Mobile Proactive Secret Sharing

by

David Andrew Schultz

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2007

Author .....
Department of Electrical Engineering and Computer Science
January 29, 2007

Certified by .....
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Mobile Proactive Secret Sharing

by

David Andrew Schultz

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

This thesis describes mobile proactive secret sharing (MPSS), an extension of proactive secret sharing. Mobile proactive secret sharing is much more flexible than proactive secret sharing in terms of group membership: instead of the group of shareholders being exactly the same from one epoch to the next, we allow the group to change arbitrarily. In addition, we allow for an increase or decrease of the threshold at each epoch. We give the first known efficient protocol for MPSS in the asynchronous network model. We present this protocol as a practical solution to the problem of long-term protection of a secret in a realistic network.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

I would like to thank my advisor, Barbara Liskov, for her encouragement and support throughout the process of researching and writing this thesis. Her meticulous eye caught many problems with the protocol design early on, and her many reviews and suggestions of this document significantly improved the clarity of the presentation. I am greatly indebted to Moses Liskov as well, who was particularly helpful in designing the cryptographic elements of the protocol and verifying their correctness.

I would also like to thank my past and present officemates at MIT: Winnie Cheng, James Cowling, Ben Leong, Daniel Myers, Alice Reyzin, Rodrigo Rodrigues, Liuba Shrira, and Ben Vandiver. In addition to the many fruitful discussions I have had with them on aspects of this thesis, they have been a pleasure to work with, and have provided the occasional distractions that helped preserve my sanity. Several of my friends and collaborators at Berkeley, in particular David Molnar and David Wagner, helped spark my interest in threshold cryptography and the topic of this thesis, so I would like to thank them as well.

I am deeply indebted to my parents, Jean and Randy, and my sister Laura, for all of the love and support they have shown me throughout my lifetime. They helped me through good times and bad, and the examples they set for me have been invaluable. I thank God above all for my family and my many blessings, and for His love and guidance in my life.

Finally, I thank Shafi Goldwasser, Stanislaw Jarecki, and Ron Rivest for useful discussions about secret sharing and other topics related to this thesis.

Contents

List of Figures

Chapter 1

Introduction

Secret sharing allows a collection of parties to possess shares of a secret value (such as a secret key), such that any $t + 1$ shares can be used to reconstruct the secret, yet any t shares provide no information about the secret. Sharing of cryptographic keys is crucial in distributed systems intended to withstand Byzantine faults, that is, failures that cause servers to behave arbitrarily badly, perhaps because they have been compromised. This is in contrast to systems that tolerate only fail-stop failures, in which servers stop responding altogether. In the context of Byzantine faults, secret sharing allows systems to perform cryptographic operations securely, preserving the secrecy of the keys despite up to t malicious servers.

In long-lived systems, however, servers can be compromised over time, giving an adversary the opportunity to collect more than t shares and recover the secret. Additionally, systems may fail to function properly, for instance due to hardware failure or an attack. To prevent the number of failures from exceeding the threshold the system is designed to tolerate, servers must be repaired or replaced over time, perhaps with newly-installed servers. Moreover, this replacement must be performed

periodically even in the absence of detected faults due to the potential for lie-in-wait attacks. In this type of attack, faulty servers appear to behave correctly while an attacker compromises additional machines; once $t+1$ servers have been compromised, they start behaving badly or simply reveal the secret to the attacker.

Proactive secret sharing (PSS) schemes, e.g., [CKLS02, DJ97, HJKY95, OY91, WWW02, ZSvR05], address the problem that shares can be exposed or lost over time due to Byzantine faults. In PSS, servers execute a share regeneration protocol, in which a new set of shares of the same secret is generated and the old shares discarded, rendering useless any collection of t or fewer shares the adversary may have learned. Furthermore, PSS schemes typically provide a share recovery protocol so that a full set of new shares can be generated even if some of the old shares (up to some maximum number of faults tolerated) have been lost.

However, proactive secret sharing has limitations that make it impractical in many situations. Traditional proactive secret sharing imagines a world in which servers are "recovered" at a rate equal to that at which they are compromised. A significant flaw in PSS is that "recovery" of nodes that have been compromised or suffered irreparable hardware failures is problematic. Merely replacing the machine's hardware or software might eliminate the fault, but it does not undo all of the damage; each machine's identity on the network is defined by some secret state (e.g., private keys), and after a failure, this state might be deleted, corrupted, or known to the adversary. Consequently, the recovered server needs new secret state and in essence, a new identity, and we may as well consider it to be a new server. Ordinary PSS schemes [CKLS02, HJKY95, OY91] generate new shares of the shared secret, but they assume that other secret state associated with each node is never compromised, so the identities of the shareholders are fixed over the lifetime of the system.

A better approach is to allow shareholders to be replaced with different nodes. This provides a reasonable methodology for a system administrator: identify a set of "fresh" nodes, e.g., recently added nodes or newly recovered nodes with new keys, and direct the secret shares to be moved to them. However, PSS schemes do not permit the replacement of one shareholder with another. Furthermore, a difficulty in defining such schemes is that a naïve transfer of shares from an old group of servers to a new group may reveal the secret if more than t faulty servers exist between the old group and the new group.

An additional point is that PSS schemes do not allow the threshold t to change, which may be desirable in a long-lived system. The threshold has a meaning: it represents an assumption about how easily nodes can become corrupted. If current events dictate a reevaluation of this assumption, it would be better to change to a new threshold rather than start over. For instance, a new vulnerability in Windows might lead to a decision to increase t, whereas the patch of that vulnerability might later lead to it being decreased.

1.1 Motivating Proactive Secret Sharing

This section describes a typical scenario in which proactive secret sharing plays a key role, and clarifies why secret sharing is important. Byzantine-fault-tolerant systems are typically intended to implement long-running, reliable services that process requests on behalf of clients. They function correctly despite up to a threshold number of Byzantine faults among the servers that carry out the operations. If the system operates over an untrusted network such as the Internet, it must also be able to certify the results of these operations using cryptography (threshold signatures). Additionally, the system may need to selectively decrypt messages, such that the

authority to decrypt is distributed amongst the servers and faulty servers cannot decrypt messages unilaterally (threshold encryption).

Byzantine-fault-tolerant systems that require selective decryption have a clear need for proactive secret sharing. If one were to use an ordinary public key encryption scheme in which the private key were distributed to all of the servers, a single malicious server could decrypt ciphertexts arbitrarily. Threshold encryption schemes [BBH06, CG99] address this issue by assigning a *key share* to each server and distributing the power to decrypt amongst the servers such that more than a threshold number of servers must participate in order to decrypt any message. As more and more servers become faulty, share refreshment via PSS is needed to compensate.

If the system needs to sign messages, it might be less obvious to the reader why PSS is important, so this section describes a simpler approach and shows why it fails. An alternative to proactive secret sharing is to assign a public/private key pair to each server and make the corresponding public keys well-known, then have servers sign the result of operations using their private keys. Any sufficiently large set of signatures that all agree is accepted as a valid certificate. However, this is a poor solution because it poses a significant key management problem. All clients must have a trusted certificate that enumerates the public keys of the current set of servers, and each time the set of servers changes, the old servers must certify the new ones to the clients. The current server group must be able to prove its authority to clients that are arbitrarily out of date, which results in unwieldily certification chains that grow linearly over time. Furthermore, if each certificate is comprised of a threshold number of signatures from individual servers, the certificates themselves might be relatively large.

Proactive secret sharing (PSS) and threshold signature schemes can address these problems. A single public/private key pair is generated by a trusted dealer when the

system is first initialized, the public key is published, and *shares* of the private key are distributed to the initial set of servers. The servers use threshold signature schemes [DF91, GJKR96, Lan95] to produce signatures using the shared secret key. To handle share exposure due to Byzantine faults, the servers execute a share regeneration protocol, in which a new set of shares of the same secret is generated and the old shares discarded, rendering useless any collection of t or fewer shares the adversary may have learned. Hence, PSS solves the key management problem because there is only one key that never changes; instead, only the *sharing* of that key changes over time. The adversary can learn nothing about the key unless it is able to acquire $t+1$ shares before the next share regeneration operation takes place.

1.2 Contributions

This thesis introduces MPSS, a new mobile proactive secret sharing scheme. In MPSS, as in other proactive secret sharing schemes, servers execute a periodic share refreshment protocol. Additionally, MPSS allows the shares to be transferred to a new group of share holders, which may be larger or smaller than the old group, without revealing any extra information to corrupted nodes. Hence, proactive secret sharing can be thought of as a special case of MPSS where the old and new groups are restricted to be identical. This thesis gives an efficient and practical protocol for MPSS in the asynchronous network setting.

This thesis makes the following major contributions:

- It describes a new technique for carrying out MPSS. Prior schemes were either limited to ordinary PSS (e.g., [CKLS02, HJKY95, ZSvR05]), or they required an amount of communication that was exponential in the number of sharehold-

ers (e.g., [WWW02, ZSvR05]). Hence, we describe the first *practical* MPSS scheme.

- It shows how our MPSS scheme can be extended to support changing the threshold of faults tolerated from one epoch to the next.

- It describes an efficient way of carrying out the secret resharing, based on the use of a *coordinator* that determines the resharing proposals under consideration. This allows us to achieve significantly better performance in the common case where the coordinator is not faulty.

- It describes a simple extension to the forward-secure encryption scheme of Canetti, Halevi, and Katz [CHK03] to support identity-based forward-secure encryption. This extension allows us to improve system performance by enabling efficient provable accusations that can be verified by others without revealing secret information. Note that the use of this extension is not required in order for our basic protocol to work.

- It describes a new correctness condition for secret sharing systems. The condition is important because it rules out a particularly attractive attack in which the adversary isolates a node and then, without compromising it, fools it into revealing its secret information. We also show that implementing this condition requires the use of forward-secure signing.

A further point is that we show how to use *publically verifiable accusations* (PVAs) to increase the efficiency of our protocol. Corrupted nodes can attempt to disrupt resharing by producing incorrect information; PVAs allow honest servers to prove the information is incorrect and thus more easily come to agreement on which of their

peers have behaved correctly. Verifiable accusations are a general-purpose technique that may also be applied to other protocols that rely upon Byzantine agreement; prior work has had to either assume synchrony to implement accusations or avoid them altogether.

Thesis Outline

The thesis is organized as follows. We begin in Chapter 2 by discussing related work, including systems we build upon, as well as other proactive secret sharing schemes and their limitations. Chapter 3 defines and justifies the network model and our assumptions. Chapter 4 gives an overview of our approach, and in particular describes the mathematics behind it, and how new shares are computed and verified. The thesis goes into more detail on our network protocol in Chapter 5, and Chapter 6 describes some important optimizations that improve performance. Chapter 7 presents modifications to the protocol that allow the group size to grow and shrink. Validation of our approach can be found in Chapter 8, which details correctness conditions, theorems, and proofs. Chapter 9 evaluates the performance of our scheme and analytically compares it against several other techniques. Finally, we conclude in Chapter 10.

Chapter 2

Related Work

We first briefly mention basic secret sharing and verifiable secret sharing schemes. Then we discuss proactive secret sharing, which attempts to address the problem that shares learned by the adversary are compromised, posing a problem for long-lived systems. However, proactive secret sharing schemes are limited in that they assume that the set of shareholders remains the same forever. Hence, we discuss mobile proactive secret sharing schemes that allow the set of shareholders and possibly the size of this set to change.

In this thesis, t denotes the threshold. That is, in the secret sharing schemes we discuss, any t shares will not reveal the secret, yet $t + 1$ shares can be combined to recover it. We let n denote the total number of shareholders.

2.1 Basic Schemes

2.1.1 Secret Sharing

Secret sharing was first proposed by Shamir [Sha79] and independently by Blakley [Bla79]. These seminal schemes operate under a very simple model: a trusted dealer has a secret and distributes a different share of that secret to each server. Shamir demonstrates that a passive adversary who learns up to t shares of the secret gains no partial information about the secret, yet any $t + 1$ servers can combine their shares to recover the secret. Blakley's scheme makes a similar guarantee, except that it does not provide perfect secrecy; combinations of t or fewer shares reveal partial information about the secret, and additional modifications are needed to ensure perfect secrecy. Shamir's scheme is based on interpolation of polynomials over a finite field, whereas Blakley's scheme encodes the secret as an intersection of n-dimensional hyperplanes. Each share in Shamir's scheme is the same size as the original secret, but shares in Blakley's scheme are t times as large. Shamir's scheme is more widely used because it provides stronger guarantees and better space efficiency using only relatively simple mathematics. Most of the other schemes discussed in this paper are based on Shamir's scheme, which we describe in more detail in Chapter 4. The PSS scheme of Zhou et al. [ZSvR05] (see Section 2.3.3) is based on a different secret sharing mechanism that is even simpler but more limited than both Shamir's and Blakley's schemes.

2.1.2 Verifiable Secret Sharing

Feldman [Fel87] and Pedersen [Ped91a] introduced verifiable secret sharing schemes based on Shamir's work. These schemes allow shareholders to determine whether

the dealer sent them valid shares of the secret, hence allowing them to come to a consensus regarding whether the secret was shared successfully. In this context, the dealer is *semi-trusted*; it does not reveal the secret, but it might attempt to fool servers into accepting an invalid sharing of the secret. Verifiable secret sharing is an important component in many distributed secret sharing protocols involving untrusted participants because the protocols typically involve each server acting as a semi-trusted dealer to all of the others.

Feldman's and Pedersen's schemes have similar efficiency but slightly different security guarantees. Feldman's scheme is perfectly binding (meaning that an untrusted dealer cannot fool shareholders into accepting an invalid sharing) and computationally binding (meaning that secrecy is subject to computational hardness assumptions and the amount of computation available to the shareholders). Pedersen's scheme, on the other hand, is computationally binding and perfectly hiding. It has been shown that schemes that are both perfectly hiding and perfectly binding do not exist. In the context of the proactive secret sharing schemes we discuss, a computationally unbounded attacker can exploit the VSS to expose the secret regardless of which choice we make, so the distinction is a non-issue for us. We focus on Feldman's scheme because it is notationally easier to describe than Pedersen's scheme; however, the systems we describe could be adapted to either scheme. This thesis describes Feldman's scheme in more detail in Section 4.

2.2 Proactive Secret Sharing

2.2.1 Ostrovsky and Yung

Proactive secret sharing was introduced by Ostrovsky and Yung in [OY91] as a way to cope with network worms or viruses. In their model, an adversary infects shareholders at a constant rate, but shareholders are also rebooted and restored to their correct state at an equal rate. Hence, they assume that in any given time period (we use the term *epoch* herein), $t < \lfloor n/2 \rfloor$ shareholders may be faulty. (Note that this threshold is better than the $t < \lfloor n/3 \rfloor$ typically required for asynchronous schemes, and is possible only because the correctness of their protocol is based on the unrealistic synchrony assumption that servers that fail to respond within some fixed amount of time are faulty.) Shareholders preserve the privacy of the shared secret by executing a *refresh protocol* to generate a new sharing of the secret, discarding their old shares, and using the new shares for the next epoch. In [OY91], the refresh protocol is implemented via a generic secure multi-party computation protocol on the existing shares. These multi-party protocols (e.g., [BGW88, CCD88, RBO89]) are general but inefficient. Some are implemented in terms of many instances of verifiable secret sharing, with the number of rounds proportional to the depth of a circuit that implements the function to be computed.

This seminal work is important because it was the first to demonstrate that proactive secret sharing is theoretically possible; however, the Ostrovsky and Yung scheme is infeasible in practice because performing nontrivial calculations using generic secure multi-party protocols is expensive. Furthermore, Ostrovsky and Yung assume that the network is synchronous, and that secure channels are uncompromised by

past corruptions of the endpoints. Practical implementations of secure channels involve secret keys that would be exposed by a compromise of the endpoints, and hence it is unclear how to recover the node in that case, since the adversary now knows the node's secret keys. Also, although they show that "recovery" of a machine's state is possible in theory by having all of the other participants construct it via a secure multi-party computation, it is unclear how one might perform recovery efficiently in practice.

2.2.2 Herzberg et al.

Herzberg et al. [HJKY95] address the efficiency problem by introducing a protocol specialized to the problem of generating a new secret sharing. In their scheme, discussed in more detail in Section 4, participants numbered $i = 1 \ldots n$ have an initial Shamir sharing with polynomial P (i.e., secret $s = P(0)$ with shares $P(1) \ldots P(n)$), and in the refresh protocol they construct a new sharing $P+Q$, where Q is a random polynomial with $Q(0) = 0$. To handle the case where a previously-corrupted node k has lost its old share and needs to recover its correct state, other participants execute a *recovery protocol* in which each other party i sends $P(i) + R_k(i)$ to k, where R_k is a random polynomial with $R_k(k) = 0$. Note that in an asynchronous network, recovery may additionally be needed for nodes that have never been faulty, simply because they never received their share from a previous execution of the protocol.

Herzberg et al.'s scheme is difficult to translate into an asynchronous network protocol partly because it has an accusation/defense phase in which the network is assumed to be reliable. Each server sends a message to each other server, and if any senders misbehave, the recipients broadcast accusations against them. Then the accused servers must broadcast a defense, or else they will be deemed faulty by the

other servers. However, in an asynchronous network, we do not know how long it will take for us to receive defenses from honest servers, and if we establish a specific timeout, we may spuriously deem honest servers to be faulty if their responses are delayed. The authors claim in a footnote that for certain encryption schemes such as RSA, the defense step can be eliminated, which might simplify the translation. However, we show that the encryption scheme must also be forward-secure. Furthermore, fixing the problem in an asynchronous network requires a property of the encryption primitive that is stronger than chosen ciphertext security, and neither RSA nor RSA under the Fujisaki-Okamoto transformation[FO99] satisfy this property. (More specifically, in addition to revealing the plaintext, the decryption oracle discloses all randomness used in the encryption computation.) In Section 6.1, we present a verifiable accusation scheme that avoids this problem, although our protocol does not require verifiable accusations.

2.2.3 Cachin et al.'s Asynchronous Scheme

The protocol of Cachin, Kursawe, Lysyanskaya, and Strobl [CKLS02] is the first efficient scheme in the asynchronous model, also for $t < \lfloor n/3 \rfloor$. Whereas the Herzberg et al. scheme [HJKY95] computes each new share $P'(i)$ as a function of a *corresponding* old share $P(i)$, the Cachin scheme is based on *resharing* the shares of the secret and combining the resulting subshares to form new shares of the secret. Their paper first presents a protocol for asynchronous verifiable secret sharing, then shows how to build an asynchronous proactive secret sharing scheme by having each honest shareholder create a VSS of its share. Their VSS scheme is similar to the one of Stinson and Wei [SW99], and the method of computing new shares from subshares is based on the linearity of Lagrange interpolation, which was proposed by Desmedt

and Jajodia in [DJ97]; however, the authors seem to be unaware of either of these earlier works.

To handle share recovery for participants who have lost or never received their shares, Cachin et al. use a two-dimensional sharing $P(\cdot, \cdot)$ in which shares are one-dimensional projections $P(i, y)$ and $P(x, i)$; thus, any participant can interpolate its share given the points of overlap with at least $t + 1$ other shares. Cachin et al.'s protocol requires that a significant amount of information be broadcast by each participant to each other participant even in the absence of faults, whereas our scheme achieves better efficiency in the common case by using a coordinator. Moreover, their protocol does not support changing the set of shareholders.

This thesis describes additional details of the Cachin et al. scheme, primarily in the context of performance, in Section 9.3.

2.3 Mobile Proactive Secret Sharing

2.3.1 The Desmedt and Jajodia Scheme

Desmedt and Jajodia [DJ97] were the first to propose an extension of proactive secret sharing, which they call secret redistribution and we call mobile proactive secret sharing, that allows the set of shareholders, number of shareholders, and threshold to change. They use the same strategy as Cachin et al. [CKLS02] (albeit in more generic group-theoretic terms), in which each "old" shareholder acts as a dealer and shares its share of the secret to the new shareholders. The new shareholders then combine the subshares from some set of at least $t + 1$ old shareholders to produce new shares of the secret. However, their scheme is not verifiable, and thus faulty nodes in the old group that behave incorrectly can cause the new shareholders to

generate an invalid sharing of the secret. Furthermore, their scheme is not formulated in terms of a concrete network protocol, so it is unclear, for instance, how the new shareholders are to decide which old shareholders to accept shares from if there are faulty old shareholders and lost network messages. A direct implementation of their proposal would only work in a synchronous network with a passive adversary that can eavesdrop and corrupt nodes, but not generate spurious messages.

2.3.2 Wong, Wang, and Wing Scheme

Wong, Wang, and Wing [WWW02] improve upon Desmedt and Jajodia [DJ97] in two significant ways. First, they provide a complete, implementable, network protocol. Second, their scheme is verifiable, so cheating "old" shareholders can't compromise the validity of the sharing or prevent it from completing. However, their scheme relies upon all of the new shareholders being honest for the duration of the protocol, which is an unrealistic assumption. Furthermore, their scheme is inefficient in the presence of malicious old shareholders because it gives the new shareholders no way to determine which old shareholders sent bad information. Hence, they must restart their protocol potentially an exponential number of times using different subsets of old shareholders, until a set of entirely honest shareholders is chosen.

2.3.3 APSS

Zhou et al. [ZSvR05] proposed the first technique that works in an asynchronous model, which they call APSS. Here the threshold is modified from $t < \lfloor n/2 \rfloor$ to $t < \lfloor n/3 \rfloor$, which is optimal for protocols relying upon asynchronous Byzantine agreement [CL02].

Their construction is based on an exclusive-or sharing scheme rather than on

Shamir's secret sharing. The exclusive-or scheme is simpler and more limited because it only supports k-out-of-k sharings, i.e., all the shares are required to reconstruct. In the exclusive-or scheme, given a secret s, generate random values $r_1, r_2, \ldots, r_{t-1}$ and output shares $r_1, \quad r_2, \quad \ldots, \quad r_{t-1}, \quad r_1 \oplus r_2 \oplus \cdots r_{t-1} \oplus s$, where $\oplus$ denotes bitwise exclusive or. Any combination of $k-1$ of these shares is indistinguishable from random values, but the exclusive-or of all of the shares is s. For every possible subset of honest shareholders of size $t+1$, they produce a trivial $t+1$-out-of-$t+1$ sharing of the secret using the exclusive-or sharing; hence, any $t+1$ shareholders can reconstruct the secret. However, this construction results in exponentially large shares of the secret; hence, the communication required to refresh those shares is exponential in n, the number of shareholders. Chen [Che04] implemented and analyzed their scheme and found the communication overhead (total data exchanged for all servers) to be 47 kB for $t = 1$, 3.4 MB for $t = 2$, 220 MB for $t = 3$, and unacceptable for larger thresholds, at least in her implementation. Unfortunately, it seems that in order to ensure that the probability that the threshold is exceeded is reasonably small in a real-world system, using realistic assumptions about failure rates, the value of t must be 6 or greater [Rod]. Hence, to be practical, it seems the protocol must have subexponential complexity, regardless of optimizations we might be able to apply to this exponential scheme. Our protocol requires $O(n^4)$ bytes of network traffic with reasonable constant factors. The thesis contains an analysis of performance of various schemes in Chapter 9.

2.3.4 MPSS — Our Scheme

Our approach uses a simple Feldman VSS, and the technique for generating new shares is based on the one of Herzberg et al [HJKY95]. However, our protocol as-

sumes a much weaker (asynchronous) network and allows the group to change. When the group changes, it is able to handle a threshold of up to t Byzantine shareholders in the old group and an additional threshold of t Byzantine servers in the new group. Furthermore, unlike the scheme of [WWW02], we achieve worst-case polynomial communication complexity, and moreover our protocol has low overhead in the optimistic case where there are no failures. Our protocol makes use of accusations as part of choosing the new shares, as does Herzberg et al. However Herzberg et al. make use of an accusations/defense phase, which require extra interaction that is undesirable in the asynchronous setting. In particular, when servers receive invalid messages, they must accuse the sender, and if the sender is honest it must broadcast a defense to prove that the accusation is specious. But if message delays can be arbitrary, it is impossible to ensure that all accusations and all defenses from honest parties have been received, and hence we cannot tell which servers are misbehaving. Our protocol does not require accusations, but as an optimization, this thesis presents an optional extension called *verifiable accusations*. Unlike Herzberg et al.'s accusations, verifiable accusations require no defense phase, as any party can determine the validity of the accusation.

Chapter 3

Network Model and Assumptions

In this section we discuss how we model the network and the power of the adversary, as well as the cryptographic assumptions our protocols require. Briefly, we assume that the network is asynchronous and under the control of the adversary. The adversary may adaptively corrupt a limited number of nodes over a certain period we call an *epoch*, thereby learning all of their stored information and causing them to behave arbitrarily badly.

We assume that once corrupted, a node remains corrupted forever. This is reasonable because once the adversary knows a node's secret keys, it would be unsafe to consider that node recovered without changing its keys, in which case it would effectively be a different node.

3.1 Network Assumptions

We assume some number n of nodes $S_1, \ldots, S_n$, each of which represents a server on the network. In a multiparty protocol, these parties send messages to one another.

