APSS: Proactive Secret Sharing in Asynchronous Systems*

Lidong Zhou†, Fred B. Schneider‡, and Robbert van Renesse‡

October 10, 2002

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

†Microsoft Silicon Valley Research Center, 1065 La Avenida, Mountain View, CA 94043.

‡Department of Computer Science, Upson Hall, Cornell University, Ithaca, New York 14853.

Abstract

APSS, a proactive secret sharing (PSS) protocol for asynchronous systems, is derived and proved correct. A PSS protocol enables a set of secret shares to be periodically refreshed with a new, independent set, thereby thwarting so-called mobile adversary attacks. APSS tolerates certain attacks that PSS protocols for synchronous systems cannot, because protocols for asynchronous systems are inherently less vulnerable to denial of service attacks, which slow processor execution or impede message delivery and thus violate the defining assumptions of a synchronous system.

Categories and Subject Descriptors: C.2.0 [Computer-Communication Networks]: General—security and protection; C.2.4 [Computer-Communication Networks] Distributed Systems—client/server; D.4.5 [Operating Systems]: Reliability—fault-tolerance; D.4.6 [Operating Systems]: Security and Protection—authentication, cryptographic controls; E.3 [Data]: Data Encryption—public key cryptosystems.

1 Introduction

An $(n, t + 1)$ *secret sharing* [34, 2] for a secret s is a set of n random *shares* such that (i) s can be recovered with knowledge of $t + 1$ shares, and (ii) no information about s can be derived from t or fewer shares. Thus, if each share is assigned to a different server in a distributed system then the secret remains available provided more than t servers are available and the secret remains confidential unless more than t servers have been compromised.

Secret sharing alone does not defend against *mobile adversaries* [28], which attack, compromise, and control one server for a limited period before moving to the next. Over time, a mobile adversary might compromise $t+1$ servers, obtain $t+1$ shares, and thus learn the secret. The defense here is *share refreshing*, whereby servers periodically create a new and independent secret sharing and then replace old shares with new ones. Because the new and old secret sharings are independent, a mobile adversary cannot combine new shares with old shares in order to reconstruct the secret. And because old shares are deleted when replaced by new shares, a mobile adversary must compromise more than t servers between share refreshings to succeed.

Secret sharing with share refreshing is known as *proactive secret sharing* (PSS) [23, 20]. PSS reduces the *window of vulnerability* during which an adversary must compromise more than t servers in order to learn the secret. Without share refreshing, the window of vulnerability is unbounded; with PSS, the window of vulnerability is shortened to the period between share refreshings.

Prior work on PSS protocols assumes a *synchronous* system, meaning bounds on message delivery delays and processor execution speed are known. Any assumption constitutes a vulnerability, and the assumption of a synchronous system is no exception. Denial of service attacks, in particular, might delay messages and/or consume processor cycles, thereby invalidating the defining assumption for a synchronous system.

This paper describes APSS, a PSS protocol for *asynchronous* systems—systems in which message delivery delays and processor execution speeds do not have fixed bounds. We are eliminating an assumption and thus eliminating a vulnerability. Besides implementing secret sharing, APSS can be used for *threshold cryptography* [12, 3] where processors store shares of a private key and perform cryptographic operations using these shares (without ever materializing the entire private key).

The protocol is derived in four steps, each the result of a transformation to defend against increasingly hostile adversaries. This form of exposition, though perhaps a bit longer, elucidates the role played by each element of APSS. And the transformations themselves are interesting for their applicability in other settings.

The paper is organized as follows. In §2, the system model and the correctness requirements for APSS are specified. Next, §3 describes cryptographic building blocks used in the protocol. The protocol itself is derived in §4. Various ways in which the protocol can be optimized and extended are explored in §5. Related work is discussed in §6, and the appendix contains correctness proofs.

2 System Model and Correctness

Consider a system comprising a set of *processors* that communicate through a network. Of these, n processors are designated *servers* and hold the shares of a secret. Each processor is assumed to have an *individual* public/private key pair and to know the public keys of all other processors. As is traditional in the proactive secret sharing literature (for example, see [23]), we assume:

- Attackers lack the computational power to compromise the cryptographic building blocks used in protocols.
- A processor's private keys are stored in a tamper-proof cryptography co-processor which performs all operations involving these keys.[1]

2.1 Attacks and Failures

We intend APSS for use in an environment, like the Internet, where failures and attacks could invalidate assumptions about timing:

Asynchronous System: There is no bound on message delivery delay or processor execution speed. $\boxtimes$

[1]In the absence of tamper-proof co-processors, keys must be refreshed in addition to refreshing the shares. One simple approach has trusted administrators for each processor invent and propagate new public keys through secure channels implemented by having an *administrative* public/private key pair. The administrative public key is known to other administrators (and all processors); the administrative private key, kept off-line most of the time as a defense against on-line attacks, is used to sign notification messages for the new public key. Other rekeying schemes are discussed in [8].

We also restrict the fraction of processors that (for whatever reason) fail to execute the prescribed protocol, but we otherwise make no assumptions about processor behavior:

Compromised Processors: At any point in time, a processor is either *correct* or *compromised*. A compromised processor might stop executing, deviate arbitrarily from its specified protocols (i.e., Byzantine failure), and/or corrupt or disclose information stored locally; a correct processor follows the protocols and does not corrupt or disclose information stored locally.

Given an interval $\mathcal{T}$ of time, a processor is considered *correct within* $\mathcal{T}$ if and only if that processor is correct throughout interval $\mathcal{T}$; otherwise, the processor is deemed *compromised within* $\mathcal{T}$.

Assume at most t processors are compromised within each window of vulnerability, where $3t + 1 \leq n$ holds. ⊠

Without some assumption about the communications network, an adversary could prevent processors from communicating with each other and with clients. No protocol could be expected to work in that setting, so we assume:

Fair Links: A *fair link* is a communication channel between processors that does not necessarily deliver all messages sent, but if a processor sends sufficiently many messages to a single destination then one of those messages is correctly delivered. Messages in transit may be disclosed to or altered by adversaries.

The network is assumed to provide fair links. ⊠

The Asynchronous System, Compromised Processors, and Fair Links assumptions together give adversaries considerable power. Adversaries can

- attack servers and other processors, provided fewer than $n/3$ are compromised within any window of vulnerability,
- launch eavesdropping, message insertion, corruption, deletion, and replay attacks, and
- conduct denial of service attacks that delay messages or slow processors by arbitrary finite amounts.

2.2 Window of Vulnerability Definition

The Asynchronous System assumption means that the window of vulnerability for APSS must be defined in terms of events rather than the passage of time. This is not a panacea.

Using a protocol's events to delimit its window of vulnerability could afford attackers leverage. In particular, denial of service attacks can delay the protocol's completion, thus lengthening the window of vulnerability and increasing the interval available to perpetrate an attack. But whether longer windows of vulnerability are significant depends on your model for the adversary. Two plausible models are:

- The adversary is an agent whose ability to compromise servers depends on the time available for attacks.
- The adversary is an agent whose ability to compromise servers depends on intrinsic aspects of those servers, such as operating system, application software, *etc.*

If the first model is more realistic for the anticipated threat, then the Asynchronous System and Compromised Processor assumptions taken together limit what the adversary can do. This is because their effect is to bound the number of servers that the adversary can compromise during intervals (*viz.*, a window of vulnerability) that might be arbitrarily long. If the second model is the more realistic, then the possibility for longer windows of vulnerability need not be of concern.

To further complicate matters, whichever of these models for the adversary is embraced, there are ways in which APSS is going to be more robust than a PSS protocol would be. Consider a synchronous system and assume that APSS and a PSS protocol have comparable execution times there (a reasonable supposition, given the implementation of APSS in [37]). APSS will successfully defend against every attack that the PSS protocol does. Moreover, APSS will also defend against some attacks that compromise the PSS protocol—namely, denial of service attacks that invalidate the defining assumptions of a synchronous system. So there is a sense in which APSS is the more robust.

In short, the choice of models and its consequences can be subtle. There is something philosophically appealing, however, about protocols like APSS whose correctness does not depend on assumptions about execution timings.

Runs, Versions, and Epochs. Each execution, or *run*, of APSS generates a new secret sharing, so events associated with this share refreshing are what define the APSS window of vulnerability. To give a precise definition for the window of vulnerability, *version numbers* are assigned to shares, secret sharings, and runs:

- Servers initially store shares with version number 0.
- If a run is executed with shares having version number v, then this run and the resulting new shares have version number $v + 1$.
- A secret sharing is assigned the same version number as its shares.

Run v *starts* when some correct processor initiates run v locally or starts participating in run v.[2] Run v *terminates locally* on a correct processor p participating in APSS once p has become quiescent[3] and, if a server, it has (i) stored corresponding shares with version number v, (ii) deleted shares it stores that have version number less than v, and (iii) deleted other locally stored information that can be used to infer anything about those deleted shares. Run v *terminates globally* at the earliest time t that the run has terminated locally on each processor p correct within the interval delimited by the global start of run v and time t.

Define *epoch* v to be the interval from the start of run v to the global termination of run $v + 1$. Figure 1 illustrates how epochs relate to runs and version numbers on shares. Notice, processors that are correct either within epoch $v - 1$ or correct within epoch v are, by definition, correct within run v because run v belongs to epoch $v - 1$ and to epoch v.

In a successful attack, an adversary must collect $t + 1$ or more shares, all with the same version number. Since $t + 1$ or more servers must be compromised during the same epoch in order to collect these shares, we define the window of vulnerability mentioned in the Compromised Processors assumption to be an epoch.

[2]In practice, the start of a run by a correct server would be preceded by execution of a recovery routine to exorcise any residue of prior compromise at that server.

[3]A *quiescent* processor is one that sends protocol messages only in response to receiving a protocol message.

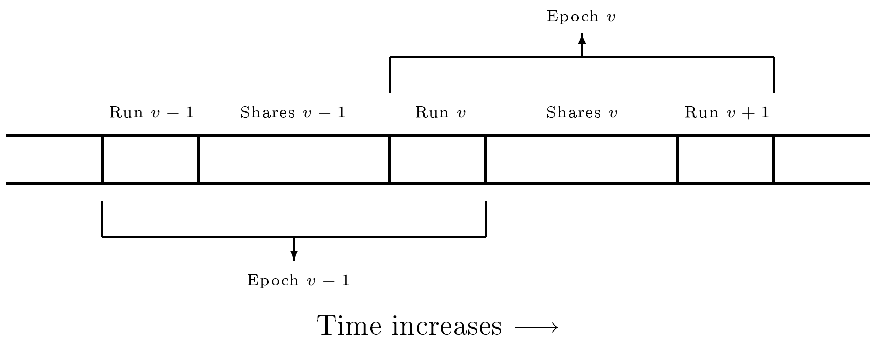

Figure 1: Epochs and Versions.

2.3 APSS Correctness

Correctness of APSS stipulates that not only does the secret remain confidential and available but also that adversaries cannot forever delay share refreshing by preventing runs from starting or from terminating globally:

APSS Secrecy: Secret s cannot be learned by adversaries. $\boxtimes$

APSS Availability: Secret s can be recovered using shares stored by correct servers. $\boxtimes$

APSS Progress: Adversaries cannot prevent an APSS run from starting, and every run eventually terminates globally. $\boxtimes$

Typically, a PSS protocol is executed in concert with some *client protocol*, which reads shares and recovers the secret or computes some function where the secret is an argument. A threshold signature protocol [13], for example, uses shares in constructing a digital signature for a message; the private (signing) key is the secret being shared. The defining characteristic of PSS—that shares from different sharings of a secret s cannot be combined to reveal information about s—means client protocol execution must be synchronized with share refreshing. This is an instance of the well known readers/writers problem [11], with client protocols the readers and the share refreshing protocol a writer.

One solution, in the spirit of Lamport's concurrent reading while writing [25], is to use version numbers in conjunction with verifiable secret sharing [10]: the client protocol detects whether shares it read constitute a sharing of s and iterates if they do not. The Cornell On-line Certification Authority (COCA) [37] does this, using APSS in conjunction with client protocols that read and update digital certificates stored by quorums of servers. Share refreshing in COCA is executed hourly, and COCA's read, update, and share refreshing protocols each completes in under a few seconds thereby reducing the chances that a read will overlap share refreshing.

An open question is how one could eliminate the iteration by employing synchronization to delay and coordinate execution of the client protocol and share refreshing. Eliminating all overlap between client protocol and share refreshing execution is certainly sufficient but might not be necessary—client protocol execution might be permitted at those servers having new shares and be delayed at all others. Note that any synchronization must accommodate APSS Progress, which requires client protocol execution (perhaps instigated by an adversary) not be able to forever delay the start of a share refreshing protocol run.

3 Building Blocks for APSS

Secret sharing involves two operations: split and reconstruct. Operation split generates a set of random shares from a secret s; these constitute a *sharing* of s. Operation reconstruct recovers s from certain sets of shares. A distinct label is associated with each share and sharing of s, as follows.

> $\left[\overline{s}^{:\Lambda}\right]$ denotes a sharing of s; the sharing is labeled Λ.
>
> $\left[\overline{s}^{:\Lambda}\right]_i$ denotes the i^{th} share[4] of sharing $\left[\overline{s}^{:\Lambda}\right]$.

Labels that are clear from context or unnecessary will be omitted to avoid clutter.

Version numbers associated with shares, sharings, and runs form the basis for some useful relations on share and sharing labels. For label Λ (Λ') on a sharing generated during a run v (v'), $\Lambda \prec \Lambda'$ holds if and only if $v < v'$ holds and $\Lambda \simeq \Lambda'$ holds if and only if $v = v'$ holds.

