanisms discussed for COCA in Section 3.3 [14]. A performance comparison would be interesting but no suitable data for BFT are yet available.

The PASIS (Perpetually Available and Secure Information Systems) architecture [107] is intended to support a variety of approaches—decentralized storage system technologies, data redundancy and encoding, and dynamic self-maintenance—that have been used in constructing survivable information storage systems. Once PASIS has been implemented, it should be possible to program COCA's Query and Update in any number of ways. What is not clear is whether PASIS will support COCA's optimizations (see Chapter 4) or defense against denial of service attacks, since doing so would depend on PASIS selecting a weak model of computation and supporting access to low-level details of the PASIS building-block protocols.

Replication and secret sharing are the basis for a fault-tolerant and secure key distribution center (KDC) described in [48]. In this design, each client/KDC-server pair shares a separate secret key. The KDC allows two clients to establish their own shared secret key, and does so using protocols in which no single KDC-server ever knows that shared secret key. In fact, an attack must compromise a significant fraction of the KDC's servers before any keys the KDC establishes to link clients would be revealed.

Also related to COCA are various distributed systems that implement data repositories with operations analogous to Query and Update. Phalanx [70] is particularly relevant, because it is intended for a setting quite similar to COCA's (*viz.* asynchronous systems in which compromised servers exhibit arbitrary behavior) and can be used to implement shared variables having similar semantics to COCA's certificates. (COCA's certificates can be regarded as shared variables that are being queried and updated.)

Phalanx [70] supports two different implementations of read (Query) and write (Update) for shared variables. One implementation is optimized for *honest writers*, clients that follow specified protocols or exhibit benign failures (crash, omission, or timing failures); a second implementation tolerates *dishonest writers*, clients that can exhibit arbitrary behavior when faulty. Phalanx employs a masking Byzantine quorum system [69] for dishonest writers and employs a dissemination quorum system for honest writers.[14]

In Phalanx's honest writer protocol, writers must be trusted to sign the objects being stored. Although, as with this honest writer protocol, COCA also uses a dissemination quorum system, COCA's protocols do not require clients to be trusted—COCA servers store objects (certificates) that are signed by COCA's service key, and that prevents compromised COCA servers from undetectably corrupting objects they store. Another point of difference between COCA and Phalanx is the manner in which clients verify responses from the service. In Phalanx, every client must know the public key of every server, whereas in COCA each client need know only the single public key for the service.

[14]In a masking Byzantine quorum system, Quorum Intersection is strengthened to stipulate that the intersection of any two quorums always contains more correct replicas than compromised replicas. A masking Byzantine quorum system can tolerate compromise of as many as one fourth of servers. A dissemination quorum system tolerates one third of its servers being compromised.

The e-vault data repository at IBM T.J. Watson Research Center implements Rabin's information dispersal algorithm [88] for storing and retrieving files [56, 40]. Information is stored in e-vault with optimal space efficiency. But the e-vault protocols assume a synchronous model of computation and, thus, involve stronger assumptions about execution timing and delivery delays than we make for COCA. Such stronger assumptions constitute a denial of service vulnerability—an attacker that is able to overload processors or clog the network can invalidate these assumptions and cause protocols to fail. Like with COCA, clients of e-vault communicate with the system through a single server (there called a gateway).

Public Key Infrastructure. Most previous work on public key infrastructure (e.g., [42, 104, 68, 60]) advocates an off-line CA, which issues certificates and certificate revocation lists (CRLs). Trade-offs associated with CRLs and related mechanisms are discussed in [95, 78, 63, 35, 71]. Stubblebine [103] compares different mechanisms to deal with revoked certificates and argues that a single on-line service is impractical for both performance and security reasons, advocating a solution with an off-line identification authority and an on-line revocation authority. COCA could be used to implement such a solution.

Alternatives to using an off-line CA include on-line certificate status checking (OCSP) [79, 78, 63] and on demand revocation lists [71]. These rely on some sort of trusted on-line service (a responder, a validation authority, and so on) and therefore our experience implementing and deploying COCA is directly applicable.

3.5 Concluding Remarks

Off-line operation of a CA—an air gap—is clearly an effective defense against network-borne attacks. For that reason, the traditional wisdom has been to keep a CA off-line as much as possible. This approach, however, trades one set of vulnerabilities for another. A CA that is off-line cannot be attacked using the network but it also cannot update or validate certificates on demand. Vulnerability to network-borne attacks is decreased at the expense of increased client vulnerability to attacks that exploit recently invalidated certificates.

By being an on-line CA, COCA makes the trade-off between vulnerabilities differently. COCA's vulnerability to network-borne attacks is greater, but its clients' vulnerability to attacks based on compromised certificates is reduced. Marrying COCA with an off-line CA would achieve the advantages of both [68, 103, 79]. The off-line CA would issue certificates for clients, and COCA would validate (on demand) these certificates. Revocation of a certificate would thus be achieved by notifying COCA; issuance of a new certificate would require interacting with the off-line CA.

COCA, in composing mechanisms for fault-tolerance and security, implements a secure multi-party computation [108, 47, 5, 23]. Just as agreement protocols and their kin have become part of the vocabulary of system builders concerned with fault-tolerance, so too must protocols for secure multi-party computation if we aspire to build trustworthy systems. Query and Update have relatively simple semantics. For building richer services that are fault-tolerant and secure, we must become facile with implementing richer forms of secure multi-party computation—protocols that enable n mutually distrusted parties to compute a publicly known function on a secret input they share without disclosing the input or what input shares are held by the parties.

Careful attention paid to the assumptions that characterize COCA's environment led to a system with inherent defenses to denial of service attacks. While additional denial of service defenses are described in Section 3.3, enumerating and countering specific attacks can be unsettling as a sole means of defense: What if some unanticipated attack is launched? Defenses based on weak assumptions are, by construction, accompanied by a characterization of the vulnerabilities—the assumptions themselves. And, by their very nature, weak assumptions are difficult to violate.

Chapter 4

COCA Implementation and Performance Measurements

Our COCA prototype is approximately 35K lines of new C source; it employs a threshold RSA scheme and a proactive threshold RSA scheme [89] (using 1024-bit RSA keys) that we built using OpenSSL [83]. Certificates stored on COCA servers are in accordance with X.509 [17], with the COCA's serial number embedded in the X.509 serial number.

Much of the cost and complexity of COCA's protocols is concerned with tolerating failures and with defending against attacks, even though failures and attacks are infrequent today. We normally expect:

N1: Servers will satisfy stronger assumptions about execution speed.

N2: Messages sent will be delivered in a timely way.

Our COCA prototype is optimized for these normal circumstances. Wherever possible, redundant processing is delayed until there is suspicion that assumptions N1 and N2 no longer hold.

In particular, our COCA prototype sequences when servers start serving as delegates for client requests already in progress. This reduces the number of delegates when N1 and N2 hold, hence it reduces the cost of request processing in normal circumstances. The refinements to the protocols of Section 3.2 are:

- A client sends its request only to a single delegate at first. If this delegate does not respond within some timeout period, then the client sends its request to another t delegates, as required by the protocols in Section 3.2.

- A server that receives a message in connection with processing some client request $\mathcal{R}$ and that is not already serving as a delegate for $\mathcal{R}$ does not become a delegate until some timeout period has elapsed.

- A delegate p sends a response to all COCA servers, in addition to sending the response to the client initiating the request, after the request has been processed. After receiving such a response, a server that is not a delegate for this request will not become one in the future; a server that is serving as a delegate aborts that activity.

  A cached response will be forwarded to a server q whenever q instructs p to participate in the processing of a request that has already been processed. Upon receiving the forwarded response, q immediately terminates serving as a delegate for that request.

A more general description of such an optimization with applications to the threshold signature protocol and to the APSS protocol can be found in Section 5.3.

This chapter is organized as follows: Section 4.1 describes performance measurement results for COCA deployment on a local area network. COCA performance on an Internet deployment is the main subject of Section 4.2. In Section 4.3, COCA

Table 4.1: Performance of COCA over a LAN.

COCA Operation	**Mean** (*msec*)	**Std dev.** (*msec*)
Query	629	16.7
Update	1109	9.0
PSS	1990	54.6

performance under various simulated denial of service attacks is reported and analyzed.

4.1 Local Area Network Deployment

These experiments over a local area network (LAN) involved a COCA prototype comprising four servers (i.e., $n = 4$ and $t = 1$) communicating through a 100Mbps Ethernet. The servers were Sun E420R Sparc systems running Solaris 2.6, each with four 450 MHz processors. The round-trip delay for a UDP packet between any two servers on the Ethernet is usually under 300 microseconds.

Table 4.1 gives times for COCA functions executing in isolation when assumptions N1 and N2 hold. We report the delay for Query, for Update, and for a round of proactive secret sharing (PSS). The reported sample means and sample standard deviations are based on 100 executions. All samples are located within 5% of the mean.

To better understand the origin of these delays, we report in Table 4.2 the (percentage) contribution of certain CPU-intensive cryptographic operations. For Query and Update, we measured the time spent in generating partial signatures and in signing messages. For proactive secret sharing, we measured the delay associated with the one-way function (oneWay)[1], with message signing, and with computation

[1]The one-way function involves expensive modular exponentiation and is needed to implement verifiable secret sharing, as described in Section 2.2.3.

Table 4.2: Breakdown of Costs for COCA over a LAN.

	Query	Update	PSS
Partial Signature	64%	73%	
Message Signing	24%	19%	22%
One-Way Function			51%
SSL			10%
Idle	7%	2%	15%
Other	5%	6%	2%

involved in establishing an SSL (Secure Socket Layer) connection to transmit confidential information between servers. Notice that improved hardware for performing cryptographic operations could have a considerable impact. Idle time, because servers must sometimes wait for one another, is also listed in Table 4.2. Only 2% to 6% of the total execution time is unaccounted. That time is being used for signature verification, message marshaling and un-marshaling, and task management.

To evaluate the effectiveness of the optimizations outlined above for when assumptions N1 and N2 hold, Figure 4.1 compares performance with and without the optimizations. The results summarize 100 executions; very small sample standard deviations were observed here. The optimizations can be seen to be effective.

4.2 Internet Deployment

Communication delays in the Internet are higher than in a local area network; the variance of these delays is also higher. To understand the extent, if any, this affects performance, we deployed four COCA servers as follows.

- University of Tromsø, Tromsø, Norway. (300 MHz, Pentium II)
- University of California, San Diego, CA. (266 MHz, Pentium II)
- Cornell University, Ithaca, NY. (550 MHz, Pentium III)

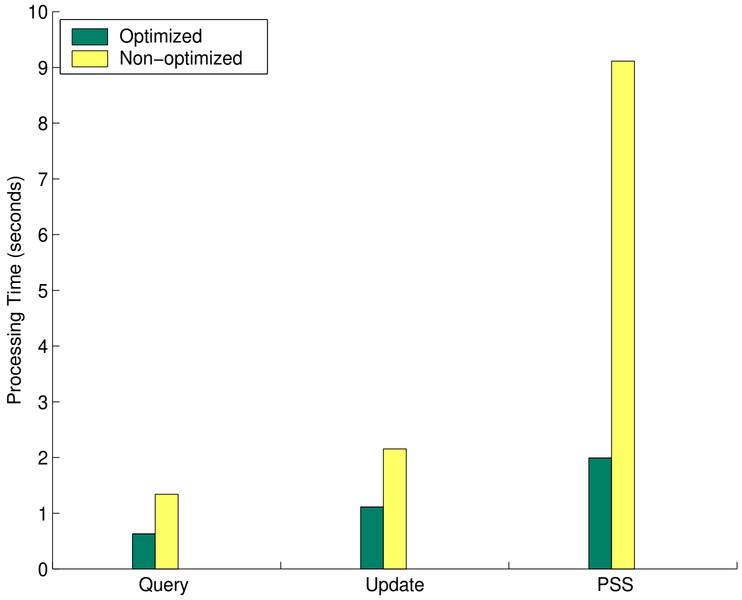

Figure 4.1: Effectiveness of Optimization.