We assume messages are peer-to-peer; there is no broadcast channel. However, the network is asynchronous and unreliable: messages may be lost or delivered out of order. To capture the worst possible network behavior, we assume an intelligent adversary controls all communication in the network: it receives all messages, and determines what messages are to be delivered, and in what order. Furthermore, messages may appear that were not purposely sent by any party. To be more precise, we imagine the following:

Definition 3.1 ***Asynchronous Network Model.***

- *Communication takes place via messages sent from one node to another.*
- *The adversary decides what messages get delivered and when.*
- *In particular, the adversary may deliver messages out of order or not at all, and may also create and deliver arbitrary new messages.*

These assumptions allow the adversary, for instance, to permanently prevent any messages from getting through to their recipients, which will allow the adversary to effectively shut down any protocol.

Our protocol is *correct* given only these assumptions (and the usual theoretical assumption that the adversary has a bounded amount of computation available for constructing spurious messages). However, our protocol will not necessarily *terminate* given only these assumptions. In order to prove that *any* protocol terminates, we will have to assume at least that message delivery is not completely disrupted. The property we require to ensure termination is called *strong eventual delivery.*

Definition 3.2 ***Strong Eventual Delivery.*** *The maximum delay in message delivery for messages* repeatedly *sent from uncorrupted nodes is bounded (with some*

unknown bound), and while that bound can change over time, it does not increase exponentially indefinitely.

This definition does not include any mention of messages that are lost entirely. This is because in practice, lost messages are eventually retransmitted by the sender, so losses can be modeled as delays.

The basis for our assumption is the BFT agreement protocol [CL02], which requires strong eventual delivery in order to guarantee termination. We invoke BFT as a subprotocol, so we assume what BFT assumes.

This type of restriction applies to any asynchronous network protocol. In particular, it has been proven in [FLP82] that it is impossible to ensure both safety (informally, "bad things don't happen," e.g., the protocol does not fail or produce incorrect results) and liveness (informally, "good things eventually happen," i.e., the protocol eventually terminates) at the same time without making a synchrony assumption. Our protocol always provides correctness, but we require a relatively weak synchrony assumption to guarantee termination. This is in contrast to fundamentally *synchronous* protocols such as Jarecki's [HJKY95] that assume that the network is reliable, and fail if certain synchrony assumptions are not met.

We call the requisite property for our protocol *strong* eventual delivery to contrast it with ordinary eventual delivery, which is the property assumed by other proactive secret sharing protocols such as the one of Zhou et al. [ZSvR05].

Definition 3.3 ***Eventual delivery***. *Messages* repeatedly *sent from uncorrupted nodes will be delivered in a finite amount of time.*

This definition is weaker and hence one might think that it is more satisfactory than strong eventual delivery. However, a system that satisfies eventual delivery but not

strong eventual delivery has message delays that increase exponentially over time and indefinitely. Such a system would not be usable in practice.

3.2 Adversary Assumptions

We assume the existence of a powerful active adversary who observes all network traffic and can corrupt any node in the network at any time (although with some overall restriction on the number of such corruptions). When the adversary corrupts a node, all secret information held by that node becomes known to the adversary, and in particular, the adversary is able to forge messages from that node and decrypt messages sent to it. Furthermore, since the adversary sees all network traffic, it can attempt to use secret state stored on corrupted nodes to decrypt messages sent to those nodes before the corruption took place. Corrupted nodes no longer send messages according to the protocol, so such a node might, for instance, send messages to some parties but not others, participate in only particular phases of the protocol, or send out bad information. Once a node is corrupted, it remains corrupted forever. Note that the adversary is *adaptive*, that is, it can make decisions about which nodes to corrupt at any time, based on information available to it from past corruptions and from snooping network traffic.

Note that unlike some other protocols such as that of Wong et al. [WWW02], we do not assume that the new group of shareholders is initially uncorrupted, nor do we assume that all of the new shareholders will remain uncorrupted during the execution of the resharing protocol. Since the machines participating in a resharing protocol are necessarily connected to the network, allowing corruptions in both the new and old groups is important. We should not assume that any particular networked computer will be invulnerable to corruption for any amount of time, e.g., the amount of time

required to execute the resharing protocol.

3.3 Cryptographic Assumptions

We assume certain cryptographic tools are secure. Specifically, we use the following primitives:

- A secure cryptographic hash function H that is resistant to second preimage attacks. In particular, given a message m_1, it should be computationally infeasible to find a message m_2 such that $H(m_1) = H(m_2)$.

- Feldman's verifiable secret sharing (VSS) scheme [Fel87]. (Note that we can alternatively use Pedersen's VSS scheme [Ped91a] with only minor modifications, despite the fact that the version of our protocol presented here uses Feldman's scheme.) These schemes are computationally secure under the discrete logarithm assumption.

- A forward-secure encryption scheme secure against chosen-ciphertext attacks, such as the scheme of Canetti, Halevi, and Katz [CHK03]. This scheme is secure under the bilinear Diffie-Hellman (BDH) assumption [BF01].

- A forward-secure signature scheme. In [Kra00], Krawczyk shows how an unforgeable forward-secure signature scheme can be constructed from any unforgeable signature scheme.

Forward-secure signatures and Feldman secret sharing exist under the discrete logarithm assumption in the random oracle model, which is implied by the bilinear Diffie-Hellman assumption. In addition we need to limit the power of the adversary:

since we need computational assumptions for our cryptographic primitives, we need to assume the adversary operates in probabilistic polynomial time (that is, that each decision it makes is decided efficiently).

Our verifiable accusation scheme, which is an optional extension to our protocol that improves performance, requires an identity-based forward-secure encryption scheme. We describe such a scheme, which is secure under the bilinear Diffie-Hellman assumption [BF01], in Section 6.1.4. For verifiable accusations, the scheme must also be secure against a *fake key attack* (see Definition 3.4). Informally, this says that an adversary who knows the private key cannot generate a "bad" private key that works for certain messages but not others. In the scheme we present in Section 6.1.4, it is infeasible for an adversary who knows the private key to generate a second key at all.

Definition 3.4 ***Security against fake key attacks.*** *A public key cryptosystem (E, D) is secure against fake key attacks if, given a properly-generated key pair (PK, SK) and a ciphertext $C = E_{PK}(m)$, it is computationally infeasible to generate a* fake key *SK' such that $D_{SK'}(E_{PK}(C)) \neq m$ with non-negligible probability, but for a random m', $D_{SK'}(E_{PK}(m') \neq m'$ with negligible probability.*

3.4 Epochs and Limitation of Corruptions

System execution consists of a sequence of "epochs," each of which has an associated epoch number. In an epoch e a particular set of nodes, U_e, is responsible for holding the shares of the secret, and we assume that the adversary can corrupt no more than a threshold t of the servers in U_e before the end of epoch e. For our scheme, $n = |U_e| = 3t + 1$, which is optimal for an asynchronous protocol. In practice, the

scheme works with any $n \geq 3t + 1$, but $n > 3t + 1$ does not improve reliability and only decreases efficiency.

We would like to think of an epoch as a fixed amount of time, for instance, a day. Given this, the assumption is quite reasonable: although an adversary may be powerful, we assume it takes enough effort to corrupt a node that the adversary cannot corrupt too many nodes too quickly. Unfortunately, in an asynchronous network, epochs cannot be defined in terms of a window of absolute time relative to an absolute clock (or even bounded by one): rather, they must be defined in terms of events in the protocol.

We first define epochs locally. Each node in the system has internal events that define the end of an epoch for that party. Specifically, each party has forward-secure signature and encryption keys based on epochs, which it discards (along with its secret shares if it has any) when it moves to the next epoch. We say a party is *in epoch e* if it has its epoch e keys but no older keys. Definition 3.5 captures this idea more formally. Note that the notion of a local epoch is only well-defined with respect to non-faulty servers, as the state of Byzantine-faulty servers could be arbitrary.

Definition 3.5 ***Local epochs.*** *A non-faulty server is in* local epoch e *if it has secret shares or secret keys associated with epoch e. It leaves epoch e when it wipes all such information from its memory and is no longer able to recover it.*

Globally, for any epoch e, the total number of nodes in U_e that the adversary may corrupt *while they are in a local epoch* $e' \leq e$ is no more than t. Recall that we assume that corrupted nodes remain corrupted, thus, if a node is corrupted in epoch $e' \leq e$ it is also corrupted in epoch e.

3.5 Our Epoch Definition and the Importance of Forward-Secure Signatures

It is tempting to define an epoch more simply than we have defined it. We could limit the adversary to corrupting no more than t nodes up to the point at which all honest nodes in U_e have left epoch e, as is done by Zhou et al. [ZSvR05]. We call this the global epoch definition.

Definition 3.6 ***Global epochs.*** *A* global epoch *e is the spam of time starting when the first honest server enters local epoch e and ending when the last honest server has left local epoch e.*

Indeed, since our protocol is guaranteed to terminate given our network assumptions (Section 3.1) and limitations on corruption, this time period is finite, so the simpler definition is tempting. However, Definition 3.6 is less desirable than Definition 3.5 because it allows a relatively easy attack where an adversary isolates a node, prevents it from leaving an epoch, and uses the extra time to corrupt other group members. Our approach requires forward-secure signing to prevent group members that are corrupted after they leave the epoch from fooling the isolated node into revealing its secret.

For the sake of refutation, suppose we implemented our system with ordinary signatures, rather than forward-secure ones. We show a specific attack, which we call an *isolation attack*, that may occur when the adversary is restricted by local epochs but is ruled out when the adversary is restricted by the weaker global epoch definition.

Definition 3.7 ***Isolation attack.*** *Let $U_e = \{S_1, \ldots, S_{3t+1}\}$ be the set of shareholders in epoch e. The attack is as follows.*

- *During epoch e, the adversary corrupts S_1 through S_t, learning t shares. The adversary also prevents honest server S_{3t+1} from communicating (which, in the real world, amounts to a simple denial of service attack against S_{3t+1}.)*

- *The other $3t$ replicas execute the protocol without S_{3t+1}, and the transfer to epoch $e+1$ completes. The non-faulty servers in epoch e discard their shares, except for S_{3t+1} which is still unable to communicate.*

- *Subsequently, the adversary corrupts t additional servers in U_e. This is allowed because the adversary is only restricted to corrupt t servers in local epoch e; after epoch e has ended locally and those servers have discarded their shares, it could potentially corrupt all of the servers in U_e given enough time.*

- *S_{3t+1}, in collaboration with the $2t$ corrupted servers, now executes another instance of the protocol, but since there are now $2t$ corruptions, the adversary can ensure that all of the proposals that are selected (see Section 4.2.1) are known to it.*

- *S_{3t+1} generates share transfer messages for epoch $e+1$ based on the bogus information supplied by the $2t$ faulty servers, and these transfer messages reveal S_{3t+1}'s secret share. Since the adversary corrupted S_1 through S_t prior to the end of epoch e, it had t shares already; hence, with the addition of S_{3t+1}'s share, it now has $t+1$ shares. Thus the adversary learns the secret.*

Note that the isolation attack just described is only possible because nodes that were corrupted after the end of epoch e behave as though they are still in epoch e. Forward-secure signatures prevent this attack. With forward-secure signatures, nodes that are uncorrupted when they exit epoch e locally evolve their signature keys

so that they are no longer capable of signing new messages associated with epoch e. Thus, if they later become corrupted, the adversary cannot use them to coerce isolated servers still stuck in epoch e into revealing their shares.

This attack is prevented *by assumption* if we use the global epoch definition. Using this weaker definition, it would not be legitimate for the adversary to corrupt any additional nodes while node S_{3t+1} hasn't finished epoch e. Essentially, the global epoch definition places additional restrictions on the adversary model by extending the lifetime of each epoch e as long as there is any correct but isolated server in epoch e. As far as we are aware, the need for forward-secure signatures in proactive secret sharing schemes in asynchronous networks has not been observed because most prior work (e.g., [ZSvR05, WWW02]) has used the global epoch definition, or swept the entire issue under the rug by assuming more abstract primitives such as "secure channels" [HJKY95].

3.6 Pragmatics

The above definition will allow us to prove that the adversary is unable to learn the secret, if more than t shares are needed to do this. However it leaves open the question of how to ensure this condition in practice.

We assume the system has a "user" who is responsible for making a number of decisions including (1) the duration of epochs, (2) the threshold t for each epoch, (3) the membership of the group in each epoch, (4) and the membership of nodes in the system. All these decisions may be made by a person, or some may be made automatically by a management system.

Group membership is an especially important decision, because nodes that have been in the system for a long time are vulnerable to attack by the adversary over that

entire time period. One way to minimize the risk of corrupted nodes is to choose members of the next group to be "fresh" nodes that have joined the system only recently. An alternative is to keep group membership secret until epoch e is about to begin, to prevent the adversary from targeting new group members in advance.

The user also needs to decide when to move to the next epoch. This decision is based on assumptions about the threshold, the vulnerability of the old group, and the time it will take to make the transition to the new epoch. Note that the transition can be slow because of network delays that prevent the old nodes from learning that it is time to end the current epoch and that further prevent the share transfer from completing.

In practice it isn't difficult for the user to decide what to do. For example, the user might choose a system configuration in which nodes are located in secure, failure-independent locations, so that the probability of more than t failures in a group in a 24-hour period is acceptably low for some relatively small value of t, e.g., $t = 3$. Then the user might choose to start a new epoch every 12 hours because this is likely to allow the transition to the next epoch to complete before the old group has been in operation for a day.

A potential objection to the way we have modeled the adversary is that we limit the number of nodes the adversary can corrupt per epoch, yet network delays can cause an epoch to last arbitrarily long. One could argue that a more faithful model of the real world would have an adversary who corrupts nodes at a fixed rate with respect to real time. We counter that in the real world, widespread network outages tend to be relatively short in duration. Hence, it is overwhelmingly likely that nodes will be able to communicate well enough to complete the protocol in a reasonable amount of time. In particular, $2t + 1$ honest nodes in the old group and $2t + 1$ honest nodes in the new group must be able to communicate; long-term

local network outages at the remaining nodes do not pose a problem because we use forward-secure signatures and our system is secure under the local epoch definition (see Definition 3.5).

Chapter 4

Our Secret Redistribution Scheme

This section describes our redistribution scheme for the case when the old and new group have the same failure threshold t. We explain how to extend the scheme to allow the threshold to change in Section 7. Mobile proactive secret sharing consists of the redistribution protocol along with a protocol for the initial sharing. We focus on the redistribution protocol, deferring the details of the relatively uninteresting protocol for the initial sharing to Section 5.

We begin by summarizing the secret sharing and verifiable secret sharing schemes we use. Then we explain our redistribution technique. The exact details of how nodes communicate to carry out redistribution are given in Section 5.

4.1 Preliminaries

Here we sketch some of the tools we use in our scheme, but omit a discussion of components we treat as "black boxes" such as forward secure encryption [CHK03] and signatures [Kra00]. Section 3.3 enumerates the cryptographic primitives we

require and refers to the papers that establish their correctness. We use Feldman's verifiable secret sharing scheme [Fel87], based on Shamir secret sharing [Sha79], to encode the secret. The method we use for deriving a new sharing of the secret is similar to the one proposed by Herzberg et al. in [HJKY95].

4.1.1 Shamir's Secret Sharing Scheme

Here we describe Shamir's secret sharing scheme. All arithmetic in this scheme is carried out with respect to some finite field $\mathbf{F}_q$, where q is a large prime and a public parameter to the system. We use s to denote the secret and t to denote the desired threshold. In order to distribute shares of s so that any group of size $t+1$ can reconstruct s from their shares, while any group of size t or smaller learns nothing, we use a large prime number q as a parameter. Random values $p_1, \ldots, p_t \in \mathbf{F}_q$ are generated such that $p_t \neq 0$ and P is defined to be the polynomial $P(x) = s + \sum_{i=1}^{t} p_i x^i$.

Each party is assigned an id number i different from 0; party i's share is $P(i)$. With $t+1$ points on P, we can recover P by Lagrange interpolation and thus learn $P(0) = s$. However, with only t points, there is still a degree of freedom left; therefore, the secret may be any value.

4.1.2 Verifiable Secret Sharing

In a secret sharing scheme, players receive shares and must trust that their shares are correct. In a *verifiable* secret sharing scheme (VSS), additional information is given that allows each player to check whether or not its share is correct.

To summarize, in Feldman's verifiable secret sharing (VSS) scheme, the secret is shared using Shamir's secret sharing. For the verification information, a cyclic

group G of order q and a generator g for G must be given as a system parameter.[1] The security of Feldman's scheme is predicated on the discrete logarithm assumption (DLA). Briefly, DLA is the generally accepted assumption that given a generator g for some finite cyclic group and some number c in that group, it is computationally infeasible to find an x such that $g^x = c$ (i.e., it is hard to compute $\log_g c$.) Technically speaking, it may be possible to compute particular bits (e.g., the low-order bit) of the discrete logarithm [PH78], but it is well known how to circumvent this difficulty via padding such that the bits of s representing the actual secret are hard-core. In particular, Long and Wigderson [LW88] show that the high-order bits are secure.

When the Shamir polynomial P is generated, "commitments" are given for each coefficient: $c_0 = g^{p_0} = g^s$, and generally $c_i = g^{p_i}$. Now note that each party can compute

$$g^{P(i)} = \prod_{j=0}^{t} {c_j}^{i^j},$$

and thus check if the share s_i it receives is equal to $P(i)$ by checking if $g^{s_i} = g^{P(i)}$. (In cyclic group G with generator g and elements x and y, $g^x = g^y$ if and only if $x = y$.) Furthermore, if the discrete logarithm assumption is true, no party i can learn another party's share $P(j)$ from the commitment $g^{P(j)}$.

In this thesis we use Feldman's scheme for concreteness and simplicity of presentation, but note that we could just as easily have used another VSS scheme such as Pedersen's scheme[Ped91b]. Therefore, we do not concern ourselves with the theoretical differences between these schemes, in particular the fact that Feldman's scheme is perfectly binding and computationally hiding, whereas Pedersen's scheme

[1]This may be accomplished, for instance, by choosing q such that $p = 2q + 1$ is also prime, and letting G be the subgroup of $\mathbb{Z}_p^*$ of elements of order dividing q. But any such q, G, and g will do.

is perfectly hiding and computationally binding. In any case, it is easy to show that a computationally unbounded attacker can exploit the VSS to expose the secret regardless of which choice we make.

4.1.3 Herzberg et al.'s Proactive Secret Sharing

The formula by which a new sharing is derived from the old one in our scheme is based on the proactive secret sharing scheme of Herzberg et al. [HJKY95], presented in more detail in S. Jarecki's master's thesis [Jar95]. In this scheme, a new set of shares is generated from the old Shamir share polynomial P by choosing a random polynomial Q such that $Q(0) = 0$, that is,

$$Q(x) = 0 + q_1 x + q_2 x^2 + \cdots + q_f x^f$$

where the coefficients q_i are random. Node i's new share is then $P'(i) = P(i) + Q(i)$. This is still a valid sharing of the same secret since P' is a randomly chosen polynomial with constant coefficient s, but all the other coefficients are random and independent of the coefficients of P.

Herzberg et al.'s scheme also includes a share recovery protocol to allow a previously faulty node j to recover its share. Share recovery for j is done by choosing a random polynomial R_j such that $R_j(j) = 0$. Each other node i sends $P(i) + R_j(i)$ to node j. Node j then reconstructs the polynomial $P + R_j$ by Lagrange interpolation and evaluates it at j to obtain its share $P(j) + R_j(j) = P(j)$.

The polynomials Q and R_j are generated in a distributed fashion in such a way that each participant i gets $Q(i)$ and each $R_j(i)$, but learns no other information about Q or the R_js. This generation process involves each node constructing *proposal* polynomials and sending out points on these polynomials. Then the nodes

execute an agreement protocol to determine which proposals are valid. The Q and R_j polynomials are the sum of the valid proposals. Our protocol generates these polynomials in a similar way, but using a different agreement protocol and with additional considerations to handle changing the group membership; see Sections 4.2.3 and 5.3.

4.1.4 Group Size

Until now, we have discussed resharing schemes with respect to two parameters: t, the maximum number of shares that may be revealed without exposing the secret, and n, the total number of shareholders. However, we have not yet discussed the relationship between t and n.

Herzberg et al.'s scheme assumes an active adversary that may attempt to cheat in order to learn the secret, construct an invalid resharing, or prevent the secret from being reconstructed. Up to t servers may be corrupted, and the faulty servers may simply refuse to participate in the protocol. Herzberg et al.'s scheme is based on Shamir's secret sharing scheme, in which at least $t + 1$ nodes are required to reconstruct the secret (or else the t faulty shareholders could reconstruct). Hence, Herzberg et al. require $n \geq 2t+1$ to ensure that at least $t+1$ non-faulty shareholders are always available.

Our scheme additionally assumes that the network is asynchronous and unreliable, that is, that messages may be delivered after some unknown delay or lost entirely. Thus, if we do not hear from a particular server, we cannot tell whether that server is faulty, or whether its response has been delayed by the network. We require $n \geq 3t + 1$, with the following justification. Out of $3t + 1$ shareholders, the protocol can only wait to hear from $2t + 1$, as the remaining t servers may be slow

or faulty and we cannot tell which is the case. Of the $2t + 1$ that respond, t may be faulty, leaving us with the required minimum of $t + 1$ to reconstruct the secret. In fact, $n = 3t + 1$ is the minimum n for any protocol that requires Byzantine agreement in an asynchronous network [CL02]. This issue is further clarified in Chapter 5 where proposal selection and agreement are discussed, and revisited in Chapter 7, which describes how the threshold can be changed between successive epochs. In this chapter, we will simply assume there are $n = 3t + 1$ old shareholders and $n = 3t + 1$ new shareholders.

4.2 Secret Redistribution

We want to generate new shares for the same secret and move the new shares to a new group of nodes, which may be completely disjoint from the old ones. Since the attacker can corrupt up to t in a group of nodes, in this system it can control $2t$ nodes between the two groups.

An important point is that Herzberg et al.'s resharing scheme, unmodified, is insecure when applied to secret redistribution to a new group because it leaks information about the new shares to the old group. This is because each member i of the old group gets a point $Q(i)$ so that it can compute $P'(i) = P(i) + Q(i)$ and send it to the new group. Alternatively, we might imagine that $P(i) + Q(i)$ is computed in the new group, but this would result in the new group learning old shares. In either case, the result is that the adversary potentially learns up to $2t$ new shares. For instance, suppose that $t = 1$ and that node 2 is corrupted in the old group and node 3 is corrupted in the old group. The adversary learns $P'(2)$ and $P'(3)$ and can recover the secret.

Our protocol has two groups, an old group of shareholders and a new group,

whereas Herzberg et al. have only one group that never changes. As we explain in the following section, one of the requirements of our protocol is that the members of the old and new groups have distinct sets of identifiers, provided that they are distinct nodes. Let S_i refer to the i^{th} member of the old group and T_k refer to the k^{th} member of the new group. Formally, we call the set of identifiers for the old group $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ and the set of identifiers for the new group $\{\beta_1, \beta_2, \ldots, \beta_n\}$. Hence, α_i is the identifier of node S_i and β_k is the identifier for T_k, and we have the restriction that for all i and k, $\alpha_i \neq \beta_k$ unless S_i is the same machine as T_k. This notation allows us to refer to both the old and new group members using the indices 1 through n, even though their identifier sets are disjoint and need not follow any particular pattern. Hence, in describing our protocol, instead of writing (for example) S_i's share as $P(i)$ as in Section 4.1.3, we would write $P(\alpha_i)$.

4.2.1 Generating New Shares

Our solution, roughly speaking, is to combine the resharing and share recovery into a single step. Instead of computing $P(\alpha_k) + Q(\alpha_k)$ in the old group and sending it to T_k, each old node S_i computes $P(\alpha_i) + \big(Q(\alpha_i) + R_j(\alpha_i)\big)$ and sends this point to T_k. Upon receiving at least $t+1$ such points, T_k can interpolate to obtain the polynomial $P + Q + R_k$, then evaluate this polynomial at β_k to obtain

$$P(\beta_k) + Q(\beta_k) + R_j(\beta_k) = P(\beta_k) + Q(\beta_k) = P'(\beta_k).$$

Since R_j is random everywhere except at β_k, this polynomial provides the new node T_k no additional knowledge except $P'(\beta_k)$. Furthermore, the old nodes learn nothing about the new share $P'(\beta_k)$ because each old node only knows a single point on any given polynomial $P + Q + R_k$, and this polynomial is random and independent of P'

except at β_k. (Recall that we ensure that $\alpha_i \neq \beta_k$ for any i, k by requiring that the old and new nodes use disjoint sets of identifiers.)

The difficulty here is in generating these polynomials Q and R_i in the old group so that no node knows too much about them; in particular, each old node S_j should learn only $Q(\alpha_j) + R_i(\alpha_j)$ for all i. If a node were capable of learning additional points, an adversary could accumulate $t + 1$ points and interpolate $Q + R_i$. Then a faulty S_i could learn share $P'(\alpha_i)$ intended for the new group. Furthermore, nodes must not be able to learn points on Q individually, because t collaborating faulty old shareholders could take their t points along with the well-known point $(0, 0)$ to interpolate Q. They could then add $Q(\alpha_i)$ to their old shares $P(\alpha_i)$ to obtain points on P', which are only supposed to be known only to new shareholders.