[4]Some canonical order on shares is assumed.

3.1 Combinatorial $(n, t+1)$ Secret Sharing

With what we call *standard secret sharing*, shares are single values; with what we call *combinatorial secret sharing*, shares are sets of values, called *share sets*, and values in share sets implement a standard secret sharing. APSS employs combinatorial secret sharing. The protocol is general enough to handle any adversary structure [21], but to simplify the presentation, we here discuss in detail support only for one family: $(n, t+1)$ secret sharing. In §5.3, we explain how that protocol would be extended for other adversary structures.

An $(n, t+1)$ secret sharing can be implemented by assigning shares obtained using a standard $(\binom{n}{t}, \binom{n}{t})$ secret sharing to share sets comprising a combinatorial secret sharing [22]. In this construction, the secret is recovered from a collection of share sets by using reconstruct on shares in the union of those share sets.

Construction of $(n, t+1)$ Combinatorial Secret Sharing [22]:

1. Create $l = \binom{n}{t}$ different sets $P_1, ..., P_l$, each containing t servers. These sets and all their subsets enumerate collections of servers whose compromise must be tolerated during a single window of vulnerability.
2. Create $[\overline{s}]$ using an (l, l) standard secret sharing.
3. Include share $[\overline{s}]_i$ in S_p, the share set for a server p, if and only if p is not in P_i. That is, for any server p, share set S_p is defined by:

   $$S_p = \{[\overline{s}]_i \mid 1 \leq i \leq l \ \wedge \ p \notin P_i\}.$$

   By not assigning $[\overline{s}]_i$ to any server in P_i, we ensure that servers in P_i do not together have all l shares needed to reconstruct s.

   Also, for any server p, define *index set* I_p:

   $$I_p = \{i \mid 1 \leq i \leq l \ \wedge \ p \notin P_i\}$$

   Index sets provide a sharing-independent description of the share-set construction because, by definition, $I_p = \{i \mid [\overline{s}]_i \in S_p\}$ and $S_p = \{[\overline{s}]_i \mid i \in I_p\}$ hold. ⊠

Share sets from the above construction exhibit two important properties:

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

Figure 2: Example of Combinatorial Secret Sharing.

SS1. For any set P of servers, where $|P| \geq t + 1$, the following holds:

$$(\bigcup_{p \in P} S_p) = [\overline{s}]$$

SS2. For any set P of servers, where $|P| \leq t$, the following holds:

$$(\bigcup_{p \in P} S_p) \subset [\overline{s}]$$

SS1 implies that secrets can be recovered (using reconstruct) from the share sets stored by any $t + 1$ or more servers; SS2 implies that share sets at t or fewer servers lack one or more of the shares needed to recover s. Thus, we have:

Threshold Availability: Secret s can be reconstructed from the shares stored by $t + 1$ or more correct servers. ⊠

Threshold Confidentiality: It is infeasible to reconstruct s from the shares stored by t or fewer servers. ⊠

Figure 2 illustrates a $(4, 2)$ (i.e., $n = 4$ and $t + 1 = 2$) combinatorial secret sharing based on a $(\binom{4}{1}, \binom{4}{1}) = (4, 4)$ standard secret sharing and using s_i to denote $[\overline{s}]_i$. Notice that the share set for each server p_j consists of all shares except $[\overline{s}]_j$. Index sets are also shown. No single share set contains all 4 shares, so a single compromised server is unable to obtain all 4 shares needed to recover the secret.

One benefit of basing APSS on combinatorial secret sharing is the simple and efficient recovery scheme it admits for assembling the share set of a compromised server that has been repaired. Each share of the underlying standard secret sharing is an element of multiple share sets and, therefore, is stored at multiple servers. A recovering server q (say) thus can rebuild share set S_q by requesting shares from other (correct) servers. Upon receiving such a recovery request from server q, a correct server p replies with a set $\{[\overline{s}]_i \mid i \in I_p \cap I_q\}$ of shares, encrypted in a way that only processor q can decrypt. For example, when p_4 of Figure 2 is recovering, it receives $\{s_2, s_3\}$ (but not s_4) from p_1, $\{s_1, s_3\}$ (but not s_4) from p_2, and $\{s_1, s_2\}$ (but not s_4) from p_3.

The $(n, t+1)$ combinatorial secret sharing construction we use is exponential with respect to t, both in the total number l $(= \binom{n}{t})$ of shares and the size $|S_p|$ $(= \binom{n}{t} - t)$ of share sets.[5] The underlying (l, l) standard secret sharing, however, is simple and cheap—modular addition and subtraction suffice. But asymptotically superior alternatives to combinatorial secret sharing for APSS do exist, and they are discussed in §5.4.

3.2 Share Refreshing

Share refreshing for the combinatorial secret sharing given in §3.1 can be implemented in terms of share refreshing for the underlying (l, l) standard secret sharing.[6] The new share sets are sets of new shares generated by share refreshing for the (l, l) standard secret sharing.

Specifically, for every old share $[\overline{s}^{:v}]_i$ from the (l, l) standard secret sharing, split is used to obtain a *subsharing* $\left[\overline{[\overline{s}^{:v}]_i}\right]$ comprising *subshares* $\left[\overline{[\overline{s}^{:v}]_i}\right]_1$, $\left[\overline{[\overline{s}^{:v}]_i}\right]_2$, $\ldots$, $\left[\overline{[\overline{s}^{:v}]_i}\right]_l$. The result is l sets that each contains l subshares. Then, $\mathsf{reconstruct}$ is used to generate new share $[\overline{s}^{:v+1}]_i$ by combining subshares $\left[\overline{[\overline{s}^{:v}]_1}\right]_i$, $\left[\overline{[\overline{s}^{:v}]_2}\right]_i$, $\ldots$, $\left[\overline{[\overline{s}^{:v}]_l}\right]_i$—notice the interchanged subscripts. Figure 3 depicts the construction, with s_{ij} an abbreviation for $\left[\overline{[\overline{s}^{:v}]_i}\right]_j$. A column in the figure depicts the subshares that comprise a subsharing for each old share; a row shows which subshares are combined to form each new share.

[5] Most distributed services involve only a relatively small number of servers, and this exponential factor is then not a concern.

[6] Any (l, l) secret sharing in which a secret can be reconstructed as a linear combination (in a certain field) of its shares supports the share refreshing outlined in this subsection. Shamir's scheme [34] and the trivial (l, l) scheme that uses modular addition are both examples of such linear secret sharing schemes [24].

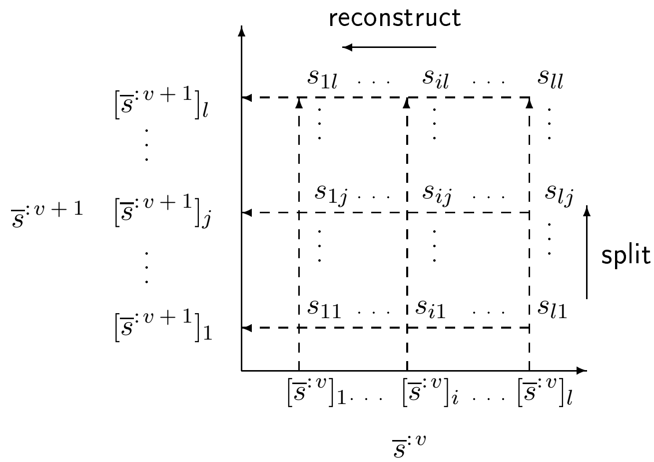

Figure 3: Share Refreshing.

In this share refreshing scheme, a subshare is made available to those and only those servers storing a share constructed from that subshare. This can be enforced by encrypting subshares; index set I_p defines what subshares should be encrypted in a way that only server p can decrypt.

Henceforth, we employ the convention that upper case lambda Λ and decorated variants Λ', Λ_i, *etc.* are labels on sharings, lower case lambda λ and decorated variants λ', λ_i, *etc.* are labels on subsharings, and the share set for server p that corresponds to a sharing with label Λ is denoted S_p^Λ.

4 APSS Protocol Derivation

We describe the protocol for a single run v. Call epoch $v-1$ the *previous epoch*, run $v-1$ the *previous run*, epoch v the *current epoch*, run v the *current run*, shares with version number $v-1$ *old shares*, and shares with version number v *new shares*. Unless noted otherwise, the term "correct servers" refers to servers that are correct within the current run.

The derivation of APSS is divided into four steps. We start by giving a share-refreshing protocol that works under relatively strong assumptions. Each subsequent step introduces modifications for further weakening these assumptions. The appendix contains a proof that the final protocol works in any distributed system satisfying the Asynchronous System, Compromised

Processors, and Fair Links assumptions.

4.1 Version 1: Benign Compromise

We build on the share refreshing protocol of §3.2. As a starting point, we consider systems in which the Compromised Processors and Fair Links assumptions (§2.1) are strengthened to:

Crash-Disclosure Failures: A compromised server might halt prematurely. Before it halts, it behaves correctly.[7] A compromised server might also disclose any information stored locally to adversaries. Up to t servers may be compromised in an epoch. ⊠

Secure and Reliable Links: Messages sent are reliably delivered. The content of messages in transit is never disclosed or corrupted. ⊠

The share refreshing protocol of §3.2 presumes that a single subsharing has been generated for each old share. However, share sets at servers are not disjoint and split is nondeterministic. So a protocol in which each correct server generates a subsharing for every old share contained in its share set does not produce a single subsharing for each old share—it produces multiple subsharings for each old share. A single new sharing can be obtained, though, by selecting one of the subsharings for each old share and executing the share refreshing protocol from §3.2 with the selected subsharings. That approach is the basis for the share refreshing protocol of this subsection.

Define a *candidate set of subsharings* to be a set of subsharings that contains exactly one subsharing for each old share. We identify a candidate set $\mathcal{C}$ of subsharings derived from an (l, l) secret sharing $\left[\overline{s}^{:\Lambda}\right]$ by a label $\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l$, where λ_i labels the subsharing $\left[\overline{\left[\overline{s}^{:\Lambda}\right]_i}^{:\lambda_i}\right]$ in $\mathcal{C}$ for share $\left[\overline{s}^{:\Lambda}\right]_i$. Label $\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l$ also identifies the new sharing of s based on $\mathcal{C}$:

$$\begin{aligned}
\left[\overline{s}^{:\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l}\right] &\triangleq \left\{\left[\overline{s}^{:\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l}\right]_1, \left[\overline{s}^{:\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l}\right]_2, \ldots, \left[\overline{s}^{:\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l}\right]_l\right\} \\
\left[\overline{s}^{:\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l}\right]_j &\triangleq \mathsf{reconstruct}(\left[\overline{\left[\overline{s}^{:\Lambda}\right]_1}^{:\lambda_1}\right]_j, \left[\overline{\left[\overline{s}^{:\Lambda}\right]_2}^{:\lambda_2}\right]_j, \ldots, \left[\overline{\left[\overline{s}^{:\Lambda}\right]_l}^{:\lambda_l}\right]_j)
\end{aligned}$$

[7]These failures are called crash failures in the fault tolerance literature [18].

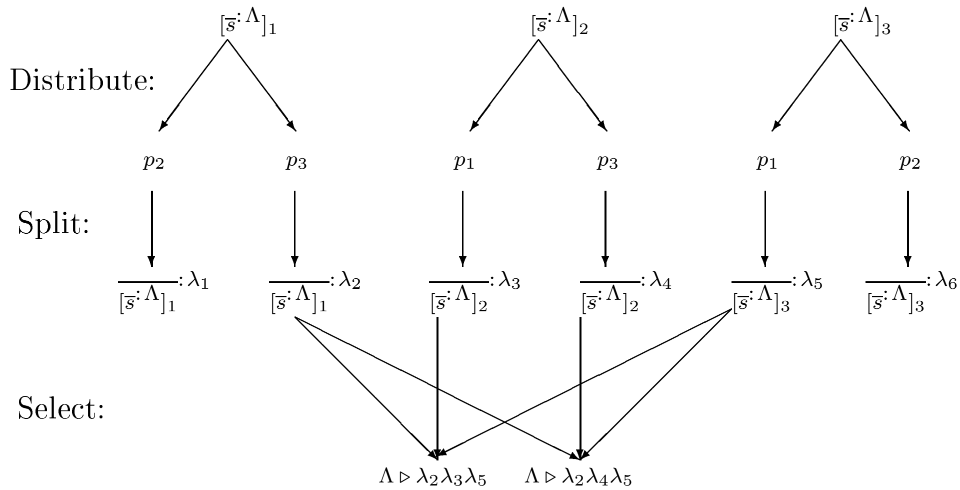

Figure 4: Candidate Sets of Subsharings.

Figure 4 illustrates the construction of labels for possible candidate sets of subsharings (and corresponding new sharings). Assume for some sharing Λ, server p_1 stores $\left[\overline{s}^{:\Lambda}\right]_2$ and $\left[\overline{s}^{:\Lambda}\right]_3$; server p_2 stores $\left[\overline{s}^{:\Lambda}\right]_1$ and $\left[\overline{s}^{:\Lambda}\right]_3$; and server p_3 stores $\left[\overline{s}^{:\Lambda}\right]_1$ and $\left[\overline{s}^{:\Lambda}\right]_2$. Servers p_1, p_2, and p_3 can generate 6 subsharings from the shares they hold, and from these subsharings, there are 8 candidate sets of subsharings; the labels for two of them are shown in Figure 4: $\Lambda \triangleright \lambda_2\lambda_3\lambda_5$ and $\Lambda \triangleright \lambda_2\lambda_4\lambda_5$.