- Dartmouth College, Hanover, NH. (450 MHz, Pentium II)

All ran Linux.[2] Figure 4.2 depicts the average message delivery delay (measured using ping) between these servers. Delivery delays on the Internet vary considerably [65] but the values observed during the experiments we report did not differ significantly from those in Figure 4.2.

Table 4.3 gives measurements for the Cornell host in our 4-site Internet deployment. In comparing Table 4.1 and Table 4.3, we see the impact of the Internet's longer communication delays (which also lead to longer server idle time). The sample standard deviation is also higher for the Internet deployment, due to higher load variations on servers and due to the higher variance of delivery-delays on the Inter-

[2] Beggars can't be choosers. For making measurements, we would have preferred having the same hardware at every site, though we have no reason to believe that our conclusions are affected by the modest differences in processor speeds. For a real COCA deployment, we would recommend having different hardware and different operating systems at each site so that common-mode vulnerabilities are reduced.

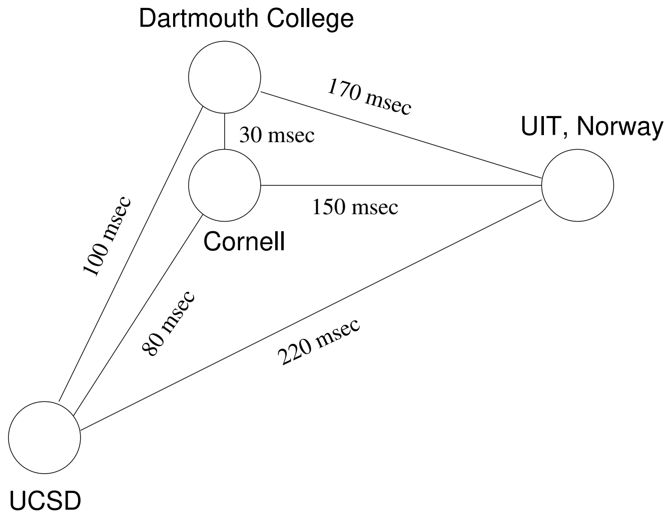

Figure 4.2: Deployment of COCA Servers over the Internet.

Table 4.3: Performance of COCA over the Internet.

COCA Operation	**Mean** (*msec*)	**Std dev.** (*msec*)
Query	2270	340
Update	3710	440
PSS	5200	620

net; all samples are located within 25% of the mean. See Table 4.4 for a breakdown of delays (analogous to Table 4.2) for our Internet deployment of COCA.

A Note on Scalability

The performance results presented above show that processing a client request (Query or Update) is expensive, which in turn would seem to limit the number of clients that COCA can support. While the performance of COCA could be enhanced by adopting faster machines, by using hardware implementation for cryptographic operations, or by having multiple processors on each server to enable parallel computation, we here point out some directions for architectural improvements that could potentially lead to better scalability. More research is needed to explore the feasibility and the impact of these proposals.

Table 4.4: Breakdown of Costs for COCA over the Internet.

	Query	Update	PSS
Partial Signature	8.0%	8.7%	
Message Signing	3.2%	2.5%	2.6%
One-Way Function			7.8%
SSL			1.6%
Idle	88%	87.7%	87.4%
Other	0.8%	1.1%	0.6%

Hierarchy. As proposed in [42], we can construct a hierarchy, where each node in the hierarchy is a CA. Each CA at the lowest level is in charge of a small subset of clients, and each CA is a client of its parent CA which is located at the next higher level. COCA could be used to implement any CA in the hierarchy. To decide whether a specific CA should provide on-line capabilities, we have to assess the frequency of certificate invalidation—if the bindings between the CA's clients and their public keys change often, then the CA could employ COCA to reduce the risks related to invalidated certificates; if certificates are relatively stable, then the CA could stay off-line, thereby eliminating risks related to on-line attacks. Therefore, a sensible configuration could be to implement each CA at the lowest level as an on-line CA using COCA—because the certificates of its clients are likely to be dynamic. But since the service key pair of a lowest-level CA does not change regularly, it might suffice to have any CA at a higher level off-line. Using this scheme, we could achieve better scalability because each lowest-level CA, which is implemented using COCA, is in charge of only a small subset of clients.

Recency requirement. Instead of querying COCA for up-to-date certificates, clients have the freedom to decide, based on the perceived risks and security policies, whether to trust a certificate that has previously been verified. One way is to employ *recency requirements* [103, 95] that specify the maximum time that may elapse before validity of the certificate is rechecked. Most applications do not require that the validity of a certificate be checked before each use. Consequently, these applications will associate a non-zero recency requirement with these uses, thereby reducing the load on COCA and improving the scalability of COCA.

To support recency requirements, we have to strengthen our assumption of the asynchronous system model and assume approximately synchronized clocks. Such strengthening introduces new vulnerabilities—if an attacker can slow client clocks, then the use of a compromised certificate is prolonged without violating any recency requirements. And an attacker that prevents COCA servers from maintaining synchronized clocks also succeeds. Clients have to take these factors into account when specifying their recency requirements. When the risks of these new vulnerabilities are beyond their tolerance, clients could always employ on-demand validation (i.e., setting recency requirement to be zero).

4.3 COCA Performance and Denial of Service Attacks

Any denial of service attack will ultimately involve some combination of compromised clients and/or servers (i) sending new messages, (ii) replaying old messages, and (iii) delaying message delivery or processing. COCA defends against these attack manifestations with a combination of request-processing authorization, resource management, and caching. To evaluate how effective these classical defenses are, we simulated certain attacks. The results of those experiments for our local area network deployment of COCA are discussed in this section.

4.3.1 Message-Creation Defense

New messages sent by servers are not nearly as effective in denial of service attacks against COCA as new messages sent by clients. This is because messages from servers are rejected unless they self-verify. Such messages must contain a correctly signed client request as well as correctly signed messages from all servers involved in previous protocol steps—the collusion and compromise of more than t COCA servers is thus required to get past COCA's request-processing authorization mechanism. Moreover, once any message from a given server is found by a COCA server p to be invalid, subsequent messages from that server will be ignored by p, considerably blunting their effectiveness in a denial of service attack to saturate p.

In contrast, a barrage of requests from compromised clients, if correctly signed, cannot be rejected by COCA's request-processing authorization mechanism (unless the identities of these compromised clients are already known by the receiver). The impact of such a barrage should be mitigated by COCA's resource management mechanism, which ensures that messages from a small set of senders do not monopolize server resources. How effective as a defense this mechanism is depends on the exact configuration of COCA's resource management mechanism: the number of input queues, on which input queues various clients are grouped, and the scheduler used in servicing these input queues.

To measure the effectiveness of COCA's resource management mechanism, it suffices to investigate the simple case of two clients. A *compromised client* sends a barrage of new requests to the service at rates we control;[3] a *correct client* sends

[3]Because the compromised client does not await responses before sending additional requests, these experimental results apply directly to the case where a group of compromised clients all share the same input queue on each server.

a request, awaits a response or a timeout[4]. Of interest is by how much the correct client's requests become delayed due to requests the compromised client sends, since this information can then be used in predicting COCA's behavior when there are more than two queues and clients.

Once a client's request $\mathcal{R}$ is appended to some input queue on a (correct) COCA server, two factors contribute to delay processing $\mathcal{R}$. The first source of delay arises from multiplexing the server as it processes a number of requests. This number of requests is referred to as the *level of concurrency.* Assuming a modest load from correct clients, the delay due to sharing the processor with other, concurrent requests is not affected by actions an attacker might take and thus is not of interest here; our experiments therefore assume servers process requests to completion serially (*viz.* the level of concurrency is 1). The second source of delay is affected by the compromised client's barrage of new messages—requests in input queues whose processing will precede $\mathcal{R}$. A mechanism to defend against a barrage of client requests must control this source of delay, and it is this delay that we measure.

Our first experiment adjusted the rate of requests from the compromised client while measuring the performance of requests from the correct client. To start, each server was configured to store all client requests on a single input queue. The capacity of this queue was 10 requests. We found that the correct client would get no service whenever the compromised client sent requests at a rate in excess of 10 requests per second. At 10 requests per second, requests from the compromised client fill the (fixed capacity) input queue virtually all the time—a Query request from the correct client has a 9 in 10 chance of being discarded because it arrives when there is no room in the input queue, and an Update request has half that (due

[4]The timeout is 1 second for Query and 2 seconds for Update.

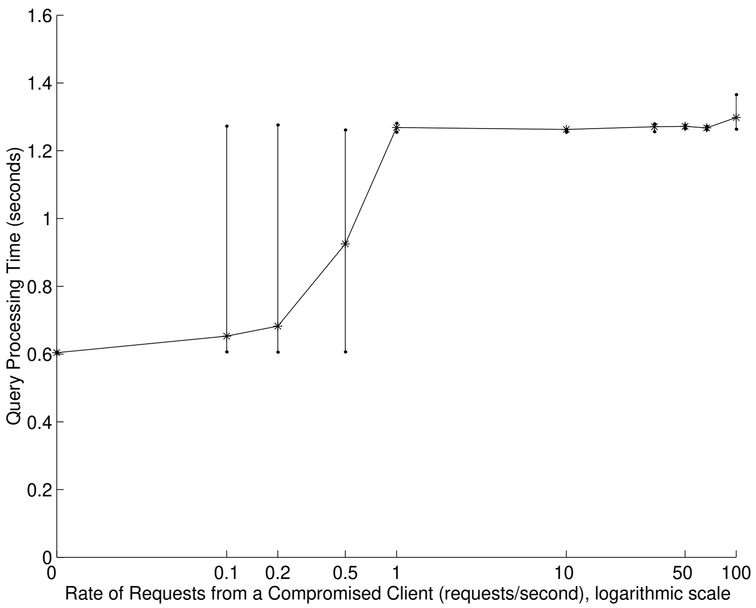

Figure 4.3: Performance of Query under Message-Creation Attacks.

to the 1 and 2 seconds timeout respectively). Needless to say, the denial of service attack is a success.

For the next experiment, each server was configured to have separate queues for the correct client and the compromised client. A round-robin scheduler serviced the two queues. Figures 4.3 and 4.4 show performance of Query and Update requests from the correct client for various rates of requests from the compromised client. Every reported data point is the average processing time over 100 experiments; the error bars depict the range for 95% of the samples.