To this end, it is crucial that no old shareholder and new shareholder have the same identifier. Otherwise S_i would be able to learn $Q(\alpha_j) = Q(\alpha_j) + R_i(\alpha_j)$ and compute $P'(\alpha_j) = P(\alpha_j)+Q(\alpha_j)$. This point on P', combined with the t other points on P' that the adversary learns from corrupting t nodes in the new group, suffice for the adversary to recover P' and hence the secret. Furthermore, clearly no server can have id 0, since this would imply that its share of the secret is identical to the secret.

In practice, it is easy to ensure that indices between the old and new groups are distinct and nonzero. One way is to give each server an identifier that is unique across the entire system. This identifier might simply be a cryptographic hash of the node's public key, for instance.

4.2.2 Protocol Sketch

In this section, we give an overview of what information is transmitted between the old and new groups and how the new shares are constructed and verified. We defer

the details of the protocol to Section 5. Here, S_i refers to a member of the old group and T_k refers to one of the new shareholders. As a matter of convention, the index i denotes a sender in the old group, j denotes a recipient in the new group, and k denotes a recipient in the new group. Roughly, our protocol proceeds as follows.

1. **Proposal Selection.** Each party S_i creates a random polynomial Q_i modulo q such that $Q_i(0) = 0$, and for each T_k in the new group, it creates a random polynomial $R_{i,k}$ modulo q such that $R_{i,k}(\beta_k) = 0$.[2] S_i creates Feldman VSS commitments for the coefficients of each polynomial, and sends this out along with its points on the polynomials $Q_i + R_{i,k}$, for each k, to every other S_j in the old group. The information about the points is encrypted for its recipient, and this list of encryptions, along with the verification information, is signed by S_i as S_i's *proposal*.

   A node receiving a proposal determines whether or not the points sent to it are consistent with the verification information: if they are, the node will approve of the proposal. In particular, each recipient verifies that the points it received are consistent with the Feldman commitments, and that these are commitments to polynomials with the appropriate properties, i.e., $Q_i(0) = 0$ and $R_{i,k}(\beta_k) = 0$. The mechanics of this verification procedure are discussed below.

2. **Agreement.** The old nodes reach an agreement about a set of proposals, each proposal from a different sender, and each approved by at least $2t+1$ nodes. Let $\mathcal{S}$ be the set of i such that server i's proposal was one of the agreed-upon ones. $Q(x)$ is defined to be $\sum_{i \in \mathcal{S}} Q_i(x)$, and $R_k(x)$ is defined to be $\sum_{i \in \mathcal{S}} R_{i,k}(x)$.

[2] A simple way to generate such an $R_{i,k}$ is to construct a polynomial $R'_{i,k}(x) = r'_{i,k,0} + r'_{i,k,1}x + \cdots + r'_{i,k,t}x^t$ with random coefficients $r'_{i,k,0}, \ldots, r'_{i,k,t}$ and let $R_{i,k}(x) = R'_{i,k}(x) - R'_{i,k}(\beta_k)$.

3. **Transfer.** Each old node i that knows its secret share and approves of all proposals in the set sends to each new node k the value $P(\alpha_i)+Q(\alpha_i)+R_k(\alpha_i)$. This is calculated by adding node S_i's share, $P(\alpha_i)$, to the sum over $S_j \in \mathcal{S}$ of $Q_j(\alpha_i) + R_{j,k}(\alpha_i)$, which are points S_i received from other nodes. Along with the share, the old node also sends the verification information for all the polynomials in the proposal set, as well as the verification information for P.

   The new nodes check that the points sent to them are properly generated. To perform this check, each new node T_k needs Feldman commitments to the old share polynomial P, as well as commitments to Q and R_k. The details of how this information is generated in each subsequent epoch is discussed in Section 4.2.3.3. Once T_k has $t+1$ properly generated points for the same set of proposals, it reconstructs the polynomial $P+Q+R_k$ and evaluates it at β_k to obtain its share.

4.2.3 Verifying Proposals

4.2.3.1 Goals of Verification

The first two steps of the protocol outlined above are intended to produce $n+1$ polynomials $Q, R_1, R_2, \ldots, R_n$ with particular properties. In particular, we must have the following properties, despite nodes that attempt to cheat:

$$\begin{aligned} Q(0) &= 0 \\ \forall k \quad R_k(\beta_k) &= 0 \end{aligned} \tag{4.1}$$

However, any given node S_j in the old group shouldn't know anything about Q or any R_k. S_j has just one point on each of the sum polynomials

$$\begin{gathered} Q(\alpha_j) + R_1(\alpha_j) \\ Q(\alpha_j) + R_2(\alpha_j) \\ \vdots \\ Q(\alpha_j) + R_n(\alpha_j) \end{gathered}$$

To ensure that the required properties hold for Q and all R_k, we ensure that these properties hold for all of the contributions in the agreed-upon set $\mathcal{S}$ in step 2, i.e.,

$$\forall i \in \mathcal{S} \qquad \begin{cases} Q_i(0) & = & 0 \\ \forall k \quad R_{i,k}(\beta_k) & = & 0 \end{cases} \tag{4.2}$$

Since, for instance, Q is a sum of the chosen Q_is, if all of the Q_is have a zero at 0, then so does Q. (A further point is that if the unconstrained coefficients of at least one of the Q_is is random and independent of the other polynomials, then Q will be random and independent of the other Q_is. But this property is guaranteed assuming at most t failures and $|\mathcal{S}| > f$, so we need not attempt to verify the randomness of the proposals received.) Thus, the problem is reduced to arranging so that each S_j can verify that each S_i's proposal is properly formed, i.e., S_j can determine whether the proposal satisfies the equations above. As we discuss in chapter 5, the agreement in the second step of the protocol outlined above is carried out in such a way that only nodes that sent well-formed proposals to $2t+1$ nodes (possibly including themselves) are included in $\mathcal{S}$.

4.2.3.2 Mechanics

In the first step of the protocol, each old node S_i produces a set of polynomials

$$\begin{aligned}
Q_i(x) &= q_{i,1}x + q_{i,2}x^2 + \cdots + q_{i,t}x^t \\
R_{i,1}(x) &= r_{i,1,0} + r_{i,1,1}x + r_{i,1,2}x^2 + \cdots + r_{i,1,t}x^t \\
&\vdots \\
R_{i,n}(x) &= r_{i,n,0} + r_{i,n,1}x + r_{i,n,2}x^2 + \cdots + r_{i,n,t}x^t.
\end{aligned}$$

Then S_i sends to S_j the points $\langle p_1, p_2, \ldots, p_n \rangle$, where $p_k = Q_i(j) + R_{i,k}(j)$, along with Feldman-style commitments to each of the constituent polynomials. The commitment matrix generated by S_i has the form

$$\begin{pmatrix}
 & g^{q_{i,1}} & g^{q_{i,2}} & \ldots & g^{q_{i,t}} \\
g^{r_{i,1,0}} & g^{r_{i,1,1}} & g^{r_{i,1,2}} & \ldots & g^{r_{i,1,t}} \\
 & & \vdots & & \\
g^{r_{i,n,0}} & g^{r_{i,n,1}} & g^{r_{i,n,2}} & \ldots & g^{r_{i,n,t}}
\end{pmatrix}.$$

S_j now has to verify two things: first, that S_i's proposal (namely Q_i and all the $R_{i,k}$) satisfy equations 4.2 according to the commitments provided, and second, that the points S_i received are consistent with those commitments. The first property ensures that S_i generated its polynomials correctly, and the second property is used to verify that S_i is presenting points on the same polynomials to all the other nodes in the group.

As in Feldman's scheme [Fel87], we apply the homomorphic properties of exponentiation to verify these properties. For the first property we have:

$$\begin{aligned} 0 = R_{i,k}(\beta_k) \quad \Longleftrightarrow \quad g^0 &= g^{R_{i,k}(\beta_k)} \\ &= g^{\sum_{l=1}^{t} r_{i,k,t} {\beta_k}^l} \\ &= \prod_{l=1}^{t} \left(g^{r_{i,k,t}}\right)^{{\beta_k}^l} \end{aligned} \tag{4.3}$$

Similarly, after some simple algebra, we derive the check equation for the second property:

$$\begin{aligned} p_k = Q_i(\alpha_j) + R_{i,k}(\alpha_j) \quad \Longleftrightarrow \quad g^{p_k} &= g^{Q_i(\alpha_j)} g^{R_{i,k}(\alpha_j)} \\ &= \prod_{l=0}^{t} \left(g^{q_{i,t}} g^{r_{i,k,t}}\right)^{{\alpha_j}^l} \end{aligned} \tag{4.4}$$

Of course, the $g^{q_{i,t}}$ and the $g^{r_{i,k,t}}$ values are not computed by S_j; rather, they are provided as part of S_i's commitment matrix. Note also that $g^{q_{i,0}}$ isn't part of the matrix provided by S_i. This is because we require that $Q_i(0) = 0$, so $g^{q_{i,0}}$ must be 1.

4.2.3.3 Computing Verification Information for the New Secret

As mentioned in Section 4.2.1, each node T_k in the new group needs to obtain valid commitments to the old share polynomial P, as well as commitments to Q and R_k in order to determine which of the points it receives from the old group are valid. These commitments can be computed by each member of the old group and sent to T_k, and T_k will accept any $t+1$ identical sets of commitments as valid. (In practice, T_k might instead receive one copy of the commitments and t signatures that attest to the authenticity of that copy.)

Given a point $\nu_{j,k}$ received from S_j, T_k uses the commitments to check that

$$\nu_{j,k} = P(\alpha_j) + Q(\alpha_j) + R_k(\alpha_j).$$

The mathematical technique is the same as discussed in the previous section in the context of verifying proposals in the old group and not repeated here. Instead we describe how the commitments to P, Q, and R_k are generated within the old group.

Let

$$\begin{array}{lcccccccccc} P(x) & = & s & + & p_1 x & = & p_2 x^2 & + & \cdots & + & p_t x^t \\ Q(x) & = & & & q_1 x & + & q_2 x^2 & + & \cdots & + & q_t x^t \\ R_k(x) & = & r_{k,0} & + & r_{k,1} x & + & r_{k,2} x^2 & + & \cdots & + & r_{k,t} x^t \end{array}$$

Recall from step 2 of the protocol sketch that $Q(x) = \sum_{i \in \mathcal{S}} Q_i(x)$ and $R_k(x) = \sum_{i \in \mathcal{S}} R_{i,k}(x)$, where $\mathcal{S}$ is the set of agreed-upon proposals. Therefore, for $1 \leq l \leq t$ and $0 \leq m \leq t$ we have $q_l = \sum_{i \in \mathcal{S}} q_{i,l}$ and $r_{k,m} = \sum_{i \in \mathcal{S}} r_{i,k,m}$. Each member of the old group has commitments to the coefficients of each Q_i and $R_{i,k}$, namely, $g^{q_{i,1}}, \ldots, g^{q_{i,t}}$ and $g^{r_{i,k,0}}, \ldots, g^{r_{i,k,t}}$. We compute the commitments to Q and R_k, specifically $g^{q_1}, \ldots, g^{q_t}$ and $g^{r_{k,0}}, \ldots, g^{r_{k,t}}$, using the homomorphic properties of exponentiation:

$$\begin{array}{rcl} g^{q_l} & = & \prod_{i \in \mathcal{S}} g^{q_{i,l}} \\ g^{r_{k,m}} & = & \prod_{i \in \mathcal{S}} g^{r_{i,k,m}} \end{array}$$

As for the commitments to P, we assume that the dealer provides these in the first epoch. In subsequent epoch, commitments to the new share polynomial are computed from the commitments to the old share polynomial as well as the commitments to Q. Since $P'(x) = P(x) + Q(x)$, we have for all $1 \leq l \leq t$, $p'_l = p_l + q_l$. Therefore, $g^{p'_l} = g^{p_l} g^{q_l}$.

Chapter 5

The Redistribution Protocol

This section gives the details of the communication in the redistribution protocol described in Chapter 4. The protocol works in an asynchronous, unreliable network; nodes resend messages to ensure delivery.

There are three phases to the protocol: *proposal selection*, *agreement*, and *transfer*. Chapter 4 gave an overview of each of these phases and formulated the mathematics behind the construction and verification of new shares. In this section, we describe in detail what messages are exchanged and how the protocol proceeds.

All information sent in these protocols is encrypted for the recipient using a forward-secure encryption scheme so that an attacker cannot decrypt a message sent in epoch e without corrupting its recipient before the end of local epoch e (see Definition 3.5). In addition, messages are signed using forward-secure signatures. Improperly signed messages are discarded.

Our system makes use of BFT [CL02] to carry out agreement. As in BFT, at any moment one of the group members is the *coordinator* (also called the *primary*). The coordinator directs the activities of the group, e.g., it chooses the order of the

operations that clients ask the group to carry out. Some communication steps in which each node sends a different message to each other node that would otherwise be point-to-point are directed through the coordinator so that recipients have a consistent view of who has spoken so far.

However, we cannot assume that the coordinator is behaving correctly, as this would make it a single point of failure. To address this potential problem, the protocol proceeds in a series of *views*. All messages have an associated view number. The other nodes watch the coordinator, and if it is not behaving properly, they carry out a view change protocol that increments the view number, selects a different node to be the coordinator, and starts a new instance of the protocol using the new view number. Nodes are chosen to be the coordinator in subsequent views in a deterministic (round-robin) fashion, so that the attacker is unable to control this choice. Our protocol ensures that if the coordinator is behaving correctly as far as other nodes can tell, then the worst it can do is prevent the protocol from completing in the current view. Hence, nodes will initiate a view change after a timeout, and this timeout increases exponentially with each subsequent view (and hence the strong eventual delivery assumption, Definition 3.2). Details of the view change protocol, as well as further justification and proofs of correctness for this approach, can be found in [CL02] and we do not reproduce them here.

Note that epoch numbers are different from view numbers, although each message is labeled with both an epoch number and a view number. Each epoch is identified with a particular set of shareholders and a particular sharing of the secret s. Our share redistribution protocol is used to transition from one epoch to the next. Internally, the protocol proceeds through a sequence of views, each with an associated coordinator, until it selects a coordinator that behaves honestly and the protocol completes.

In BFT, there is a single Byzantine fault tolerant group whereas every time our protocol is run there are two groups of interest: the old shareholders (in epoch e) and the new shareholders (in epoch $e+1$). Each of these groups can perform agreement with up to their respective threshold number of faults in each group, so the execution of the protocol could be coordinated via either group in principle. In our protocol, the coordinator is always chosen to be a party in the old group because this is where most of the communication and all of the agreement takes place. Once the members of the old group have agreed to a set of valid proposals from each other, each honest node in the old group can independently send data to the members of the new group to allow the members of the new group to compute their shares, and no agreement operations are needed in the new group.

To run the protocol, each party needs to have a signature and encryption key pair to ensure that messages sent between parties are authentic, non-malleable, and secret. Both the signature scheme and the encryption scheme will need to be *forward-secure*. Specifically, we suggest the use of the Bellare-Miner forward-secure signature scheme [BM99] and the Canetti-Halevi-Katz forward-secure encryption scheme [CHK03]. Forward-secure encryption is needed because the adversary may save network messages from past epochs, then later break into more than the threshold number of nodes from a past epoch, revealing the state of those nodes, including encryption keys; forward-secure encryption schemes prevent the adversary from decrypting these past messages. Forward-secure signatures are needed to prevent corrupted nodes from sending authentic-looking messages to isolated but correct nodes that still believe they are in some past epoch. When we discuss our optional verifiable accusation scheme in section 6.1, we describe additional requirements on the encryption scheme if verifiable accusations are used. However, note that verifiable accusations are merely an optimization.

We assume the epoch number is e ("the current epoch"), and that each party is aware of the nodes in epoch e and $e+1$ ("the next epoch"), as well as the public encryption and signature keys of every node. We define t_e as the maximum number of faults tolerated in the epoch e and $n_e = 3t_e+1$ as the number of shareholders comprising epoch the epoch e. The set of shareholders for epoch e is $\{S_1, \ldots, S_{n_e}\}$, the shareholders for epoch $e+1$ is $\{T_1, \ldots, T_{n_{e+1}}\}$, and the unique identifiers associated with these nodes are $\{\alpha_1, \ldots, \alpha_{n_e}\}$ and $\{\beta_1, \ldots, \beta_{n_e}\}$, respectively. As shorthand, t and n refer to the current epoch e, that is, $t = t_e$ and $n = n_e$. We assume every message a node sends is signed under its signature key.

Messages are ignored if they aren't properly signed by the sender or have the wrong format. The latter check includes checking embedded messages (e.g., messages from the coordinator that contain $2t+1$ messages from other nodes): a message is ignored if any embedded messages aren't properly signed by their sender or are in the wrong format. Recipients also ignore messages that have incorrect view or epoch numbers.

5.1 BFT

Our protocols make use of Castro and Liskov's BFT algorithm [CL02], which provides asynchronous agreement with $t < \lfloor n/3 \rfloor$ Byzantine faults. In particular, given *proposals* (explained in Section 5.3.1) from at least $2t+1$ servers and a sufficient number of votes on those proposals, our redistribution protocols choose a subset of those proposals and invokes BFT so that all honest servers agree on which proposals to use. BFT ensures that if at least $2t+1$ servers (of whom at least $t+1$ are honest) agree on a set of proposals, then no honest server will complete the agreement protocol with a different set. This subsection briefly reviews the operation of BFT,

but for full details refer to Castro and Liskov [CL02].

BFT provides liveness (it terminates with $2t+1$ servers reaching agreement) and safety (honest servers that complete the protocol all agree on the same proposals) subject to $t < \lfloor n/3 \rfloor$ faults and the strong eventual delivery assumption described in Definition 3.2. Thus, secret redistribution requires these assumptions as well. As in our protocol, BFT uses a primary to coordinate the activities of the group, and the primary in BFT is the same server as the coordinator in our scheme.

BFT executes in three phases. In the first phase, the primary broadcasts a PRE-PREPARE message, which contains the value to be agreed upon—a set of votes, in our case. In the next phase, each server, upon receiving a PRE-PREPARE or PREPARE message, broadcasts a PREPARE to all other servers if it is willing to accept the proposed value. In [CL02], servers will accept the value if it is properly signed by a client, whereas in our system, servers accept a proposal set as valid if it contains the requisite number of signed proposals and votes on those proposals. Finally, upon receiving $2t+1$ PREPARE messages, each server broadcasts a COMMIT. The protocol commits locally at a server when that server receives $2t+1$ COMMIT messages.

If the primary is faulty, a view change protocol is executed, which starts the protocol over again with a new primary in a new view. Views are contained within epochs; BFT may progress through multiple views in attempt to reach agreement and transition from epoch e to epoch $e+1$. Any node may request a view change when it believes the primary to be faulty or reaches a timeout. If recipients of this message agree with the view change they send an acknowledgement to the new primary, which generates a view change certificate consisting of $t+1$ acknowledgements. When a node sends a request for a view change and the recipient has already seen a commit message from the primary, it forwards the primary's commit message to the requester.

Timeouts increase exponentially with each successive view to match the synchrony assumptions described in Chapter 3.

5.2 The Share and Recon protocols

For the sake of completeness, we give a Share protocol that can be used by a dealer to establish the initial sharing of the secret, and a Recon protocol that can be used by the shareholders to reconstruct the secret. Both of these protocols are quite straightforward and of relatively little practical interest, but they demonstrate that our MPSS is a valid secret sharing scheme. In practice, rather than reconstruct the secret, one would like to use it to carry out a secure multiparty computation, e.g., signing or decrypting messages [DF91, GJKR96, Lan95]. Our protocol uses a linear sharing, which these schemes typically assume.

In the Share protocol, we assume there is a special party, known as the dealer, that has the secret s as an input. The dealer generates the random polynomial P by generating random values $p_1, \ldots, p_t$ modulo q, and makes a list of shares, $P(i) \bmod q$ for each $1 \leq i \leq t$, as well as commitments $g^s, g^{p_1}, \ldots, g^{p_t}$ modulo q. The dealer encrypts each share $P(i)$ using encryption key PK_i, and the collection of shares is signed with the dealer's signing key. The dealer then broadcasts this complete list of encrypted shares, along with all the commitments, to every member S_i in the initial group. Nodes other than the dealer will continually request their share from all other nodes until they receive it. Nodes who have received the message from the dealer will forward it to any nodes that request it.

In the Recon protocol, every node sends its share to every other node. Each node receiving a share checks whether it is valid by using the Feldman commitment to that share. Once a node receives $t + 1$ valid shares, the node reconstructs the polynomial

P using Lagrange interpolation and computes the secret, namely, $P(0)$.

5.3 The Redist_0 protocol

This section describes Redist_0, the protocol for performing secret redistribution when the threshold does not change. We describe Redist_{-1} and Redist_{+1}, which handle decreasing and increasing the threshold, respectively, in Chapter 7.

5.3.1 Redist_0 Messages

Before delving into the details of the protocol itself, we discuss the messages used by Redist_0. The protocol uses four basic messages in normal case operation, plus one additional message to handle the case where a non-faulty node in the current epoch does not have its share (e.g., because it was unable to communicate earlier.) BFT, which we invoke as a subprotocol, uses three additional messages, plus several others to handle view changes, which we do not list here. Section 5.1 gave a brief overview of BFT.

All messages are acknowledged (although the acknowledgement messages are not shown here) to cope with network packet loss. Honest servers continue to retransmit messages until either they receive an acknowledgement from the recipient or the recipient no longer needs to receive the message (e.g., because the protocol terminated or advanced to a later phase). As in BFT, acknowledgements may be authenticated to prevent a denial-of-service attack in which spoofed acknowledgements are injected.

Figure 5-1 describes the format of all of the messages. We use a **`bold teletype`** for literals and the metalanguage, a **bold serif font** for data structures and variables, and a plain serif font for functions. Node i's public and private encryption

$\mathbf{Proposal}_{i,j}$ = $[\mathrm{Enc}_{PK_{\mathtt{e},j}}(Q_i(\alpha_j) + R_{i,1}(\alpha_j), \ldots, Q_i(\alpha_j) + R_{i,n}(\alpha_j))]$

$\mathbf{Commitment}_F$ = $[g^{f_0|0\ldots0}, g^{f_1|0\ldots0}, \ldots]$ *where* $F(x) = f_0 + f_1 x + \cdots$

(a) Common data structures

$\mathbf{MsgProposal}_i$ =
$[\mathtt{proposal}, \mathbf{epoch\#}, \mathbf{proposal}_{i,1}, \ldots, \mathbf{proposal}_{i,n},$
$\mathbf{commitment}_{Q_i}, \mathbf{commitment}_{R_{i,1}}, \ldots, \mathbf{commitment}_{R_{i,n}}]_{SK_{\mathtt{s},i}}$

$\mathbf{MsgProposalSet}$ =
$[\mathtt{proposalset}, \mathbf{epoch\#}, \mathbf{view\#}, \forall i \in \mathcal{S}_A \quad \mathbf{msgproposal}_i]_{SK_{\mathtt{s},\mathrm{coord}}}$
where $\mathcal{S}_A$ *is the set of nodes whose proposals were received by the coordinator*

$\mathbf{MsgProposalResponse}_i$ =
$[\mathtt{proposalresponse}, \mathbf{epoch\#}, \mathbf{view\#}, \mathbf{BadSet}, \mathrm{H}(\mathbf{msgproposalset})]_{SK_{\mathtt{s},i}}$
where **BadSet** *is the set of the identifiers of all the proposals* S_i *claims are invalid*

$\mathbf{MsgNewPoly}_j$ =
$[\mathtt{newpoly}, \mathbf{epoch\#},$
$\mathrm{Enc}_{PK_{\mathtt{e},1}}(P(j) + Q(j) + R_1(j)), \ldots, \mathrm{Enc}_{PK_{\mathtt{e},n}}(P(j) + Q(j) + R_n(j)),$
$\mathbf{commitment}_{P+Q+R_1}, \ldots, \mathbf{commitment}_{P+Q+R_n}]_{SK_{\mathtt{s},j}}$

$\mathbf{MsgMissingShare}_i$ =
$[\mathtt{missingshare}, \mathbf{epoch\#}]_{SK_{\mathtt{s},i}}$

(b) Protocol messages

Figure 5-1: Message Formats for the Unmodified Redistribution Protocol

keys are denoted by $PK_{\mathtt{e},i}$ and $SK_{\mathtt{e},i}$, respectively, and i's signing keys are $PK_{\mathtt{s},i}$ and $SK_{\mathtt{s},i}$. The notation $\texttt{[}message\texttt{]}_\kappa$ denotes the message *message* concatenated with $\mathrm{Sig}_\kappa(message)$.

Two common data structures, $\mathbf{Proposal}_{i,j}$ and $\mathbf{Commitment}_F$, are shown in Figure 5-1(a). These structures contain the proposal points and commitment matrices described in Chapter 4, for each node's proposal. As in Chapter 4, α_j denotes node S_j's unique identifier, Q_i and the $R_{i,k}$s are S_i's proposal polynomials, g is a group generator, and F is any polynomial.