To select a single candidate set of subsharings for use in constructing new shares, we postulate a coordinator C_p and (unrealistically) assume the processor p executing C_p is never compromised.[8]

Correct Coordinator: Coordinator C_p is correct. ⊠

Coordinator C_p also starts the share refreshing protocol at each server and controls when servers delete old shares (ensuring that a sufficiently large set of servers can, at all times, reconstruct the secret because either enough old shares or new shares are available).

[8] This assumption will later be relaxed.

(1) C_p sends an $\mathsf{init}(\Lambda)$ message to all servers, where $\Lambda = \mathtt{V}_p$ holds.

(2) C_p awaits receipt of the $\mathsf{contribute}(\lambda_i)$ messages. λ_i is a label for a subsharing the sender computes for some share with label Λ it stores.

(3) Once enough labels have been received in $\mathsf{contribute}(\lambda_i)$ messages to construct a candidate set of subsharings, C_p does so, constructing label $\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l$ and sending that label for the new sharing in a $\mathsf{compute}(\Lambda \triangleright \lambda_1, \lambda_2, \ldots, \lambda_l)$ message to all servers.

(4) Once $\mathsf{computed}(\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l)$ messages have been received from $2t+1$ servers:

    (a) $\mathtt{V}_p := \Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l$

    (b) C_p runs the Share and Subshare Deletion Protocol.

Figure 5: Protocol for Coordinator C_p.

Figure 5 outlines the coordinator's protocol for refreshing a sharing that has label Λ; Figure 6 gives the server protocol.[9] In these, a program variable[10] $\mathtt{S}_p^{\Lambda}$ stores shares from $[\overline{s}^{:\Lambda}]$ that server p has received and/or constructed; a program variable $\mathtt{V}_p$ stores the label for a sharing; and between runs, $\mathtt{S}_q^{\mathtt{V}_p} = S_q^{\mathtt{V}_p}$ holds for $t+1$ or more servers q that are correct within that interval. This is formalized by an assertion $\mathcal{I}(p)$ that holds during the interval between runs:

$\mathcal{I}(p)$: For all correct processors q and all labels Λ: $\mathtt{S}_q^{\Lambda} \subseteq S_q^{\Lambda}$
For all processors q in some set comprising $t+1$ correct
processors: $\mathtt{S}_q^{\mathtt{V}_p} = S_q^{\mathtt{V}_p}$

Share refreshing then works as follows.

To start things off, coordinator C_p sends an $\mathsf{init}(\Lambda)$ message (step (1)), where $\Lambda = \mathtt{V}_p$ holds, to all servers. Upon receipt of that message, a correct

[9] The protocol presented in Figure 6 sacrifices efficiency in favor of simplicity. For example, rather than sending multiple store messages between each a pair of servers as is now done, a more efficient protocol would send a single, combined message.

[10] By convention, program variables are typeset in fixed-width fonts and are subscripted by the processor where they are located.

Generate and Propagate Subshares:

Upon receipt of an $\mathsf{init}(\Lambda)$ message, for every share $\left[\overline{s}^{:\Lambda}\right]_i$ in $\mathtt{S}_q^{\Lambda}$, server q does the following:

(a) Calculates a subsharing for $\left[\overline{s}^{:\Lambda}\right]_i$:

$$\left[\overline{\left[\overline{s}^{:\Lambda}\right]_i}^{:\Lambda \circ C_p \circ i \circ q}\right] := \mathsf{split}(\left[\overline{s}^{:\Lambda}\right]_i).$$

(b) Using a $\mathsf{store}_r(\Lambda \circ C_p \circ i \circ q, x)$ message, sends to each server r the label $\Lambda \circ C_p \circ i \circ q$ along with (in x) subshares $\left[\overline{\left[\overline{s}^{:\Lambda}\right]_i}^{:\Lambda \circ C_p \circ i \circ q}\right]_j$ where $j \in I_r$ holds.

(c) Using a $\mathsf{contribute}(\Lambda \circ C_p \circ i \circ q)$ message, sends label $\Lambda \circ C_p \circ i \circ q$ to C_p.

Construct New Shares:

Upon receipt by server q of a $\mathsf{compute}(\Lambda \triangleright \lambda_1, \lambda_2, \ldots, \lambda_l)$ message from C_p and $\mathsf{store}_q(\lambda_1, x_1)$, $\mathsf{store}_q(\lambda_2, x_2)$, ... , $\mathsf{store}_q(\lambda_l, x_l)$ messages from other servers:

(a) For each j such that $j \in I_q$, server q calculates new share

$$\left[\overline{s}^{:\Lambda \triangleright \lambda_1 \lambda_2 \ldots \lambda_l}\right]_j := \mathsf{reconstruct}(\left[\overline{\left[\overline{s}^{:\Lambda}\right]_1}^{:\lambda_1}\right]_j, \left[\overline{\left[\overline{s}^{:\Lambda}\right]_2}^{:\lambda_2}\right]_j, \ldots, \left[\overline{\left[\overline{s}^{:\Lambda}\right]_l}^{:\lambda_l}\right]_j)$$

(b) Using a $\mathsf{computed}(\Lambda \triangleright \lambda_1, \lambda_2, \ldots, \lambda_l)$ message, server q notifies C_p that shares $\left[\overline{s}^{:\Lambda \triangleright \lambda_1 \lambda_2 \ldots \lambda_l}\right]_j$ such that $j \in I_q$ holds have been computed.

Figure 6: Two Phase Share Refreshing Protocol.

server q begins its execution of the Generate and Propagate Subshares phase of the server protocol.

During this phase, each old share labeled Λ and stored by a correct server q is split into a subsharing (step (a)). Since every share is stored by at least one correct server (due to $\mathcal{I}(p)$ and the way share sets have been constructed), at least one subsharing will be produced for every old share, even if as many as t servers are compromised. Each individual subshare is sent in a store_r message to every server r that might require this subshare in constructing new shares (step (b)). The labels for each of these subsharings are also sent to C_p (step (c)) in contribute messages.

C_p next selects a candidate set of subsharings and uses compute messages to inform all servers of the label[11] $\Lambda \triangleright \lambda_1 \lambda_2 \ldots \lambda_l$ for a new sharing defined by that candidate set of subsharings (step (3)). Correct servers q now enter their Construct New Shares phases and calculate new shares (step (a)) based on the candidate set of subsharings identified by the label in the compute message received from C_p and the subshares received from other servers in store_q messages. Correct servers then send computed messages to notify C_p that new shares have been computed (step (b)).

Once the coordinator receives computed messages (step (4)) from $2t+1$ servers then, because of the way share sets have been constructed, for each old share there exists some correct server that has generated and stored the corresponding new share. C_p now updates variable V_p (step (4a)) and initiates a protocol (step (4b)) to delete old shares, subshares, and other information that would allow inferences about the old shares. The details of this share deletion protocol are fleshed out in §4.3.

4.2 Version 2: Send-Omission Failures and Fair Links

The above protocol assumes that up until a sender becomes compromised (and halts), it has sent all messages specified by the protocol. That assumption is unrealistic. It is also unrealistic to assume that messages sent are always delivered, never corrupted, and never disclosed. Therefore, in this subsection, we relax that Secure and Reliable Links assumption, returning to the Fair Links assumption of §2.1, and we relax Crash-Disclosure failures to the following.

[11] Forming new labels by concatenating old labels facilitates a simple syntactic test for evaluating the $\prec$ and $\simeq$ relations on labels.

Send-Omission Failures: A compromised processor may disclose locally stored information, halt prematurely, and/or send only a subset of the messages specified by its protocol.

Assume at most t processors are compromised in each epoch, where $3t + 1 \leq n$ holds. ⊠

Additional defenses must be brought to bear now that adversaries can insert, modify, delete and/or replay messages, and now that a compromised server might send only a subset of the messages specified by its protocols:

- To deal with messages not sent by compromised processors and message deletions due to Fair Links, acknowledgments are employed in conjunction with message retransmission. And where this does not work, additional messages are sent to retrieve information that should have been but never was delivered.

- To tolerate message insertion and modification, senders use their individual keys and sign messages; receivers check those signatures to detect and discard extraneous or corrupted messages.

- To combat replay attacks, information (e.g., the version number of the run) is added that allows detection of duplicates, and processors store for the current run each message received along with the response it elicited. Message processing is then made idempotent by checking to see when a message is received whether it is a duplicate of a message previously received in this run and, if it is, replying with the previously stored response.[12]

Henceforth, we presume—without mention—that signature checking is included in the semantics of our send and receive primitives. The details for message retransmission and acknowledgments are a bit subtle, however, so they are given explicitly.

The protocols of Figure 5 and Figure 6 are now replaced by a set of concurrent *protocol actions* (separated by "||" when they appear within the same figure): Coord-1, Gen-1, Gen-2, Const-1, and Const-2. See Figures 7 through 9. Protocol actions are not atomic—each comprises a sequence of labeled *steps* that specify sequential execution, typically in response to

[12]Responses to duplicates must be sent, because the prior response might have been a victim of the Fair Links assumption.

receipt of a specific type of message. So, for instance, protocol action Const-1 (Figure 9) is started upon receipt of a compute message, and its step Const-1a executes to completion before step Const-1b is started.

Some protocol actions employ repeated sending of messages in order to overcome compromised processors that do not perform all sends specified by their protocols and in order to overcome message loss or deletion. The description of such a send always specifies what messages the sender must receive in order for that repeated sending to stop. Repeated sending is not always necessary, though—a reply r to some message m need not employ repeated sending if r would be regenerated in response to the repeated sending of m.

Repeated sending alone does not suffice, however. Additional message exchanges also have to be added. Consider a server q that propagates subshares by repeatedly sending store messages to other servers. Any server r receiving such a store message might respond with a stored message as an acknowledgment, and the repeated sending would terminate when $n-t$ stored messages are returned.[13] But after receiving $n-t$ stored messages, some correct servers still might not have received a store message from q. These correct servers will ultimately need the subshares contained in that store message, so we add a new message exchange (the retrieve and retrieved messages of Figure 9) so these subshares can be fetched from other servers.

The revised protocol for coordinator C_p to refresh a sharing labeled Λ is implemented by protocol action Coord-1 in Figure 7. As before, execution starts with C_p sending an init(Λ) message (step Coord-1a) to all servers.

Protocol action Gen-1 (Figure 8) implements the Generate and Propagate Subshares phase of Figure 6 with the addition of repeated message sending to accommodate message loss. So, Gen-1 commences with the receipt of the coordinator's init(Λ) message, splits every subshare into a subsharing (step Gen-1a), repeatedly sends these subshares in store_r messages to other servers r that will require this information (step Gen-1b) until stored_r messages are received from $2t+1$ servers, and sends labels for these subshares back to coordinator C_p in contribute messages (step Gen-1c). Repeated sending of contribute messages is not necessary, because they are a reply to an init message which will be repeatedly resent until enough contribute messages are received.

[13] Server q is only guaranteed to receive $n-t$ stored messages, because as many as t servers might be compromised.

Coord-1: (a) C_p repeatedly sends to all servers an $\mathsf{init}(\Lambda)$ messages where $\Lambda = \mathtt{V}_p$ holds; sending stops when C_p receives $\mathsf{contribute}(\lambda_i)$ messages for some set of labels $\lambda_1,\ \lambda_2,\ \ldots\ ,\lambda_l$ comprising a candidate set of subsharings.

(b) C_p constructs label $\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l$ and repeatedly sends that to all servers in $\mathsf{compute}(\Lambda \triangleright \lambda_1, \lambda_2, \ldots, \lambda_l)$ messages; sending stops when a $\mathsf{computed}(\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l)$ message is received from $2t+1$ servers.

(c) Once $\mathsf{computed}(\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l)$ messages have been received from $2t+1$ servers:

i. $\mathtt{V}_p := \Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l$

ii. C_p runs the Share and Subshare Deletion Protocol.

Figure 7: Revised Protocol for Coordinator C_p.

Upon receipt of contribute messages containing enough different labels for subsharings (step Coord-1a) that servers have calculated, C_p selects a candidate set of subsharings and forwards the identifying label to all servers (step Coord-1b) using compute messages. Correct servers now calculate new shares based on the candidate set of subsharings identified by the label in that compute message. Const-1 (Figure 9) extends the Construct New Shares phase of Figure 6 with repeated sendings to overcome lost messages. Note also the addition of $\mathsf{retrieve}_j$ messages (Const-1(a)i and Const-2) so server q can, if necessary, fetch subshares that are prescribed by the coordinator's compute message but have not been received by q in store_q messages.

Once C_p receives computed messages from $2t+1$ servers (step Coord-1c), new shares have been calculated and stored in the share sets of at least $t+1$ correct servers. Variable $\mathtt{V}_p$ can now be updated to reflect this (step Coord-1(c)i), and the deletion of old shares and subshares can commence (step Coord-1(c)ii).

4.3 Version 3: Replicated Coordinators

The Correct Coordinator assumption is unrealistic. So next we modify the above protocol for use in environments where compromised coordinators C_p

Gen-1: Upon receipt of an $\mathsf{init}(\Lambda)$ message from coordinator C_p, server q does the following for each share $\left[\overline{s}^{:\Lambda}\right]_i$ in share set S_q^{Λ}:

(a) Calculates a subsharing for $\left[\overline{s}^{:\Lambda}\right]_i$:

$$\left[\overline{\left[\overline{s}^{:\Lambda}\right]_i}^{:\Lambda \circ C_p \circ i \circ q}\right] := \mathsf{split}(\left[\overline{s}^{:\Lambda}\right]_i).$$

(b) Repeatedly sends a $\mathsf{store}_r(\Lambda \circ C_p \circ i \circ q, x)$ message to every server r; sending stops when server q receives $\mathsf{stored}_r(\Lambda \circ C_p \circ i \circ q)$ messages from $2t+1$ different servers r. A $\mathsf{store}_r(\Lambda \circ C_p \circ i \circ q, x)$ message sent to r contains label $\Lambda \circ C_p \circ i \circ q$ along with (in x), for each $j \in I_r$, subshare $\left[\overline{\left[\overline{s}^{:\Lambda}\right]_i}^{:C_p \circ \Lambda \circ i \circ q}\right]_j$ encrypted in a way that only server p can decrypt.