The curves for Query and Update in Figures 4.3 and 4.4 comprise two segments. In the first segment, an increase in the rate of requests that the compromised client sends causes an increased delay for requests from the correct client. As the rate of requests from the compromised client increases, so does the probability that COCA—with its round-robin servicing of input queues—will have to process one of those requests $\mathcal{R}$ before processing a request from the correct client. The processing

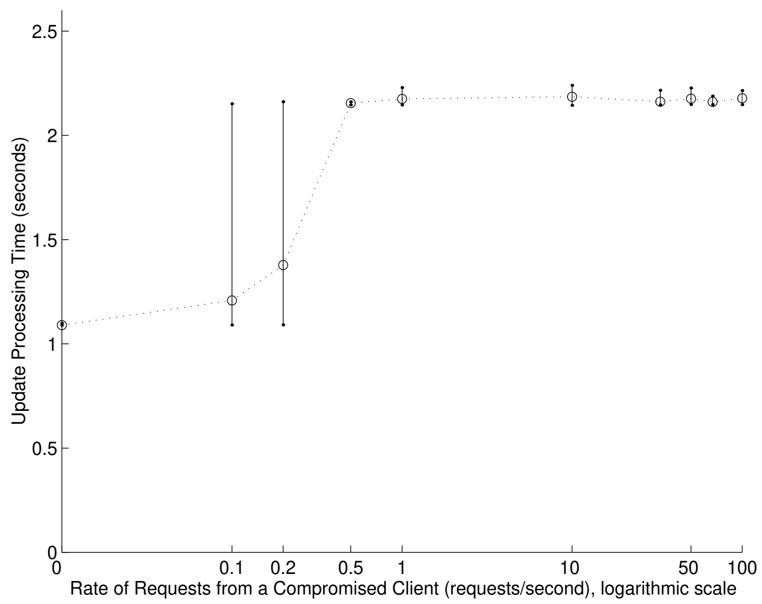

Figure 4.4: Performance of Update under Message-Creation Attacks.

of $\mathcal{R}$ thus increases the processing time for a request from the correct client. We see in this segment almost identical wide ranges of samples for each rate measured. The worst case occurs when the request from the correct server arrives just after a request from the compromised client starts to get processed, while the best case occurs when the request from the correct server arrives when no request from the compromised client is being processed. Even though we see the same worst and best case, the mean of samples increases as the rate of requests from the compromised client increases, reflecting an increasing probability that the request from the correct client has to wait for the processing of a request from the compromised client.

Once the compromised client is sending requests at approximately the same rate as the normal client (i.e., approximately 1 request per second for Query and 0.5 requests per second for Update), the second segment of the curve begins. Throughout this segment, further increases in the request rate from the compromised client do not further degrade the processing of requests from the correct client. This is because requests from the two clients are being processed in alternation, and the delay for requests from the correct client remain at about double what is measured when there is no compromised client. Note that, as the rate of requests from the compromised client increases, more and more of those requests are discarded by servers—the fixed-capacity input queue for the compromised client is full when those requests arrive.

COCA's request-processing authorization mechanism starts saturating at 100 requests per second and thereafter the server would have diminished processing capacity to execute protocols for Query and Update.

In an actual deployment, clients will be partitioned over a set of input queues. But the worst-case performance for this case is easy to bound in light of the above experiments for two clients. Suppose b queues are serving only compromised clients, c queues are serving only correct clients, and d queues are serving both kinds of clients. Requests from compromised clients will starve requests from correct clients that share the same input queue, because the first experiment above established that if the rate of requests to a single input queue from compromised clients exceeds 10 requests per second then requests from correct clients to that input queue are unlikely to succeed. And the second experiment established that COCA's resource-management mechanisms will guarantee that $c/(b+c+d)$ of each server's processing time and other resources are devoted to processing requests on the queues that serve only correct clients.

4.3.2 Message-Replay Defense

COCA employs caching to defend against denial of service attacks involving message replays. We do not consider replays of client requests in our experiments, because their impact on COCA is considerably smaller than the impact of processing new requests from a compromised client. Specifically, for new requests, COCA must expend resources in executing the protocol for the operation being requested, but for replays of client requests, processing (by design) involves considerably fewer resources—the request is one that can be rejected because its sequence number is too small, one that can be satisfied from the server's cache, or one that can be ignored because it is already being processed. The curves of Figures 4.3 and 4.4 thus give the bounds we seek on the worst-case performance of COCA when client-request replays form the basis for a denial of service attack.

Replays of messages from servers in COCA are not immobilizing, because relatively expensive cryptographic computations are cached. To validate this, we simulated an attacker replaying server messages at varying rates to all other COCA servers. The message being replayed was designed to cause a defenses-disabled COCA server to compute partial signatures, which takes approximately 200 milliseconds on a 450 MHz Sun E420 Sparc server—a relatively expensive operation and thus particularly effective in a denial of service attack.

We measured the average delay for Query, Update, and proactive secret sharing as a function of the rate of message replays sent by the compromised server. We compared the performance in the case where caching is enabled to that in the case where caching is disabled. This information appears in Figures 4.5 through 4.7.

For the case where caching is enabled, the average delay for each operation is largely unaffected as the rate of message replay increases, because caches satisfy most of the computational needs in handling those messages. We witnessed a slight increase in the average delay when the rate of message replay reaches 100 messages per second. This is the point where the request-processing authorization mechanism

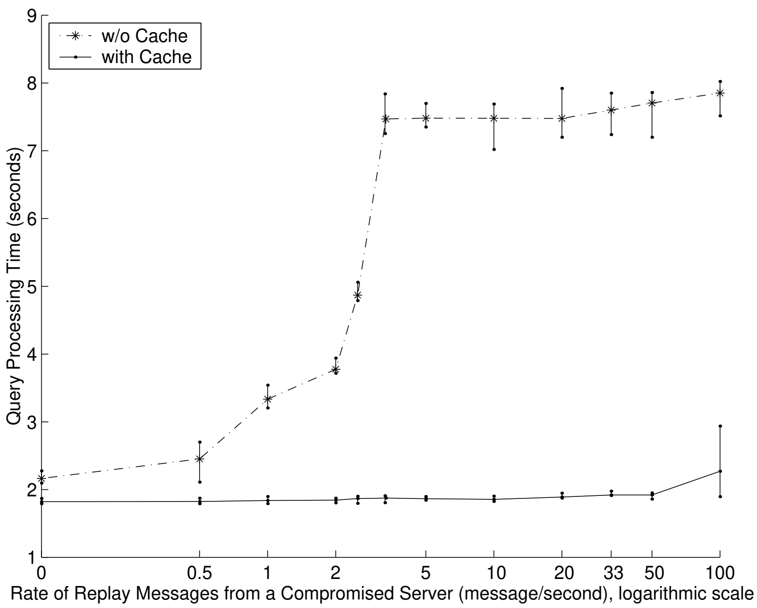

Figure 4.5: Performance of Query under Message-Replay Attacks.

becomes saturated by incoming messages.

For the case where caching is disabled, each curve consists of two segments. The first segment (which ends at approximately 3 replays per second for Query and Update, and 10 replays per second for PSS) resembles the first segment in the curves of Figures 4.3 and 4.4, and it reflects the increased use of processing resources by replays to recompute values that were not cached as the replay rate increases. The second segment only gradually increases. Over this range, additional computation is not required (so additional delay is not incurred) since the resource management mechanism bounds the number of attacker messages that are processed.

Even without the compromised server launching the attack (i.e., when the rate of replay messages is 0), the average delay for each operation in the case where caching is enabled is lower than that in the case where caching is disabled. This is because, with one fewer server participating, repeated executions of certain expensive operations is necessary since normal circumstances assumption N1 no longer

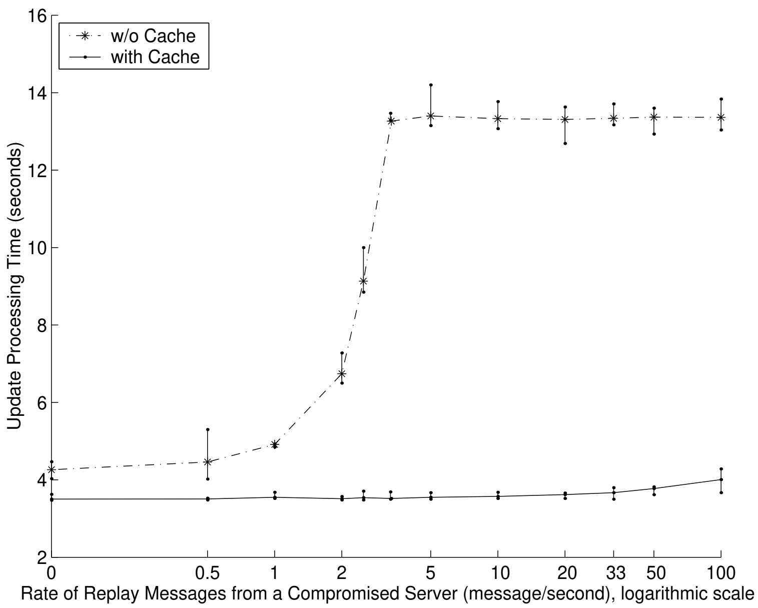

Figure 4.6: Performance of Update under Message-Replay Attacks.

holds, so correct servers are unable to finish processing in an optimized execution. The switch back to the fault-tolerant version causes repeated executions of certain expensive cryptographic operations, which can be avoided when caching is enabled.

4.3.3 Delivery-Delay Defense

To measure the impact of message transmission and processing delays on the performance of COCA, we added code to the implementation so that messages delivered to a client or server could be delayed a specified amount before becoming available for receipt. We investigated both the case where messages sent to one specific server are delayed and the case where messages sent to all servers and clients are delayed.

Figure 4.8 gives the average time and the interval containing 95% of the samples for COCA to process three operations of interest—Query, Update, and a round of proactive secret sharing—when messages from a single server are delayed. The case where this server is unavailable is also noted as inf on the abscissa.

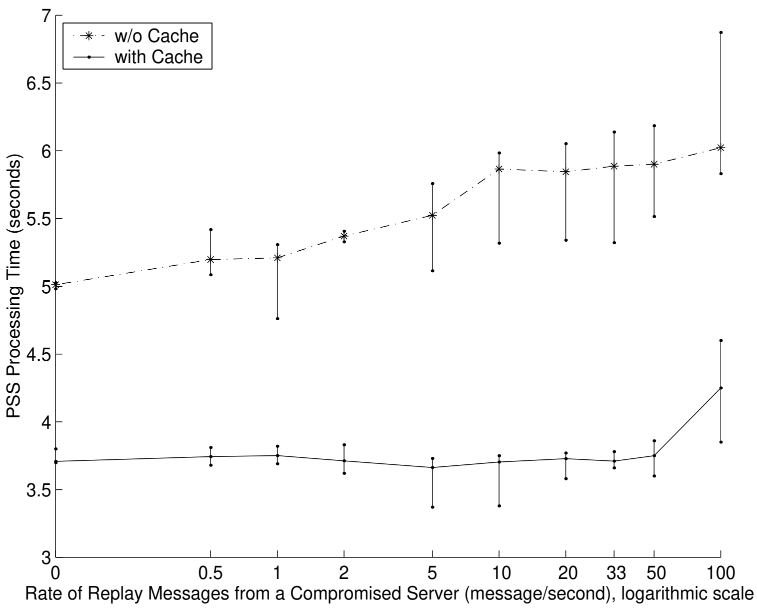

Figure 4.7: Performance of PSS under Message-Replay Attacks.

As delay increases, the processing time is seen to move through three phases. During the first phase, as server p (say) increases its delay in processing messages, so does the delay for the operation of interest. This occurs because COCA protocols initially assume normal circumstances assumptions N1 and N2 hold, and the optimized protocols require participation by p. A delay in messages from p thus delays the protocols.