Recall that the 0-padding at the end of each exponent in the commitment vector is needed because an attacker may be able to compute the low-order bits of the exponents (and hence the shares of the secret) from the commitment vector, since we are using Feldman VSS[Fel87]. However, the high-order bits are provably secret under the discrete logarithm assumption.

We briefly describe the purpose and contents of the messages here, but defer details to Section 5.3.2. The $\mathbf{MsgProposal}_i$ message contains node S_i's proposals for all other shareholders, along with commitments to the coefficients of the corresponding polynomials. The **MsgProposalSet** is a set of $2t + 1$ or more **MsgProposal**s collected by the coordinator. Nodes vote on which of the proposals sent by the coordinator are valid by sending a **MsgProposalResponse**, which enumerates the proposals that the sender believes are invalid. The response contains a hash of the coordinator's original **MsgProposalSet** to ensure that the coordinator is unable to cheat by sending different proposal sets to different nodes.

To perform share transfer to the new group, nodes in the old group send a **MsgNewPoly** message to the new group. Each **MsgNewPoly** message contains information for *all* members of the new group so that any $t + 1$ such messages generated by honest nodes allows *any* node in the new group to compute its share. The

contents are encrypted appropriately, so that the private key $SK_{\mathsf{e},k}$ is required to decrypt the data that is used to compute T_k's share. If some members of the new group receive their shares but others do not, the ones that are missing shares use **MsgMissingShare** messages to request the needed **MsgNewPoly** messages from the rest of the group.

In Section 6.2 we describe modifications to the message format whereby we hash and sign particular message components in a different manner to increase efficiency.

5.3.2 Redist_0 Steps

The following are the steps in the Redist_0 protocol.

1. Proposal Selection.

(1a) Each node S_i in the old group generates a random degree-f polynomial Q_i such that $Q_i(0) = 0$. It also generates n random polynomials $R_{i,1}, \ldots, R_{i,n}$ such that $R_{i,k}(\beta_k) = 0$, as described in Section 4. These polynomials have the form

$$\begin{aligned}
Q_i(x) &= q_{i,1}x + q_{i,2}x^2 + \cdots + q_{i,t}x^t \\
R_{i,1}(x) &= r_{i,1,0}x + r_{i,1,1}x + r_{i,1,2}x^2 + \cdots + r_{i,1,t}x^t \\
&\vdots \\
R_{i,n}(x) &= r_{i,n,0}x + r_{i,n,1}x + r_{i,n,2}x^2 + \cdots + r_{i,n,t}x^t.
\end{aligned}$$

Then S_i generates a commitment matrix C_i for these polynomials:

$$C_i = \begin{pmatrix} & g^{q_{i,1}|0\ldots0} & g^{q_{i,2}|0\ldots0} & \ldots & g^{q_{i,t}|0\ldots0} \\ g^{r_{i,1,0}|0\ldots0} & g^{r_{i,1,1}|0\ldots0} & g^{r_{i,1,2}|0\ldots0} & \ldots & g^{r_{i,1,t}|0\ldots0} \\ & & \vdots & & \\ g^{r_{i,n,0}|0\ldots0} & g^{r_{i,n,1}|0\ldots0} & g^{r_{i,n,2}|0\ldots0} & \ldots & g^{r_{i,n,t}|0\ldots0} \end{pmatrix}$$

Finally, S_i uses these polynomials to generate a $\mathbf{MsgProposal}_i$, which it broadcasts to all other servers in the old group. This signed message, detailed in Figure 5-1, has a **`proposal`** tag, the current epoch and view numbers, the proposals for all the other nodes in the old group encrypted with their respective public keys, and the commitment matrix.

(1b) The coordinator collects properly signed and well-formed **MsgProposal**s from at least $2t + 1$ distinct nodes (possibly including itself). To check that a message is well-formed, the coordinator checks the signature, the message format, and that the commitments prove that the proposed Q_i and $R_{i,k}$ polynomials have the requisite properties. Details of this verification step are found in section 4.2.3. We are guaranteed to receive $2t+1$ well-formed proposals because there are $n \geq 3t + 1$ nodes and at most t of these are faulty. For each S_j in the old group, the coordinator sends a **MsgProposalSet** message containing the proposals it collected.

(1c) If some node S_j receives a **MsgProposal** or **MsgProposalSet** and does not know *it's own* share, it broadcasts a **MsgMissingShare** message. Each S_i that receives a **MsgMissingShare** message responds by sending S_j all of the **MsgNewPoly** messages S_i received in the previous epoch, as explained in step (3b). These messages allow S_j to construct its share.

This extra step is sometimes needed because in an asynchronous network, we cannot guarantee that every member of the old group knows its share from the previous execution of the protocol. In particular, when we begin Redist_0 to move from epoch e to epoch $e+1$, there may be nodes in epoch e that don't have their shares, even though nodes in epoch $e-1$ have each received $t+1$ acknowledgements and discarded their information. However, we can guarantee that there are $t+1$ properly generated **MsgNewPoly** messages from epoch $e-1$, which all nodes in epoch e can use to compute their shares, and each of these messages is stored by at least one honest node in epoch e. Any node that is missing its share can request the requisite encrypted data to compute its share, which effectively allows us to assume that all honest nodes know their shares.

2. Agreement

(2a) When node S_j receives the **MsgProposalSet** from the coordinator, it verifies the format of every **MsgProposal** in the set, just as the coordinator did in step (1b). S_j also checks that the set itself is properly generated, i.e., it contains at least $2t+1$ **MsgProposal**s, all from distinct nodes in the old group. If the **MsgProposalSet** or any of the **MsgProposal**s it contains are not well-formed, the coordinator is faulty and S_j requests a view change. If the public information appears to be valid, for each $\mathbf{msgproposal}_i$ in the set, S_j decrypts $\mathbf{proposal}_{i,j}$, then verifies that the proposal matches the public commitment, as described in section 4.2.3.

(2b) S_j builds a list of all the nodes that sent bad proposals to it, i.e., proposals that didn't match the public commitments. It sends a signed **MsgProposalResponse** to the coordinator with this list. The response

Proposal Selection Algorithm

```
1.   d ← 0,   satisfied ← ∅,   rejected ← ∅
2.   props ← set of all proposals in MsgProposalSet
3.   foreach MsgProposalResponse R from distinct node i
4.     if i ∈ rejected
5.       continue
6.     if ∃j ∈ R.BadSet such that j ∈ props
7.       props ← props − {i, j},   rejected ← rejected ∪ {i, j}
8.       satisfied ← satisfied − {j}
9.       d ← d + 1
10.    else
11.      satisfied ← satisfied ∪ {i}
12.    if |satisfied| = 2t + 1 − d
13.      stop
```

Figure 5-2: Proposal Selection Algorithm

also contains a hash of the coordinator's **MsgProposalSet** message to ensure that a bad coordinator can't trick honest nodes by sending different proposal sets to each of them.

(2c) As the coordinator receives **MsgProposalResponse**s, it executes the Proposal Selection Algorithm shown in Figure 5-2. This algorithm is online; it processes responses as the coordinator receives them, and the coordinator stops waiting for additional responses as soon as the algorithm terminates. The coordinator verifies that each **MsgProposalResponse** is properly signed and well-formed before passing it to the Proposal Selection Algorithm.

The essence of this algorithm is that every time a response from a node that has not been accused accuses one or more of the remaining proposals,

we remove the proposals for both the accuser (if it is still on the **props** list in the first place) and one of the accusees, since at least one of these two is bad. To ensure determinism when multiple accusations against remaining proposals are available, we always choose the accusation against the lowest-numbered node in **props**. We prove in Sections 8.2 and 8.3 that the termination condition guarantees liveness (i.e., it is always reached) and also ensures that $t + 1$ honest nodes are satisfied with the remaining proposals and there is at least one proposal from an honest node left.

(2d) The coordinator runs BFT [CL02] to attempt to get the other replicas to agree to the proposal set it has chosen. The 'value' that replicas agree to is the set of proposals selected by the coordinator, along with the list of **MsgProposalResponse**s it used to do so, in the order the coordinator processed them. (Since the Proposal Selection Algorithm is deterministic, the list of responses is sufficient since all correct replicas will compute the same proposal set.)

Each correct recipient S_j, when asked to agree to this value, executes the proposal selection algorithm on the list of proposal responses. If the coordinator is non-faulty, then the algorithm successfully terminates on S_j with the same list of proposals that the coordinator chose, and S_j proceeds with the agreement protocol. S_j participates in the BFT protocol even if it believes some of the proposals the coordinator chose are invalid; S_j's compliance merely means that the coordinator correctly chose a proposal set according to the $2t + 1$ or more **MsgProposalResponse**s it received, which may or may not include S_j's response. (Furthermore, S_j will only agree if the **MsgProposalResponse**s contain identical hashes of the co-

ordinator's **MsgProposalSet**; this ensures that the coordinator sent the same list of proposals to all respondents.)

If a view change occurs, the new coordinator must repeat steps (2a) through (2c) using the set of proposals it collected, but there is no need to repeat step (1) of rebroadcasting all the proposals.

3. Transfer

(3a) Upon successfully completing the BFT protocol, each old node S_j that is satisfied with the agreed-upon proposal set sends the appropriate information to the new group in the form of a **MsgNewPoly** message. The **MsgNewPoly** message contains $3t + 1$ points, one for each of the members of the new group, each encrypted for epoch $e + 1$ with the respective recipient's public key.

Note that (unless our optional verifiable accusation scheme is used) not all non-faulty S_js will necessarily be able to participate in this step, because the proposal set chosen by the coordinator might contain proposals that S_j found to be invalid. However, the proposal selection algorithm guarantees that at least $t + 1$ honest nodes are happy with the selected proposals, and this is enough for the new shareholders to compute their shares.

As soon as an honest S_j has generated its **MsgNewPoly** messages for the new group, it can can discard its share and advance its forward secure encryption and signature keys. (If it is unable to generate these messages, it discards its secret information immediately.) Henceforth, an attack on S_j that results in more than f corruptions in the old group cannot cause the secret to become corrupted or revealed. However, to ensure that the

integrity of the sharing is preserved, S_j must remain correct until step (3c) to ensure that enough nodes in the new group received its **MsgNewPoly** message.

(3b) Each new shareholder T_k receives **MsgNewPoly** messages from members of the old group and computes its share. All of the **MsgNewPoly** messages from honest S_j's contain identical commitments to $P + Q + R_k$ for each k; hence, the first $t + 1$ identical commitments from distinct nodes are the correct ones. As noted in step (1c), T_k retains the **MsgNewPoly** messages so that in the following epoch, T_k will be able to send these messages to other members of the new group that did not receive them.

(3c) Each old shareholder S_j that was able to generate **MsgNewPoly** messages may delete these messages after receiving acknowledgements from $t + 1$ members of the new group, because the acknowledgements prove that at least one honest node in the new group got the message. At this point S_j is no longer needed; if it later becomes corrupted, the new group will still be able to reconstruct the secret.

Chapter 6

Improving the Performance of the Basic Protocol

Here we present several modifications to our protocol that yield better performance by reducing the number of rounds of communication and the number of responses that must be awaited in some steps and by lowering communication cost. Some of the extensions, such as our verifiable accusation scheme, may be independently useful in other protocols and come into play again when we discuss reducing the threshold in Chapter 7.

6.1 Verifiable Accusations

The proposal selection algorithm described above doesn't check accusations; it simply acts on them. The logic is that if one node accuses another, one of the two must be corrupt, so we remove both of them. At termination, the proposal set chosen by an honest coordinator in our unmodified protocol has the property that it contains at

least one proposal from an honest node and at least $t + 1$ honest nodes are happy with all the proposals in the set.

Here we describe a slightly different scheme in which accusations can be verified. The main advantage of this scheme is that the coordinator only requires $t + 1$ proposals in order to carry out proposal selection, so that the amount of communication and the message sizes are reduced. Furthermore, we can guarantee that all of the nodes from which an honest coordinator received a **MsgProposalResponse** will be able to use the proposal set that is selected.

Verifiable accusations are a way for other nodes to check the accuser's claim that the information sent to it was invalid. The idea is that nodes that receive invalid proposals in step 1 of the protocol can generate a verifiable accusation, and all other nodes can examine the accusation and accurately determine whether the accuser or the accusee is faulty. It is important that the accusation does not reveal any secret information belonging to honest nodes. However, if either the sender or recipient of an encrypted message is faulty, it is safe to reveal the contents of the message because the faulty party may be assumed to be under the control of the adversary, and hence could have revealed it anyway.

6.1.1 A Straw-Man Scheme

For illustration, we describe a bad scheme that is verifiable, but leaks secret information. We later show how to fix the inadequacies in this scheme. The straw-man proposal is to have the accuser reveal the encrypted and authenticated message it receives along with its secret decryption key. Other nodes can check accusations as follows:

1. Verify the signature on the alleged bad message using the accusee's public key.

If the signature is valid, then the message truly came from the sender, and if not the accuser is faulty.

2. Check that the accuser provided its correct private key for decrypting messages sent to it. This can be accomplished by encrypting a random message using the accuser's well-known public key, then decrypting it with the supplied private key and comparing the result to the original random message.

3. Decrypt the encrypted contents using the private key provided by the accuser. If the contents are not in the appropriate format or do not decrypt properly, the accusee is faulty.

4. Verify that the encrypted contents satisfy the appropriate properties, e.g., the ones described in section 4.2.3. If so, then the accuser is faulty, and if not, the accusee is faulty.

In essence, this is the kind of verification procedure we would like to have. Unfortunately, this straw-man scheme forces the accuser to reveal its private key, which would prevent the accuser from communicating privately with other nodes.

6.1.2 Fixing the Straw-Man Scheme: Forward-secure IBE

One possible fix to the broken scheme just described would be to negotiate sets of public/private key pairs, with a different key pair between every pair of servers for every epoch. However, this solution is inefficient because it requires an extra protocol round for key generation and dissemination. Furthermore, in an asynchronous network, we can only wait for $2t + 1$ out of the $3t + 1$ servers to publish their keys, which means that not all honest servers will be able to communicate. It turns out that this restriction makes it impossible to ensure that the protocol terminates.

A much better solution, and the one we describe in the following pages, is to parameterize the public/private key pair on the identities of the sender, recipient, and epoch, instead of distributing new keys for every sender/recipient pair and every epoch. Forward-secure encryption schemes [CHK03] allow the private key to evolve over time (e.g., every epoch) such that messages from past epochs cannot be decrypted using the current epoch key. Identity-based encryption (IBE) schemes [BF01, CHK03, Sha84] allow for the public key to be parameterized on an arbitrary string (e.g., the tuple consisting of the identities of the sender and recipient). Hence, our solution is to use an identity-based encryption scheme that is additionally forward-secure. We give brief definitions of identity-based encryption and forward-secure encryption (omitting formal statements of what it means for these schemes to be secure) in Figures 6-1 and 6-2, respectively.

In Section 6.1.4, we show how to construct a forward-secure identity-based encryption scheme by modifying the forward-secure encryption scheme of Canetti, Halevi, and Katz [CHK03]. This scheme builds forward-secure encryption out of hierarchical identity-based encryption (HIBE), so retrofitting the forward-secure scheme to also be identity-based is straightforward.

Security of HIBE (and hence security of verifiable accusations) is predicated on the bilinear Diffie-Hellman (BDH) assumption [BF01], which our unmodified protocol does not require. Note that our unmodified protocol *does* require forward-secure encryption and signatures, but it does not require that encryption be identity-based. The rest of this section explains our forward-secure identity-based encryption scheme and how we use it to implement verifiable accusations.

Definition 6.1 ***Identity-Based Encryption.*** *Identity-based encryption is a tuple of four algorithms: KeyGen, GetPrivateKey, Encrypt, and Decrypt, which work as follows.*

- *KeyGen():*
  Generate a public key PK and a master secret key SK.

- *GetPrivateKey(PK, SK, ID):*
  For the given public/private master key pair, return a private key k_{ID} for the identity ID. ID can be an arbitrary string.

- *Encrypt(PK, ID, m):*
  Compute the encryption of the message m under public key PK parameterized by identity ID, and return the ciphertext c.

- *Decrypt(k_{ID}, c):*
  Return m, the decryption of the ciphertext c. This operation should succeed if and only if c is a valid result of Encrypt(PK, ID, m) and k is the output of GetPrivateKey(PK, SK, ID) for the same PK and ID.

As an illustrative example, suppose we have an email server with a well-known public key, with the associated private master key SK safeguarded by the ISP. The ISP uses GetPrivateKey() to generate private identity keys $k_{\texttt{alice}}$ and $k_{\texttt{bob}}$, which it transmits securely to users Alice and Bob. Subsequently, anyone can encrypt a message for Alice using the *Encrypt()* function given only the email server's public key and the string `alice`. Only Alice and the ISP are able to decrypt this message.

Figure 6-1: Identity-Based Encryption

Definition 6.2 ***Forward-Secure Encryption.*** *A forward-secure encryption scheme is a tuple of four algorithms: KeyGen, UpdatePrivateKey, Encrypt, and Decrypt.*

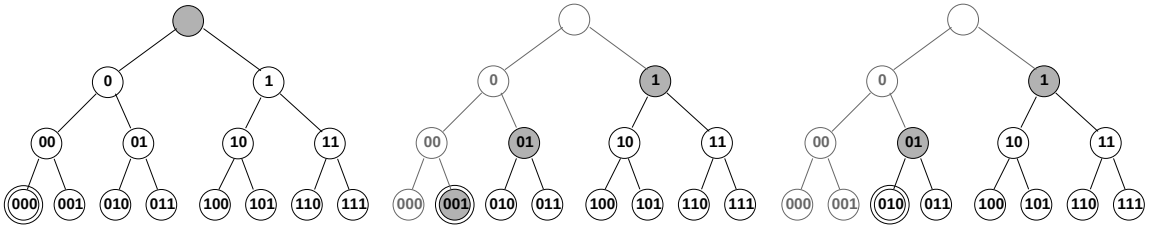
- *KeyGen(T):*
  Return a public key PK and an initial secret key SK_0 for a forward-secure encryption scheme designed to operate for a maximum of T epochs.

- *UpdatePrivateKey(SK_e):*
  Given the private key for epoch e, return the private key for epoch $e+1$. The inverse operation (computing the key for epoch e from the key for epoch $e+1$) should be infeasible.

- *Encrypt(PK, e, m):*
  Return c, the encryption of m for epoch e.

- *Decrypt(SK_e, c):*
  Return m, the decryption of c. The ciphertext c must have been encrypted with respect to epoch e.

Figure 6-2: Forward-Secure Encryption



(a) $e = 0$ (b) $e = 1$ (c) $e = 2$

Here, $r = 3$, and the first 3 epochs are shown. A double circle indicates the identity used for encryption in the current epoch, and shaded circles indicate state that the BTE scheme retains during that epoch.

Figure 6-3: Binary Tree Encryption Illustration

6.1.3 Canetti-Halevi-Katz Forward-Secure Encryption

The paper of Canetti, Halevi, and Katz [CHK03] shows how to turn a hierarchical identity-based encryption scheme (HIBE) into a forward-secure encryption scheme through the concept of *pebbling*. A hierarchical identity-based encryption scheme is simply an ordinary identity-based encryption scheme (see Figure 6-1) with the additional property that k, the output of GetPrivateKey(PK, SK, ID), can itself be used as a master secret key for identities "under" ID. For instance, we could have

$$k_{\texttt{foo|}} \leftarrow \text{GetPrivateKey}(PK, SK, \texttt{foo|})$$

$$k_{\texttt{foo|bar}} \leftarrow \text{GetPrivateKey}(PK, k_{\texttt{foo|}}, \texttt{bar}).$$

Hence, the holder of a particular private key for a particular identity ID can decrypt messages for any identities for which ID is a prefix.

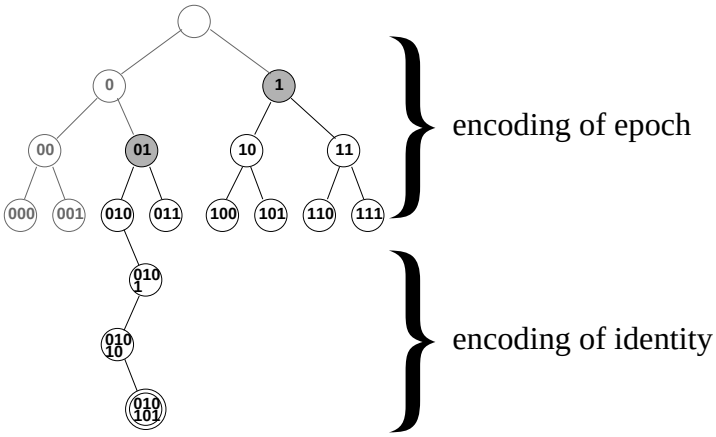
Canetti et al describe *binary tree encryption*, in which an identity is restricted to be a sequence of 1 or more bits, and a sequence $b_1, \ldots, b_{k-1}$ is hierarchically "above" any sequence that starts with $b_1, \ldots, b_k$. Given the identity key associated with any

node in the tree, it is possible to derive the keys for its children. BTE is used to create a forward-secure encryption scheme as follows: in epoch e, where the binary representation of e as a r-bit binary number is $e_1 e_2 \ldots e_r$ for a fixed r, the identity used for encryption is $e_1, e_2, \ldots, e_r$. Initially, the master secret key is kept, but as an epoch ends, higher-level master keys are replaced by lower-level ones that they can derive, such that the old epoch key can no longer be derived. Figure 6-3 shows how an example BTE scheme with $r = 3$ evolves over the first three epochs. The keys used for encryption are the leaves of the tree, and in any given epoch, the state retained makes it possible to derive the keys for all future epochs, but not past epochs. The number of identity keys that must be stored is $O(r)$ since we need only remember at most one node per level in the tree. The system supports up to 2^r epochs.

6.1.4 Our Forward-Secure IBE Scheme

The scheme discussed above can easily be expanded to allow for identity-based forward-secure encryption, by using a further level of hierarchy for the identity. That is, the identity used as input to the BTE is the concatenation of the epoch e and the identity ID given as input to the identity-based forward-secure encryption algorithm. The first r levels of the tree encode the epoch number, and the remaining levels encode ID, as shown in Figure 6-4. Note that the specific secret key for a certain epoch with a certain identity is not hierarchically "above" any key that would be used in the future, since it is a leaf; therefore, revealing it exposes only that key and nothing else.

This scheme is both simpler and less flexible than the forward-secure hierarchical identity-based encryption scheme of Yao et al. [YFDL04]. The Yao et al. scheme has higher overhead than ours, but allows for additional flexibility that is not required



Extending the BTE tree for $e = 2$ from Figure 6-3(c), we derive the forward-secure IBE key for $e = 2$ and identity $5 = 101_2$.

Figure 6-4: Tree Encoding for Forward-Secure IBE

for our purposes. In particular, they allow a key for identity ID and epoch e to be evolved into a key for ID and epoch $e+1$ without the master key for epoch e, whereas our scheme only allows the master key for the epoch to be evolved. The additional property they provide is undesirable for our verifiable accusation scheme because our accusations reveal keys for identities corresponding to nodes that generated bad proposals, and we would not like the adversary to use these keys to compute keys for the same identity in future epochs. Strictly speaking, however, we could use their scheme safely because our threat model assumes that corrupted nodes are always replaced, rather than being "recovered" in a future epoch while retaining the same identity.

There is one caveat here: the properties of identity-based encryption assume that the generator of the master key—the root of the tree—is honest. However, this should not be a problem for our application: nodes may be corrupted, but they are not corrupted prior to generating their own master keys, which is an offline operation.

6.1.5 Verifiable Accusations from Forward-Secure Encryption

In an identity-based encryption scheme, each public key is really a master public key, and can be used to encrypt under any "identity" (that is, any string). The secret keys for each identity can be derived from the master secret key.

With an identity-based forward-secure encryption scheme, we have sender S_i encrypt the proposals for S_j as $\text{Enc}_{PK_j,i,e}(\text{prop})$ where PK_j is S_j's public key. The result is a forward-secure encryption for epoch e, parameterized under identity i. S_j decrypts this message using a secret epoch key $k_{e,i}$ and to accuse it simply reveals this key in the response in step 2 of the protocol. Other nodes can use this key to determine whether S_j's accusation is honest or not, but releasing the key doesn't help the adversary decrypt messages from other senders to S_j, for either the current or later epochs.

Note that our use of identity-based encryption is unusual in that messages are encrypted using the recipient's public key and the sender's identity. More familiar applications of IBE involve senders encrypting messages using a global public key and the recipient's identity.

Figure 6-6 shows the new proposal selection algorithm that is used for verifiable accusations. Whereas the original algorithm potentially required more than $2t + 1$ **ProposalResponse**s, the modified algorithm for verifiable accusations requires exactly $2t + 1$ responses, and it is able to guarantee that all the honest nodes that responded are happy with the final set of proposals it selects.