(c) Sends to C_p a $\mathsf{contribute}(\Lambda \circ C_p \circ i \circ q)$ message that contains label $\Lambda \circ C_p \circ i \circ q$.

||

Gen-2: Each server r, upon receipt of a $\mathsf{store}_r(\Lambda \circ C_p \circ i \circ q, x)$ message from a server q, decrypts x, stores it, and replies to q, sending a $\mathsf{stored}_r(\Lambda \circ C_p \circ i \circ q)$ message.

Figure 8: Revised Generate and Propagate Subshares Phase.

satisfy Send-Omission Failures. Thus, we are making the assumptions of §2.1 for all processors, except that Compromised Processors is strengthened to Send-Omission Failures.

The idea behind this next version of the protocol is simple. By executing at least $t+1$ coordinators, each on a distinct processor, at least one executes on a correct processor. That correct coordinator executes the protocol of §4.2 and, therefore, produces the desired new sharing. The invariant for this new protocol, then, is:

$\hat{\mathcal{I}}$: For some correct processor p: $\mathcal{I}(p)$.

Of course with multiple coordinators, multiple new sharings could be pro-

Const-1: A server r, upon receipt of a $\mathsf{compute}(\Lambda \triangleright \lambda_1\lambda_2 \ldots \lambda_l)$ message:

(a) For each $j \in I_r$:

i. For each i such that $1 \leq i \leq l$: If $\left[\overline{\left[\overline{s}^{\,:\Lambda}\right]_i}^{\,:\lambda_i}\right]_j$ has not been received by r, then r repeatedly sends to all servers $\mathsf{retrieve}_j(\lambda_i)$; sending stops upon receipt of a $\mathsf{retrieved}_j(\lambda_i, x)$ message, whereupon x is decrypted to obtain $\left[\overline{\left[\overline{s}^{\,:\Lambda}\right]_i}^{\,:\lambda_i}\right]_j$.

ii. Server r executes:

$$\left[\overline{s}^{\,:\Lambda \triangleright \lambda_1\lambda_2 \ldots \lambda_l}\right]_j$$
$$:= \mathsf{reconstruct}(\left[\overline{\left[\overline{s}^{\,:\Lambda}\right]_1}^{\,:\lambda_1}\right]_j, \left[\overline{\left[\overline{s}^{\,:\Lambda}\right]_2}^{\,:\lambda_2}\right]_j, \ldots, \left[\overline{\left[\overline{s}^{\,:\Lambda}\right]_l}^{\,:\lambda_l}\right]_j)$$

and stores $\left[\overline{s}^{\,:\Lambda \triangleright \lambda_1\lambda_2 \ldots \lambda_l}\right]_j$ in share set $\mathsf{S}_r^{\Lambda \triangleright \lambda_1\lambda_2 \ldots \lambda_l}$.

(b) Server r sends to C_p a $\mathsf{computed}(\Lambda \triangleright \lambda_1\lambda_2 \ldots \lambda_l)$ message.

$||$

Const-2: A server q, upon receipt of a $\mathsf{retrieve}_j(\lambda_i)$ message from server r replies if $j \in I_q \cap I_r$ holds and $\left[\overline{\left[\overline{s}^{\,:\Lambda}\right]_i}^{\,:\lambda_i}\right]_j$ is stored at q. The reply is a $\mathsf{retrieved}_j(\lambda_i, x)$ message, where x equals $\left[\overline{\left[\overline{s}^{\,:\Lambda}\right]_i}^{\,:\lambda_i}\right]_j$ encrypted in a way that only server p can decrypt.

Figure 9: Revised Construct New Shares Phase.

duced. But note that the total number of new sharings stored by servers remains bounded by the number of coordinators, because each coordinator produces at most one new sharing.

A protocol involving multiple new sharings will satisfy the correctness specification in §2.3 provided:

- Adversaries cannot benefit from having access to the various new and old share sets that might now exist at compromised servers.

- Servers store and label the shares from the multiple sharings, so client protocols have a way to distinguish between shares that come from different sharings.

We next turn to these two issues.

Adversaries Cannot Exploit Multiple Share Sets. If the sharings produced by the concurrently executing coordinators were independent then adversaries would be unable to reconstruct the secret by combining the multiple share sets now at compromised servers. But the sharings produced by the above protocols are not necessarily independent. This is because different coordinators might use overlapping candidate sets of subsharings (since new sharings generated from common subsharings are not independent).

However, the lack of independence that accompanies overlapping candidate sets of subsharings cannot be exploited by adversaries. Each subshare $\left[\overline{\left[\overline{s}^{:\Lambda}\right]_i}^{:C_p \circ \Lambda \circ i \circ q}\right]_j$ (produced by Gen-1a) is used only in the construction of a new share with index j (see Const-1(a)ii), and shares with the same index j are (in share sets) stored by the same servers. Thus, shares that might not be independent are, in fact, stored at the same servers. So the lack of independence among shares does not help an adversary gain information about any shares not stored at compromised servers. (See Appendix A.3 for a full proof of APSS Secrecy.)

Client Management of Multiple New Sharings. Multiple new sharings can be produced with more than one coordinator. A client protocol must know which shares to use. This is easily arranged. By construction, distinct sharings have distinct labels. So by transmitting the labels, confusion is eliminated about which sharing is being used—each message containing a share or the result of a computation involving a share is accompanied by the label for the sharing that contains that share.

Share and Subshare Deletion Protocol

APSS Progress (§2.3) requires global termination of a run, which implies that all correct servers eventually delete old shares (along with subshares and other information derived from those shares) and then become quiescent. Shares and associated information can be deleted by overwriting server variables; quiescence can be achieved by aborting protocol actions that have not yet terminated. So that is what the protocol of this subsection does.

We conclude from invariant $\hat{\mathcal{I}}$ that, for non-empty $\mathtt{S}_q^\Lambda$, assignment statement

$$\mathtt{S}_q^\Lambda := \emptyset \tag{1}$$

deletes old shares at a server q provided $\Lambda \prec \mathtt{V}_p$ holds for some correct coordinator C_p. Thus, whenever any correct coordinator C_p assigns the label for a new sharing to $\mathtt{V}_p$, assignment statement (1) may be executed for non-empty server variables $\mathtt{S}_q^\Lambda$ where label Λ satisfies $\Lambda \prec \mathtt{V}_p$. Similar assignment statements would be employed for deleting subshares and other information that can be used to infer old shares and is stored in server variables. Since our descriptions of protocol actions have not named those variables explicitly, mention of executing assignment statement (1) should henceforth be taken to include overwriting the information stored in those, too.

Assignment statement (1) subjects variable $\mathtt{S}_q^\Lambda$ to concurrent access, as follows. A run of our protocol with multiple concurrent coordinators can lead to multiple instances of Gen-1 executing concurrently at each server. All Gen-1 instances that have received $\mathsf{init}(\Lambda)$ messages read $\mathtt{S}_q^\Lambda$. If any of those Gen-1 instances at a given server q has not terminated when assignment statement (1) is executed there, then $\mathtt{S}_q^\Lambda$ becomes the object of concurrent reads and writes. Concurrent access brings the possibility of interference [29]. And even though $\hat{\mathcal{I}}$ is, by design, not invalidated by assignment statement (1), the update to $\mathtt{S}_q^\Lambda$ does have the potential to cause interference by preventing quiescence. The following scenario illustrates.

Suppose $\mathtt{V}_r = \Lambda'$ holds because some coordinator C_r has reached step Coord-1(c)ii. Consider instances of protocol action Gen-1 associated with a coordinator C_p that executes concurrently with C_r and sends an $\mathsf{init}(\Lambda)$ message to start, where $\Lambda \prec \Lambda'$ holds. If at enough servers q, because assignment (1) has executed, $\emptyset = \mathtt{S}_q^\Lambda \subset S_q^\Lambda$ holds before the Gen-1 protocol action at q has sent a $\mathsf{contribute}$ message for every share in S_q^Λ, then C_p will not receive enough $\mathsf{contribute}$ messages to constitute a candidate set of subsharings. So C_p will remain forever in step Coord-1a awaiting additional $\mathsf{contribute}$ messages. C_p does not reach quiescence.

Quiescence in the face of this interference is easily arranged, though. Once any correct coordinator C_r has reached step Coord-1(c)ii, subsequent execution for this run by other coordinators and their associated protocol actions becomes unnecessary—with one new sharing already established, actions concerned with establishing a second new sharing serve no useful purpose and thus can be aborted.

To this end, associate with each executing protocol action an *instantiation identifier*. Protocol actions associated with a coordinator C that starts by sending an $\mathsf{init}(\Lambda)$ message are assigned instantiation identifier $\Lambda \circ C$, with relation $\preccurlyeq$ on instantiation identifiers defined by:

$$\Lambda \circ C \preccurlyeq \Lambda' \circ C' \text{ if and only if } (\Lambda \prec \Lambda') \vee (\Lambda \simeq \Lambda').$$

Quiescence is now easily achieved.

Protocol Action Abort Rule: Once execution of step Coord-1(c)ii with instantiation identifier $\Lambda \circ C_r$ has commenced, abort protocol actions Coord-1, Gen-1, Gen-2, Const-1, and Const-2 having instantiation identifiers $\mathcal{X}$ that satisfy $\mathcal{X} \preccurlyeq \Lambda \circ C_r$.

Note how aborting Coord-1 has the effect of aborting all concurrently executing coordinators, as we require.

The Share and Subshare Deletion Protocol is implemented by a protocol action Term-1 that is executed by each processor hosting a coordinator or server. Execution of Term-1 at a server q commences with receipt of a $\mathsf{henceforth}(\Lambda')$ message to signify that a new sharing with label Λ' has been disseminated and, therefore, the following holds.

$\mathcal{D}(\Lambda')$: For all processors q in some set comprising $t+1$ correct processors: $\mathsf{S}_q^{\Lambda'} = S_q^{\Lambda'}$ holds.

Sending a single $\mathsf{henceforth}(\Lambda')$ message to every processor does not suffice to ensure that Term-1 executes at each correct processor, because of Fair Links. Repeated sending also does not suffice—whatever protocol action is repeatedly sending the $\mathsf{henceforth}(\Lambda')$ might itself be aborted by the Protocol Action Abort Rule before the $\mathsf{henceforth}(\Lambda')$ message has been sent enough times to reach all correct processors.

One means to ensure that Term-1 is executed at all correct processors becomes feasible if coordinators are hosted only by servers. So, we now stipulate:

Pervasive Coordinator Deployment: Each of the n servers p also concurrently executes a coordinator C_p.

Since $3t + 1 \leq n$ holds, the goal of having at least $t + 1$ coordinators is achieved. And Pervasive Coordinator Deployment provides two additional points of leverage:

- A $\mathsf{henceforth}(\Lambda')$ message sent by a correct coordinator C_p to its hosting server p is always reliably delivered (despite the Fair Links assumption) because the network is not involved in this communication.
- If a server p is not quiescent because repeated sends in Coord-1, Gen-1, or Const-1, are executing, then either (a) coordinator C_p will reach step Coord-1(c)ii even if no subsequent messages are delivered to p or else (b) C_p is in the midst of executing or will start executing a repeated send.[14]

The first ensures that Term-1 will commence execution at some correct processor if a correct coordinator that reaches step Coord-1(c)ii sends a $\mathsf{henceforth}(\Lambda')$ message to all servers, since any message sent by a correct coordinator C_p will be received by server p.[15] The second enables system-wide dissemination of $\mathsf{henceforth}(\Lambda')$ messages: Term-1 replying with $\mathsf{henceforth}(\Lambda')$ to every message received suffices to activate Term-1 at every correct server, as we now show.

Consider a correct server p that has not received a $\mathsf{henceforth}(\Lambda')$ message and is not quiescent. And suppose Term-1 is executing at some correct server q. We must show that Term-1 will be activated at server p. There are two cases:

Case 1. Execution of C_p will reach step Coord-1(c)ii even if no subsequent messages are delivered to server p. C_p will thus send a $\mathsf{henceforth}(\Lambda')$ when it reaches step Coord-1(c)ii and, because C_p is correct and that message does not traverse the network, execution of Term-1 at p is guaranteed.

Case 2. C_p is in the midst of executing or will start executing a repeated send. Because C_p is correct, every send will cause a message to be

[14]Here is a proof sketch. If neither (a) nor (b) holds, then C_p must have been aborted. But all protocol actions on server p must then have been aborted and, therefore, server p is quiescent. Thus, if neither (a) nor (b) holds then server p is quiescent. Taking the contrapositive yields what we seek.

[15]In fact, it suffices for C_p to send the $\mathsf{henceforth}(\Lambda')$ message only to server p. But $\mathsf{henceforth}(\Lambda')$ messages reaching the other correct servers will hasten termination of the run.

Term-1: A server q, upon receipt of a $\mathsf{henceforth}(\Lambda')$ message:

(a) Abort protocol actions Coord-1, Gen-1, Gen-2, Const-1, and Const-2 executing at server q and associated with instantiation identifiers $\mathcal{X}$ such that $\mathcal{X} \preccurlyeq \Lambda' \circ C_p$ holds.