The second phase is entered after the delay for p causes servers to suspect that normal circumstances assumptions N1 and N2 do not hold. These servers initiate redundant processing, creating additional delegates for in-process operations, for example. Participation by p is no longer required for the operation to terminate; increasing the delay at p does not delay completion of the operation. But p will continue to send messages requiring servers to compute replies. The time that servers devote to generating these replies decreases as the delay for p increases, simply because p sends fewer such messages when the delay is greater. Servers thus

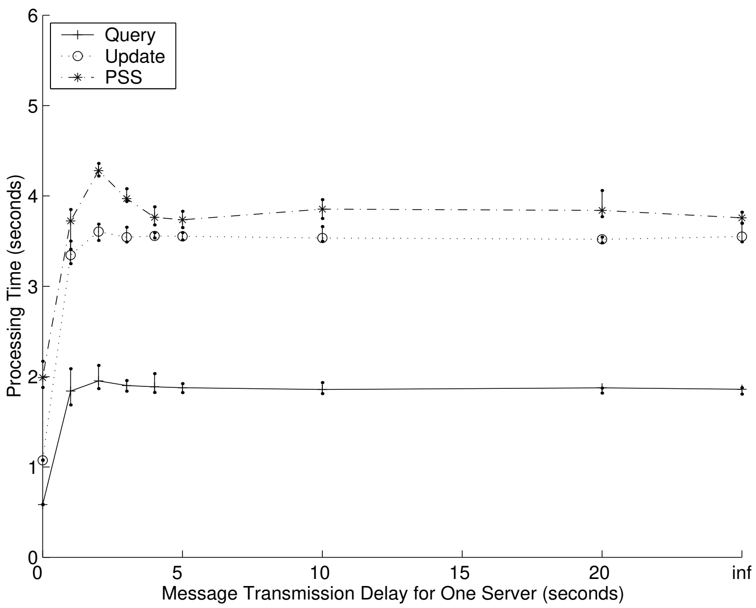

Figure 4.8: Performance of COCA vs. Message Delay for One Server.

have more cycles to devote to generating replies for servers other than p; these are the replies needed in order for the protocols to terminate. So, the increasing delay for p frees server resources to speed the termination of the protocol, and average processing time decreases in this second phase.[5]

The third phase—a plateau in response time—is reached when the delay for p is sufficiently high so that it imposes little load on other servers.

Figure 4.9 gives average measured delay and the interval containing 95% of the samples when message delay increases at all servers and clients. Observe that the execution time increases linearly with the increase of message delay. The curves are consistent with how the protocols operate: processing a Query involves 6 message

[5]We see that the decrease in processing time is more significant in the case of proactive secret sharing than in the cases of Query and Update. This is because, in the case of proactive secret sharing, processing messages from server p involves some new (therefore not cached) expensive cryptographic operations, while, in other two cases, expensive cryptographic operations can be avoided due to caching.

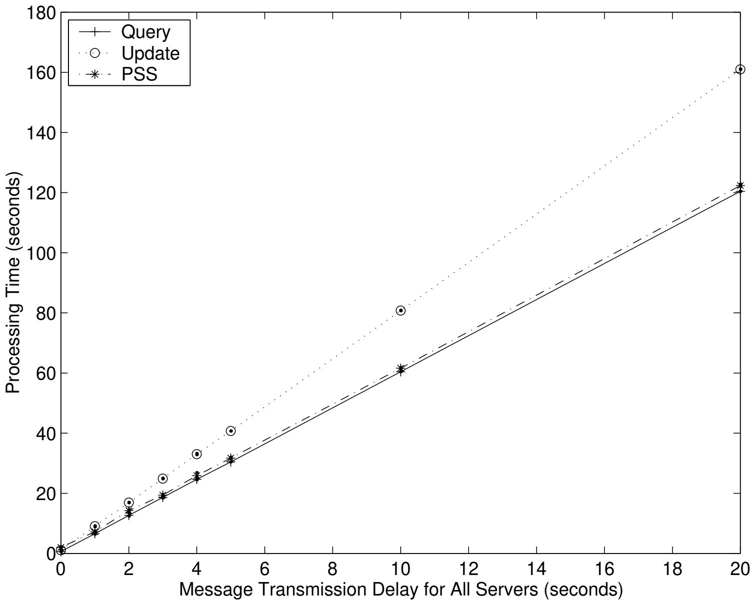

Figure 4.9: Performance of COCA vs. Message Delay for All Servers.

delays, processing an Update involves 8 message delays, and a round of proactive secret sharing involves 6 message delays.

Chapter 5

Conclusion

This dissertation not only describes the design and implementation of a specific on-line certification authority (COCA), but also offers a general framework for building trustworthy on-line services that demand both fault tolerance and security. The framework comprises the following components:

Replication. Employ replication to achieve fault tolerance. Create sufficient diversity among servers that implement the service, so that server compromise can be regarded as independent. Design protocols for coordinating servers to implement the intended semantics of the service under the given system model. Different approaches (e.g., replicated state machine approach or Byzantine quorum systems) can be adopted for implementing different services under different system models.

Secret sharing and secure multi-party computation. For any secret maintained by the service, instead of storing that secret on every server, use a secret sharing scheme and only distribute shares of that secret to servers. Employ secure multi-party computation schemes (e.g., threshold cryptography schemes) so that servers can compute functions (e.g., generating a digital signature) using these shares without the need to reconstruct the secret.

Proactive recovery. Employ proactive recovery to reduce the window of vulnerability of the service. Each execution of the proactive recovery protocol reconstitutes local states of servers, refreshes shares—stored on servers—of any secret maintained by the service, and refreshes any secret that is local to a server (e.g., the private key of a server).

Defense against denial of service attacks. Identify vulnerabilities to denial of service attacks by pinpointing imbalances between the cost for an adversary to submit a request and the cost for a server (or the service) to process that request. Deploy defense mechanisms (e.g., request-processing authorization, resource management, and caching) against denial of service attacks accordingly.

To be fault-tolerant and secure, an on-line service should assume a weak system model. The weaker the system model, the harder it is for failures and adversaries to compromise the service by invalidating assumptions in the system model. However, a service that works in a weaker system model is usually harder to design, because of the wide variety of adversary attacks such a model admits; all these attacks have to be taken into account. Furthermore, a service for a weak system model could exhibit poor performance, due to expensive mechanisms required to tolerate failures and attacks. Good performance is challenging to achieve but necessary for a service to be practical. Our experience with COCA demonstrates approaches for addressing these challenges, as well as revealing future research directions related to these approaches. The remainder of this chapter elaborates.

5.1 Successive Model Weakening

In the design of the APSS protocol of Chapter 2 and the COCA protocols of Chapter 3, we employed a single technique for controlling complexity of protocol design for a weak system model—evolve the protocols by considering a series of more and more malicious adversaries: A first protocol is designed for a strong system model characterizing a relatively benign environment. That protocol is then repeatedly revised to work in a (weaker) system model that admits more malicious adversaries, until a protocol for the intended system model is obtained. The complexity of protocol design is thus managed through a form of divide-and-conquer—at each step, a designer focuses on a small number of additional attacks now permitted in the newly weakened system model.

This idea of deriving a protocol is not new. For example, it has been proposed in [9, 26, 102, 81, 3, 4] as a way to obtain fault-tolerant protocols for different failure models. Prior work proposes mechanical transformations between protocols for different system models. Although theoretically interesting, the mechanical transformations too often yielded unnecessarily complex and inefficient protocols. This is because such transformations, due to their generality, are unable to take advantage of semantics in doing optimization.

The design of COCA extensively employs successive model weakening. The following identifies two examples where this technique was applied.

(1) In Chapter 2, a first APSS protocol was constructed for the strong system model in which Passive Server-Adversaries with Crash Failures, Passive Link-Adversaries, and Correct Coordinator were assumed. This first protocol was then revised to work in a weaker system model that did not assume Correct Coordinator holds. Next, the assumption of Passive Link-Adversaries was re-

laxed to that of Active Link-Adversaries, thereby allowing attacks that could insert, delete, modify, reorder, and replay messages in transit on links. Finally, the assumption of Passive Server-Adversaries with Crash Failures was weakened to that of Active Server-Adversaries, thereby admitting Byzantine failures of servers.

(2) In Chapter 3, Update and Query protocols were first proposed for a system model assuming a correct delegate and reliable links. Then, these protocols were improved to tolerate crash failures of delegates. Next, new protocols were derived to work in a system model that admits also Byzantine failures of delegates. Finally, the protocols were revised to work with fair links.

Different sequences of system models were used to guide the derivation of the APSS protocol and the Update and Query protocols, respectively. Deciding the sequence of gradually weakening system models is a crucial and often the most challenging step of this design process. It remains more an art than a science.

Certain weakening of a system model showed up repeatedly; for example, from Passive Link-Adversaries (which assumes reliable links) to Active Link-Adversaries (which assumes fair links), and from crash failures to Byzantine failures. And, similar mechanisms are used for transforming protocols with respect to the same weakening of a system model; for example, retransmissions (e.g., in the form of group_send) are used for protocols to cope with fair links assumed by Active Link-Adversaries, while self-verifying messages are used for protocols to constrain Byzantine failures. Gaining additional insight into these mechanisms is likely to be a valuable endeavor, because the mechanisms could prove effective for building other trustworthy services—especially when the same system model is assumed for these services.

5.2 Circumventing Agreement

Although a service designed for the asynchronous system model exhibits reduced vulnerability to denial of service attacks, achieving agreement in the asynchronous system model is known to be problematic. The need to achieve agreement seems to arise naturally when designing a replicated service, because (correct) servers, as replicas, are generally assumed to be in consistent states—maintaining such consistency often seems to require agreement among these servers.

COCA is a replicated service but does not require an agreement protocol for coordinating servers. Here is how COCA circumvents the need for an agreement protocol.

(1) For the APSS protocol in Chapter 2, servers are not expected to generate a single new sharing of the service private key from an old sharing in an execution of that protocol—requiring that a single new sharing be generated seems to require an agreement among servers on which subshares to use for generating that single new sharing. Rather, multiple new sharings can be generated, each with a unique label. Having multiple new sharings generated turns out not to be a problem—when constructing a signature for a message, servers use the label to indicate which sharing should be used.

(2) As described in Chapter 3, COCA accepts and processes Query and Update requests; each of COCA's servers stores a copy of every client's certificate. It seems convenient to think of (correct) servers as identical replicas and thus always storing the same set of certificates. So, a natural implementation would be to use the replicated state machine approach, which ensures that (correct) servers agree on the requests to be processed and on the order of the requests

to be processed. The need for an agreement in such an implementation is evident.

Instead, by adopting a Byzantine quorum system, COCA ensures only that (correct) servers in some quorum have processed each request. The need for an agreement protocol is thus averted. COCA also does not guarantee that (correct) servers process requests in the same order. Instead, COCA associates with each certificate a unique serial number that encodes certain causality information and orders certificates for the same *cid* accordingly. By taking advantage of some specific properties of Query and Update (e.g., idempotence), these guarantees, although weaker than what a replicated state machine provides, are sufficient for implementing the intended semantics of COCA.

Pondering on how COCA circumvents agreement leads to some interesting research questions:

- COCA has demonstrated that, in certain cases, a seemingly necessary agreement can be avoided. It would be interesting to identify the general circumstances where agreement protocols can be excised and also to identify exactly what can or cannot be achieved without agreement.

- Our APSS protocol avoids establishing agreement by allowing multiple new sharings to be generated in each execution. It remains an open question whether agreement is necessary to have a single new sharing generated in an APSS protocol.

- Although a Byzantine quorum system can be implemented even in the asynchronous system model with Byzantine failures of servers, a Byzantine quorum system provides a weaker semantics than a replicated state machine. In which cases are Byzantine quorum systems applicable and what are the limitations of Byzantine quorum systems in comparison with the replicated state machine approach?