If $\textbf{accusation}_{j,i}$ is valid, then honest nodes that receive it will be convinced that S_i generated an invalid proposal; if it is invalid, honest nodes will be convinced that S_j is faulty. All of the following conditions must be met for $\textbf{accusation}_{j,i}$ to be

$\mathbf{Proposal}_{i,j}$ = [$\mathrm{Enc}_{PK_{\mathtt{e},j},i,e}(Q_i(j) + R_{i,1}(j), \ldots, Q_i(j) + R_{i,n}(j))$]

$\mathbf{Accusation}_{j,i}$ =[i, **GetPrivateKey**($PK_{\mathtt{e},j}$, $SK_{\mathtt{e},j}$, e)]

$\mathbf{AccusationSet}_j$ =[**accusationset**, $\forall i$ **accused** **accusation**$_{j,i}$]

$\mathbf{MsgProposalResponse}_i$ =
[**proposalresponse**, **epoch#**, **view#**,
AccusationSet, H(**msgproposalset**)]$_{SK_{\mathtt{s},i}}$

Figure 6-5: Modified Message Formats for Verifiable Accusations

Proposal Selection Algorithm (Verifiable Accusations)

1. **responses** $\leftarrow 0$

2. **props** $\leftarrow$ set of all proposals in **MsgProposalSet**

3. **foreach MsgProposalResponse** R from distinct node i

4. **foreach accusation**$_{i,j} \in R$.**AccusationSet**

5. **if accusation**$_{i,j}$ is valid

6. **props** $\leftarrow$ **props** $- \{j\}$

7. **else**

8. **props** $\leftarrow$ **props** $- \{i\}$

9. **responses** $\leftarrow$ **responses** $+ 1$

10. **if responses** $\geq 2t + 1$

11. **stop**

Figure 6-6: Proposal Selection Algorithm for Verifiable Accusations

considered valid.

- The proposal contained within the accusation is a properly signed and tagged **MsgProposal** message from S_i.

- When $\mathbf{proposal}_{i,j}$ is decrypted using the key contained in the accusation, either the decryption fails, or the decrypted message fails one of the validity checks described in section 4.2.3.

- When a random message is encrypted using S_j's public key under epoch e and identity i, the ciphertext is correctly decrypted using the key contained in the accusation. This ensures that the decryption key supplied by S_j is indeed the correct one.[1]

Checked accusations ensure that only invalid proposals are discarded in the proposal selection phase. Therefore it is sufficient to start with only $t+1$ proposals since one of them must be from an honest node, and that proposal will not be discarded. Since the random polynomials used to compute the new shares are the sum of all the accepted proposals, having at least one proposal from an honest node ensures that the sum is random as well.

Strictly speaking, line 8 of the algorithm is not needed. It says that if we receive an invalid accusation from S_i, we don't use S_i's proposals, even if those proposals appear to be valid. However, it seems wise to ignore provably faulty nodes to the extent possible.

[1]For this check to be effective, it must be computationally infeasible for faulty nodes to construct "bad" private keys that decrypt some messages encrypted with their public key but not others; see Definition 3.4 of Section 3.3. For the Canetti-Halevi-Katz cryptosystem, it is infeasible to find a second private key that decrypts *any* message with non-negligible probability. We do assume that each server's public key was properly generated when the server was initialized, since most cryptosystems assume an honest key generator.

6.1.6 Problems with Accusations in the Scheme of Herzberg et al.

Herzberg et al.'s proactive secret sharing protocol [HJKY95] also uses accusations, but it has several significant differences from our scheme that make it less plausible. Foremost, their scheme requires verifiable accusations, whereas our scheme merely uses them as an optimization. Their accusations also require synchrony assumptions about the network that are too strong to work in practice for unreliable networks such as the Internet.

In Herzberg et al.'s scheme [HJKY95], each time a server S_j accuses another server S_i, S_i must respond with a *defense* against this accusation. The accusation itself is merely a cryptographically signed statement of the fact that S_j believes S_i's proposal to be invalid. The defense is a decryption of the proposal, along with any information S_i used to compute the encryption. (All semantically secure public key encryption algorithms must be randomized, so this additional information generally consists of the initialization vector, or random seed, used by the encryption function.) Anyone can then check whether the message S_i claims to have sent encrypts to the ciphertext S_i signed earlier.

This approach does not work when the network is asynchronous. If S_i does not send a defense, it might be faulty, or it might be honest but unable to respond in time. The coordinator has no way to determine which of these two scenarios occurred, and therefore Herzberg et al.'s accusation scheme fails. Note that Herzberg et al.'s scheme was only designed for reliable, synchronous networks, in which the inability of a node to respond within a fixed time period implies that it has failed.

Herzberg et al. also claim that for some encryption algorithms (in particular, they cite RSA), the defense step is unnecessary. Presumably the authors refer to

the usual hybrid RSA scheme in which the message is encrypted using a symmetric key encryption algorithm and a random key, then the random key is encrypted using RSA. Revealing the random key suffices to allow others to decrypt the message and verify that it corresponds to the signed ciphertext. However, this proposed scheme is vulnerable to adaptive chosen ciphertext attacks. For example, suppose S_i sends a proposal to S_j, encrypted with S_j's public key, and call the ciphertext c. An attacker A wishing to learn information about the decryption of c sends a proposal c' to S_j, which is either identical or similar to c. S_j accuses A by decrypting c', which reveals information about c. It has been demonstrated that attacks of this nature are effective against RSA [Ble98]. Modifications to RSA such as the Fujisaki-Okamoto transformation[FO99] that add chosen-ciphertext security are not an effective solution to this problem because they add additional random parameters to the encryption function that the accusee would need to reveal, and revealing these parameters would once again make the scheme subject to a chosen ciphertext attack.

6.2 Reducing Load on the Coordinator

Using a cryptographically-secure hash H (e.g., $H = \text{SHA256}$), we can reduce message sizes significantly, especially in the common case in which very few if any servers are actually behaving badly. In the original protocol, even though each sever broadcasts its proposals to other servers in the old group, the coordinator rebroadcasts the proposals its receives. Doing so ensures that all nodes that can communicate with the coordinator receive (minimally) the same $2t + 1$ proposals, which our protocol requires, but it also places an undue burden on the coordinator.

Reducing load on the coordinator is important because the coordinator is the bottleneck in our protocol. The coordinator receives as much data as every other

$\mathbf{MsgProposalSet}$ =
$\quad$ [**proposalset**, **epoch#**, **view#**, $\forall i \in \mathcal{S}_A \quad \langle i, H(\mathbf{msgproposal}_i)\rangle$]$_{SK_{\mathrm{s,coord}}}$

$\mathbf{MsgMissingProposals}_j$ =
$\quad$ [**missingproposals**, **epoch#**, **view#**, **MissingSet**]$_{SK_{\mathrm{s},j}}$

Figure 6-7: Message Formats using Hashing. Only messages that differ from the original protocol are shown.

node, asymptotically speaking. In broadcasting all the proposals via point-to-point links, however, the coordinator sends about a factor of n more data than it receives. To reduce the amount of data sent, the coordinator can instead send a list of hashes of the proposals (and commitments) it receives, and recipients can use the hashes to verify that the proposals and commitments they were sent are identical to the ones that the coordinator received. This ensures that faulty servers cannot send one set of proposals to some nodes and a different set of proposals to others.

If there are faulty replicas that send messages to the coordinator but to nobody else, the burden of sending these proposals still falls on the coordinator as it does in the unmodified protocol, but in the common case where the number of faults is small, this optimization is significant. Servers that do not receive proposals included in the coordinator's proposal set must request them from the coordinator in the optimized protocol.

Figure 6-7 shows the protocol messages using hashing to reduce message size. Instead of including all the proposals in the **MsgProposalSet**, the coordinator merely sends a list of the indices of the servers that sent the proposals, along with hashes of the proposals. When a server S_j receives a **MsgProposalSet** from the coordinator, it checks to see if it is missing any of the proposals enumerated in the **MsgProposalSet**, and it determines whether some of the proposals it received don't match the hashes contained in the **MsgProposalSet**. S_j waits for additional

proposals until the number of missing or mismatched proposals is at most t (since there are at most t faulty servers, and S_j will eventually hear from all of the non-faulty ones). S_j optionally delays for a small, fixed amount of time to give honest servers more time to respond, then if it still has missing or mismatched proposals, it sends a **MsgMissingProposals** message to the coordinator with the list of such proposals. A correct coordinator will respond with the required **MsgProposal**s. If the coordinator is faulty and sends inconsistent information, or neglects to send to particular servers, the protocol may not complete in the current view; in this case, a view change will occur after a timeout, and a new coordinator will be chosen as usual.

If the coordinator is honest, faulty servers and network delays can force it to send a large number of proposals. In particular, there are three situations where the coordinator may have to do additional work:

- Each Byzantine faulty server can request that the coordinator send it up to t proposals.

- By sending its own proposals to the coordinator and nobody else, a faulty server can force the coordinator to forward its proposals to the $2t + 1$ or more honest replicas.

- If honest servers do not receive the required proposals from other honest servers within a fixed amount of time due to message delays, they may need to request the information from the coordinator instead. Honest servers must do this because they cannot tell whether they are missing proposals due to some of the proposers being bad, or because messages from those proposers to them were not delivered in time.

In the optimistic case where these problems do not occur, the coordinator sends a factor of n less information. However, even in the worst case, the coordinator can be forced to send no more than t proposals to each server, instead of $2t + 1$ proposals as in the original protocol—a factor of 2 improvement.

Chapter 7

Changing the Threshold

The protocol as we have described it performs a proactive secret resharing from one group of size $n = 3t + 1$ to another group of size $n' = n$, where the group members may be arbitrary. In fact, it is trivial to extend the protocol so that n and n' can be arbitrary numbers as long as $n \geq 3t + 1$ and $n' \geq 3t + 1$. However, merely changing the number of shareholders without changing the threshold t is not useful in and of itself. A secret sharing with $n > 3t+1$ is less efficient in terms of communication than a $n = 3t + 1$ sharing, and less secure because the maximum number of permissible faults is a smaller fraction of the entire group.

A much more useful operation is changing the threshold itself. That is, we fix $n = 3t+1$ and $n' = 3t'+1$ but allow for $t' \neq t$. In practice, an administrator may want to change the threshold to adapt to changing security needs and new assumptions about the reliability of the group members or fate-sharing amongst servers.

Changing the threshold is interesting and requires significant additional effort. This chapter discusses techniques for increasing and decreasing the threshold.

7.1 Decreasing the Threshold

In many situations, it may be desirable to tweak the threshold multiple times over a small range. For instance, an administrator might temporarily increase the threshold in response to an Internet worm that may affect a significant fraction of the servers, then decrease it again when the servers have been patched for that worm. This section presents a simple technique for handling this situation.

To decrease the threshold by one, we create a virtual server (or an additional one if previous decreases have occurred). The virtual server's share is given to all real servers—effectively making the share public—so that any group of m nodes will know $m + v$ shares, where v is the number of virtual servers. Thus, if we increase v by one, the threshold effectively decreases by one. During the protocol, virtual servers do not generate proposals, but do get shares, check them, and send them to the next group. These tasks are carried out by the actual servers in a deterministic fashion so that each honest server can perform the simulation locally.

This scheme works because reducing the threshold from t to $t' = t - v$ but continuing to use a degree-t polynomial essentially permits v additional faults. Hence, if the adversary has $t - v$ shares due to corruptions in the new group, and it additionally knows the public share for the v virtual servers, it still does not have enough points to interpolate a degree-t polynomial. However, it is important that virtual servers "behave" correctly. All v of them must produce the appropriate information for the next group in the following epoch, because recipients will need $t + 1$ correct **MsgNewPoly** messages and only $t' + 1 = t - v + 1$ honest senders are guaranteed to be able to produce these. The new groups in subsequent resharings will only accept **MsgNewPoly** messages if $t' + 1$ members of the old group produce the same message according to their local simulations. In essence, virtual servers can be modeled

as passive adversaries; they may reveal information, but they behave correctly.

This approach is somewhat unsatisfying theoretically because using this method to reduce the threshold does not reduce the asymptotic computational overhead of the protocol. However, it decreases the number of physical servers; eliminates some of the encryption and signing; and significantly reduces the communication overhead because the virtual servers do not generate polynomials or commitments, commitments being the chief source of overhead in our scheme.

7.2 Increasing the Threshold

The protocol from Section 7.1 for decreasing the threshold relies on creating virtual servers, which act like real shareholders, but are merely simulations. If we have decreased the threshold using this protocol, we simply decrease the number of virtual servers when we move to the next epoch, and only resort to one of the protocols described in this section when there are no more virtual servers. This means that if the threshold is increased and decreased many times within a fixed range, the incremental changes are extremely cheap. Below, we describe what to do if there are no virtual servers.

To perform a resharing to a new group while simultaneously increasing the threshold by c, we need to construct a new sharing polynomial of degree $t' = t + c$. This can be done by having each node generate Q and R_k polynomials of degree $t' = t + c$ rather than degree t. Then the resulting polynomial $P' = P + Q$ will also have degree $t' = t + c$.

Transferring the shares is a problem for the original Redist_0, however, because new servers must receive valid points from $t + c + 1$ old nodes in step (3b) so that they can interpolate the degree $t + c$ polynomial. However, Redist_0 guarantees only

that $t+1$ non-faulty servers in the old group are able to send shares. This is because the proposal selection algorithm (Figure 5-2) only guarantees that $t+1$ non-faulty servers are able to use the set of proposals it chooses, and some accepted proposals may be invalid for the remaining t good servers in the old group.

There are several ways of dealing with this problem. To demonstrate some of the issues, we begin in Section 7.2.1 by showing a simple solution without verifiable accusations and demonstrate that it does not work even for $c = 1$. In Section 7.2.2, we show a two-phase protocol based on Redist_0 that overcomes these limitations. In Section 7.2.3 we describe $\mathsf{Redist}^{\nu}_{+c}$, which inherits some of the basic structure of the straw man, but solves the problems of the straw man in a different way. A discussion of the relative merits of the two approaches can be found in Section 7.2.4.

7.2.1 A Straw Man Protocol

Our first proposal for increasing the threshold from t to $t' = t + c$ attempts to use the usual group sizes $n = 3t + 1$ and $n' = 3t' + 1$ and eschews verifiable accusations. Before we describe the scheme, we argue that any adaptation of Redist_0 that satisfies these properties is limited to $c = 1$.

Suppose that the initial proposal set contains t proposals from faulty nodes and $t + 1$ proposals from good nodes, but the bad proposals are specially generated to contain good information, except for a particular set of $t - 1$ good nodes. Assume optimistically that the protocol is somehow able to wait for proposal responses from all non-faulty nodes. Then $t + 2$ good nodes will accept all the proposals, and the remaining $t - 1$ good nodes will accuse the t bad ones. But without verifiable accusations, each good node can eliminate at most one bad proposal; thus, one bad proposal will remain, and only $t + 2$ good nodes will be satisfied with the chosen

proposal set. The new group uses a polynomial of degree $t + c$, so we must choose a proposal set such that at least $t + c + 1 \leq t + 2$ non-faulty nodes in the old group are able to use. Therefore $c \leq 1$.

Our straw man protocol works as follows, using $c = 1$:

1. Run Redist_0, but use proposal polynomials of degree $t + 1$. As soon as the proposal selection algorithm terminates, perform Byzantine agreement on the current proposal set and attempt the share transfer, as in Redist_0.

2. If the share transfer cannot complete (i.e., there are only $t+1$ honest servers that are satisfied with the chosen proposal set), this means that the coordinator has yet to hear from t honest servers. Hence, the coordinator can await additional **MsgProposalResponse** messages in parallel with the share transfer. If an additional response is received, the agreement and share transfer steps are repeated with the new information. The second attempt is guaranteed to complete because at least $t + 2$ honest servers will be able to send information to the new group.

Note that we must perform these two attempts in parallel. If the first share transfer completes, the coordinator may never receive an additional **MsgProposalResponse** because it has heard from all of the honest servers already; it is impossible to tell which case we are in.

In any secure secret redistribution protocol, there must be a well-defined point at which honest shareholders in the old group discard their secret information. Otherwise, an adversary who continues to corrupt old nodes after the resharing has completed will eventually be able to recover the secret. In Redist_0, for instance, old nodes wait until BFT completes, compute messages to send to the new group if they

are able to, and then discard their secret information. As we shall see, the pivotal failure of the straw man scheme is that old nodes cannot tell when it is safe for them to discard their secret information. We discuss three methods for old nodes to make this determination, and show that all three fail.

Approach 1. Each old node waits until step (2) of the straw man protocol completes, and discards its secret information after the second agreement has completed and the **MsgNewPoly** messages have been generated. This approach does not work because we will never reach step (2) if the coordinator already received responses from all honest nodes in step (1).

Approach 2. Each old node waits only until step (1) of the straw man protocol completes, and discards its secret information as soon as it has generated its **MsgNewPoly** messages for step (1) (assuming it is satisfied with the chosen proposal set). However, the share transfer in step (1) may not succeed because we can only be certain that $t+1$ honest servers are able to use the chosen proposal set, and $t+2$ are needed to ensure that the new group is able to reconstruct the secret. If this happens, the protocol is forced to proceed to step (2), but step (2) cannot complete because some of the honest nodes have discarded their secret information too early. The result is that neither the old nor the new groups can recover the secret s.

Approach 3. We can attempt to fix approach 2 by using acknowledgements from the new group as a basis for deciding when it is okay for servers in the old group to discard their secret information. Suppose each server T_k in the new group delays its acknowledgement for as long as possible. In particular, T_k only sends an acknowledgement upon receipt of $t'+1 = t+2$ **MsgNewPoly** messages that contain valid shares for T_k. However, T_k could have received $t+1$ **MsgNewPoly** messages from honest servers in the old group, and one **MsgNewPoly** message from a faulty server that just happened to send a valid share for T_k. Hence T_k's acknowledgement

only proves that T_k can reconstruct its own share; it does not guarantee that the **MsgNewPoly** messages provided to T_k are sufficient to allow other members of the new group to compute their shares. Even if each server S_i in the old group awaits $2t' + 1$ acknowledgements (i.e., the maximum number it can wait for), this only guarantees that $t' + 1$ honest servers in the new group can reconstruct their shares (since t' of the acknowledgements may be from faulty servers). However, given that only $t' + 1$ non-faulty servers in the new group know their shares, the next instance of the protocol that transitions from epoch $e + 1$ to $e + 2$ will be unable to complete, because the precondition we require is that all honest nodes know their shares.

The key insight is that using a protocol structured like the straw man scheme presented in this section does not work because shareholders in the old group don't have enough information to tell when it is safe to discard their secret information. Two resharings take place in parallel, and while we know that one of them will complete successfully, do not know which one. If nodes wait for the second resharing to complete, they may wind up retaining their secret information forever if the first resharing was successful. If they wait for only the first phase to complete, they may discard their secret information too soon.

7.2.2 A Two-Step Redist_{+c}

This section describes the Redist_{+c} protocol, which uses two resharings to work around the problems with the straw man protocol of Section 7.2.1. In particular, the old shareholders in Redist_{+c} have a well-defined point at which they can discard their secret information safely, and we remove the restriction that $c = 1$.

The straw man protocol illustrates that to increase the threshold using a variant of Redist_0, $t + c + 1$ honest servers in the old group must be able to perform share

transfer to the new group. To guarantee that this is possible, we need $3t + c + 1$ servers total, of which no more than t are faulty. Thus, we use the following two step Redist_{+c} protocol:

1. Run Redist_0 to perform a resharing from the current group $\{S_1, \ldots, S_{3t+1}\}$ to a new *temporary* group $\{S'_1, \ldots, S'_{3t+c+1}\}$ that contains c additional members. The threshold remains unchanged.

2. Run Redist_0 in the temporary group to transfer the sharing to the new group $\{T_1, \ldots, T_{3t'+1}\}$. Modify the protocol to use proposal polynomials of degree $t' = t + c$, and in the proposal selection algorithm, wait for $2t + c + 1 - d$ **MsgProposalResponse**s instead of $2t + 1 - d$. This guarantees that at least $t' + 1$ nodes in the temporary group can perform the transfer.

The first step of resharing to a group of size $3t + c + 1$ is identical to executing the original Redist_0 except that each S_i generates c additional polynomials, $R_{i,3t+2}, \ldots, R_{i,3t+c+1}$, and hence it adds c extra points to each $\mathbf{proposal}_{i,j}$ it sends. As noted in Section 4.1.4, Redist_0 works for any $n \geq 3t + 1$, not just $n = 3t + 1$; we have thus far fixed $n = 3t + 1$ merely as a matter of efficiency. The second instance of Redist_0 also uses additional polynomials—$3t' + 1$ of them—and the degree of these polynomials is $t + c$. But since we have $2t + c + 1$ honest servers, of whom $t + c + 1$ are able to participate in share transfer, we are ensured that the new group will be able to interpolate a polynomial of degree $t + c$. Note that the intermediate group may overlap arbitrarily with the old and new groups.

7.2.3 A One-Step $\mathsf{Redist}^{\nu}_{+c}$

This section describes the $\mathsf{Redist}^{\nu}_{+c}$ protocol, which is based directly on the straw man protocol of Section 7.2.1, but uses verifiable accusations to solve the problems with that protocol. Recall that the straw man failed because old shareholders could not determine when it was okay for them to discard their secret information. $\mathsf{Redist}^{\nu}_{+c}$ is an alternative to the two-step Redist_{+c} protocol from Section 7.2.2, which worked around the problems of the straw man by adding more shareholders.

We presented three flawed approaches for deciding when to discard secret information in the straw man scheme. Approach 1 illustrated that if old shareholders wait too long, they may never discard their secrets. Approach 2 showed that without verifiable accusations, the old group alone cannot tell whether the proposal set it chooses in step (1) is acceptable for $t' + 1$ honest new nodes. Approach 3 showed that even if we rely upon acknowledgements from the new group, not all members of the new group are able to recover their shares. This section demonstrates that verifiable accusations overcome the problems associated with approaches 2 and 3.

The lack of verifiable accusations also limited the straw man to $c = 1$. $\mathsf{Redist}^{\nu}_{+c}$ also removes this restriction, and allows the threshold to increase by up to $c \leq t$. In other words, we may increase the threshold by no more than a factor of two in each resharing. Larger increases are not allowed because each member of the new group needs $t' + 1 = t + c + 1$ properly-generated **MsgNewPoly** messages to compute its share. We can only guarantee that there are $2t + 1$ correct servers in the old group, so we require that $t + c + 1 \leq 2t + 1$.

$\mathsf{Redist}^{\nu}_{+c}$ proceeds in a series of *iterations*, and up to $c + 1$ iterations may be required to complete the protocol. These iterations serve a similar purpose to the steps in the straw man scheme, but the straw man fixed $c = 1$, so it had $c + 1 = 2$

iterations. We begin in the same way as the straw man: we start agreement and share transfer in iteration 0 as soon as the proposal selection algorithm terminates (after having received at most $2t + 1$ responses, since we are using verifiable accusations). However, the coordinator awaits up to c additional **MsgProposalResponse**s, and each time it receives an additional response containing a valid accusation, it removes the accused proposal from the proposal set, increments the iteration number, and restarts the agreement and share transfer with the amended set.

Each old node erases its secret information and quiesces when it receives receipts from $2t' + 1$ servers in the new group indicating that those servers successfully computed their shares. As discussed in Section 7.2.1, this only guarantees that $t' + 1$ honest servers in the new group know their shares; however, we show that by using verifiable accusations, we can perform a recovery operation within the new group so that all honest servers in the new group learn their shares.

7.2.3.1 Redist_{+c} Steps

To make Redist_{+c} work, each of the **MsgProposalSet**, **MsgProposalResponse**, and **MsgNewPoly** messages, as well as the BFT protocol messages, are annotated with an **iteration#** field, which makes messages for each iteration distinct. Iterations are units smaller than epochs but larger than views; each epoch may require up to $c + 1$ iterations of agreement in order to complete the transition to the next epoch, and each iteration may require multiple views. (However, the total number of view changes due to faulty coordinators over all iterations is still at most t.[1])

We also add a **MsgCompletionCertificate** message, which has the following format:

[1] It is possible for additional view changes to occur due to timeouts, but these are infrequent and have minimal impact in practice. See Section 5.1 for details.

MsgCompletionCertificate$_i$ **=**

[completioncertificate, $\forall j \in \mathcal{S}_C$ **msgtransfersuccess**$_j$**]**

A **MsgCompletionCertificate** is generated by each node S_i in the old group. It is essentially a collection of $2t'+1$ **MsgTransferSuccess** messages, all for the same iteration (namely, the ones in $\mathcal{S}_C$, the set of the first $2t'+1$ such messages received by S_i). This collection proves that the share transfer to the new group was successful for that iteration. Note that the **MsgCompletionCertificate** need not be signed because the constituent **MsgTransferSuccess** messages are signed, and any collection of properly authenticated **MsgTransferSuccess** messages corresponding to the same iteration is a valid certificate, even if the collection is assembled by a dishonest party.

The steps of Redist_{+c} are as follows:

1. Run Redist_0 with the verifiable accusation option, but use proposal polynomials of degree $t+c$ and use several modifications described in the following steps. Perform agreement and attempt share transfer when the proposal selection algorithm terminates (modified for verifiable accusations, see Figure 6-6).