(b) If $\mathtt{V}_q \prec \Lambda'$ then $\mathtt{V}_q := \Lambda'$

(c) For every label Λ such that $\Lambda \prec \Lambda'$ holds:

- $\mathsf{S}_q^{\Lambda} := \emptyset$
- Delete all locally stored subshares generated from those shares.

(d) Thereafter, upon receipt of any message containing a label Λ such that $\Lambda \prec \Lambda'$ holds, q replies with a $\mathsf{henceforth}(\mathtt{V}_q)$ message.

Figure 10: Share and Subshare Deletion Protocol.

sent. Repeated sends in Coord-1 (Figure 7) are directed to all servers. Thus, eventually a message m from C_p will reach server q or C_p will reach step Coord-1(c)ii. If the latter, Case 1 applies. If the former, then eventually some copy of m will be received by Term-1 at q, and a $\mathsf{henceforth}(\Lambda')$ response will eventually be received by p.

Figure 10 pulls together the pieces for the Term-1 protocol action of the Share and Subshare Deletion Protocol just developed. When executed at a server q, step Term-1a runs the Protocol Action Abort Rule; step Term-1b establishes the second conjunct of invariant $\mathcal{I}(q)$ (because $\mathsf{henceforth}(\Lambda')$ messages signify that $\mathcal{D}(\Lambda')$ holds) and that in turn implies $\hat{\mathcal{I}}$; and step Term-1c deletes any old share information being stored on q.

To complete the picture, some means for C_p to run the share and subshare deletion protocol, as specified in in step Coord-1(c)ii of Figure 7, must be given. It is:

Coord-1(c)ii : C_p sends a $\mathsf{henceforth}(\mathtt{V}_p)$ message to all servers.

Anarchy and Coordination

By definition, a run of the multiple coordinator protocol just outlined starts when any coordinator begins step Coord-1a. Notice that correctness of the protocol in no way depends on how that decision to start is made. Moreover, the different coordinators need not be synchronized in any way. One or more correct coordinators that store version $v-1$ shares will be the first to complete run v and compute version v sharings. Thereafter, any tardy coordinator C_q sending init messages to start a run v will in reply receive a $\mathsf{henceforth}(\Lambda')$ message, where Λ' is the label on some version v sharing whose shares servers already store. Receipt of this $\mathsf{henceforth}(\Lambda')$ message by C_q causes protocol action Term-1 to execute and terminate run v locally.

In an actual APSS deployment, bounding the real-time that can elapse between protocol runs might be important. Frequent runs prevent client protocols from performing useful work; infrequent runs mean long windows of vulnerability. These difficulties can be avoided if local processor clocks are available. Frequent runs are prevented by making assumptions about the relative clock rates at different processors—messages to start a new epoch are ignored until enough time has elapsed on the receiver's local clock since the old epoch started. Infrequent runs are prevented by making assumptions about the real-time rates of processor clocks and having coordinators use elapsed time to trigger the start of a new run.

Note, a system in which local processor clocks are used in this way does not violate the Asynchronous System assumption. So APSS continues working correctly even if assumptions about local processor clocks are violated. What is affected by invalidating the local processor clock assumptions is guarantees about the elapsed time between share refreshings, but such guarantees are not part of the correctness requirements (§2.3) for APSS.

4.4 Version 4: Defending Against Malicious Servers

The effects of compromised processors are visible only through the messages they send. So a mechanism that allows correct processors to detect and ignore bogus messages constitutes the obvious defense, since it transforms an environment satisfying the Compromised Processors assumption into one satisfying the Send-Omission Failures assumption (§4.1), an environment in which the protocol of §4.3 already works.

Messages convey assertions about current and past system states. In

APSS, a logical formula can be associated with each type[16] of message; values in a message serve as arguments to the formula. Substitution of these arguments into the formula produces the assertion being conveyed. See Figure 11 for the assertions APSS messages convey. That these assertions suffice is demonstrated by their adequacy for the APSS protocol correctness proof given in the Appendix.

Define a message to be *valid* during an APSS execution if the associated assertion is *true* when that message is sent; otherwise, the message is considered *invalid*. Correct processors send only valid messages; invalid messages are sent only by compromised processors. A *self-verifying* message is one for which the receiver can determine whether that message is valid based solely on values contained in the message. Clearly, having all messages be self-verifying would allow correct processors to ignore invalid messages they receive.

Some messages are self-verifying because they are always valid. Both stored_r and $\mathsf{computed}$ messages are examples. The assertion associated with each is of the form

SVA: "If r is correct then $A(r)$ holds"

where r is the message sender and $A(r)$ concerns the local state of r. Assertion SVA trivially holds if r is compromised, because the antecedent of SVA is then *false*; and if r is correct, then $A(r)$ must hold because correct processors follow the protocol, which presumably has $A(r)$ *true* before sending the message.

Most of the messages in APSS are not self-verifying. Some (e.g., store_r and $\mathsf{retrieved}_j$) make assertions about secret sharings, shares, and labels; others (e.g., init, $\mathsf{contribute}$, $\mathsf{compute}$, $\mathsf{retrieve}_j$, and $\mathsf{henceforth}$) make assertions that correlate states at processors. But all can be made self-verifying if senders include additional information in messages. Two building blocks are involved:

- Verifiable secret sharing [10], which provides a way to check that labels, shares, and sharings are what a sender purports.
- Inference of predicates from collections of $2t + 1$ messages with only t from compromised servers.

[16] Message types have all along been given in sans serif and include: init, $\mathsf{contribute}$, $\mathsf{compute}$, $\mathsf{computed}$, store_r, stored_r $\mathsf{retrieve}_r$, $\mathsf{retrieved}_r$, and $\mathsf{henceforth}$.

Messages	Assertions	Attachments (Labels are assumed to contain validity checks)
$\mathsf{init}(\Lambda)$	A1. Each share of sharing Λ was stored at one or more servers that are correct within the previous epoch.	Attach the $\mathsf{henceforth}(\Lambda)$ message (obtained during the previous run in step Term-1, Figure 10) to the init message.
$\mathsf{store}_r(\Lambda \circ C_p \circ i \circ q, x)$	A2. $x = \{ \left[\overline{\left[\overline{s}\colon \Lambda\right]_i}\colon \Lambda \circ C_p \circ i \circ q \right]_j \mid j \in I_r \}$.	Employ verifiable secret sharing.
$\mathsf{stored}_r(\lambda)$	A3. If server r is correct, then $\{ \left[\overline{\left[\overline{s}\colon \Lambda\right]_i}\colon \lambda \right]_j \mid j \in I_r \}$ was stored on server r.	Always valid.
$\mathsf{contribute}(\lambda)$	A4. Every subshare of subsharing λ was stored at one or more correct servers.	Attach the $\mathsf{stored}_r(\lambda)$ messages received from $2t+1$ different servers r (step Gen-1b, Figure 8).
$\mathsf{compute}(\Lambda \triangleright \lambda_1 \lambda_2 \ldots \lambda_l)$	A5. For each subsharing λ_i $(1 \leq i \leq l)$, every subshare of the subsharing was stored at one or more correct servers.	Attach the $\mathsf{contribute}(\lambda_i)$ message (obtained in step Coord-1a, Figure 7) for each λ_i $(1 \leq i \leq l)$.
$\mathsf{computed}(\Lambda \triangleright \lambda_1 \lambda_2 \ldots \lambda_l)$	A6. If sender r is correct, then shares in $\{ \left[\overline{s}\colon \Lambda \triangleright \lambda_1 \lambda_2 \ldots \lambda_l\right]_j \mid j \in I_r \}$ were stored on server r.	Always valid.
$\mathsf{retrieve}_j(\lambda_i)$	A7. Every subshare of subsharing λ_i was stored on one or more correct servers.	Attach the $\mathsf{compute}(\Lambda \triangleright \lambda_1 \lambda_2 \ldots \lambda_l)$ message (obtained in step Const-1, Figure 9), where λ_i is one of the labels in the $\mathsf{compute}$ message. (In fact, only the $\mathsf{contribute}(\lambda_i)$ message, which can be found in the attachment to the $\mathsf{compute}(\Lambda \triangleright \lambda_1 \lambda_2 \ldots \lambda_l)$ message, needs to be attached.)
$\mathsf{retrieved}_j(\lambda_i, x)$	A8. $x = \left[\overline{\left[\overline{s}\colon \Lambda\right]_i}\colon \lambda_i \right]_j$	Employ verifiable secret sharing.
$\mathsf{henceforth}(\Lambda')$	A9. Each share of sharing Λ' was stored at one or more servers that are correct within the current epoch.	Attach the $\mathsf{computed}(\Lambda')$ messages received from $2t+1$ different servers (obtained in step Coord-1b, Figure 7).

Figure 11: Self-verifying messages in APSS: Assertions and Attachments

Verifiable Secret Sharing. Feldman's non-interactive verifiable secret sharing [15] provides the foundation for our scheme to check whether a given value x is, as purported, an element of some (l, l) standard secret sharing $\left[\overline{s}^{:\Lambda}\right]$, and to do that checking without knowledge of secret s or other shares comprising $\left[\overline{s}^{:\Lambda}\right]$.[17]

Without loss of generality, what follows is formulated in terms of secrets, shares, and sharings with the understanding that it also applies to subshares and subsharings, as they too are shares and sharings. For each x a secret, share, or sharing, we define $\lessdot x \gtrdot$, the *validity check* for x, in terms of a collision-resistant one-way function oneWay as follows.[18]

$$\lessdot x \gtrdot \triangleq \begin{cases} \mathsf{oneWay}(x) & \text{if } x \text{ is a secret or a share} \\ \langle \mathsf{oneWay}(x_1), \mathsf{oneWay}(x_2), & \text{if } x \text{ is a sharing comprising} \\ \quad \ldots, \mathsf{oneWay}(x_l) \rangle, & \qquad \text{shares } x_1, x_2, \ldots, x_l \end{cases} \tag{2}$$

Validity checks for secrets and shares can be made public, even if the secrets or shares themselves cannot, because by definition it is infeasible to compute the inverse of oneWay. This means that a function $\mathsf{vcConstr}$, which relates $\lessdot s \gtrdot$ to the validity checks for shares $s_1, s_2, \ldots, s_l$ that comprise a sharing of s,

$$\lessdot s \gtrdot = \mathsf{vcConstr}(\lessdot s_1 \gtrdot, \lessdot s_2 \gtrdot, \ldots, \lessdot s_l \gtrdot) \tag{3}$$

can provide a means for checking whether a share s_i in a message is as purported. It suffices that validity checks $\lessdot s \gtrdot$, $\lessdot s_1 \gtrdot$, $\lessdot s_2 \gtrdot$, ..., and $\lessdot s_l \gtrdot$ be publicly known or included in any message containing s_i so (3) can be evaluated when that message is received—if (3) holds then (with high probability) $s_1, s_2, \ldots, s_l$ are a sharing of s. Examples of the oneWay and $\mathsf{vcConstr}$ functions that satisfy (3) are given in [15].

The assertions for store_r and $\mathsf{retrieved}_j$ in Figure 11 are the only ones that involve shares (actually subshares). Both assertions require checking whether one or more values y in a message satisfy:

$$\mathit{ShareInteg}(y, \lambda, j): \quad y = \left[\overline{\left[\overline{s}^{:\Lambda}\right]_i}^{:\lambda}\right]_j$$

[17] Pedersen's verifiable secret sharing schemes [30] could also be used but provide different security guarantees than Feldman's. See [23] for a comparison.

[18] Notation $\langle v_1, v_2, \ldots, v_n \rangle$ is used to denote a vector whose values are $v_1, v_2, \ldots, v_n$.

To check whether $\mathit{ShareInteg}(y, \lambda, j)$ holds, it suffices that $\lessdot s \gtrdot$ be known to all servers and that the message containing y also contains validity checks $\langle a_1, a_2, \ldots, a_l \rangle$ purportedly for $\left[\overline{s}^{:\Lambda}\right]$ and $\langle b_1, b_2, \ldots, b_l \rangle$ purportedly for $\left[\overline{\left[\overline{s}^{:\Lambda}\right]_i}^{:\lambda}\right]$. Each of the following must then be checked.

(i) $b_j = \lessdot y \gtrdot$,

(ii) $a_i = \mathsf{vcConstr}(b_1, b_2, \ldots, b_l)$, and

(iii) $\lessdot s \gtrdot = \mathsf{vcConstr}(a_1, a_2, \ldots, a_l)$.

Condition (i) checks whether index j correctly identifies the share that y purportedly contains, condition (ii) checks whether the set of subshares with validity checks $b_1, b_2, \ldots, b_l$ constitutes a subsharing for a share with validity check a_i, and condition (iii) checks whether shares with validity checks $a_1, a_2, \ldots, a_l$ constitute a sharing of s.

Implicit in using (i) – (iii) to check $\mathit{ShareInteg}(y, \lambda, j)$ is the presumption that each sharing label uniquely identifies a sharing. If multiple sharings were given the same label then $\mathit{ShareInteg}(y, \lambda, j)$ and $\mathit{ShareInteg}(y', \lambda, j')$ might hold even though y and y' are not shares in the same sharing. In particular, APSS depends on subshares with the same label being from the same subsharing so that the subshares can be combined to obtain a (new) sharing of s.

Uniqueness of sharing and subsharing labels is straightforward to enforce. We include as part of each label the validity check for the sharing being labeled. This works because, with high probability, collision-resistant function oneWay produces different values for different sharings. The inclusion of validity checks in labels also has the practical advantage of providing needed information in messages containing shares, since shares in APSS messages are always accompanied by labels.

In revising the protocol of §4.3 to employ labels that include validity checks, steps must be added to generate those validity checks. There are two cases.