5.3 Normal-Case Optimization

A fault-tolerant and secure service might well exhibit poor performance, because achieving fault tolerance necessarily involves redundancy and because defending against attacks often requires the use of expensive cryptographic schemes. These mechanisms are only needed for the service to survive in worst-case scenarios. And, in reality, worst-case scenarios are rare. In normal circumstances, when there are no failures or malicious attacks, mechanisms for fault-tolerance and security are unnecessary.

Ideally, a service would incur the cost of fault tolerance and security mechanisms only when they are needed (e.g., when components are failing or under attack). Such ideal behavior could be approximated as follows.

Consider a service implemented by a set of protocols that work in some intended, weak, system model. These protocols will be referred to as *heavyweight protocols*, because they require expensive mechanisms to defend against failures and attacks admitted by the weak system model.

For each heavyweight protocol, deploy a corresponding *lightweight protocol* that accomplishes the same task but only in a strong system model expected to hold under the normal circumstances. Each lightweight protocol, when run in a strong system model, will exhibit better performance than the corresponding heavyweight protocol. The lightweight protocol, however, might fail to do its job when assumptions of the strong system model are violated.

Each lightweight protocol is assumed to come with mechanisms to detect violations to the strong system model. The detection mechanisms must work in the weak system model, especially when the strong system model no longer holds, because that is the case where violations need to be detected.

The service runs the lightweight protocols first. If and when a violation to the strong system models needed by these lightweight protocols is detected, the heavyweight protocols are activated. Assuming the strong system model is rarely violated, the service would be running the lightweight protocols most of the time. Therefore, the service would exhibit performance symptomatic of the lightweight protocols, without sacrificing defense against failures and attacks—the detection mechanisms guarantee the heavyweight protocols would be invoked when failures and attacks do occur.

This optimization is not without costs, though. When the strong system model is violated, the performance of the service could be worse than that of executing only the heavyweight protocols, because of the overhead of executing the lightweight protocols and the overhead of the detection mechanisms. However, the performance improvements achieved by the lightweight protocols in cases where the strong system model does hold should outweigh this cost.

Here are two examples of the lightweight/heavyweight optimization from this dissertation.

Optimization of the threshold signature protocol. In the original heavyweight threshold signature protocol in Chapter 3, the same partial signature could be generated by multiple different servers using the same share. (Recall that, for a combinatorial secret sharing, the same share of the underlying standard secret sharing might be distributed to share sets of different servers, as shown in Figure 2.2.)

Such redundancy is necessary for fault tolerance.

For a corresponding lightweight threshold signature protocol, the strong system model assumes the following:

- For every server contacted by the (correct) initiator of the protocol, the initiator receives partial signatures from that server within a certain period of time (based on the local clock of the initiator).

- Partial signatures the initiator receives from servers are correct.

Given this strong system model, for a lightweight threshold signature protocol, the initiator requests that, for every share of the selected sharing, only a single partial signature be generated and submitted. (This avoids redundant computation of partial signatures.) And, the lightweight threshold signature protocol detects violations to the strong system model using the following mechanisms.

- The initiator starts a timer[1] for every message it sends. If requested partial signatures fail to arrive within a specified bound, then the timeout of the timer indicates that the strong system model does not hold.

- After receiving all needed partial signatures, the initiator combines these partial signatures to obtain the signature for the message being signed. The initiator verifies the correctness of the generated signature. If bogus partial signatures have been received and used, then the verification fails and a violation to the strong model is detected.

[1] Here, we are assuming that the timer on a correct server cannot be subverted and that the timer always advances at a reasonable rate, even in the weak system model. Similar assumptions are implicit for COCA's heavyweight protocols (e.g., COCA's servers are assumed to start proactive recovery periodically based on their local clocks). Therefore, the assumption about timers at correct servers is not new.

Whenever the initiator detects a violation to the strong system model, it starts the heavyweight protocol and requests additional partial signatures to be generated and submitted.

Optimization of APSS protocol. The heavyweight APSS protocol in Chapter 2 invokes multiple threads, causes multiple new subsharings to be generated from the same old share by different servers, and has multiple new sharings generated by different threads. Such redundancy is needed for fault tolerance.

For a corresponding lightweight APSS protocol, we stipulate a strong system model that assumes the following:

- A designated coordinator p carries out its thread to have a new sharing established.
- Every correct server q (say) receives from coordinator p a finished message signifying the termination of p's thread. The message arrives within a certain period of time from when q started execution of the APSS protocol, based on q's local clock.
- When requested, every server will correctly generate subshares from the selected shares and propagate these subshares.
- For any message that the coordinator sends, the coordinator receives the response from the intended recipient of that message within a certain period of time.

The lightweight APSS protocol can then execute as follows. Coordinator p, in its thread (the only thread for the lightweight APSS protocol), requests that, for every share of a selected sharing, a single subsharing is generated from that share by a server. Subshares of these subsharings are then used to construct a new sharing. This results in a single new sharing being established when execution of the lightweight APSS protocol terminates. Thus, the lightweight APSS protocol eliminates the unnecessary cost of generating other subshares and new sharings.

The lightweight protocol is equipped with some detection mechanisms.

- Both the coordinator and the servers use timers, as in the threshold signature protocol shown earlier, to detect messages that fail to arrive within a specified period of time.

- As with the original heavyweight APSS protocol, coordinator p sends self-verifying finished messages, so that any bogus notification from p will be detected.

  Similarly, each server is required to use the self-verifying contribute messages as notifications that a subsharing has been generated and propagated correctly, so that a bogus notification from that server will be detected.

  Note that, even in the weak system model, an adversary cannot forge a bogus self-verifying message without being detected.

- Verifiable secret sharing is used with the subsharings and the sharings to enable detection of bogus subshares and shares that are generated and propagated.

  Note that, since verifiable secret sharing is used in the heavyweight protocols, employing it as a detection mechanism for the lightweight protocols does not involve strengthening the system model.

In the combined heavyweight/lightweight protocol, if the coordinator detects the strong system model was violated, then it requests participation of more servers in its thread and requests that more subsharings be generated and propagated, if necessary. And, if a server detects violation to the strong system model, then the server starts functioning as a new coordinator and starts its own thread.

Issues in Designing Optimization

Although conceptually simple, lightweight/heavyweight optimizations are hard to discover. Picking an appropriate strong system model for a lightweight protocol is often challenging. A promising approach is to start by identifying possible detection mechanisms. This is relatively easy, because many detection mechanisms are standard. Examples for COCA include timeouts for detecting any violation to timing assumptions, digital signatures for detecting bogus partial signatures, self-verifying messages for detecting bogus messages, and verifiable secret sharing for detecting bogus shares and subshares. Often, these mechanisms (e.g., self-verifying messages and verifiable secret sharing) can be found in the heavyweight protocols.

Each detection mechanism checks violations to a certain assumption. That assumption might then be considered a candidate for the strong system model. Whether or not to include the assumption in the strong system model can depend on other factors, such as whether that assumption holds in normal circumstances and whether making that assumption leads to significant performance gain for a lightweight protocol.

Detection mechanisms in a lightweight protocol need not be accurate. It suffices that those circumstances where the lightweight protocol could fail are detected. But the mechanism can be conservative and raise false alarms when the strong model actually holds. A conservative detection mechanism thus can cause an unnecessary switch to a heavyweight protocol. Such switches do not endanger correct operation of the service, because the heavyweight protocol will work in the strong system model. There is an obvious performance penalty for false alarms, though.

As an example of a conservative detection mechanism, consider a strong system model that assumes that a server eventually receives a certain message. No detection mechanism can accurately detect a violation to this assumption within a finite period time. Instead, a timer can be used, so the switch from lightweight to heavyweight protocol occurs if the server does not receive that message within a certain finite period of time. But a false alarm will be raised if the message is delivered after the timeout. To reduce a performance penalty, choose the timeout period to be sufficiently large, so that false alarms are rare in normal circumstances.

Finally, note that, in the earlier examples of the threshold signature protocol and the APSS protocol, detection mechanisms run on each individual server and make decisions (about violations) based on local information available to that server (e.g., a local timeout, signature verification failure, or detection of an ill-formed self-verifying message). When a violation to the strong system model is detected locally, a server starts the heavyweight protocol (e.g., by sending new messages or by starting a new thread) without coordinating with other servers. Having both violation detection and protocol switching performed locally avoids coordination among servers. Such coordination is often complicated and expensive, because it necessarily involves distributed protocols. But one could imagine having a system-wide mode shift occur in response to detecting a violation to the strong system model.

5.4 Final Remarks

This dissertation represents our first step in investigating how to marry fault tolerance and security when building a trustworthy on-line service. Our experience with COCA has enabled us to identify interesting issues surrounding the problem, to propose a general framework that addresses these issues, and to demonstrate the feasibility of the framework. We believe that extending our experience with COCA to other fault-tolerant and secure services, with richer semantics, is likely to foster new insights.

Appendix A

Complete COCA Protocols

This appendix presents the details of the COCA protocols, including the APSS protocol, the client protocol, the threshold signature protocol, the Query processing protocol, and the Update processing protocol. The following notational conventions are used throughout the appendix:

- p, q, r: servers
- c: COCA client
- $\langle m \rangle_k$: message m signed by COCA using its service private key k
- $\langle m \rangle_p$: message m signed by a server p using p's private key
- $\langle m \rangle_c$: message m signed by a client c using c's private key
- $PS(m, [\overline{s}^{:\Lambda}]_i)$: a partial signature for a message m generated using share $[\overline{s}^{:\Lambda}]_i$
- $[p \longrightarrow q :\ m]$: message m is sent from server p to server q
- $[\forall q.\ p \longrightarrow q :\ m_q]$: message m_q is sent from server p to server q for every server q

- $\mathcal{E}(m)$: m is encrypted in a way that only the intended recipient can decrypt m.

Each message includes a type identifier to indicate the purpose of the message. These type identifiers are presented in the sans serif font. Note that the mechanisms and optimizations described in Section 3.3, Chapter 4, and Section 5.3 are not included in this presentation.

A.1 APSS Protocol

This section describes the APSS protocol. Only the protocol for one thread with p being the coordinator is given; protocols for other threads are identical. Certain operations are to be performed whenever a server receives a message from other servers. These operations are factored out and presented as *general actions*. To clarify the protocol, we also present two sub-protocols (*subshare propagation protocol* and *subshare recovery protocol*) separately.

General actions:

G-1. For each message a server (or coordinator) q receives, q first checks whether the message is a valid finished message (the format of a finished message is shown in step 3.3 of the main protocol) showing that a sharing of version number v' has been established, with v' higher than cvn_p, the current version number of q. If so, q stores this finished message (and removes any other finished messages that are stored) and updates cvn_p to v'. If q is a coordinator, then q terminates its thread and forwards this finished messages to all servers. If q is a server, then q deletes the old shares and subshares.

G-2. For each message a server q receives from a server (or coordinator) r, if the message is not a finished message, then, let v' be the version number of the run for which the message is sent (such information is included in each message), server q checks whether $v' < cvn_q$ holds. If so, then q sends its stored finished message to r (M' is given in step 3.3 of the main protocol and p could be any server).

$$[q \longrightarrow r : \langle \mathsf{finished}, v, p, M' \rangle_p]$$

Subshare propagation protocol:

To propagate shares of a subsharing $\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}$, where $\Lambda = (v-1) \circ \overline{vc}.\overline{s}^{:\Lambda}$ and $\lambda = \Lambda \circ i \circ \overline{vc}.\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}$, server q invokes the following protocol:

P-1. Server q employs $\mathsf{group\_send}(q, \mathsf{establish}, \mathsf{established}, 2t+1)$, where

$$\mathsf{establish}_{q \to r} = \langle \mathsf{establish}, v, \lambda, q, r, \mathcal{E}(S_r^{\lambda}) \rangle_q$$

is the message q repeatedly sends to each server r, and established messages, as shown in step P-2.3, serve as acknowledgments to the establish messages.