2. If the share transfer cannot complete (i.e., there are fewer than $t+c+1$ honest servers that are satisfied with the chosen proposal set), this means that the coordinator has yet to hear from some of the honest servers. The coordinator awaits additional **MsgProposalResponse** messages in parallel with the share transfer. Each time a new **MsgProposalResponse** arrives that contains a valid accusation against a proposal in the current proposal set, remove that proposal from the set and restart the agreement and share transfer steps with the new proposal set. After at most c restarts, $t+c+1$ honest servers will be satisfied with the proposal set and the coordinator stops processing responses.

3. Upon receiving $t+c+1$ valid **MsgNewPoly** messages from the old group, all for the same iteration, servers in the new group generate **MsgTransferSuccess** messages, which they broadcast to all members of the old group.

4. When a server in the old group collects $2t'+1$ valid **MsgTransferSuccess** messages, all with identical iteration numbers, the server discards all its secret share and all other secret information for epoch e. Then it concatenates the **MsgTransferSuccess** messages to form a **MsgCompletionCertificate** and broadcasts the certificate to all servers in the new group as a BFT request. Each server in the old group stops executing the protocol as soon as $t'+1$ servers in the new group acknowledge receipt of the **MsgCompletionCertificate**.

5. Servers in the new group carry out BFT to agree to some valid completion certificate out of all the completion certificates they have received.

6. At this point, it could be the case that only $t'+1$ non-faulty servers in the new group are able to reconstruct their shares; the other t' of the $2t'+1$ **MsgTransferSuccess** messages may have come from faulty servers. If any server T_k does not know its own share, it executes the Recover protocol (see below) to recover its share. Upon completion of Recover, T_k learns its share and broadcasts a **MsgTransferSuccess** message to all members of the old and new groups, which ensures that all honest servers in the old group eventually receives $2t'+1$ **MsgTransferSuccess**es, generate completion certificates, and delete their secret information.

The purpose of the **MsgTransferSuccess** and **MsgCompletionCertificate** messages in $\mathsf{Redist}^{\nu}_{+c}$ is twofold. They tell old nodes when it is safe to throw away secret information and terminate the protocol, and they allow new nodes to pin down

and agree upon a *particular* resharing that completed. Recall that $\mathsf{Redist}^{\nu}_{+c}$ executes multiple iterations, each of which is associated with different Q and R_k polynomials. Distinct iterations produce distinct and incompatible resharings of the secret, and it is possible for more than one of these iterations to reach the share transfer stage. When executing Recover and any other protocol that requires the secret shares, it is important that servers in the new group agree as to which set of shares to use. Running BFT to choose a valid completion certificate ensures that this consensus is met.

7.2.3.2 The Recover Protocol

The Recover protocol serves the same purpose for $\mathsf{Redist}^{\nu}_{+c}$ that **MsgMissingShare** messages served for Redist_0. In Redist_0, **MsgMissingShare** messages were sent whenever a server T_k in the new group could not participate in some future protocol because it did not know its share, and honest servers that *did* have their shares could simply reply to T_k with the **MsgNewPoly** messages they received from the old group. In Redist_0, we could guarantee that at least one honest server in the new group would receive **MsgNewPoly** messages from $t + 1$ honest servers in the old group, and since these messages were generated by honest parties, they contained the requisite information for every member of the new group to compute its share. In $\mathsf{Redist}^{\nu}_{+c}$, this is no longer true; of the $t' + 1$ **MsgNewPoly** messages received by $t' + 1$ honest servers in the new group, some of those messages may be from faulty servers in the old group that happen to contain valid information for the original recipients but invalid data for some honest T_k. Hence, simple **MsgMissingShare** messages do not work for $\mathsf{Redist}^{\nu}_{+c}$.

Each server T_k in the new group may initiate the Recover protocol to request

$\mathbf{ProposalR}_{i,j}$ = $\mathrm{Enc}_{PK_{\mathtt{e},j}}(R'_{i,k}(\alpha_j))$

$\mathbf{MsgRecover}_k$ =
[**recover**, **epoch#**, **sharing#**] $_{SK_{\mathtt{s},k}}$

$\mathbf{MsgProposalR}_l$ =
[**proposalr**, **epoch#**, **sharing#**,
$\mathbf{proposalr}_{l,1}, \ldots, \mathbf{proposalr}_{l,n}$, $\mathbf{commitment}_{R_{l,k}}$] $_{SK_{\mathtt{s},l}}$

MsgProposalSetR =
[**proposalsetr**, **epoch#**, **iteration#**, **sharing#**,
$\forall l \in \mathcal{S}_A$ $\mathbf{msgproposalr}_l$] $_{SK_{\mathtt{s},k}}$

$\mathbf{MsgProposalResponseR}_l$ =
[**proposalresponser**, **epoch#**, **iteration#**, **sharing#**,
AccusationSet, H(**msgproposalsetr**)] $_{SK_{\mathtt{s},l}}$

$\mathbf{MsgNewPolyR}_l$ =
[**newpolyr**, **epoch#**, **iteration#**, **sharing#**, $\mathrm{Enc}_{PK_{\mathtt{e},k}}(P'(l) + R'_k(l))$] $_{SK_{\mathtt{s},l}}$

Messages for Recover are similar to messages for Redist_0. However, there is no BFT subprotocol, view numbers are replaced with iteration numbers, and **MsgProposalR**s are much smaller than **MsgProposal**s because they only contain proposals and commitments for $R_{i,k}$ for one server T_k (the one whose share is to be recovered) instead of $3t + 1$ servers.

Figure 7-1: Message Formats for Recover

that the other servers generate its share for it. Honest servers will execute this protocol if they are missing their shares and servers in the old group have already received enough **MsgTransferSuccess** messages and terminated. Specifically, T_k starts Recover(k) when it receives a **MsgProposalResponse**

The messages used by Recover are shown in Figure 7-1, and the Recover(k) protocol operates as follows:

1. T_k broadcasts a **MsgRecover** message to all servers in the new group.

2. Each honest T_l in the new group creates a polynomial $R_{l,k}$ at random, except that $R_{l,k}(\beta_k) = 0$. For each $T_m \neq T_k$ in the new group, T_l generates a proposal

$R_{l,k}(\beta_m)$ and encrypts it with T_m's public key. It broadcasts a **MsgProposalR** message containing a vector of proposals, and Feldman commitments to the coefficients of $R_{l,k}$.

3. Upon receiving $t'+1$ proposals, T_k broadcasts a **MsgProposalSetR** message that specifies which proposals to use.

4. Each honest T_l *that knows its own share* sends a **MsgProposalResponseR** message that contains verifiable accusations against any proposals T_l deemed invalid.

5. Upon receiving $t'+1$ valid responses, T_k sends a **MsgResponseSetR** message that contains these responses.

6. Each recipient T_l that knows its own share validates the **MsgResponseSetR** by running the proposal selection algorithm on the responses. If the proposal selection algorithm terminates successfully, T_l uses the chosen set of proposals to generate a **MsgNewPolyR** message for T_k.

7. When T_k receives valid $t'+1$ valid **MsgNewPolyR** messages, it can reconstruct its share. This may not be possible immediately because the $t'+1$ responses T_k received in step 5 may contain responses from faulty nodes. T_k also awaits additional responses, and if it receives any before it is able to interpolate its share, it starts a new iteration of the protocol beginning at step 5. For each honest server T_l that has already sent a **MsgNewPolyR**, T_l will only send a new **MsgNewPolyR** if the new proposal set is a subset of the previous one.[2]

[2]This property ensures that running multiple iterations does not reveal any additional information, because the proposal set in each successive iteration differs from the previous one only in the removal of invalid proposals, which are already known to the adversary. Honest servers must verify that this property holds so that a dishonest T_k cannot coerce them into revealing information about

Recover is similar to Redist_0, but no agreement or view changes take place; T_k acts as the coordinator, and liveness is subject to T_k being correct. However, since the whole purpose of running the protocol is to recover T_k's share, T_k can only hurt itself by cheating. Because there is no agreement, T_k *can* coerce different sets of honest nodes to use different sets of proposals in computing their **MsgNewPolyR**s, but honest servers will validate their proposal sets to ensure that each contains at least one proposal from an honest server. Hence, the point in a **MsgNewPolyR** from an honest server will be random, and not known or controlled by the adversary.

Recover is required because the postcondition of Redist_{+c} is that $t'+1$ honest servers in the new group know their share, but the precondition of the protocol that does the resharing to the *following* epoch is that *all* honest servers know their shares. Recover bridges the gap by allowing honest servers in the new group to recover their shares if they didn't get them from Redist_{+c}, given only that $t'+1$ honest servers in the new group know their share.

Verifiable accusations are needed to make Recover work because Recover has to operate even if only $t'+1$ honest servers know their shares. An honest T_k must be able to choose a proposal set that is acceptable for $t'+1$ honest servers that know their shares, so in the worst case, the proposal set it chooses must satisfy *all* of the honest servers. Recall that the version of the Proposal Selection Algorithm that did not use verifiable accusations (Figure 5-2) could not guarantee that all honest servers were able to use the selected proposal set; for each response, the algorithm removes at most one of the accused servers, even if an honest server accuses multiple bad proposals. With verifiable accusations, it is possible to arrive at such a proposal set eventually. T_k may not arrive at this set right away, but as it receives responses from more servers, T_k continues to refine the set by eliminating bad proposals.

---

their secret shares.

7.2.4 Comparing $\mathsf{Redist}^{\nu}_{+c}$ and Redist_{+c}

We presented two protocols for increasing the group size by c because both $\mathsf{Redist}^{\nu}_{+c}$ (Section 7.2.3) and Redist_{+c} (Section 7.2.2) have advantages and drawbacks, which we discuss here.

Redist_{+c} has the undesirable property that the intermediate sharing is a t-out-of-$3t + c + 1$ sharing, and the number of faults permitted by this sharing is a smaller fraction of the total group size than in an optimal t-out-of-$3t+1$ sharing. For example, with $t = 8$, Redist_0 is secure when up to 32% of the shareholders are faulty. However, if we want to increase the threshold from 8 to 15, the intermediate group used by Redist_{+7} can have no more than 25% faulty shareholders. Theoretically speaking, $n = 3t + 1$ is provably optimal, and Redist_{+c} falls short. In practice, however, the impact of the intermediate group is minimal because the group is transitory. Servers in the intermediate group are needed only until second sharing completes, so the adversary's ability to corrupt nodes in that group is severely limited.

Redist_{+c} works for arbitrarily large values of c. In practice, however, it is good to keep c small relative to t because larger values of c imply that the fraction of allowable bad servers in the intermediate group is smaller. Large increases in the threshold can be performed incrementally, if needed. In real systems, changes in the threshold are motivated by changes in assumptions about the security of the underlying servers, so large variations are expected to be rare in any case.

By contrast, $\mathsf{Redist}^{\nu}_{+c}$ only supports increasing the threshold in increments of $c \leq t$, but it is more appealing theoretically because it uses the optimal group sizes with respect to the threshold (i.e., $n = 3t + 1$ and $n' = 3t' + 1$).

From a practical point of view, however, $\mathsf{Redist}^{\nu}_{+c}$ is less appealing than Redist_{+c} for efficiency reasons. The overhead of Redist_{+c} is comparable to that of simply run-

ning Redist_0 twice, whereas $\mathsf{Redist}^{\nu}_{+c}$ may require c iterations, and subsequently, up to $2t$ instances of $\mathsf{Recover}$ may be executed. Furthermore, in $\mathsf{Redist}^{\nu}_{+c}$, old shareholders cannot discard their secret information until $t' + 1$ servers in the new group acknowledge that they have received enough information to compute their shares. In Redist_{+c}, old shareholders can discard their secret information sooner, right when the second agreement operation completes.

The number of bits sent is the same asymptotically for all the protocols, since the dominant source of overhead is in broadcasting the proposals; however $\mathsf{Redist}^{\nu}_{+c}$ can require many more rounds of communication.

Chapter 8

Correctness

In this section, we prove the correctness of our scheme. In particular we want to establish that our scheme satisfies three properties: secrecy (the adversary never learns the secret), integrity (the secret remains intact from sharing to sharing), and liveness (the resharing protocol always terminates). We formally define what these terms mean in the following sections.

Sections 8.1, 8.2, and 8.3 establish that our protocol satisfies appropriate secrecy, integrity, and liveness properties, respectively. Subsequently, in Section 8.4, we show that all three of these properties are still satisfied when the verifiable accusations extension of Section 6.1 is used. Section 8.5 provides a similar justification for the hashing optimization of Section 6.2. The last three sections of this chapter, 8.6, 8.7, and 8.8, justify the correctness of our schemes for decreasing and increasing the threshold.

Our proofs rely on the system model and assumptions described in Chapter 3. To briefly review, we assume that the network is asynchronous and under control of the adversary. The system proceeds through a series of epochs, where each epoch

is associated with a different sharing of the secret, and our redistribution protocol is used to transition between epochs. A non-faulty server is in *local epoch* e if it has shares or secret keys associated with epoch e, and it discards this information when it leaves the epoch (see Definition 3.5). Liveness is additionally subject to the strong eventual delivery assumption. Informally, this is a slightly stronger version of the assertion that all messages repeatedly sent from correct servers are eventually delivered (see Definition 3.2 for details). The adversary is computationally bounded, active, and adaptive; it may corrupt up to t servers in local epoch e. The adversary learns all secret information on corrupted servers, and these servers may behave arbitrarily. To get the cryptographic tools we need, we assume the existence of a secure cryptographic hash function H and that the bilinear Diffie-Hellman problem is intractable [BF01].

8.1 Secrecy

In this section, we prove that the adversary cannot learn the secret except with negligible probability, given our system model. We establish that the exchange of proposals within the old group does not reveal any sensitive information and results in the selection of Q and R_k polynomials that have the requisite properties, provided that at least one proposal comes from an honest server. Then we show that transfer to the new group, which also contains corrupted servers, does not leak enough information to the adversary to reveal anything about the secret. Finally, we show that the Proposal Selection Algorithm 5-2 does indeed ensure that at least one of the selected proposals comes from an honest server, which implies that the previous theorems are true unconditionally.

First, consider the case of a single resharing, from epoch e to epoch $e + 1$, and

- For all corrupted old servers S_i, the points $P(\alpha_i)$
- For all $k \in [1, 3t+1]$ and all corrupted old servers S_i, the polynomials Q_i and $R_{i,k}$ (corrupted proposers)
- For all $i, k \in [1, 3t+1]$ and all corrupted old servers S_j, the point $Q_i(\alpha_j) + R_{i,k}(\alpha_j)$ (corrupted recipients of proposals)
- For all corrupted new servers T_k, the polynomial $P + Q + R_k$
- Feldman commitments to all coefficients of each polynomial P, Q_i, and $R_{i,k}$

Figure 8-1: Information Learned by the Adversary

suppose there are t faulty servers in epoch e and t faulty servers in epoch $e+1$. Recall that we have a degree-t Shamir sharing polynomial P in the old group, sharing polynomial $P' = P + Q$ in the new group. The resharing is based on the sum of polynomials Q and R_k, which are the sums of proposals from individual nodes in the old group, i.e., $Q = \sum_{i \in \mathcal{S}} Q_i$ and $R_k = \sum_{i \in \mathcal{S}} R_{i,k}$. These polynomials have particular properties, as described in Chapter 4. The information the adversary learns immediately from its corruptions is summarized in Figure 8-1.

We discount the impact of the Feldman commitments, which allows us to state theorems based on the remaining information in information-theoretic terms. The security of Feldman's scheme can be proven under the discrete logarithm assumption [Fel87], but the guarantees are probabilistic and applicable only to polynomially-bounded adversaries.

We begin by showing that the adversary cannot learn the polynomials Q or R_k or the secret s given that the proposal set used contains at least one honest proposer. Lemmas 8.1 and 8.2 show that the Q and R_k polynomials are random and independent of any information the adversary learns via corruptions in the old group, and Lemma 8.3 shows that the R_ks are pairwise independent of each other.

Of course these properties do not hold at points α_j such that S_j is corrupted, but there are only t such points.

The caveat that makes these lemmas nontrivial is that the adversary may generate its proposals after it knows t points from each of the honest proposals, corresponding to the t nodes in the old group it has corrupted. However, each of these proposals is generated from t independent degree-t polynomials of the form $Q_i+R_{i,k}$, so knowledge of t points still leaves one free variable. If the points on Q_i and $R_{i,k}$ were sent separately instead of as a sum, these lemmas would be false.

Lemma 8.1 *If $\mathcal{S}$, the set of proposals chosen by the proposal selection algorithm 5-2 contains at least one proposal from an uncorrupted server, then for any uncorrupted S_j, $R_k(\alpha_j)$ is random and independent of the $R_{i,k}$ polynomials from corrupted proposers.*

Proof. $R_k = \sum_{i\in\mathcal{S}} R_{i,k}$, and there exists an i such that $R_{i,k}$ was generated by an honest server. This $R_{i,k}$ will be generated randomly, so the sum R_k will be random, except that $R_k(\beta_k) = 0$ for $\beta_k \neq \alpha_j$. But this point is fixed and based on information the adversary can learn without executing the protocol. □

Lemma 8.2 *If $\mathcal{S}$, the set of proposals chosen by the proposal selection algorithm 5-2 contains at least one proposal from an uncorrupted server, then for any uncorrupted S_j, $Q(\alpha_j)$ is random and independent of the Q_i polynomials from corrupted proposers.*

Proof. $Q = \sum_{i\in\mathcal{S}} Q_i$, and there exists an i such that Q_i was generated by an honest server. This Q_i will be generated randomly. ($Q_i(0) = 0$, but $\alpha_j \neq 0$.) The adversary may generate the other proposals after the honest proposals have been sent, but it can only do so with respect to points it knows, i.e., t points on each $Q_i + R_{i,k}$. But

these are pairwise-independent degree-t polynomials, so knowledge of t points still leaves a free coefficient. Hence the sum Q will be random. □

Lemma 8.3 *If $\mathcal{S}$, the set of proposals chosen by the proposal selection algorithm 5-2 contains at least one proposal from an uncorrupted server, then for any $k' \neq k$ and uncorrupted S_j, $R_k(\alpha_j)$ and $R_{k'}(\alpha_j)$ are random and independent.*

Proof. Each honest proposer S_i will construct its proposals such that each pair $(R_{i,k}, R_{i,k'})$ is random and independent, so this follows immediately from Lemma 8.1. □

The first three lemmas focused on corruptions in the old group. The following lemma shows that when we take into consideration information learned from corruptions in both the old and new groups, the adversary still learns nothing about $Q + R_k$. Thus, the points sent to the new group don't invalidate Lemmas 8.1–8.3.

Lemma 8.4 *If $\mathcal{S}$, the set of proposals chosen by the proposal selection algorithm 5-2 contains at least one proposal from an uncorrupted server, then for any uncorrupted S_j, $Q(\alpha_j) + R_k(\alpha_j)$ is independent of the information the adversary learns (Figure 8-1.)*

Proof. $Q(\alpha_j) + R_k(\alpha_j)$ is independent of the proposals by Lemma 8.1, and independent of each $Q(\alpha_j) + R_{k'}(\alpha_j)$ for $k' \neq k$ by Lemma 8.3. S_i sends $P(\alpha_i) + Q(\alpha_i) + R_k(\alpha_i)$ to T_k, but even if T_k is corrupted, this reveals nothing because $P(\alpha_i)$ is secret and independent of Q and R_k. □

Now we are ready to prove the main theorem, which states that the execution of the protocol, including the combined view of the adversary of the old and new sharings, does not reveal anything about the secret.

Theorem 8.1 *If $\mathcal{S}$, the set of proposals chosen by the proposal selection algorithm 5-2 contains at least one proposal from an uncorrupted server, then the information the adversary learns (Figure 8-1) is independent of the secret $s = P(0)$.*

Proof. The adversary is given t points on P and t *distinct* points on $P + Q$ (since we assume $\alpha_i \neq \beta_k$ for all i, k). None of these points is at $x = 0$. Q is random, except at 0, and by Lemma 8.2, Q is independent of the information the adversary learns, so these two sets of points are random and indepenent of each other. The t points on P are independent of $P(0)$ because P is of degree t and hence has at least one free coefficient, and similarly for $P + Q$. □

The information revealed by our protocol is similar to what is revealed in Herzberg et al.'s [HJKY95] proactive refresh protocol, plus $3t + 1$ instances of their recovery protocol, except that we have combined the steps in such a way that the adversary cannot learn the polynomial Q. Recall that Q is the polynomial that makes it possible to map between shares in the old group and shares in the new group. Herzberg et al. do not require that Q remain secret if there are a full t corruptions while the resharing protocol executes because the old and new groups in their protocol are identical. Hence, a full t corruptions during the transfer process from epoch e to $e + 1$ implies that the same servers are corrupted in both epochs, and thus Q reveals nothing new. Pages 49–77 of Jarecki's master's thesis [Jar95] contain more comprehensive proofs of secrecy for the composition of their two protocols. Their work also shows that a complete "view" of the adversary's information similar to Figure 8-1 that includes the Feldman commitments does not reveal any information, and they also prove secrecy for a modified scheme using Pedersen commitments.

Our theorems are all predicated on the proposal set $\mathcal{S}$ containing at least one honest proposer. We prove that the proposal selection algorithm guarantees this

property.

Theorem 8.2 ***PSA Secrecy.*** *If $\mathcal{S}$ is the set of proposals generated by the proposal selection algorithm (Figure 5-2), then $\mathcal{S}$ contains a proposal from at least one honest server.*

Proof. Let $\mathcal{B}$ denote the set of faulty servers whose proposals appear in the initial proposal set, and let $b = |\mathcal{B}|$. Clearly $b \leq t$. We split the responses processed by the algorithm into three categories:

1. Responses that are satisfied with the current proposal set at the point when they are processed do not result in the removal of any proposals from the set.

2. Consider responses generated by good servers that accuse bad servers in the current proposal set (which must also be in $\mathcal{B}$) *and* responses generated by bad servers in $\mathcal{B}$. These accusations result in the removal of one proposal from a bad server and at most one proposal from a good server. There are at most b such responses.

3. Now consider responses generated by bad servers not in $\mathcal{B}$. Each such response results in the removal of at most one good proposal and no bad proposals. There are at most $t - b$ such responses.

Only a response in categories (2) and (3) result in the removal of a good proposal, and there are at most t such responses. The initial proposal set consists of $2t + 1$ proposals, of which $2t + 1 - b$ are from honest servers, so $2t + 1 - b - t = t + 1 - b \geq 1$ proposals from honest servers remain. □

Remark 8.3 Notice that in some cases, the number of proposals removed that are from good servers may exceed the number of proposals removed that are from bad

servers, by up to $t-b$. However, this only happens when not all of the proposals from bad servers were included in the initial proposal set, and hence the initial proposal set contains more than $t+1$ good proposals.

Corollary 8.4 *The information the adversary learns from the execution of* Redist_0 *(Figure 8-1) is independent of the secret* $s = P(0)$.

Proof. Follows directly from Theorems 8.1 and 8.2. □

8.2 Integrity

In this section, we prove that the secret remains intact from epoch to epoch. In particular, it should always be the case that after a resharing protocol completes,[1] any honest node T_k in the new epoch is able to compute its share $P'(\beta_k)$. This implicitly assumes that honest servers can identify and discard bogus information from corrupted servers, which is true as a consequence of our use of forward-secure signatures and our verification scheme based on Feldman's scheme, described in Section 4.2.3. The focus of our proofs is on ensuring that there is enough *good* information to establish the new sharing.

First, we show that the Proposal Selection Algorithm (Figure 5-2) produces a set of proposals that is acceptable to at least $t+1$ honest servers. This will allow us to establish that at least $t+1$ servers perform the share transfer to the new group.

Theorem 8.5 ***PSA Admissibility.*** *Upon termination of the proposal selection algorithm (Figure 5-2), at least* $t+1$ *non-faulty servers are satisfied with the proposals in the* ***props*** *set.*

[1] We assume here that the protocol does complete and defer the issue of liveness to Section 8.3.

Proof. When a response contains an accusation against the current set (i.e., the sender is unsatisfied), either the accuser or the accusee is faulty, and we add both to the **rejected** set and increment d. Hence at any point, at least d faulty servers are in the **rejected** set. No server can be both satisfied and rejected at the same time, so at most $t - d$ faulty servers are in the **satisfied** set. The algorithm terminates when $|\textbf{satisfied}| = 2t + 1 - d$, so at least $t + 1$ of the servers in the **satisfied** set are non-faulty. □

Next, we show that *if* the protocol terminates successfully, then all honest members of the new group have their shares of the secret. Together with Theorem 8.9, this establishes that the share transfer always succeeds.