- **Subsharings.** New subsharings are generated in step Gen-1a (Figure 8). Validity check $\lessdot \left[\overline{\left[\overline{s}^{:\Lambda}\right]_i}^{:\Lambda \circ C_p \circ i \circ q}\right] \gtrdot$ for these new subsharings should be computed using (2).

- **Sharings.** New sharings are generated from the shares computed in step Const-1(a)ii. Validity check $\lessdot \left[\overline{s}^{:\Lambda \triangleright \lambda_1 \lambda_2 \ldots \lambda_l}\right] \gtrdot$ for the new sharing cannot be computed directly using (2), because no server computes or stores all the shares. However, from (3) we have

  $$\begin{aligned} &\lessdot \left[\overline{s}^{:\Lambda \triangleright \lambda_1 \lambda_2 \ldots \lambda_l}\right]_j \gtrdot \\ &\quad = \mathsf{vcConstr}\left(\lessdot \left[\overline{\left[\overline{s}^{:\Lambda}\right]_1}^{:\lambda_1}\right]_j \gtrdot, \lessdot \left[\overline{\left[\overline{s}^{:\Lambda}\right]_2}^{:\lambda_2}\right]_j \gtrdot, \ldots, \lessdot \left[\overline{\left[\overline{s}^{:\Lambda}\right]_l}^{:\lambda_l}\right]_j \gtrdot\right), \end{aligned}$$

  so the desired validity check is the right-hand side of this equation.

With validity checks now included in labels, a significantly terser representation of labels becomes possible. APSS performs only certain operations on labels:

- It evaluates relations $\prec$ on labels and $\preccurlyeq$ on instantiation identifiers (which in turn requires evaluating $\prec$ and $\simeq$ on labels) in Term-1 (Figure 10).
- It uses labels to distinguish different shares and sharings produced in the same run by different coordinators as part of Client Management of Multiple New Sharings (§4.3).
- It uses validity checks in labels to make messages self-verifying.

The recursive label construction used by APSS above supports these operations, but label length grows exponentially with the number of runs. Better is to represent the label on a sharing $\left[\overline{s}^{:\Lambda}\right]$ by the pair $\langle v, \lessdot \left[\overline{s}^{:\Lambda}\right] \gtrdot \rangle$, where v is the run in which the sharing was created, and to represent the label on a share $\left[\overline{s}^{:\Lambda}\right]_i$ by the label $\langle v, \lessdot \left[\overline{s}^{:\Lambda}\right] \gtrdot \rangle$ on the sharing and the index i. Such labels grow only logarithmically with the number of runs (because the size of v is logarithmic in the number of runs).

The label on a subsharing $\left[\overline{\left[\overline{s}^{:\Lambda}\right]_i}^{:\lambda_i}\right]$ must not only distinguish among different subsharings but must also convey the *source share* $\left[\overline{s}^{:\Lambda}\right]_i$ from which that subsharing is derived.[19] Therefore, a terse encoding for label on a subsharing can be a 4-tuple

$$\langle v,\ \lessdot \left[\overline{s}^{:\Lambda}\right] \gtrdot,\ i,\ \lessdot \left[\overline{\left[\overline{s}^{:\Lambda}\right]_i}^{:\lambda_i}\right] \gtrdot \rangle,$$

[19] Information about the source share is required by step Coord-1a in determining whether subsharings constitute a candidate set of subsharings.

where $\langle v, \lessdot \left[\overline{s}^{:\Lambda}\right] \gtrdot\rangle$ is the label on the sharing that provides the source share.

Inferring Predicates from Collections of Messages. Tests and attachments that make messages self-verifying compose in the expected way. Recall that a self-verifying message m' attached to another is signed by the sender of m', so validity of m' can be determined not only by the original recipient but by any server obtaining a copy of m'. Thus, if a message m' is self-verifying for assertion A' then attaching m' (including the attachments needed to make m' self-verifying) to a message m makes m self-verifying for A'. Such composition of self-verifying messages is used in Figure 11 for init, $\mathsf{compute}$, and $\mathsf{retrieve}_j$ messages.

Second, a set of messages sent by $2t + 1$ distinct processors during the same run will necessarily contain messages from $t+1$ correct processors. Even without knowing exactly which of these $2t+1$ messages were sent by correct processors, a receiver can deduce interesting assertions about the system state from the $2t + 1$ messages. For example, receipt of $2t + 1$ $\mathsf{computed}$ messages from distinct processors in the same run implies that $t+1$ of those messages are valid and, therefore, every share has been stored at some correct server because of property SS1 of share sets. Both $\mathsf{henceforth}$ and $\mathsf{contribute}$ messages are made self-verifying in Figure 11 by attaching sets of $2t + 1$ messages in this manner.

5 Variations on a Theme

5.1 Protocol Optimizations

Cost was not a paramount concern in our choice of mechanisms to defend against compromised processors. Rather, we selected defenses that facilitated a straightforward derivation of the protocol. With a simple and correct protocol now in hand, we turn in this section to optimizations for reducing its run-time costs.

Absence of Attacks. A system comprising all correct processors would not need the redundant processing that replicated coordinators bring. Moreover, in the absence of denial of service attacks, it becomes reasonable to assume that the system is synchronous. More efficient protocols are possible when correct processors satisfy the (stronger) assumptions of a synchronous system. So here is an opportunity to eliminate unnecessary work in settings where compromise and attack is rare—the likely case when the number of processors is small.

The key insight for the optimization is to note that actions can be delayed arbitrarily in protocols that work correctly in asynchronous systems. APSS works correctly in asynchronous systems and, therefore, the execution of all but one correct coordinator could be delayed without ill effect. In particular, an optimized version of APSS is obtained by (i) employing a single coordinator C_p whose execution is not delayed and (ii) delaying other coordinators so they do not start executing until C_p should have finished were it correct and the system synchronous.

So in the absence of attacks, C_p will propagate a henceforth message containing the label for a new sharing. Receipt of this henceforth message causes the remaining (delayed) coordinators and other protocol actions to terminate (due to Term-1 of Figure 10). A new sharing is thus produced without redundant coordinators. If, on the other hand, C_p does not succeed in producing a new sharing—because it is compromised or there is a denial of service attack—then the other coordinators' delays will expire. These other coordinators will execute their Coord-1 actions and produce a new sharing, albeit delayed relative to an APSS implementation in which no coordinator is delayed at the start.

Weaker Assertions. For certain messages in APSS, the assertions given in Figure 11 can be weakened without affecting correctness. Such weakenings offer the potential for improved performance when the cost to senders of adding attachments and/or the cost to receivers of checking attachments is lowered.

For example, the assertion that Figure 11 associates with $\mathsf{retrieve}_j(\lambda_i)$ messages—that the information being sought is available on correct servers—is the obvious precondition for a response. That assertion, however, can be weakened to *true*, making all $\mathsf{retrieve}_j(\lambda_i)$ messages trivially valid. No checking would need to be performed by the recipient of a $\mathsf{retrieve}_j(\lambda_i)$ message and no attachments would need to be provided by the sender, so the weakening yields a cheaper protocol. Protocol correctness is unaffected by a receiver ignoring $\mathsf{retrieve}_j(\lambda_i)$ messages conveying requests it cannot satisfy; protocol correctness is also unaffected by additional executions of Const-2 to reply to $\mathsf{retrieve}_j(\lambda_i)$ messages that it can satisfy.

Another weakening-based optimization is possible for stored_r and $\mathsf{retrieved}_j$ messages, because the existence of bogus subshares these messages convey can be detected later, when the subshares are combined to form the new shares. That detection would be done in connection with step Const-1(a)ii to compute each new share $\left[\overline{s}^{:\Lambda \triangleright \lambda_1\lambda_2\ldots\lambda_l}\right]_j$. Provided validity checks for all subshares computed in a run are included in the stored_r and $\mathsf{retrieved}_j$ messages, (even though only a subset of the subshares are), each processor can use $\mathsf{vcConstr}$ and (3) to compute the validity checks for all new shares. If the result of combining those validity checks using $\mathsf{vcConstr}$ does not yield $\lessdot s \gtrdot$ then at least one of the shares must be bogus, which implies that at least one of the subshares was bogus too. APSS must then be restarted at step Coord-1 but no longer ignoring the validity checks received with subshares in stored_r and $\mathsf{retrieved}_j$ messages.

In this optimization, the assertions Figure 11 associates with $\mathsf{store}_r(\Lambda, x)$ and $\mathsf{retrieved}_j(\lambda_i, x)$ are being weakened. Each assertion $A(r)$ is transformed into a valid assertion by replacing it with SVA. Validity checks for subshares are still attached to the message, but the receiver no longer checks the subshare received in each message against the corresponding validity check—shares are being checked instead. And there are fewer shares to check than subshares, because every new sharing is generated from l subsharings. The subshares do have to be checked, though, if the validity check for a share has failed, so the optimization yields a savings only when server compromise is rare.

5.2 Mitigating Denial of Service Attacks

APSS was designed to ensure that correctness properties APSS Availability, APSS Progress, and APSS Secrecy cannot be violated by the actions of t or fewer compromised servers. Nevertheless, denial of service attacks do allow compromised servers to slow down processing, and causing servers to generate large numbers of subsharings and/or sharings is a particularly effective way to accomplish that. In particular, a compromised processor could launch such an attack by sending init messages containing different labels Λ in step Coord-1a (Figure 7), by generating multiple different subsharings from each share in step Gen-1a (Figure 8) and propagating these subshares using store messages to servers in step Gen-1b (Figure 8), and by sending compute messages containing different choices of candidate sets of subsharings in step Coord-1b (Figure 7).

We can eliminate these APSS vulnerabilities by requiring that correct servers never process two messages of the same type and *pedigree* but bearing different content, where two messages are defined to have the same pedigree if they are sent on behalf of the same coordinator, in the same run, and by the same sender. This defense can be implemented if a server stores the first message of each type and pedigree that it receives in the current run and then discards subsequent incoming messages of the same type and pedigree but containing different content. Messages from a correct processor will never be discarded, because no correct processor ever sends two messages with the same type and pedigree but different content. Therefore, correctness of the APSS protocol is unaffected by this denial of service defense.

5.3 General and Dynamic Adversary Structures

An *adversary structure* [21] is a collection of sets of processors, where each set names processors that might all be compromised in the same epoch and all subsets of each set in the collection are also in the collection. APSS was described for t *threshold* adversary structures—collections of all sets containing t or fewer processors. Other adversary structures exist. And for a given system and threat, these other adversary structures might well be better models of what sets of processors could be compromised during a window of vulnerability.

APSS is easily extended to accommodate arbitrary adversary structures. For this purpose, it is helpful to recall that APSS

- refers frequently to the sets of $t + 1$ servers and to the sets of $2t + 1$ servers, and
- employs the construction of [22] to formulate share sets.

Two important things about the sets containing $t + 1$ servers are: (i) each such set contains at least one correct server, and (ii) the secret can be reconstructed from shares stored on the $t + 1$ servers. The collection of sets containing $2t + 1$ servers also is interesting, as an instance of a *dissemination Byzantine quorum system* [26] with each such set of servers a *quorum*, and therefore the following holds.

Quorum Availability: There always exists a quorum consisting only of correct servers. ⊠

Quorum Intersection: The intersection of any two quorums contains a correct processor. ⊠

Moreover, the quorums in APSS also satisfy

Quorum Recoverability: Share sets at correct servers in a quorum contain sufficient shares to reconstruct the secret. ⊠

so we will refer to them as forming a *recoverable quorum system*.

The construction in [22], in fact, encompasses the generation of share sets for arbitrary adversary structures such that a secret is reconstructible by a set of servers R if and only if R is not in the adversary structure. And provided no three sets in an adversary structure $\mathcal{A}$ cover the set of U all servers, then the collection $\mathcal{Q} = \{U - F | F \in \mathcal{A}\}$ of server sets will constitute a recoverable quorum system[20] because $\mathcal{Q}$ satisfies Quorum Availability, Quorum Intersection, and Quorum Recoverability.[21] Thus, simply replacing appearances of "$t + 1$ servers" in the description of APSS with "set of servers not in the adversary structure" (so above conditions (i) and (ii) hold for the new sets) and replacing appearances of "$2t + 1$ servers" with "quorum of servers" yields a variant of APSS for any given adversary structures.

It is also not difficult to generalize APSS to support dynamically changing sets of servers and dynamically changing adversary structures.[22] Given old and new server sets and adversary structures, a run must delete all old shares on correct old servers and store new shares on the new servers. This is accomplished if

- step Gen-1b is changed so that old servers now generate and propagate the subsharings to the new servers based on the new adversary structure, and

- step Coord-1(c)ii is changed to forward henceforth messages to both the old and new servers.

[20] Note, recoverable quorum systems that have smaller quorums than $\mathcal{Q}$ might also exist.

[21] See [26] for a proof that Quorum Availability and Quorum Intersection hold. $\mathcal{Q}$ satisfies Quorum Recoverability, because $Q - F \notin \mathcal{A}$ for any $Q \in \mathcal{Q}$ and $F \in \mathcal{A}$.

[22] The underlying mathematics for schemes that allow such dynamic changes of sets of servers and adversary structures is discussed in [14].

5.4 Share Refreshing and Recovery Alternatives

When viewed at a high level, the scheme APSS uses to construct new shares involves three steps:

(i) Servers generate information (*viz.* subshares) from the shares they hold.

(ii) Different subsets of that information are forwarded to various other servers.

(iii) Each server then uses the subsets it receives to compute its new shares.

Steps (i) and (iii) together comprise an implementation of share refreshing. And implicit in step (ii) is a need for a *share recovery* mechanism so that servers can obtain information they need in step (iii) despite the Compromised Processors and Fair Links assumptions. Alternatives to the share refreshing and share recovery employed by APSS are possible; they are the subject of this subsection.