P-2. Each server r, upon receiving an establish message from q in a format shown in step P-1, performs the following steps:

P-2.1. Check whether it has received a conflicting establish message from q (that is, one that propagates a different subsharing generated from the same share). If so, alert the administrator with the conflicting messages as evidence and ignore the message.

P-2.2. Check whether the following conditions hold:[1]

[1]Note that the validity checks for the sharing and the subsharing are contained in the label λ sent in the established message.

- $\mathsf{vcConstr}(\overline{vc}.\overline{s}^{:\Lambda}) = y$, where y is the validity check for secret s,
- $\mathsf{vcConstr}(\overline{vc}.\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}) = vc.[\overline{s}^{:\Lambda}]_i$, and
- $\mathsf{oneWay}([\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}]_j) = vc.[\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}]_j$ for every $j \in I_r^{\lambda}$.

If not, alert the administrator with the signed $\mathsf{establish}$ message as evidence and ignore the message.

P-2.3. Store the subshares, together with the validity check for the subsharing, and send the following $\mathsf{established}$ message to q.

$$[r \longrightarrow q : \langle \mathsf{established}, v, \lambda, r, q \rangle_r]$$

P-3. Upon termination of the $\mathsf{group\_send}$ in step P-1, server q obtains $2t + 1$ $\mathsf{established}$ messages, which serve as the endorsements to a $\mathsf{compute}$ message in step 2.3 of the main protocol.

Subshare recovery protocol:

To recover subshares of a subsharing $\overline{[\overline{s}^{:\Lambda}]_j}^{:\lambda}$, where $\Lambda = (v - 1) \circ \overline{vc}.\overline{s}^{:\Lambda}$ and $\lambda = \Lambda \circ i \circ \overline{vc}.\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda}$, server r invokes the following protocol:

R-1. Server r employs $\mathsf{group\_send}(r, \mathsf{recover}, \mathsf{recovered}, t + 1)$, where

$$\mathsf{recover}_{r \rightarrow q} = \langle \mathsf{recover}, v, \lambda, r, q \rangle_r$$

is the message r repeatedly sends to each server q, and $\mathsf{recovered}$ messages, as shown in step R-2, serve as acknowledgments to the $\mathsf{recover}$ messages.

R-2. Each server q, upon receiving a $\mathsf{recover}$ message from r for subshares of $\overline{[\overline{s}^{:\Lambda}]_j}^{:\lambda}$, checks whether it holds the requested subshares. If so, q sends r a $\mathsf{recovered}$ message.

$$[q \longrightarrow r : \langle \mathsf{recovered}, v, \lambda, q, \mathcal{E}(S_q^{\lambda} \cap S_r^{\lambda}), r \rangle_q]$$

R-3. Upon termination of the $\mathsf{group\_send}$ in step R-1, server r obtains all subshares in S_r^{λ}. (Again, the validity of these subshares are checked using validity checks enclosed in the label.)

Main Protocol:

1. Coordinator p selects an established sharing labeled Λ, where $\Lambda = (v-1) \circ \overline{vc}.\overline{s}^{:\Lambda}$, and employs $\mathsf{group\_send}(p, \mathsf{init}, \mathsf{contribute}, t+1)$, where

$$\mathsf{init}_{p \to q} = \langle \mathsf{init}, v, \Lambda \rangle_p$$

is the message p repeatedly sends to every server q, and servers send back $\mathsf{contribute}$ messages (in step 2.3) as acknowledgments to the init messages.

2. Subshare generation and propagation.

Each server q, upon receiving from coordinator p an init message choosing a sharing labeled Λ, performs the following steps.

2.0. Check whether it has received a conflicting init message from p that selects another sharing of version $v-1$. If so, alert the administrator with the two conflicting messages as evidence and ignore this message.

2.1. Generate $\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda_i} := \mathsf{split}([\overline{s}^{:\Lambda}]_i)$ from each old share $[\overline{s}^{:\Lambda}]_i$ that q holds, if no subsharing has been generated from $[\overline{s}^{:\Lambda}]_i$. Also, generate the validity check $\overline{vc}.\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda_i}$ using oneWay:

$$\forall j : (1 \leq j \leq l).\ vc.[\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda_i}]_j := \mathsf{oneWay}([\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda_i}]_j).$$

Here, λ_i is $\Lambda \circ i \circ \overline{vc}.\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda_i}$.

2.2. Initiate the subshare propagation protocol to propagate each subsharing $\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda_i}$ that q has generated but has not propagated to other servers.

2.3. Notify coordinator p, in a $\mathsf{contribute}$ message, that all the subsharings in $\{\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda_i} \mid i \in I_q\}$ have been established. For each such subsharing, attach the label of that subsharing, together with the aggregation of the $2t+1$ $\mathsf{established}$ messages obtained in the subshare propagation protocol as the endorsements, to the $\mathsf{contribute}$ message. We use M_q to denote the set consisting of every such aggregation for every subsharing q has generated.

$$[q \longrightarrow p : \langle \mathsf{contribute}, v, p, q, M_q \rangle_q]$$

3. Share generation.

3.1. Coordinator p chooses a candidate set of subsharings $\{\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda_i} \mid 1 \leq i \leq l\}$ from subsharings whose labels are enclosed in the $\mathsf{contribute}$ messages that p has received. p then employs $\mathsf{group\_send}(p, \mathsf{compute}, \mathsf{computed}, 2t+1)$, where

$$\mathsf{compute}_{p \to q} = \langle \mathsf{compute}, p, \lambda_1, \lambda_2, \ldots, \lambda_l \rangle_p$$

is the message p repeatedly sends to every server q, and servers send $\mathsf{computed}$ messages in step 3.2.3 as acknowledgments to the $\mathsf{compute}$ messages.

3.2. Each server r, upon receiving from p a $\mathsf{compute}$ message in the form shown in step 3.1, performs the following steps.

3.2.0. Check whether it has received a conflicting $\mathsf{compute}$ message from p. If so, alert the administrator with the two conflicting messages as evidence and ignore this message.

3.2.1. Check whether it has received all subshares in

$$\bigcup_{1\le i\le l}(S_r^{\lambda_i}) = \{[\overline{[\overline{s}^{:\Lambda}]_i}^{:\lambda_i}]_j \mid (1\le i\le l)\wedge(j\in I_r)\}$$

If not, initiate the subshare recovery protocol to recover the missing subshares.

3.2.2. Compute new shares

$$[\overline{s}^{:\Lambda'}]_j := \mathsf{reconstruct}([\overline{[\overline{s}^{:\Lambda}]_1}^{:\lambda_1}]_j,\ldots,[\overline{[\overline{s}^{:\Lambda}]_l}^{:\lambda_l}]_j),$$

for each $j\in I_r$.

Compute the corresponding validity check for the new sharing

$$vc.[\overline{s}^{:\Lambda'}]_j := \mathsf{vcConstr}(vc.[\overline{[\overline{s}^{:\Lambda}]_1}^{:\lambda_1}]_j,\ldots,vc.[\overline{[\overline{s}^{:\Lambda}]_l}^{:\lambda_l}]_j)$$

for each $1\le j\le l$.

Here, Λ' is $(v, vc.[\overline{s}^{:\Lambda'}]_j)$.

Let y be the validity check of the secret the service maintains, check whether the following conditions hold:

- $\mathsf{vcConstr}(\overline{vc}.\overline{s}^{:\Lambda'}) = y$
- $\mathsf{oneWay}([\overline{s}^{:\Lambda'}]_j) = vc.[\overline{s}^{:\Lambda'}]_j$ for each $j\in I_r$

If not, alert the administrator with the compute message as evidence and ignore the message.

3.2.3. Send a computed message notifying p that r holds all shares in its share set of sharing Λ'.

$$[r \longrightarrow p : \langle\mathsf{computed}, \Lambda', r\rangle_r]$$

3.3. Upon termination of the group_send in step 3.1, coordinator p stores the $2t+1$ computed messages and constructs the following self-verifying finished message containing a set M' of the $2t+1$ computed messages for new sharing Λ' as endorsement.

$$\langle \mathsf{finished}, v, p, M' \rangle_p$$

Coordinator p executes step G-1 as if it receives this finished message.

A.2 COCA Client Protocol

Every client request has the form:

$$\langle type, c, seq, parm, cred \rangle_c,$$

where *type* indicates the type of the request, c is the client issuing the request, *seq* is the sequence number for the request, *parm* refers to the parameters related to the request, and *cred* contains the credentials that authorize the request.

Clients use the following protocol to communicate with COCA.

1. To invoke Query for the certificate associated with name *id*, client c composes a request:

   $$\mathcal{R} = \langle \mathsf{query}, c, seq, id, cred \rangle_c$$

   To invoke Update to establish a new binding of *key* with name *id* to replace an existing certificate ζ' for *id*, client c composes a request:

   $$\mathcal{R} = \langle \mathsf{update}, c, seq, \zeta', \langle id, key \rangle, cred \rangle_c$$

2. Client c sends $\mathcal{R}$ to $t+1$ servers. It periodically re-sends $\mathcal{R}$ until it receives a response to its request. For a Query, the response will have the form $\langle \mathcal{R}, \zeta \rangle_k$, where ζ is a certificate for *id*. For an Update, the response will have the form $\langle \mathcal{R}, \mathsf{done} \rangle_k$.

A.3 COCA Threshold Signature Protocol

The following describes a threshold signature protocol $\mathit{threshold\_sign}(m, E)$,[2] where m is the message to be signed and E is the evidence used in self-verifying messages to convince receivers to generate partial signatures for m. As detailed in Appendices A.4 and A.5, different types of evidence are used in the protocols for Query and Update.

1. Server p sends to each server q a sign_request message with message m to be signed and evidence E, as well as the label Λ of an established sharing to be used.

$$[\forall q.\ p \longrightarrow q:\ \langle \mathsf{sign\_request}, p, \Lambda, m, E\rangle_p] \qquad \text{(i)}$$

2. Each server q, upon receiving a sign_request message (i), verifies evidence E with respect to m and ignores the message if E is invalid. If E is valid, then q checks whether the version number of the selected sharing Λ is lower than cvn_q. If so, then q executes step G-2 of the APSS protocol and ignores the message. Otherwise, q generates a partial signature using every share in its share set S_p^Λ and sends the partial signatures back to p.

$$[q \longrightarrow p:\ \langle \mathsf{sign\_response}, q, p, m, E, \{PS(m, [\overline{s}^{:\Lambda}]_i) \mid [\overline{s}^{:\Lambda}]_i \in S_p^\Lambda\}\rangle_q]$$

3. Server p repeats step 1 until it receives $t+1$ sign_response messages or until it receives a finished message containing a version number that is higher than cvn_p.

[2] While this protocol is appropriate for schemes such as threshold RSA, the protocol might not be applicable to other threshold signature schemes, such as those based on discrete logarithms (e.g., [18, 52]). Those schemes may require an agreed-upon random number in generating partial signatures. Our protocol can be modified to work for these schemes, though.

If a finished message is received, then p executes step G-1 of the APSS protocol and restarts the threshold signature protocol using a new sharing.

If $t + 1$ sign_response messages are received, then p uses the partial signatures in these messages to construct signature $\langle m \rangle_k$. If the resulting signature is invalid (which would happen if compromised servers submit erroneous partial signatures), then p repeats step 1 for more sign_response messages and tries partial signatures from another set of $t + 1$ servers.[3] This process continues until the correct signature $\langle m \rangle_k$ is obtained.