Theorem 8.6 *If $2t + 1$ honest servers in epoch e that know their shares complete* Redist_0*, every honest server T_k in epoch $e + 1$ will be able to compute its share.*

Proof. Of the $2t + 1$ servers in epoch e that are honest, at least $t + 1$ are satisfied with the proposal set chosen by the proposal selection algorithm by Theorem 8.5, and hence each such S_i can compute $P(\alpha_i) + Q(\alpha_i) + R_k(\alpha_i)$ for all k. These servers complete the protocol when they receive $t + 1$ acknowledgements from servers in the new epoch, of which at least 1 is non-faulty. If T_k is not one of these $t + 1$ servers, it may query all of the non-faulty new servers that did receive the info from the old group to obtain the points that the old group sent, so we may assume T_k has access to this information. Hence T_k will receive $t + 1$ distinct points on $P + Q + R_k$ and will be able to interpolate this polynomial and evaluate $P(\beta_k) + Q(\beta_k) + R_l(\beta_k) = P(\beta_k) + Q(\beta_k) = P'(\beta_k)$, which is T_k's share. □

Corollary 8.7 *In any epoch e, each honest server is able to obtain its share.*

Proof. We assume that when the system is first initialized (i.e., in epoch 0), all honest servers know their share. Thus the corollary holds by Theorem 8.6 and induction on e. □

8.3 Liveness

We need to establish that Redist_0 terminates successfully, given our limitations on corruption and the strong eventual delivery assumption (Definition 3.2). The assumptions needed by this section are stronger than the ones needed in Sections 8.1 and 8.2 because it no longer suffices to say that the adversary has complete control over the network. In order to prove that Redist_0 eventually completes (or that any protocol can make progress at all), we must assume that messages repeatedly sent between honest servers eventually reach their destinations (eventual delivery). Strong eventual delivery is essentially a slightly stronger version of eventual delivery; see Chapter 3 for the formal definition.

First, we show that the proposal selection algorithm terminates. Recall that the coordinator runs proposal selection after it broadcasts a list of $2t+1$ proposals from distinct servers and begins to receive **MsgProposalResponse**s from those servers. Agreement and share transfer cannot begin until the algorithm determines that it has seen enough responses.

Theorem 8.8 ***PSA termination.*** *If the initial proposal set contains $2t+1$ proposals, each from distinct servers, then the proposal selection algorithm (Figure 5-2) terminates after processing at most $2t+1$ responses from distinct non-faulty servers.*

Proof. When a response contains an accusation against the current set, either the accuser or the accusee is faulty, and we add both to the **rejected** set and increment

d. Hence d represents an upper bound on the number of non-faulty servers in the **rejected** set. At any point, each server whose response has been processed by the algorithm so far is in the **satisfied** set or the **rejected** set. Therefore, upon processing $2t + 1$ responses from honest servers, $|\textbf{satisfied}| \geq 2t + 1 - d$, which is the termination condition for the algorithm. □

Next, we show that when we invoke the BFT agreement protocol [CL02] as a sub-protocol in step (2d) of our protocol, BFT terminates. To ensure that the agreement eventually takes place, we assume that when our protocol is invoked, it is invoked on all non-faulty servers. This assumption ensures that if the coordinator is faulty, honest servers will eventually notice the lack of progress and initiate a view change. In BFT, honest clients broadcast their requests to all servers to ensure that view changes will happen if needed. In Redist_0, a "client request" is simply a request to perform a resharing to the next group, and we assume that clients of our system do the same as clients in BFT.[2]

We treat BFT as a black box except in one case; specifically, when a view change occurs, there is an upcall from the BFT library back into our protocol. Hence, liveness follows from liveness of BFT, provided that we can demonstrate that the upcall always returns. When a view change occurs, the new coordinator in Redist_0 may have to do more work before it can enter the PRE-PREPARE phase of the next view. In particular, it may have to re-execute steps (2a) through (2c) of Redist_0, in which it asks other nodes to vote on the $2t + 1$ proposals it has collected. The new coordinator will always get proposals from the $2t + 1$ non-faulty servers under the eventual delivery assumption, and the proposal selection process will terminate by

[2]The client in our system need not be a physical machine. The start of the transition to the next epoch could be initiated by another agreement operation, by a human administrator, or by a separate protocol responsible for managing the system configuration.

Theorem 8.8. Subsequently the new coordinator can begin agreement in the new view, returning to BFT. Hence, the additional steps required at the start of each view do not invalidate BFT's liveness guarantees.

Now we are ready to prove that the entire protocol Redist_0 terminates.

Theorem 8.9 *If at least $2t+1$ honest servers in epoch e initiate Redist_0, the coordinator is honest, and the proposal selection algorithm terminates, then Redist_0 will terminate at all honest servers in epoch e.*

Proof. By Corollary 8.7, all honest servers in epoch e have their shares.

In step (1) of Redist_0, the coordinator awaits $2t+1$ well-formed **MsgResponses**, which it will eventually receive from honest servers by the eventual delivery assumption and the assumption that there are at least $2t+1$ non-faulty servers. It broadcasts these proposals and collects responses, which are input to the proposal selection algorithm. PSA terminates by Theorem 8.8, and when it does, the coordinator will run the BFT protocol. The $2t+1$ honest servers will accept the coordinator's proposal set and be complicit in the agreement; hence BFT will terminate as shown in [CL02], and all honest servers will learn the result of this agreement.

By Theorem 8.5, at least $t+1$ of the honest servers will be able to use the chosen proposal set and generate **MsgNewPoly** messages for the new group. The honest servers that can use the proposal set terminate the protocol upon receipt of $t+1$ acknowledgements from the new group, which they will receive since $2t+1$ members of the new group are honest. Honest servers that cannot use the proposal set terminate immediately after completing BFT. □

8.4 Verifiable Accusations

Verifiable accusations, described in Section 6.1, are an optimization to the basic Redist_0 protocol. In this section, we show that the changes required in order to support verifiable accusations preserve secrecy, integrity, and liveness.

There are two significant changes. First, **MsgProposalResponse** messages contain accusations, which reveal additional information, so we need to show that this change does not impact secrecy. Second, the verifiable accusations option uses a different Proposal Selection Algorithm shown in Figure 6-6, rather than the one from Figure 5-2. Furthermore, the coordinator waits for only $t + 1$ proposals instead of $2t + 1$, so fewer proposals are available at the start of the algorithm. We will show that Theorems 8.2, 8.5, and 8.8 also hold for the modified algorithm. In fact, the proofs are simpler than in the original protocol.

The modified message formats for verifiable accusations were shown in Figure 6-5. In **MsgPropsalResponse**, the **BadSet**, which merely enumerated the bad proposals, has been replaced with **AccusationSet**, which contains a list of encryption keys. When S_j accuses S_i, it includes the encryption key S_i used to encrypt its proposals intended for S_j; this means that either S_i or S_j is faulty. If S_i is faulty and S_j is honest, then revealing this key only allows others to decrypt messages encrypted using S_j's public key parameterized on identity i and epoch e, since we are using the forward-secure identity-based encryption scheme of Section 6.1.4. Since S_i is faulty, only faulty parties will use these encryption parameters, so no information from honest servers is revealed. If the accuser S_j is faulty, on the other hand, then revealing its decryption key or the contents of messages sent to it does nothing, since the adversary already has this information anyway by virtue of S_j being corrupted.

These next theorems state that the modified Proposal Selection Algorithm pre-

serves the same properties as the original algorithm.

Theorem 8.10 ***PSA Secrecy (Verifiable Accusations).*** *If $\mathcal{S}$ is the set of proposals generated by the proposal selection algorithm (Figure 6-6), then $\mathcal{S}$ contains at least one proposal from an honest server.*

Proof. The algorithm is initialized with proposals from $t + 1$ distinct servers. Of these servers, at most t can be faulty, and hence there is at least one proposal in the initial set from a non-faulty server. The algorithm only removes a proposal if there is a *valid* accusation against it (meaning the proposal is bad) or if the proposer generates an invalid accusation against another server. Non-faulty servers generate neither bad proposals nor bad accusations, so the final set contains at least one proposal from a non-faulty server. □

Theorem 8.11 ***PSA Admissibility (Verifiable Accusations).*** *Upon termination of the proposal selection algorithm (Figure 6-6), at least $t+1$ non-faulty servers are satisfied with the proposals in the* ***props*** *set.*

Proof. Replies from non-faulty servers will include valid accusations against all of the proposals they are not satisfied with, and all such proposals will be removed. Hence all non-faulty servers whose replies are processed will be satisfied with the chosen set. The algorithm terminates after processing $2t + 1$ responses, of which at least $t + 1$ must be from non-faulty servers. □

Theorem 8.12 ***PSA termination (Verifiable Accusations).*** *The proposal selection algorithm (Figure 6-6) terminates after processing at most $2t+1$ responses from distinct non-faulty servers.*

Proof. The algorithm terminates after receiving $2t + 1$ replies. At most t of the $3t + 1$ servers are faulty, and by the eventual delivery assumption (Definition 3.3), replies from all of the non-faulty servers will eventually be received. □

8.5 Reducing Load on the Coordinator

Section 6.2 specified a modification to Redist_0 that reduces the load on the coordinator by using hashes. Here, we argue that the hashing optimization preserves liveness, integrity, and secrecy. Recall that until BFT is invoked, the principal job of the coordinator is to provide the group with a consistent view of a set of proposals collected from $2t + 1$ distinct servers and process votes on those proposals. With the optimization, the coordinator sends a list of cryptographic hashes of the proposals instead of the proposals themselves. Honest servers that are missing proposals in the set can request them from the coordinator. Clearly this change does not impact secrecy, and we argue that it does not affect integrity or liveness either.

Integrity. We require that the hash function H be collision-resistant, so that a faulty coordinator even in concert with other faulty servers cannot feasibly produce a hash value that corresponds to multiple proposals. Therefore, a **MsgProposalSet** from the coordinator containing hashes along with proposals that match those hashes is just as good as a **MsgProposalSet** generated according to the original Redist_0.

Liveness. If the coordinator is dishonest, it may neglect to send missing proposals, which may make it impossible for non-faulty servers to agree to a set of proposals. However, in this case the non-faulty servers will initiate a view change and elect a new coordinator.

8.6 Decreasing the Threshold

The protocol for decreasing the threshold (Section 7.1) is based on the addition of virtual servers, which are simulated locally by each real server. If the initial threshold was t and the new threshold is $t' = t - v$, v virtual servers are added.

Secrecy. Since each real server simulates each virtual server, the adversary knows everything that each virtual server knows. Thus, in the context of secrecy, the adversary has secret information from $t' + v$ servers. But $t' + v = t$, and the reduced scheme still uses degree-t polynomials, so all the theorems from Section 8.1 are still applicable.

Integrity. Virtual servers do not generate proposals, but they do participate in share transfer. The shares of virtual servers are public, and they use their shares to generate **MsgNewPoly** messages, based on the chosen proposal set. Each non-faulty physical server sends the **MsgNewPoly** "from" each virtual server to the new group, and members of the new group use the commitments to verify the accuracy of the simulation. (In practice, physical servers would likely coordinate with each other to avoid duplication of effort.)

Liveness. Within the old group, servers never wait for responses from virtual servers, since the virtual servers are simulated locally. BFT and proposal selection are run with respect to the reduced threshold t' and physical group size $n' = 3t' + 1$, so liveness is preserved as per the theorems of Section 8.3.

8.7 The Two-Step Redist_{+c}

Section 7.2.2 discussed protocol Redist_{+c}, which increases the threshold by c by performing two resharings. This section establishes that Redist_{+c} is correct. The first

resharing increases the group size to $3t + c + 1$ without changing the threshold, and the second increases the threshold to $t' = t + c$. Only t faulty nodes are allowed in the intermediate group.

The first resharing uses additional R_k polynomials within each proposal, but the number of proposals, the structure of the first two steps of the protocol, and the number of messages exchanged remain the same. In the share transfer stage, additional **MsgNewPoly** messages are sent to account for the larger group size, but old servers still only await $t + 1$ acknowledgements, since the threshold in the new group is still t. Since we haven't changed the number of faults we allow or the number of responses we need to wait for in the protocol at this point, all of the correctness results we derived for Redist_0 still apply.

The second transfer, from the temporary group to the new group of size $3t' + 1$, uses degree-t' polynomials so that when share transfer occurs, t' corrupted servers in the new group cannot compromise secrecy. The Proposal Selection Algorithm waits for c extra responses, which ensures that c additional honest servers are able to use the chosen proposal set. Hence, there are $t' + 1$ non-faulty servers that are able to perform the share transfer involving a degree-t' polynomial, so integrity is preserved. Liveness is preserved because the temporary group has c additional non-faulty servers in it, so we can afford to wait for another c responses, and the proposal selection process will eventually terminate.

8.8 The One-Step $\mathsf{Redist}^{\nu}_{+c}$

The threshold increase protocol $\mathsf{Redist}^{\nu}_{+c}$, described in Section 7.2.3, uses proposal polynomials of degree $t' = t + c$ to increase the degree of the sharing polynomial in the new group. It uses verifiable accusations to ensure that all servers in the new

group are able to receive their shares. In this section, we prove that $\mathsf{Redist}^{\nu}_{+c}$ satisfies the secrecy, integrity, and liveness properties we established for Redist_0. There are several significant differences in the protocol, including multiple iterations of proposal selection and the addition of a Recover protocol within the new group.

8.8.1 Secrecy Protection of $\mathsf{Redist}^{\nu}_{+c}$

First we consider $\mathsf{Redist}^{\nu}_{+c}$ by itself and establish that $\mathsf{Redist}^{\nu}_{+c}$ has the same secrecy properties as described in Section 8.1. Theorem 8.13 is analogous to Theorem 8.1 for Redist_0, but applies only to a single iteration of $\mathsf{Redist}^{\nu}_{+c}$. Theorem 8.14 shows how the fact that $\mathsf{Redist}^{\nu}_{+c}$ may require multiple iterations doesn't affect secrecy.

Theorem 8.13 *If $\mathcal{S}$, the set of proposals chosen by the proposal selection algorithm 5-2 contains at least one proposal from an uncorrupted server, then the information the adversary learns in any given iteration of $\mathsf{Redist}^{\nu}_{+c}$ is independent of the secret $s = P(0)$.*

Proof. The adversary is given t points on P and t' *distinct* points on $P + Q$, and none of these points are at $x = 0$. The t points on P are independent of $P(0)$ because P is of degree t and hence has at least one free coefficient. Similarly, the t' points on $P + Q$ are independent of $P(0) + Q(0)$ because $P + Q$ is of degree t'. The rest of the proof proceeds as in Theorem 8.1. □

Theorem 8.14 *The information the adversary learns over all iterations of $\mathsf{Redist}^{\nu}_{+c}$ is the same as the information learned in iteration 0.*

Proof. Each subsequent iteration of $\mathsf{Redist}^{\nu}_{+c}$ differs from the previous only in that the proposal set is made smaller by the removal of provably bad proposals. These

proposals were generated by faulty servers, and hence are known to the adversary already. Although the adversary can collect share transfer messages for different iterations, the difference between the polynomials used in these messages is a sum of invalid proposals. Hence the adversary learns nothing new. □

Corollary 8.15 *The information the adversary learns from the execution of* $\mathsf{Redist}^{\nu}_{+c}$ *is independent of the secret* $s = P(0)$.

Proof. Follows from Theorems 8.2, 8.13 and 8.14. □

8.8.2 Liveness of $\mathsf{Redist}^{\nu}_{+c}$

The interaction between the view change mechanism and iterations is awkward, but together these two features guarantee that $\mathsf{Redist}^{\nu}_{+c}$ always terminates. Recall that views happen sequentially, and the system as a whole operates in a single view at any given time, even though individual servers may be "slow" and have to be sent a view change certificate so that they are aware of the current view. Iterations happen in parallel, but as we will show, there are at most $c + 1$ iterations. Note that if the protocol is executing in some view v with ι parallel iterations and a view change occurs, the new coordinator may start by executing iteration 0. The number of parallel iterations being run at a given time reflects the number of times the *current* coordinator has received a new accusation against its current proposal set.

First, we prove a lemma that says that in any iteration, either $\mathsf{Redist}^{\nu}_{+c}$ will be able to terminate in that iteration, or it will proceed to the next iteration. The lemma is presented with respect to "some view" of each iteration because it is difficult to say anything about views in which the coordinator is faulty and behaves badly. However, when that happens, we know that the non-faulty servers will eventually initiate a view change, and in some future view, the coordinator will behave correctly.

Recall that protocol messages contain an iteration number field and a view number field, and messages for different iterations are distinct, as are messages from different views. Proposal sets are associated with iterations, and hence it may be understood that when we speak of the messages received in a particular iteration, all of those messages that come from honest servers are based on the same proposal set.

Lemma 8.5 *In some view of any iteration of* $\mathsf{Redist}^{\nu}_{+c}$*, once BFT has completed, one of two things is true.*

1. *Each server in the new group will eventually receive* $t'+1$ *valid* ***MsgNewPoly*** *messages from distinct servers in the old group.*

2. *The coordinator will eventually receive a* ***MsgProposalResponse*** *from an honest server that accuses a proposal in the current proposal set.*

Proof. Suppose we are in some view v with an honest coordinator, and a view change does not occur. If (2) does not occur, then all honest servers that are unable to use the current proposal set have responded, so all $2t+1$ honest old servers are able to use the current proposal set. Therefore, once agreement completes, at least $2t+1$ valid **MsgNewPoly** messages are sent to every member of the new group. Since $c \leq t$, we have $t'+1 = t+c+1 \leq 2t+1$, so case (1) occurs. Such a view v exists under the strong eventual delivery assumption (Definition 3.2) because view changes will continue to occur until the coordinator is honest and the view change timeout is long enough to allow all non-faulty nodes to respond. □

Next, we show that for iteration c, only case (1) of Lemma 8.5 is possible.

Lemma 8.6 *In some view of iteration c (counting from 0) or some earlier iteration, each server in the new group will eventually receive $t' + 1$* ***MsgNewPoly*** *messages from distinct servers in the old group.*

Proof. The current coordinator in any view of iteration c is also executing in iterations $0, \ldots, c-1$ in parallel, if that coordinator is honest. Suppose servers in the new group never receive **MsgNewPoly** messages from distinct servers in the old group in any iteration less than c. Then by Lemma 8.5, the current coordinator received an additional accusation against its current proposal set in each of these earlier iterations. It began with $2t + 1$ **MsgProposalResponse**s in iteration 0, so after c iterations it has $2t+c+1$ responses, of which $t+c+1 = t'+1$ came from honest servers. If the coordinator is honest, these servers are all able to use the resulting proposal set, so each such server will send a **MsgNewPoly** message to all servers in the new group. Furthermore, the view change protocol guarantees that the protocol will eventually proceed to some view in which the coordinator is honest. □

Theorem 8.16 *After starting at most $c + 1$ iterations, $\mathsf{Redist}^{\nu}_{+c}$ terminates for all honest members of the old group.*

Proof. The two cases of Lemma 8.5 indicate that in each iteration, either each server in the new group receives $t' + 1$ **MsgNewPoly** messages from the old group, or the coordinator receives a new accusation, which prompts it to start a new iteration. Lemma 8.6 states that for iteration c, only the former case is possible. When honest servers in the new group receive $t' + 1$ **MsgNewPoly** messages, they broadcast **MsgTransferSuccess** to all honest servers in the old group. There are at least $2t' + 1$ honest servers in the new group, so all honest servers in the old group receive $2t' + 1$ **MsgTransferSuccess** messages. When this happens, the servers in the old

group send **MsgCompletionCertificate**s to the new group, and the new group runs BFT to agree to some valid completion certificate. BFT terminates as shown by Castro et al. [CL99]. □

8.8.3 Integrity Preservation for $\mathsf{Redist}^{\nu}_{+c}$ and Recover

In Section 8.2, we showed that after Redist_0 terminates, any node T_k in the new group that didn't have its share could simply ask all other members of the new group for the **MsgNewPoly** messages received during the redistribution. The responses T_k received would always suffice for T_k to recover its share. When $\mathsf{Redist}^{\nu}_{+c}$ terminates, the guarantees we are able to make about the integrity of the secret are weaker than for Redist_0, and an additional Recover protocol may be needed to ensure that T_k can recover its share.

Lemma 8.7 *After* $\mathsf{Redist}^{\nu}_{+c}$ *terminates, at least* $t'+1$ *honest servers in the new group have their shares.*

Proof. Servers in the old group only terminate the protocol upon receipt of $2t'+1$ **MsgTransferSuccess** messages, and $t'+1$ of these messages must come from honest servers. Honest servers in the new group only generate these messages when they receive $t'+1$ **MsgNewPoly** messages that are valid for them, and $t'+1$ valid points suffice for these servers to interpolate their shares. □

Lemma 8.8 *If* T_k *is honest and broadcasts a* ***MsgRecover*** *message, and at least* $t'+1$ *nodes know their shares, then* T_k *is eventually able to recover its share.*

Proof. Recover terminates when T_k collects $t'+1$ valid **MsgNewPolyR** messages from distinct nodes. Each such message contains a point on the same degree-t'

polynomial, which T_k can interpolate using the $t'+1$ points and evaluate at β_k to obtain its share. □

Theorem 8.17 *After* $\mathsf{Redist}^{\nu}_{+c}$ *terminates, all honest servers in the new group have their shares.*

Proof. Implied by Lemmas 8.7 and 8.8. □

8.8.4 Liveness and Secrecy for Recover

Recover is needed in combination with $\mathsf{Redist}^{\nu}_{+c}$ to ensure that we are able to guarantee integrity, i.e., all honest servers in the new group are able to reconstruct their shares. However, we must also show that the addition of the Recover protocol does not cause us to sacrifice secrecy, and that Recover eventually terminates.

Secrecy preservation for Recover is straightforward to establish and follows a similar argument to the one used for Redist_0. When $\mathsf{Recover}(k)$ is executed, the only node to receive information based on the sharing of the secret is T_k. The only significant concern is that if T_k is dishonest, it might try to coerce honest servers into revealing information about their shares.

Lemma 8.9 *If* T_k *is faulty,* T_l *is honest, and Feldman's scheme is secure, then* $R'_k(\beta_l)$ *is independent of the information the adversary learns from running* $\mathsf{Recover}(k)$.

Proof. Honest nodes T_l receive a set of $t'+1$ proposal polynomials $R_{i,k}$ along with $t'+1$ responses that use verifiable accusations to eliminate up to t' bad proposals. They compute R'_k as the sum of the remaining proposals, at least one of which comes from an honest server and is unknown by the adversary; therefore, R'_k is random and independent of the adversary's proposals. From the servers it has corrupted,

including T_k, the adversary learns t' points on R'_k, but one of these points is $R_k(\beta_k) = 0$, which is already public knowledge. R_k has degree t', so it has $t' + 1$ degrees of freedom; hence, $R'_k(\beta_l)$ could be arbitrary based on the information available to the adversary. □

Theorem 8.18 *The adversary learns nothing from running* Recover(k) *that it could not learn without running the protocol.*

Proof. By Lemma 8.9, if T_k is faulty and T_l is honest, then the adversary learns nothing about $R'_k(\beta_l)$. Therefore, when all honest servers T_l send $P'(\beta_l) + R'_k(\beta_l)$, all this allows the adversary to do is interpolate $P' + R'_k$ and compute $P'(\beta_k) + R'_k(\beta_k)$, i.e., T_k's share, which the adversary already knows. If, on the other hand, T_k is honest, then the points it receives are never revealed to the adversary anyway. □

In the context of Recover(k), liveness means that T_k eventually gets its share and can stop executing the protocol. We can only prove liveness of Recover(k) in the case where T_k is honest. However, this restriction is acceptable because the entire purpose of executing Recover(k) is to allow T_k to recover its own share, and we only care that *non-faulty* servers have their shares. If a faulty T_k runs Recover, that instance may never terminate. Eventually, the next resharing will take place despite T_k being faulty, and honest servers will stop participating in Recover(T_k) when they discard their secret information.

Theorem 8.19 *If T_k is non-faulty and at least $t' + 1$ honest servers know their shares, then T_k eventually acquires its share.*

Proof. T_k broadcasts a **MsgRecover** message and collects $t' + 1$ **MsgProposalR**s, which it can clearly afford to wait for, since there are $2t' + 1$ honest servers in

the new group. Then T_k broadcasts a **MsgProposalSetR** message and waits for $t' + 1$ responses from honest servers that know their shares before broadcasting a **MsgResponseSetR**. There are $t' + 1$ such servers by assumption, so T_k will eventually hear from them all. The first $t'+1$ responses may not all be from honest servers, but T_k broadcasts a revised **MsgResponseSetR** message each time it receives an additional response. Once T_k has sent the **MsgResponseSetR** that reflects responses from the $t' + 1$ honest servers that know their shares, those $t' + 1$ servers will each be able to send a **MsgNewPolyR** to T_k, and T_k terminates when it receives $t' + 1$ valid **MsgNewPolyR** messages. □

Chapter 9

Performance

We analyze the performance of our protocol and compare it to other schemes. Prior schemes that support secret redistribution to a new group require exponential communication in the worst case, whereas our protocol requires only a polynomial amount of communication. We also consider related schemes for ordinary proactive secret sharing that have performance characteristics similar to ours, but that are more limited in what they can do.