Share Refreshing. Jarecki [23] describes a share refreshing scheme that exploits the property that share-wise addition of any sharing of 0 with a sharing of s yields a sharing of s. So given an old $(n, t+1)$ sharing $\left[\overline{s}^{:\Lambda}\right]$ of s, a new sharing $\left[\overline{s}^{:\Lambda'}\right]$ is obtained by generating an $(n, t+1)$ sharing $\left[\overline{0}^{:\nu}\right]$ of 0 and letting each new share $\left[\overline{s}^{:\Lambda'}\right]_i$ be $\left[\overline{s}^{:\Lambda}\right]_i + \left[\overline{0}^{:\nu}\right]_i$.

Jarecki's scheme can be used in APSS.[23] In action Gen-1, a server will generate a distinct, random sharing of 0; shares in this sharing of 0 are then distributed in store_r messages to the other servers, much like subshares are now distributed in Gen-1b; a label for the sharing of 0 is sent to the coordinator using a $\mathsf{contribute}$ message. And the coordinator, in step Coord-1b, selects $t+1$ sharings of 0 (to ensure that at least one of them is generated by a correct server)[24] and informs all servers of those selections with $\mathsf{compute}$

[23]That scheme would need to be extended, however, if the dynamically changing adversary structures discussed in §5.3 were being supported by APSS.

[24]It is sufficient, in fact, to have only t sharings of 0 generated and added to the old sharing. Because at most t servers are compromised in an epoch, even if all t sharings of 0 are created by compromised servers, no other servers are compromised in the new epoch or the old epoch (recall that a run belongs to both epochs). Therefore, the adversary will not be able to learn enough old or new shares to reconstruct the secret, even though it knows the transformation from the old sharing to the new one.

messages. Servers then add the shares in the selected sharings of 0 to their old shares in order to generate new shares.

Share Recovery. In APSS, every server calculates and stores subshares. Each of these subshares is stored by enough different servers so that copies remain available at the correct servers provided no more than t servers are ever compromised.

An alternative, inspired by a scheme proposed in [5] involving bivariate polynomials, is the following. In addition to sending subshares to servers, we employ an $(n, t + 1)$ standard secret sharing and store shares of each subshare—*subsubshares*—at $2t+1$ servers. A subshare can now be recovered by repeatedly requesting its subsubshares from all processors until $t + 1$ of those subsubshares have been received. In order to defend against receiving bogus subsubshares from compromised servers, verifiable secret sharing should be used when originally splitting the subshare with the $(n, t + 1)$ sharing.

6 Related Work

APSS, initially described in [36], is the first proactive secret sharing protocol for asynchronous systems. The derivation herein is new, many of the protocol extensions and variations discussed are new, and the proof given in the Appendix is also more rigorous than the one in [36].

The protocol derivation is interesting not only because it clarifies the role of each element in the protocol but because it suggests techniques for defending against various classes of attack, namely the transformations we use to produce protocols for the ever-weaker models of computation that correspond to an increasingly stronger adversary. The idea of obtaining a protocol for an inhospitable environment by performing a series of transformations was perhaps first employed in connection with fault-tolerance. See, for example, the automatic translations discussed in [27].

Theoretical papers on cryptographic protocols often presume the existence of transformations from a more hostile model to a less hostile one. An author might, for instance, assume reliable links rather than our more realistic Fair Links, in order to ignore the complications of message re-transmission and associated acknowledgment messages which might obscure the workings of a new protocol [4]. But this practice can mislead unless an explicit formulation is referenced for the transformation being applied. For example, according to [1], it is not possible in general to simulate reliable links and obtain a quiescent protocol by using Fair Links in the presence of benign failures. This implies that describing APSS as a protocol that uses reliable links would have constituted an over-simplification with no guarantee that a move to Fair Links was possible.

Another proactive secret sharing protocol for asynchronous systems was recently described in [5]. This protocol differs from APSS in two significant ways:

- It employs a randomized multi-valued validated Byzantine agreement protocol [6] so that all correct servers will agree on the set of subsharings to use in generating a new sharing for the secret. APSS eschews the agreement and instead generates multiple new sharings, each with a different label. There is no need to generate only one new sharing, so APSS does not need to run an agreement protocol.

- It employs a bivariate polynomial to implement subshare recovery. The scheme, which could be retrofit into APSS as discussed in §5.4, circumvents the exponential explosion that combinatorial secret sharing causes APSS; the protocol of [5] has polynomial-time communication and message complexity.

Proactive secret sharing was introduced in [23, 20] and is an instance of *proactive security*, introduced by Ostrovsky and Yung in [28] for multiparty computations. Herzberg *et al.* [19] give proactive schemes for discrete-logarithm based public key cryptography; Frankel *et al.* [17, 16] and T. Rabin [31] give proactive RSA schemes. All are designed for the synchronous system model, but all provide a set of modules to generate shares from a secret (a private key), to generate subshares from a share, and to create new shares from subshares (and possibly also the old shares). These same modules could be used with APSS to obtain corresponding proactive schemes that work in asynchronous systems. For example, APSS has been used to construct a proactive threshold RSA scheme for COCA.

Besides COCA, some other systems efforts are notable for their attempts to compose security with fault-tolerance and thus for the weak assumptions they make about the environment. Rampart [32, 33] implements process groups in an asynchronous distributed system where compromised processors can exhibit arbitrary behavior. BFT (Byzantine Fault-Tolerance) [9] and SINTRA (Secure Intrusion-tolerant Replication on the Internet) [7] are toolkits that support asynchronous group communication primitives along with proactive security. And the PASIS (Perpetually Available and Secure Information Systems) architecture [35] is intended to support a variety of approaches—including secret sharing and threshold cryptography in synchronous systems—that have been used in constructing survivable information storage systems. Apparently, a number of groups have concluded that secret sharing, algorithms for asynchronous systems, and support for proactive security are going to be increasingly important if distributed services deployed in open networks, like the Internet, must be trustworthy.

Acknowledgments

We are extremely grateful to Lorenzo Alvisi, Christian Cachin, and Michael Marsh for taking the time and trouble to provide comments on the first draft of this paper.

A APSS Correctness Proof

Proving the correctness of APSS involves demonstrating that APSS Availability, APSS Progress, and APSS Secrecy hold in systems satisfying the Compromised Processors, Fair Links, and Asynchronous Systems assumptions.

A.1 APSS Availability

To prove APSS Availability, it suffices to show that if a correct server deletes shares from $\left[\overline{s}^{:\Lambda}\right]$ then there exists a sharing with label Λ' (say) such that (i) $\Lambda \prec \Lambda'$ holds and (ii) each share of $\left[\overline{s}^{:\Lambda'}\right]$ is stored at one or more correct servers.

Let Λ be the label with highest version number on a sharing whose shares have been deleted and, therefore:

Av1: Some correct server has deleted shares of $\left[\overline{s}^{:\Lambda}\right]$.

Av2: For any label Λ'' satisfying $\Lambda \prec \Lambda''$, no correct server has deleted shares of $\left[\overline{s}^{:\Lambda''}\right]$.

According to Term-1 (Figure 10), a correct server deletes shares of $\left[\overline{s}^{:\Lambda}\right]$ only if that server receives a valid $\mathsf{henceforth}(\Lambda')$ message satisfying $\Lambda \prec \Lambda'$. We conclude from Figure 11 that validity of the $\mathsf{henceforth}(\Lambda')$ message implies A9 is true, so each share of $\left[\overline{s}^{:\Lambda'}\right]$ was stored at one or more servers that are correct within the current epoch. Because $\Lambda \prec \Lambda'$ holds, based on Av2 we have that no correct server has yet deleted shares of $\left[\overline{s}^{:\Lambda'}\right]$. Therefore, each share of $\left[\overline{s}^{:\Lambda'}\right]$ is stored at one or more correct servers. So the protocol satisfies APSS Availability.

A.2 APSS Progress

To establish APSS Progress, we start by proving a series of lemmas. The first establishes that a run will eventually terminate globally if a valid $\mathsf{henceforth}(\Lambda')$ message for a new sharing Λ' is sent or received in this run by some correct server. Such a henceforth message is called a *new* valid henceforth. The remaining lemmas show that repeated sends in various protocol actions eventually terminate if no new valid henceforth has been sent.

Lemma A.1 *If a correct server r has sent or received a new valid* henceforth(Λ') *message, where Λ' is the label for a new sharing generated in this run, then the run eventually terminates globally.*

Proof Sketch: Proof by contradiction. Assume the run does not terminate globally. Thus, there exists a correct server p that never deletes its old shares and subshares and/or never becomes quiescent. Due to Pervasive Coordinator Deployment, we conclude coordinator C_p has not executed Term-1 (Figure 10) in this run and must be executing Coord-1 (Figure 7). Consequently, C_p is repeatedly sending messages to all servers, including r.

According to step Term-1d, r will always reply to these messages from C_p with a henceforth(Λ') message. Server p eventually will receive one of these henceforth(Λ') messages from r, due to the Fair Links assumption. Upon receipt of this henceforth(Λ') message, p will begin executing Term-1. Therefore, p becomes quiescent and deletes its old shares and subshares, contradicting the earlier assumption about p. ■

Lemma A.2 *Suppose no correct server has sent or received a valid new* henceforth *message. Any correct server q executing step Gen-1b eventually receives $2t+1$* stored *messages for each subsharing q propagates and, therefore, the repeated sending in Gen-1b eventually terminates.*

Proof Sketch: According to the Compromised Processors assumption, at least $2t+1$ servers are correct. By hypothesis, no server has sent or received a valid new henceforth message, so no correct server is executing Term-1 (Figure 10). Any correct server p whose co-resident coordinator C_p has not started executing Term-1 in this run will respond to a store message from q by executing Gen-2 (Figure 8) and sending a stored message. Server q repeatedly sends store messages if it has not received $2t+1$ stored messages. According to the Fair Links assumption, one of these store messages will reach each of the $2t+1$ correct servers p and eventually one of the stored messages sent in reply by each correct server will reach q. Thus, eventually q will receive the $2t+1$ stored messages required for Gen-1b to terminate. ■

Lemma A.3 *Suppose no correct server has sent or received a valid new* henceforth *message. Any correct coordinator C_p executing step Coord-1a* *eventually receives* contribute *messages containing labels comprising a candidate set of subsharings and, therefore, the repeated sending in Coord-1a eventually terminates.*

Proof Sketch: Execution of step Coord-1a by correct coordinators involves repeated sending of valid $\mathsf{init}(\Lambda)$ messages to all servers. From assertion A1 in Figure 11 for $\mathsf{init}(\Lambda)$ messages, we conclude that each share of $\left[\overline{s}^{:\Lambda}\right]$ was stored on one or more servers that are correct within the previous epoch (hence correct within the current run). Moreover, no correct server has deleted these shares due to the hypothesis of the lemma that no correct server has sent or received a valid new henceforth message—shares are deleted only by Term-1 actions, and a henceforth message must have been sent or received for this action to execute. So, for each share $\left[\overline{s}^{:\Lambda}\right]_i$, there exists at least one correct server q_i (say) that stores this share.

The repeated sending of $\mathsf{init}(\Lambda)$ messages to all servers and the Fair Links assumption together imply that each server q_i will repeatedly receive $\mathsf{init}(\Lambda)$ messages until C_p receives enough contribute messages to exit step Coord-1a. If each q_i responds to every $\mathsf{init}(\Lambda)$ message with a contribute message for each share $\left[\overline{s}^{:\Lambda}\right]_i$ that q_i stores then eventually, due to the Fair Links assumption, contribute messages for every share in $\left[\overline{s}^{:\Lambda}\right]$ will reach C_p.

All that remains is establishing that correct servers q_i will respond with contribute messages for every share stored. Upon receipt of an $\mathsf{init}(\Lambda)$ message, q_i generates a subsharing for each share $\left[\overline{s}^{:\Lambda}\right]_i$ it stores (step Gen-1a) and propagates that subsharing to all servers (step Gen-1b). Step Gen-1b eventually terminates (Lemma A.2), so server q_i will reach step Gen-1c and send the contribute message sought by C_p. ■

Lemma A.4 *Suppose no correct server has sent or received a valid new* henceforth *message. Any correct coordinator C_p executing step Coord-1b eventually receives* computed *messages from $2t+1$ servers and, therefore, Coord-1b terminates.*

Proof Sketch: We show that a correct server r replies with a computed message to every $\mathsf{compute}(\Lambda \triangleright \lambda_1\lambda_2 \ldots \lambda_l)$ message that r receives from a correct coordinator C_p. With $2t+1$ correct servers, the repeated sending of $\mathsf{compute}(\Lambda \triangleright \lambda_1\lambda_2 \ldots \lambda_l)$ messages in step Coord-1b by C_p and the Fair Links assumption together imply that C_p will eventually receive computed messages from $2t+1$ servers. Consider two cases.

Case 1. Server r already stores all of the subshares it needs for step Const-1a (Figure 9). In this case, r will execute Const-1a and reply with the computed message.

Case 2. Server r does not already store all of the subshares it needs for step Const-1a (Figure 9). Server r must therefore retrieve the missing subshares in step Const-1(a)i.