A.4 Query Processing Protocol

We describe the Query protocol initiated by a delegate p, although there might be more than one delegate processing the same request. This convention also applies to the Update processing protocol described in Appendix A.5.

For both Query and Update processing protocols, at any point of the protocol execution, if servers start participating in a new execution of proactive recovery, then servers notify the delegate (such notification acts as an acknowledgment for any message from the delegate). Upon receiving such notification, the delegate then propagates any in-progress accepted client requests to ensure that these requests be accepted in the new window of vulnerability. After the execution of proactive recovery is completed, the delegate restarts the protocols to process these in-progress client requests.

1. Upon receiving a request $\mathcal{R} = \langle \mathsf{query}, c, seq, id, cred \rangle_c$ from a client c, server p

[3] In the worst case, p must try $\binom{2t+1}{t+1}$ combinations. The cost is insignificant when t is small. There are robust threshold cryptography schemes [44, 43] that can reduce the cost using error correction codes.

first checks whether $\mathcal{R}$ is valid based on the credentials *cred* provided. If it is valid, then p sends a query_request message to all servers:

$$[\forall q.\ p \longrightarrow q:\ \langle \mathsf{query\_request}, p, \mathcal{R}\rangle_p] \tag{ii}$$

2. Each server q, upon receiving a query_request message (ii), checks the validity of the request. If the request is valid, then q fetches the current signed local certificate associated with name id: $\zeta_q = \langle id, \sigma(\zeta_q), key_q\rangle_k$. Server q then sends back to p the following message:

$$[q \longrightarrow p:\ \langle \mathsf{query\_response}, q, p, \mathcal{R}, \zeta_q\rangle_q]$$

3. Server p repeats step 1 until it receives the query_response messages from a quorum of servers (including p itself). p verifies that the certificates in these messages are correctly signed by COCA. Let $\zeta = \langle id, \sigma, key\rangle_k$ be the certificate with the largest serial number in these query_response messages. Server p invokes $threshold\_sign(m, E)$, where m is $(\mathcal{R}, \zeta)$ and E is the query_response messages collected from a quorum of servers, thereby obtaining $\langle \mathcal{R}, \zeta\rangle_k$.

4. Server p sends the following response to client c:

$$[p \longrightarrow c:\ \langle \mathcal{R}, \zeta\rangle_k].$$

A.5 Update Processing Protocol

1. Upon receiving a request $\mathcal{R} = \langle \mathsf{update}, c, seq, \zeta', \langle id, key\rangle, cred\rangle_c$ from a client c, server p first checks whether $\mathcal{R}$ is valid, based on the credentials *cred* provided. If it is valid, then p computes serial number $\sigma(\zeta) = (v + 1, h(\mathcal{R}))$ for a new certificate ζ, where v is the version number of ζ' and h is a public collision-free

hash function, and invokes $\mathit{threshold\_sign}(m, E)$, where m is $(id, \sigma(\zeta), key)$ and E is $\mathcal{R}$, thereby obtaining $\zeta = \langle id, \sigma(\zeta), key \rangle_k$.

2. Server p then sends an update_request message to every server q.

$$[\forall q.\ p \longrightarrow q:\ \langle \mathsf{update\_request}, p, \mathcal{R}, \zeta \rangle_p] \tag{iii}$$

3. Each server q, upon receiving an update_request message (iii), updates its certificate for id with ζ if and only if $\sigma(\zeta_q) < \sigma(\zeta)$, where ζ_q is the certificate for id stored by the server. Server q then sends back to p the following message:

$$[q \longrightarrow p:\ \langle \mathsf{update\_response}, q, p, \mathcal{R}, \mathsf{done} \rangle_q]$$

4. Server p repeats step 2 until it receives the update_response messages from a quorum of servers. p then invokes $\mathit{threshold\_sign}(m, E)$, where m is $(\mathcal{R}, \mathsf{done})$ and E is the set of update_response messages collected from a quorum of servers, thereby obtaining $\langle \mathcal{R}, \mathsf{done} \rangle_k$.

5. Server p sends the following response to client c:

$$[p \longrightarrow c:\ \langle \mathcal{R}, \mathsf{done} \rangle_k]$$

Bibliography

[1] R. Baldoni, J.-M. Helary, and M. Raynal. From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 273–282, New York, NY USA, June 25–28, 2000. IEEE Computer Society Technical Committee on Fault-Tolerant Computing, IFIP Working Group 10.4 on Dependable Computing and Fault Tolerance, IEEE Computer Society.

[2] B. Barak, A. Herzberg, D. Naor, and E. Shai. The proactive security toolkit and applications. In *Proceedings of the 6th ACM Conference on Computer and Communications Security (CCS'99)*, pages 18–27, Kent Ridge Digital Labs, Singapore, November 1999. ACM SIGSAC, ACM.

[3] R. Bazzi and G. Neiger. Optimally simulating crash failures in a Byzantine environment. In S. Toueg, P. G. Spirakis, and L. M. Kirousis, editors, *Distributed Algorithms, the 5th International Workshop (WDAG '91), Delphi, Greece, October 7–9, 1991, Proceedings*, volume 579 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 1991.

[4] R. Bazzi and G. Neiger. Simulating crash failures with many faulty processors. In S. Z. Adrian Segall, editor, *Distributed Algorithms, the 6th International Workshop (WDAG '92), Haifa, Israel, November 2–4, 1992, Proceedings*, volume 647 of *Lecture Notes in Computer Science*, pages 166–184. Springer, 1992.

[5] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, STOC'88*, pages 1–10, Chicago, IL USA, May 2–4, 1988. ACM.

[6] J. C. Benaloh. Secret sharing homomorphisms: Keeping shares of a secret secret. In A. M. Odlyzko, editor, *Advances in Cryptology—Crypto'86, Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, CA USA, 1986, August 11–15, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 251–260. Springer, 1987.

[7] G. R. Blakley. Safeguarding cryptographic keys. In R. E. Merwin, J. T. Zanca, and M. Smith, editors, *Proceedings of the 1979 National Computer Conference*,

volume 48 of *AFIPS Conference Proceedings*, pages 313–317, New York, NY USA, September 1979. AFIPS Press.

[8] C. Boyd. Digital multisignatures. In H. Baker and F. Piper, editors, *Cryptography and Coding*, pages 241–246. Clarendon Press, 1989.

[9] G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, November 1987.

[10] R. Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, Department of Computer Science and Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, June 1995.

[11] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor. Proactive security: Long-term protection against break-ins. *CryptoBytes (The technical newsletter of RSA Laboratories, a division of RSA Data Security Inc.)*, 3(1):1–8, Spring 1997.

[12] R. Canetti, S. Halevi, and A. Herzberg. Maintaining authenticated communication in the presence of break-ins. *Journal of Cryptography*, 13(1):61–106, 2000.

[13] R. Canetti and A. Herzberg. Maintaining security in the presence of transient faults. In Y. Desmedt, editor, *Advances in Cryptology—Crypto'94, the 14th Annual International Cryptology Conference, Santa Barbara, CA USA, August 21–25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 425–438. Springer, 1994.

[14] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA USA, November 2000.

[15] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating System Design and Implementation (OSDI'99)*, pages 173–186, New Orleans, LA USA, February 22–25, 1999. USENIX Association, IEEE TCOS, and ACM SIGOPS.

[16] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI'00)*, San Diego, CA USA, October 22–25, 2000. USENIX Association, IEEE TCOS, and ACM SIGOPS.

[17] CCITT. Recommendation X.509: The directory-authentication framework, 1988.

[18] M. Cerecedo, T. Matsumoto, and H. Imai. Efficient and secure multiparty generation of digital signatures based on discrete logarithms. *IEICE Transactions*

on Fundamentals of Electronics, Information and Communication Engineers, E76-A(4):532–545, April 1993.

[19] CertCo, Inc. http://www.certco.com.

[20] D. Chaum. Blind signatures for untraceable payments. In D. Chaum, R. L. Rivest, and A. T. Sherman, editors, *Advances in Cryptology—Crypto'82, A Workshop on the Theory and Application of Cryptography, Santa Barbara, CA USA, August 23–25, 1982*, pages 199–203. Plenum Press, New York, 1983.

[21] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, October 1985.

[22] D. Chaum. Blinding for unanticipated signatures. In D. Chaum and W. L. Price, editors, *Advances in Cryptology—Eurocrypt'87, Workshop on the Theory and Application of Cryptographic Techniques, Amsterdam, The Netherlands, April 13–15, 1987, Proceedings*, volume 304 of *Lecture Notes in Computer Science*, pages 227–233. Springer-Verlag, 1988.

[23] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, STOC'88*, pages 11–19, Chicago, IL USA, May 2–4, 1988. ACM.

[24] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In *Proceedings of the 26th Symposium on Foundations of Computer Science (FOCS'85)*, pages 383–395, Portland, OR USA, October 21–23, 1985. IEEE.

[25] C.-S. Chow and A. Herzberg. Network randomization protocol: A proactive pseudo-random generator. In *Proceedings of the 5th USENIX UNIX Security Symposium*, pages 55–63, Salt Lake City, UT USA, June 5–7, 1995. USENIX Association.

[26] B. A. Coan. *Achieving Consensus in Fault-Tolerant Distributed Computer Systems: Protocols, Lower Bounds, and Simulations*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA USA, June 1987.

[27] Y. Desmedt. Society and group oriented cryptography: A new concept. In C. Pomerance, editor, *Advances in Cryptology—Crypto'87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, CA USA, August 16–20, 1987, Proceedings*, volume 293 of *Lecture Notes in Computer Science*, pages 120–127. Springer, 1988.

[28] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *Advances in Cryptology—Crypto'89, the 9th Annual International Cryptology*

Conference, Santa Barbara, CA USA, August 20–24, 1989, Proceedings, volume 435 of *Lecture Notes in Computer Science*, pages 307–315. Springer, 1990.

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

[30] A. Doudou, B. Garbinato, and R. Guerraoui. Modular abstractions for devising Byzantine-resilient state machine replication. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, Nürnberg, Germany, October 16–18, 2000.

[31] T. Draelos, V. Hamilton, and G. Istrail. Proactive DSA application and implementation. Technical Report SAND--97-2939C; CONF-980554--, Sandia National Laboratories, Albuquerque, NM USA, May 3, 1998.

[32] C. Dwork and M. Naor. Pricing via processing or combating junk mail. In E. F. Brickell, editor, *Advances in Cryptology—Crypto'92, the 12th Annual International Cryptology Conference, Santa Barbara, CA USA, August 16–20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1992.

[33] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the 28th Annual Symposium on the Foundations of Computer Science*, pages 427–437. IEEE, October 12–14, 1987.

[34] M. J. Fischer, N. A. Lynch, and M. S. Peterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, April 1985.

[35] B. Fox and B. LaMacchia. Certificate revocation: Mechanics and meaning. In R. Hirschfeld, editor, *Financial Cryptography, the 2nd International Conference (FC'98), Anguilla, British West Indies, February 23–25, 1998, Proceedings*, volume 1465 of *Lecture Notes in Computer Science*, pages 158–164. Springer, 1998.

[36] Y. Frankel, P. Gemmel, P. MacKenzie, and M. Yung. Optimal resilience proactive public-key cryptosystems. In *Proceedings of the 38th Symposium on Foundations of Computer Science*, pages 384–393, Miami Beach, FL USA, October 20–22, 1997. IEEE.