We measure performance principally in the number of bytes that must be successfully exchanged by each honest party in order for the protocol to complete; we do not count traffic generated by dishonest parties, as this could be arbitrary. In protocols such as ours in which a particular node (e.g., the coordinator) represents a bottleneck, we analyze the performance with respect to that node. The number of bytes exchanged is a better metric than, say, number of messages, because it is not subject to gimmicks that trade number of messages for message size. It is also easier to analyze than metrics based on time because the adversary can influence the amount of time required for the protocol to complete by delaying network traffic.

We measure performance under three different scenarios:

Optimistic case. We assume the number of faults is a small constant, not proportional to t. We expect this to be the common case, since building a practical Byzantine-fault tolerant system that is reliable with overwhelming probability requires that the individual machines be relatively reliable. To see why this is true, suppose the probability of an individual machine failing is close to 1/3. Then regardless of the group size n, the probability that the number of faults is greater than $\lfloor n/3 \rfloor$ will always be high. Rodrigues et al. [Rod] found that in order for the system as a whole to remain secure over a large number of epochs with 99.999% probability, a reasonable group size, and independent failures, individual machines must be on the order of 99% reliable.

Average case with non-adaptive adversary. Assume that the adversary controls the maximum number of faulty replicas in each group and behaves arbitrarily badly. However, the adversary is not adaptive and must choose which nodes to corrupt in advance of the execution of the protocol. We make average-case assumptions about the number of coordinators that must be chosen until we have selected a non-faulty coordinator, the number of subsets of the group that must be chosen until we have chosen an entirely non-faulty subset, and so on. This scenario is meant to reflect the performance of real-world systems *operating in the presence of malicious faults*.

Worst case. We assume an *adaptive* adversary that controls the maximum number of faulty replicas in each group and behaves arbitrarily badly. Furthermore we make worst-case assumptions about coordinator and subgroup selection. This scenario is essentially the worst case permitted by our threat model, but is

	Optimistic	Average	Pessimistic
Our scheme	$O(t^3)$	$O(t^4)$	$O(t^4)$
Zhou et al.	$O(2^t)$	$O(2^t)$	$O(2^t)$
Cachin et al.	$O(t^4)$	$O(t^4)$	

Figure 9-1: Protocol Performance Comparison: Bytes Sent by Each Honest Server

not particularly interesting for comparing protocols; rather it lets us define a standard to be met. We believe that in order for a protocol to be *practical*, it must have tolerable performance in the presence of malicious faults. Some protocols that may perform well under optimistic assumptions fail this test by allowing an adversary to force them into an exponential number of rounds of communication. Furthermore, users of proactive secret sharing systems need to adjust the interval between resharings to reflect their assumptions about the rate at which nodes are corrupted and worst-plausible-case assumptions about protocol execution time.

Figures 9-1 and 9-2 present asymptotic performance results for our system as well as results for Zhou et al.'s mobile proactive secret sharing scheme [ZSvR05]. For purposes of comparison, we also list the overhead of Cachin et al.'s scheme [CKLS02], which is the best *non-mobile* proactive secret sharing scheme for asynchronous networks. More detailed analysis of the performance of these schemes can be found in Sections 9.1, 9.2, and 9.3. We omit Wong et al.'s protocol [WWW02] because it uses a threat model that is considerably weaker than the other schemes, and the other schemes' performance characteristics would be different if we used this weaker set of assumptions.

In our analysis, we omit explicit reference to security parameters. Security parameters are constants that are related to implementation details such as cryptographic key length, finite field cardinality, hash length, and so on. Making these parameters

	Optimistic	Average	Pessimistic
Our scheme	$O(1)$	$O(1)$	$O(n)$
Zhou et al.	$O(1)$	$O(1)$	$O(1)$
Cachin et al.	$O(1)$	$O(1)$	

Figure 9-2: Protocol Performance Comparison: Rounds of Communication

larger improves security against brute-force attacks but also increases the computational and network overhead of the protocol. All of the protocols we examine can be implemented securely (under standard computational hardness assumptions) such that their overhead is linear with respect to the security parameter.

Another detail we do not include in our analysis is additional overhead due to network packet loss or delay. In practice, packets are retransmitted periodically if not acknowledged by the intended recipient, and this constitutes a source of overhead measured in terms of number of bytes sent. Moreover, protocols such as ours that are based on BFT perform view changes after an exponentially-increasing timeout. These view changes may abort protocol instances with honest primaries until the timeout has increased to match the actual maximum message delay d, which increases the overhead by a factor of $\log d$. However, experience with BFT shows that view changes are both cheap and extremely rare; in an experimental setup, spurious view changes were too infrequent to cause poor performance even with an initial timeout of less than a second [CL02].

9.1 Performance of Our Scheme

9.1.1 Optimistic Case

We first consider our MPSS scheme in the optimistic case, since this case is the easiest to analyze. We analyze the version of our scheme with the hashing optimization of Section 6.2, which reduces load on the coordinator. In the optimistic case, we assume that the coordinator and all other nodes are honest. Most of the overhead comes from the proposal distribution and selection in the first phase of the protocol, mainly because of the size of the verification information that needs to be exchanged. The share transfer to the new group, by comparison, is relatively cheap.

In the first step of our protocol, each node S_i in the old group generates a **MsgProposal** consisting of $3t + 1$ **proposal**s and $3t + 2$ **commitment**s. Each proposal contains $3t + 2$ values and each commitment contains t or $t + 1$ values, so the size of a **MsgProposal** is $O(t^2)$. Each node broadcasts its **MsgProposal** to all members of the old group, resulting in $O(t^3)$ communication per node.

In the optimistic case, the hashing technique of Section 6.2 allows the coordinator to send only hashes of the first well-formed $2t + 1$ **MsgProposal**s it receives, rather than the full proposals. Thus, the **MsgProposalSet** message sent by the coordinator has size $O(t)$, and isn't a significant source of overhead.

Nodes reply with **MsgProposalResponse**s containing up to t constant-size accusations, regardless of whether the verifiable accusation scheme of Section 6.1 is used. The coordinator broadcasts these responses in a **MsgCommit** message of size $O(t^2)$ to $3t + 1$ servers, and in doing so it transmits $O(t^3)$ bytes.

Finally, share transfer to the new group involves sending $O(t)$ **MsgNewPoly** messages from each old node to each new node, where each such message is of size

$O(t^2)$. Hence each node in the old group sends $O(t^3)$ bytes for the share transfer step.

Asymptotically, the largest sources of overhead are each node broadcasting the commitments contained within the proposals, the coordinator broadcasting the responses it collects, and each node transferring the information to the new group. These tasks require $O(t^3)$ bytes each in the optimistic case, so we conclude that the overhead of our protocol in this case is $O(t^3)$.

9.1.2 Average case with non-adaptive adversary

We consider the case where there are t faulty nodes, but the adversary is non-adaptive. In other words, the adversary controls some chosen set of t nodes, but he cannot choose these nodes during the execution of the protocol, or with knowledge of what sequence of coordinators the system will cycle through as it performs view changes. This is a realistic model if we assume that most of the faults in the network are generally preexisting, and are not induced during the (brief) time during which the protocol is executed.

In our analysis of the optimistic case, we assumed that the coordinator was non-faulty. If the coordinator is faulty, we may be forced to perform one or more view changes to elect new coordinators until we have selected a non-faulty coordinator. We require $t \leq \lfloor n/3 \rfloor$, so the probability that the first coordinator we select is faulty is less than one third. The process of choosing subsequent coordinators can be modeled as random selection *without* replacement, which we can conservatively approximate as selection *with* replacement. Thus the expected number of view changes we must undergo in order to get a good coordinator is less than $\sum_{i=0}^{t-1}(1/3)^i < \sum_{i=0}^{\infty}(1/3)^i = 3/2$. We conclude that the difference between best-case behavior and average case

behavior of our protocol in terms of the number of view changes that must occur is a constant factor of 1.5. Furthermore, the expected number of rounds is still a constant.

The other new source of overhead we must consider in the presence of faults is extra work that an honest coordinator must do in order to ensure that all honest nodes receive the information they need. Recall that the coordinator collects commitments and proposal sets from $2t + 1$ nodes, t of which may be faulty. One of the roles of the coordinator in our protocol is to ensure that all nodes have a consistent view of which proposals are to be considered. If the t faulty nodes send their proposals and commitments to the coordinator and to nobody else, the coordinator is responsible for ensuring that all honest nodes receive the information. Hence, even if we use the hashing optimization of section 6.2, the coordinator may need to broadcast $O(t)$ **MsgProposal**s in full, which has a byte cost of $O(t^4)$. This is the essential difference between the optimistic case and the scenario with t Byzantine faults; the adversary can force the coordinator to perform t times as much work as it would otherwise.

9.1.3 Worst case

The worst-case scenario is similar to the average case discussed above, except that the adversary is adaptive and may infect the first t nodes that the protocol elects as coordinators. In the worst case, the system will need to perform t view changes. (As explained at the beginning of the chapter, we do not count spurious view changes due to message delays.) In each of the $t+1$ views, the coordinator may send $O(t^4)$ bytes. Other nodes only send proposals in the first view and perform share transfer in the last view, so in the other views they send only $O(t^2)$ bytes. Hence each honest server sends $O(t^3)$ bytes, and the honest coordinator in the final view may also send $O(t^4)$

bytes. We do not count the amount of data transmitted via *dishonest* coordinators, as this may be arbitrary. Hence the overall overhead in terms of number of bytes sent and received by each node is $t^4 + \sum_{i=0}^{t} t^2 = O(t^4)$.

9.2 Zhou et al. Scheme Performance

In APSS [ZSvR05], instead of using a standard threshold-t Shamir sharing, the secret sharing is composed of $\binom{n}{t+1} = \binom{3t+1}{t+1}$ trivial $t+1$-out-of-$t+1$ sharings: one sharing for each possible subset of $t+1$ servers. (A $t+1$-out-of-$t+1$ sharing can be implemented easily using bitwise exclusive-or; see section 2.1.1.) The APSS resharing protocol works by generating subshares of these shares, then later recombining the subshares in a different order. The communication required to transmit these subshares is proportional to the square of the size of the shares. So even in the best possible scenario, APSS incurs an overhead that is at least $O(\binom{n}{t+1}^2$.

APSS makes use of a coordinator, as in our protocol, and that coordinator may be faulty. Instead of relying on the Strong Eventual Delivery assumption (definition 3.2) as we do, APSS runs $t + 1$ resharing protocols in parallel, each with a different coordinator. Therefore, even in the optimistic case, the performance of their protocol is $t + 1$ times worse than if they were allowed to assume an honest coordinator.

Hence, in all cases, the overhead of their protocol is $\binom{3t+1}{t+1}^2 t$, which is exponential in t.

9.3 Cachin et al. Scheme Performance

We now consider the proactive secret sharing scheme of Cachin et al. as a means of demonstrating the plausibility of the performance of our scheme, MPSS. MPSS and

the Cachin et al. scheme differ in functionality; MPSS provides redistribution to a new group of shareholders, whereas the Cachin scheme only provides resharing within the same group. Nevertheless, the Cachin scheme is interesting because it is the most efficient scheme for non-mobile asynchronous PSS under reasonable assumptions.

The Cachin et al. proactive refresh scheme combines a complex hierarchy of more basic protocols: Asynchronous Verifiable Secret Sharing (AVSS) [CKLS02], Multi-Valued Validated Byzantine Agreement (VBA) [CKPS01], Binary-Valued Validated Byzantine Agreement [CKS00], Consistent Broadcast [CKS00], and Threshold Coin Tossing [CKS00, CKPS01]. Their AVSS scheme is analogous to the proposal broadcast and selection phase of our protocol, and their VBA protocol is analogous to BFT. However, rather than using a coordinator and view changes to decide which AVSS instances have completed successfully, they have a randomized agreement protocol based on threshold coin tossing and threshold signatures to make this decision.

The Cachin et al. scheme is somewhat heavyweight, but has some interesting characteristics as compared to MPSS. Buried in all the layers of the protocol are potentially $O(t)$ threshold signature operations, and furthermore, the $O(t^4)$ overhead is incurred by every node, and not just the coordinator. Nevertheless, the scheme works well when there are close to t faulty nodes. In contrast, MPSS works well in the optimistic case, where the coordinator is not faulty. However, a large number of compromised servers can force the coordinator to do t times as much work as the optimistic case under less optimistic assumptions. Under less optimistic assumptions, the overhead of our protocol is comparable to that of Cachin et al.'s protocol.

9.4 Separating the Proposal Selection and Agreement Phases

Most of the protocols discussed in this section can be decomposed into two steps. In the first step, each server generates randomized proposals, which represent that server's contribution to the computation of the next sharing. These proposals may be a sequence of polynomials with particular properties as in our scheme or a random sharing of the server's own share as in the other three schemes. In the second step of the protocol, an agreement operation takes place, in which the servers agree on a set of proposers that behaved correctly to the extent that their proposals can be used. Our protocol has a third step in which shares are transferred to the new group, since the new group in our protocol is not restricted to be the same as the old group.[1]

The first two steps are decomposable, so for instance it is possible to use the first step of Cachin's scheme with the BFT agreement protocol, or possibly with Zhou's trick of running $t+1$ protocol instances, each with a different coordinator, and using agreement to decide when one of them has completed. Similarly, our protocol could have made use of Cachin's multi-valued validated Byzantine agreement.

For all three schemes discussed in this chapter, the protocol overhead is dominated by the first step, in which proposals are exchanged. Figure 9-3 summarizes the proposal sizes for the various protocols. Our proposals are asymptotically as large as Cachin's, and all three schemes use $O(t^2)$ agreement protocols. However, our scheme is optimized for the optimistic case where most nodes broadcast their proposals correctly and the coordinator is honest.

[1] Zhou et al.'s protocol is a little bit different from the others. The authors claim that they can run their protocol with $t+1$ coordinators in parallel, one of these instances will complete, and therefore they do not require agreement. Nevertheless, their protocol for deciding when it is okay to delete old shares resembles agreement.

	Proposal Size	Agreement Overhead
Our scheme	$O(t^2)$	$O(t^2)$
Zhou et al.	$O(2^t)$	$O(t^2)$
Cachin et al.	$O(t^2)$	$O(t^2)$

Figure 9-3: Proposal and Agreement Overheads of Each Protocol

We chose to use BFT because it is known to be an efficient agreement protocol in practice. As compared to VBA [CKPS01], it achieves better performance in the optimistic case, which we believe to be the common case. In the worst case, its communication overhead is asymptotically as bad as VBA, but BFT may require more rounds of communication. Furthermore, BFT does not require expensive primitives, such as a voting scheme that uses multiple instances of a threshold-signature-based coin toss protocol.

Chapter 10

Conclusions

As we have shown, a proactive secret sharing scheme should satisfy several properties to be useful in real systems:

Asynchrony. Real networks such as the Internet are asynchronous; communication is occasionally disrupted for limited but unpredictable amounts of time. Protocols that require assumptions about the maximum delay in the network for correctness can fail in such scenarios.

Mobility. It must be possible to transfer the sharing from one set of shareholders to another. Schemes that use a fixed set of shareholders over the lifetime of the system must assume that as additional machines fail, other shareholders are "recovered". This assumption is unrealistic because, among other things, it implies that the encryption and signing keys shareholders use to communicate are never exposed or lost.

Efficiency. Proactive resharing protocols tend to be relatively heavyweight, but this is tolerable because they generally do not need to be executed frequently.

However, it is important that the overhead in terms of number of messages, message size, and rounds of communication stay within reasonable bounds as the threshold increases. Exponential schemes such as [Che04, WWW02] seem to have high overhead for even moderate thresholds, whereas our scheme is competitive with asynchronous proactive secret sharing schemes that have polynomial complexity.

Threshold adaptivity. It is often useful to be able to change the threshold of failures tolerated in order to adapt to changing requirements and changing assumptions about the reliability of individual machines.

This thesis describes what we believe is the first protocol to achieve these goals.

The paper also describes an efficient way to implement the protocol by using a primary. The primary directs the activities of the group and ensures that in the proposal selection phase, all servers consider identical sets of proposals. This technique enables very efficient proposal selection because all that is needed is to remove proposals from the initial set collected by the primary. A view change protocol is used to ensure that faulty primaries cannot prevent the resharing from being carried out properly.

Our protocol represents an improvement in performance over other PSS schemes in asynchronous networks in the "optimistic" case when the primary is honest. If the primary is faulty, we might need to restart the protocol via view changes up to t times, but still the overall communication complexity is comparable to or better than that of other schemes.

Furthermore, our approach can be used with either verifiable or non-verifiable accusations. Verifiable accusations are a new technique that allows a more efficient protocol, but requires forward-secure, identity-based encryption. The two-step pro-

tocol without verifiable accusations is less efficient by a constant factor, but makes fewer cryptographic assumptions. When we consider the protocol for increasing the group size, we find that the opposite is true, namely, the one-step protocol, which requires verifiable accusations, is more pleasing theoretically but not as good in practice.

10.1 Open Problems and Future Work

In addition to MPSS, several other proactive secret sharing schemes were discussed in Chapter 2. None of these other schemes satisfy all of the properties described in the introduction to this chapter, but several of them are based on sound and promising techniques. One question is whether schemes such as APSS [CKLS02], which is asynchronous and efficient, can be extended to support additional features such as resharing to a new group, increasing the threshold, and decreasing the threshold.

There are many possibilities for combining aspects of different secret sharing protocols. All of the PSS protocols we have discussed, including the non-mobile ones where the old and new groups are the same, can be broken down into two phases: in the first phase, members of the old group communicate amongst themselves to exchange information needed to compute the resharing, and in the second phase, the old group executes an agreement operation and transfers information to the new group. The design of the first phase depends on the structure of the secret sharing and how new shares are computed from old shares. MPSS and Jarecki's scheme [HJKY95] uses one technique, Cachin et al. [CKLS02] and Wong et al. [WWW02] use another, and Zhou et al. [ZSvR05] uses a third. Furthermore, these schemes all use different agreement protocols; our MPSS scheme uses BFT [CL02], Zhou et al. have an agreement protocol that eschews common case optimizations and resembles running BFT

in $t + 1$ views in parallel, and Cachin et al. use a *randomized* agreement protocol of their own invention. The Wong et al. scheme doesn't have a real agreement protocol, so it isn't able to efficiently decide which information to use in the presence of an active adversary. It would be instructive to consider what would happen if, for instance, one were to combine the first phase of Cachin et al.'s scheme with the BFT agreement protocol and the second phase of our scheme, or one of various other possible combinations. Another question is whether doing this makes it possible to extend other schemes to increase efficiency, support mobility, and so on.

Implementation and evaluation of these schemes is another important piece of future work. In order to explore modifications and better understand the performance of proactive secret sharing schemes, it is necessary to measure how they perform in real world scenarios. For example, in asymptotic terms, our scheme and Cachin et al.'s scheme are very similar. However, Cachin et al.'s scheme uses cryptographic tricks such as threshold signatures and coin tossing, which are very expensive in practice, to reduce asymptotic complexity. MPSS, on the other hand, uses BFT, which is optimized for the common case and may not perform well when the number of faults is large, or when message delays are extremely erratic. These tradeoffs are difficult to investigate analytically.

Lastly, we note that our protocols for increasing and decreasing the group size involve a variety of tradeoffs. Our scheme for decreasing the threshold is designed in such a way that subsequent increases are very cheap; however, if we were to increase the threshold from 10 to 20 and then back to 10 again, we would not be able to operate as efficiently as before. This property is dissatisfying theoretically, even though changes in the threshold are small in practice. A similar issue arises when increasing the group size: we have a protocol that works well in practice, and a second protocol using verifiable accusations that is better theoretically but not as

good in practice. It is unclear whether these tradeoffs are fundamental, or whether better ways of changing the threshold exist.

Bibliography

[BBH06] D. Boneh, X. Boyen, and S. Halevi. Chosen ciphertext secure public key threshold encryption without random oracles. In *RSA Conference*, pages 226–243, 2006.

[BF01] D. Boneh and M. Franklin. Identity-based encryption from the weil pairing. In Joe Kilian, editor, *Advances in Cryptology—CRYPTO 2001*, Lecture Notes in Computer Science, pages 213–229. Springer-Verlag, 19–23 August 2001.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, Illinois, May 1988.

[Bla79] G.R. Blakley. Safeguarding cryptographic keys. In *Proc. AFIPS 1979*, volume 48, pages 313–317, June 1979.

[Ble98] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on RSA encryption standard PKCS #1. pages 1–12, 1998.

[BM99] Mihir Bellare and Sara Miner. A forward-secure digital signature scheme. In Michael Wiener, editor, *Advances in Cryptology—CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–448. Springer-Verlag, 15–19 August 1999. Revised version is available from `http://www.cs.ucsd.edu/ mihir/`.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 11–19, Chicago, Illinois, May 1988.

[CG99] Ran Canetti and Shafi Goldwasser. An efficient threshold public key cryptosystem secure against adaptive chosen ciphertext attack. In *Theory and Application of Cryptographic Techniques*, pages 90–106, 1999.

[Che04] Kathryn Chen. Authentication in a reconfigurable byzantine fault tolerant system. In *MEng Thesis, Massachusetts Institute of Technology*, 2004.

[CHK03] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In Eli Biham, editor, *Advances in Cryptology—EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 255–271. Springer-Verlag, 4 – 8 May 2003.

[CKLS02] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proc. 9th (ACM) conference on Computer and Communications Security*, pages 88–97. (ACM) Press, 2002.

[CKPS01] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 524–541, London, UK, 2001. Springer-Verlag.

[CKS00] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC 2000)*, Portland, OR, July 2000.

[CL99] M. Castro and B. Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

[CL02] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[Cop95] Don Coppersmith, editor. *Advances in Cryptology—CRYPTO '95*, volume 963 of *Lecture Notes in Computer Science*. Springer-Verlag, 27–31 August 1995.

[DF91] Yvo Desmedt and Yair Frankel. Shared generation of authenticators and signatures. volume 576 of *LNCS*, pages 457–469. Springer-Verlag, 1991.

[DJ97] Y. Desmedt and S. Jajodia. Redistributing secret shares to new access structures and its applications. Technical Report ISSE TR-97-01, George Mason University, July 1997.

[Fel87] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 427–437, New York City, 25–27 May 1987.

[FLP82] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. Technical Report MIT/LCS/TR-282, Laboratory for Computer Science, MIT, Cambridge, MA, 1982. Also published in *Journal of the ACM*, 32:374–382, 1985.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Lecture Notes in Computer Science*, 1666:537–554, 1999.

[GJKR96] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold DSS signatures. In Ueli Maurer, editor, *Advances in Cryptology—EUROCRYPT 96*, volume 1070 of *Lecture Notes in Computer Science*, pages 354–371. Springer-Verlag, 12–16 May 1996.

[HJKY95] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing, or how to cope with perpetual leakage. In Coppersmith [Cop95], pages 457–469.

[Jar95] Stanislaw Jarecki. Proactive secret sharing and public key cryptosystems. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1995.

[Kra00] Hugo Krawczyk. Simple forward-secure signatures from any signature scheme. In *Seventh ACM Conference on Computer and Communication Security*. ACM, November 1–4 2000.

[Lan95] Susan Langford. Threshold DSS signatures without a trusted party. In Coppersmith [Cop95], pages 397–409.

[LW88] D. Long and A. Wigderson. The discrete log hides $O(\log n)$ bits. *SIAM Journal on Computing*, 17(2):363–72, 1988.

[OY91] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of the 10th (ACM) Symposium on the Principles of Distributed Computing*, pages 51–61, 1991.

[Ped91a] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer-Verlag, 1992, 11–15 August 1991.

[Ped91b] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party (extended abstract). In D. W. Davies, editor, *Advances in Cryptology—EUROCRYPT 91*, volume 547 of *Lecture Notes in Computer Science*, pages 522–526. Springer-Verlag, 8–11 April 1991.

[PH78] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over GF(p). *IEEE Transactions on Information Theory*, IT-24:106–110, 1978.

[RBO89] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85, New York, NY, USA, 1989. ACM Press.

[Rod] Rodrigo Rodrigues et al. Automatic reconfiguration for large-scale distributed storage systems. Unpublished.

[Sha79] A. Shamir. How to share a secret. *Communications of the (ACM)*, 22:612–613, 1979.

[Sha84] Adi Shamir. Identity-based cryptosystems and signature schemes. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology: Proceedings of CRYPTO 84*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer-Verlag, 1985, 19–22 August 1984.

[SW99] Douglas R. Stinson and R. Wei. Unconditionally secure proactive secret sharing scheme with combinatorial structures. In *Selected Areas in Cryptography*, pages 200–214, 1999.

[WWW02] T. M. Wong, C. Wang, and J. Wing. Verifiable secret redistribution for archive systems. In *Proceedings of the 1st International IEEE Security in Storage Workshop*, 2002.

[YFDL04] D. Yao, N. Fazio, Y. Dodis, and A. Lysyanskaya. ID-based encryption for complex hierarchies with applications to forward security and broadcast encryption. In *ACM Conference on Computer and Communication Security*, pages 354–363, 2004.

[ZSvR05] Lidong Zhou, Fred Schneider, and Robbert van Renesse. APSS: Proactive secret sharing in asynchronous systems. *ACM Transactions on Information and System Security*, 8(3):259–286, aug 2005.