[37] Y. Frankel, P. Gemmell, P. MacKenzie, and M. Yung. Proactive RSA. In B. S. Kaliski Jr., editor, *Advances in Cryptology—Crypto'97, the 17th Annual International Cryptology Conference, Santa Barbara, CA USA, August 17–21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 440–454. Springer, 1997.

[38] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol Version 3.0. Internet Draft, November 1996.

[39] J. A. Garay. Reaching (and maintaining) agreement in the presence of mobile faults. In G. Tel and P. M. B. Vitányi, editors, *Distributed Algorithms, the 8th International Workshop (WDAG '94), Terschelling, The Netherlands, September 29—October 1, 1994, Proceedings*, volume 857 of *Lecture Notes in Computer Science*, pages 253–264, 1994.

[40] J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1–2):363–389, July 28, 2000.

[41] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

[42] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed systems security architecture. In *Proceedings of the 12th National Computer Security Conference*, pages 305–319, Baltimore, MD USA, October 10–13, 1989. National Institute of Standards and Technology (NIST), National Computer Security Center (NCSC).

[43] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust and efficient sharing of RSA functions. In N. Koblitz, editor, *Advances in Cryptology—Crypto'96, the 16th Annual International Cryptology Conference, Santa Barbara, CA USA, August 18–22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996.

[44] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. In U. M. Maurer, editor, *Advances in Cryptology—Eurocrypt'96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12–16, 1996, Proceedings*, volume 1233 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 1996.

[45] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–159, Asilomar Conference Grounds, Pacific Grove, CA USA, December 10–12, 1979. ACM.

[46] V. D. Gligor. A note on denial-of-service in operating systems. *IEEE Transactions on Software Engineering*, 10(3):320–324, May 1984.

[47] O. Goldreich, S. Micali, and A. Wigderson. How to play ANY mental game. In *Proceedings of the 19th Annual Conference on Theory of Computing, STOC'87*, pages 218–229, New York, NY USA, May 25–27, 1987. ACM.

[48] L. Gong. Increasing availability and security of an authentication service. *IEEE Journal on Selected Areas in Communications*, 11(5):657–662, June 1993.

[49] L. Gong and P. Syverson. Fail-stop protocols: An approach to designing secure protocols. In R. K. Iyer, M. Morganti, W. K. Fuchs, and V. Gligor,

editors, *Dependable Computing for Critical Applications 5*, pages 79–99. IEEE Computer Society Press, 1998.

[50] V. Hadzilacos. Byzantine agreement under restricted types of failures (not telling lies). Technical Report 18-83, Harvard University, Cambridge, MA, 1983.

[51] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94–1425, Department of Computer Science, Cornell University, Ithaca, NY USA, May 1994.

[52] L. Harn. Group oriented (t, n) digital signature scheme. *IEE Proceedings—Computer and Digital Techniques*, 141(5):307–313, September 1994.

[53] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public-key and signature schemes. In *Proceedings of the 4th Annual Conference on Computer Communications Security*, pages 100–110, Zurich, Switzerland, April 1–4, 1997. ACM SIGSAC, ACM.

[54] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In D. Coppersmith, editor, *Advances in Cryptology—Crypto'95, the 15th Annual International Cryptology Conference, Santa Barbara, CA USA, August 27–31, 1995, Proceedings*, volume 963 of *Lecture Notes in Computer Science*, pages 457–469. Springer, 1995.

[55] M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. In *Proceedings of the IEEE Global Communication Conference (GLOBALCOM'87)*, pages 99–102, Tokyo, Japan, November 1987.

[56] A. Iyengar, R. Cahn, C. Jutla, and J. Garay. Design and implementation of a secure distributed data repository. In *Proceedings of the 14th IFIP International Information Security Conference (SEC'98)*, pages 123–135, Vienna, Austria and Budapest, Hungary, August 31–September 4, 1998. International Federation for Information Processing, TC11: Security and Protection in Information Processing Systems.

[57] S. Jarecki. Proactive secret sharing and public key cryptosystems. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA USA, September 1995.

[58] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the 1999 Network and Distributed System Security Symposium*, pages 151–165, San Diego, CA USA, February 4–5, 1999. Internet Society.

[59] P. Karn and W. Simpson. The Photuris session key management protocol. Internet Draft: draft-simpson-photuris-17.txt, November 1997.

[60] C. Kaufman. DASS: Distributed authentication security service. Request for Comments 1507, September 1993.

[61] S. T. Kent, D. Ellis, P. Helinek, K. Sirois, and N. Yuan. Internet infrastructure security countermeasures. Technical Report 8173, BBN, January 1996.

[62] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a Byzantine environment using an unreliable fault detector. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS'97)*, pages 61–76, Chantilly, France, December 10–12, 1997.

[63] P. C. Kocher. On certificate revocation and validation. In R. Hirschfeld, editor, *Financial Cryptography, the 2nd International Conference (FC'98), Anguilla, British West Indies, February 23–25, 1998, Proceedings*, volume 1465 of *Lecture Notes in Computer Science*, pages 172–177. Springer, 1998.

[64] L. M. Kornfelder. Toward a practical public-key cryptosystem. Bachelor's thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA USA, 1978.

[65] C. Labovitz, G. R. Malan, and F. Jahanian. Internet routing instability. In *Proceedings of the Annual Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'97)*, pages 115–126, Cannes, French Riviera, France, September 16–18, 1997. ACM.

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

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

[68] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

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

[70] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th Symposium on Reliable Distributed Systems*, pages 51–58, West Lafayette, IN USA, October 20–22, 1998. IEEE Computer Society.

[71] P. McDaniel and A. Rubin. A response to "Can we eliminate revocation lists?". In *Financial Cryptography, the 4th International Conference (FC'00), Anguilla, British West Indies, February 21–24, 2000, Proceedings*, 2000.

[72] C. Meadows. A formal framework and evaluation method for network denial of service. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 4–13, Mordano, Italy, June 28–30, 1999. IEEE Computer Society Press.

[73] C. Meadows. A cost-based framework for analysis of denial of service in networks. *Journal of Computer Security*, 9(1/2):143–164, 2001.

[74] R. C. Merkle. Secure communication over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978.

[75] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, Stanford, CA USA, 1979.

[76] J. K. Millen. A resource allocation model for denial of service. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, pages 137–147, Oakland, CA USA, May 1992. IEEE Computer Society Press.

[77] J. K. Millen. Denial of service: A perspective. In F. Cristian, G. L. Lann, and T. Lunt, editors, *Dependable Computing for Critical Applications 4*, pages 93–108. Springer, 1995.

[78] M. Myers. Revocation: Options and challenges. In R. Hirschfeld, editor, *Financial Cryptography, the 2nd International Conference (FC'98), Anguilla, British West Indies, February 23–25, 1998, Proceedings*, volume 1465 of *Lecture Notes in Computer Science*, pages 165–171. Springer, 1998.

[79] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet public key infrastructure online certificate status protocol (OCSP). Request For Comments 2560, June 1999.

[80] M. Naor, B. Pinkas, and O. Reingold. Distributed pseudo-random functions and KDCs. In J. Stern, editor, *Advances in Cryptology—Eurocrypt'99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2–6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 327–346. Springer, 1999.

[81] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, September 1990.

[82] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, 1994.

[83] OpenSSL Project. http://www.openssl.org.

[84] R. Oppliger. Protecting key exchange and management protocols against resource clogging attacks. In B. Preneel, editor, *Proceedings of the IFIP TC6*

and TC11 Joint Working Conference on Communications and Multimedia Security (CMS'99), pages 163–175, Leuven, Belgium, September 20–21, 1999. International Federation for Information Processing, Kluwer Academic Publishers.

[85] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of the 10th Annual Symposium on Principles of Distributed Computing (PODC'91)*, pages 51–59, Montreal, Quebec, Canada, August 19–21, 1991. ACM.

[86] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[87] T. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum, editor, *Advances in Cryptology—Crypto'91, the 11th Annual International Cryptology Conference, Santa Barbara, CA USA, August 11–15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1992.

[88] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, April 1989.

[89] T. Rabin. A simplified approach to threshold and proactive RSA. In H. Krawczyk, editor, *Advances in Cryptology—Crypto'98, the 18th Annual International Cryptology Conference, Santa Barbara, CA USA, August 23–27, 1998*, volume 1462 of *Lecture Notes in Computer Science*, pages 89–104. Springer, 1998.

[90] M. K. Reiter. The Rampart toolkit for building high-integrity services. In K. P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems, International Workshop, Dagstuhl Castle, Germany, September 5–9, 1994, Selected Papers*, volume 938 of *Lecture Notes in Computer Science*, pages 99–110. Springer, 1995.

[91] M. K. Reiter. Distributing trust with the Rampart toolkit. *Communications of the ACM*, 39(4):71–74, April 1996.

[92] M. K. Reiter, K. P. Birman, and R. van Renesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, November 1994.

[93] M. K. Reiter, M. K. Franklin, J. B. Lacy, and R. N. Wright. The Ω key management service. *Journal of Computer Security*, 4(4):267–297, 1996.

[94] M. Richtel and S. Robinson. Several web sites attacked following assaults on Yahoo. *New York Times*, February 8, 2000.

[95] R. L. Rivest. Can we eliminate revocation lists? In R. Hirschfeld, editor, *Financial Cryptography, the 2nd International Conference (FC'98), Anguilla, British West Indies, February 23–25, 1998, Proceedings*, volume 1465 of *Lecture Notes in Computer Science*, pages 178–183. Springer, 1998.

[96] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[97] F. B. Schneider. Abstractions for fault tolerance in distributed systems (Invited Paper). In H.-J. Kugler, editor, *Information Processing 86, Proceedings of the IFIP 10th World Computer Congress*, pages 727–734, Dublin, Ireland, September 1–5, 1986. North-Holland/IFIP.

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

[99] F. B. Schneider, editor. *Trust in Cyberspace*. Committee on Information Systems Trustworthiness, National Research Council, 1999.

[100] C. Schuba, I. Krsul, M. Kuhn, G. Spafford, A. Sundaram, and D. Zamboni. Analysis of a denial of service attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 208–223, Oakland, CA USA, May 1997. IEEE Computer Society Press.

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

[102] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[103] S. G. Stubblebine. Recent-secure authentication: Enforcing revocation in distributed systems. In *Proceedings of the 1995 IEEE Symposium on Research in Security and Privacy*, pages 224–234, Oakland, CA USA, May 1995. IEEE Computer Society Press.

[104] J. J. Tardo and K. Algappan. SPX: Global authentication using public key certificates. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 232–244, Oakland, CA USA, May 1991. IEEE Computer Society Press.

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

[106] T. Wu, M. Malkin, and D. Boneh. Building intrusion tolerant applications. In *Proceedings of the 8th USENIX Security Symposium*, pages 79–91, Washington, D.C. USA, August 22–26, 1999. USENIX Association.

[107] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliççöte, and P. K. Khosla. Survivable information storage system. *IEEE Computer*, 33(8):61–68, August 2000.

[108] A. C. Yao. Protocols for secure computation. In *Proceedings of the 23rd Symposium on Foundations of Computer Science (FOCS'82)*, pages 160–164, Chicago, IL USA, November 3–5, 1982. IEEE.

[109] C.-F. Yu and V. D. Gligor. A specification and verification method for preventing denial of service. *IEEE Transactions on Software Engineering*, 16(6):581–592, June 1990.

[110] L. Zhou and Z. J. Haas. Securing ad hoc networks. *IEEE Network*, 13(6):24–30, November/December 1999.

[111] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. Technical Report 2000-1828, Department of Computer Science, Cornell University, Ithaca, NY USA, December 2000.