From assertion A5 in Figure 11, we conclude from the valid compute($\Lambda \triangleright \lambda_1 \lambda_2 \ldots \lambda_l$) message sent by a correct coordinator C_p that every subshare of the subsharing λ_i ($1 \leq i \leq l$) was stored at one or more correct servers. Moreover, none of these subshares could have been deleted due to the hypothesis of the lemma that no correct server has sent or received a valid new henceforth message—shares are deleted only by Term-1 actions, and a henceforth message must have been sent or received for this action to execute. Server r is thus guaranteed to receive in a retrieved message any subshare it repeatedly requests with retrieve messages because at least one correct server stores that subshare and will respond to the server r with a retrieved message. Therefore, r will also proceed to step Const-1b and reply with a computed message. ■

Given these lemmas, proving APSS Progress is straightforward. There are two cases to consider:

Case 1. A correct server has sent or received new valid henceforth(Λ') message, where Λ' is the label for a new sharing generated in this run. Here, APSS Progress follows directly from Lemma A.1.

Case 2. No correct server has sent or received a new valid henceforth(Λ') message, where Λ' is the label for a new sharing generated in this run. Note, the hypothesis of Lemmas A.2 through A.4 hold. We show that a correct coordinator C_p is guaranteed to reach step Coord-1(c)ii (Figure 7) and send a new valid henceforth(Λ') message. Then, APSS Progress again follows directly from Lemma A.1.

An inspection of action Coord-1 which C_p executes reveals that execution can be prevented from reaching step Coord-1(c)ii only by blocking in:

- step Coord-1a waiting for contribute messages or

- step Coord-1b waiting for computed messages.

Lemma A.3 proves that a correct coordinator C_p will not be blocked forever in step Coord-1a, while Lemma A.4 shows that C_p would not be blocked forever in step Coord-1b. Therefore, C_p will reach step Coord-1(c)ii.

A.3 APSS Secrecy

The proof of APSS Secrecy is based on an indistinguishability argument. We prove that information available to the adversary is consistent with the sharing of any secret and, therefore, the adversary, by participating in executions of APSS that periodically refresh shares, learns nothing about the secret being shared. Specifically, we show how to transform an execution comprising a sequence of APSS runs periodically refreshing shares of a secret s into an execution that seems indistinguishable to the adversary and is periodically refreshing shares of a different secret, s'.

The proof of APSS Secrecy is relative to the following properties of the (l, l) secret sharing scheme, encryption, and validity checks used in APSS:

Se1: The adversary learns nothing about s from fewer than l shares of an (l, l) sharing $[\overline{s}]$.

Se2: The adversary learns no information about s or any share $[\overline{s}]_i$ from encrypted shares, encrypted subshares, or from validity checks of s, shares, or subshares.

Se1 is the defining characteristic of (l, l) secret sharing and applies both to recovering a secret from shares and recovering a share from subshares. Se2 implies that encryption and oneWay work as intended.[25] In light of Se2, we can ignore differences due to validity checks (e.g., in labels) for secrets s and s' in our indistinguishability argument.

The information $\chi_s(v)$ involved in a run v of APSS can be represented by the 4-tuple $(\mathcal{L}, \ell, \sigma, \mu)$ where

- $\mathcal{L}$ is the set of labels for sharings generated in run v,
- ℓ is the set of labels for subsharings generated in run v,

[25] Encrypted shares and validity checks are taken into account in the proof of [23], and those techniques would work here, too.

- σ maps labels of sharings to sharings (i.e., $\sigma(\Lambda)$ is the sharing with label Λ if $\Lambda \in \mathcal{L}$) and

- μ maps labels of subsharings to subsharings (i.e., $\mu(\lambda)$ is the subsharing with label λ if $\lambda \in \ell$).

Define $\chi_s|_A$ similarly but containing only information available to an adversary that has compromised the set A of processors. In light of Se2, $\chi_s|_A$ and $\chi_{s'}|_A$ are defined to be indistinguishable—denoted $\chi_s|_A \cong \chi_{s'}|_A$—provided only the different validity checks appearing as parts of labels on sharings and subsharings account for all the differences between $\chi_s|_A$ and $\chi_{s'}|_A$.

To prove that the adversary is unable to distinguish between a sharing of secret s and of any other secret s' from a domain $\mathcal{D}$, we will prove the following:

$$(\forall \chi_s :\ (\forall s' \in \mathcal{D} :\ (\exists \chi_{s'} :\ (\forall v :\ \chi_s(v)|_A \cong \chi_{s'}(v)|_A)))) \tag{4}$$

The proof is based on transforming a sharing for one secret (s) into a sharing for a different secret (s') by changing only specific shares and subshares (namely, those not visible to the adversary), and employs:

Sharing Transformation: Define $\tau(\left[\overline{s}^{:\Lambda}\right], i, s')$ to be a sharing of s', with $\tau(\left[\overline{s}^{:\Lambda}\right], i, s')_j$ denoting[26] its j^{th} share, such that

$$(\forall j :\ 1 \leq j \leq l \wedge j \neq i :\ \left[\overline{s}^{:\Lambda}\right]_j = \tau(\left[\overline{s}^{:\Lambda}\right], i, s')_j) \tag{5}$$

holds for any s and s' that are secrets from $\mathcal{D}$ and any $\left[\overline{s}^{:\Lambda}\right]$ that is an (l, l) standard secret sharing of s. $\boxtimes$

The existence of τ follows directly from Se1. Also note that the first argument to τ could be either a subsharing or a sharing.

Let R_v be the set of servers compromised in epoch v. From the Compromised Processors assumption, we have $|R_v| \leq t$ holds. Due to properties SS1 and SS2 (§3.1), for any sharing of s there exists an index i_v for some share not held by any compromised server:

$$(\forall p \in R_v :\ i_v \notin I_p) \tag{6}$$

[26] This use of subscripts to denote shares here and later is consistent with the notation for shares introduced in §3.

Thus, during a run v of APSS, no compromised servers have access to shares or subshares with index i_v generated in run v. This is because subshares with index i_v are generated by correct servers and propagated, encrypted, to correct servers only, so only correct servers are able to reconstruct shares with index i_v.

We prove (4) by giving a procedure to construct $\chi_{s'}$ from any given χ_s. We proceed by induction on the number of APSS runs that executed.

Base Case: $\chi_s = \chi_s(0)$. Prior to the first run of APSS in χ_s, a trusted entity has generated some sharing $\left[\overline{s}^{:\Lambda}\right]$ (say) of s and distributed shares to servers, according to the index sets. This sharing is the only sharing of s that exists in what we will refer to as run 0. No subsharings exist.

Without loss of generality, let $\chi_s(0)$ be $(\{\Lambda\}, \emptyset, \sigma_0, \mu_0)$. We construct $\chi_{s'}(0)$ to be $(\{\Lambda\}, \emptyset, \sigma'_0, \mu_0)$, where $\sigma'_0(\Lambda) = \tau(\sigma_0(\Lambda), i_v, s')$. holds. From (6) and the defining characteristic (5) of τ we conclude $\chi_s|_A \cong \chi_{s'}|_A$ holds.

Induction Step: χ_s *involves* w *or fewer runs.* We take as the induction hypothesis:

Induction Hypothesis: For all $v \leq w$, from $\chi_s(v-1) = (\mathcal{L}_{v-1}, \ell_{v-1}, \sigma_{v-1}, \mu_{v-1})$ we can construct[27] $\chi_{s'}(v-1) = (\mathcal{L}_{v-1}, \ell_{v-1}, \sigma'_{v-1}, \mu'_{v-1})$ satisfying:

$$\chi_s(v-1)|_A \cong \chi_{s'}(v-1)|_A$$

Given $\chi_s(v) = (\mathcal{L}_v, \ell_v, \sigma_v, \mu_v)$, we must construct $\chi_{s'}(v) = (\mathcal{L}_v, \ell_v, \sigma'_v, \mu'_v)$ in such a way that

(i) all subsharings in μ'_v are consistent with the shares that comprise the sharings in σ'_{v-1},

(ii) subsharings in μ'_v differ from corresponding subsharings in μ'_v only at subshares not known to compromised processors (namely, those subshares with indices i_v), and

[27] Strictly speaking, we should define $\chi_{s'}(v-1) = (\mathcal{L}'_{v-1}, \ell'_{v-1}, \sigma'_{v-1}, \mu'_{v-1})$. However, due to Se2, labels that differ only in the validity checks they contain are indistinguishable. So, it is safe to postulate that $\mathcal{L}'_{v-1} = \mathcal{L}_{v-1}$ and $\ell'_{v-1} = \ell_{v-1}$ hold rather than complicating the proof with the obvious mappings from sharing labels in $\mathcal{L}'_{v-1}$ to $\mathcal{L}_{v-1}$ and from subsharing labels in ℓ'_{v-1} to ℓ_{v-1}.

(iii) sharings in σ'_v differ from sharings in σ_v only at shares not known to compromised processors (namely, those shares with indices i_v) as well as being consistent with the subshares in μ'_v.

because then we can conclude $\chi_s(v)|_A \cong \chi_{s'}(v)|_A$ holds.

We start with the construction of μ'_v. For sharing labels Λ_{v-1} in $\mathcal{L}_{v-1}$, we have $\sigma'_{v-1}(\Lambda_{v-1})_j = \sigma_{v-1}(\Lambda_{v-1})_j$ holds if and only if $j \neq i_{v-1}$. We then define for subsharings λ of a share $\sigma_{v-1}(\Lambda_{v-1})_j$

$$\mu'_v(\lambda) = \begin{cases} \mu_v(\lambda) & \text{if } j \neq i_{v-1}, \\ \tau(\mu_v(\lambda), i_v, \sigma'_{v-1}(\Lambda_{v-1})_j) & \text{if } j = i_{v-1} \end{cases} \tag{7}$$

Note that this construction ensures $\mu'_v(\lambda)$ is a subsharing of share $\sigma'_{v-1}(\Lambda_{v-1})_j$ so (i) holds, and that subsharing $\mu'_v(\lambda)$ differs from subsharing $\mu_v(\lambda)$ only at index i_v, so (ii) holds as well.

We next give the construction of σ'_v. Each sharing $\Lambda \in \mathcal{L}_v$ is generated from a candidate set of subsharings $\lambda_1\lambda_2 \ldots \lambda_l$ where every λ_i is an element of ℓ_v:

$$\sigma_v(\Lambda)_j = \mathsf{reconstruct}(\mu_v(\lambda_1)_j, \mu_v(\lambda_2)_j, \ldots, \mu_v(\lambda_2)_j)$$

Define σ'_v similarly:

$$\sigma'_v(\Lambda)_j = \mathsf{reconstruct}(\mu'_v(\lambda_1)_j, \mu'_v(\lambda_2)_j, \ldots, \mu'_v(\lambda_2)_j)$$

We conclude that $\sigma'_v(\Lambda)_j$ is a sharing of s', as follows. From the induction hypothesis, $\sigma'_{v-1}(\Lambda_{v-1})_j$ constitutes a sharing of s'. Based on the above construction of μ'_v and on the manner in which $\mathsf{reconstruct}$ is used for share refreshing, we conclude for every Λ in $\mathcal{L}_v$ that $\sigma'_v(\Lambda)$ is also a sharing of s'.

Finally, due to the way in which μ'_v is constructed, $\mu_v(\lambda_i)_j = \mu'_v(\lambda_i)_j$ holds for any j satisfying $j \neq i_v$ and $1 \leq j \leq l$. Consequently, $\sigma_v(\Lambda)$ and $\sigma'_v(\Lambda)$ differ only at the shares and subshares with index i_v. We thus conclude $\chi_s(v)|_A \cong \chi_{s'}(v)|_A$, as required for (iii) above. And the induction case is proved: $\chi_s(v)|_A \cong \chi_{s'}(v)|_A$.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.

[2] G. R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the National Computer Conference*, 48, pages 313–317. American Federation of Information Processing Societies Proceedings, 1979.

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

[4] C. Cachin. Private communication.

[5] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. Unpublished draft, May 2002.

[6] C. Cachin, K. Kursawe, F. Petzold, and V. Shoulp. Secure and efficient asynchronous broadcast protocols (extended abstract). In J. Kilian, editor, *Advances in Cryptology — Crypto'2001 (Lecture Notes in Computer Science 2139)*, pages 524–541. Springer-Verlag, 2001.

[7] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002)*, pages 167–176. IEEE, June 2002.

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

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

[10] B. Chor, S. Goldwasser, S. Macali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneous broadcast. In *Proceedings of the*

26th Symposium on Foundations of Computer Science, pages 335–344, 1985.

[11] P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, October 1971.

[12] Y. Desmedt. Society and group oriented cryptography: A new concept. In C. Pomerance, editor, *Advances in Cryptology—Crypto'87, (Lecture Notes in Computer Science 293)*, pages 120–127, Santa Barbara, California, U.S.A., August 1988. Springer-Verlag.

[13] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *Advances in Cryptology — Crypto '89 (Lecture Notes in Computer Science 435)*, pages 307–315, Santa Barbara, California, U.S.A., August 1990. Springer-Verlag.

[14] 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.

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

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

[17] Y. Frankel, P. Gemmell, P. MacKenzie, and M. Yung. Proactive RSA. In B. Kaliski, editor, *Advances in Cryptology — Crypto '97 (Lecture Notes in Computer Science 1294)*, pages 440–454, Santa Barbara, California, U.S.A., August 1997. Springer-Verlag.

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

[19] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public-key and signature schemes. In *Proceedings of the Fourth Annual Conference on Computer Communications Security*, pages 100–110. ACM, 1997.

[20] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In D. Coppersmith, editor, *Advances in Cryptology — Crypto '95 (Lecture Notes in Computer Science 963)*, pages 457–469, Santa Barbara, California, U.S.A., August 1995. Springer-Verlag.

[21] M. Hirt and U. Maurer. Player simulation and general adversary structures in perfect multi-party computation. *Journal of Cryptology*, 13(1):31–60, 2000.

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

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

[24] S. C. Kothari. Generalized linear threshold scheme. In G. Blakley and D. Chaum, editors, *Advances in Cryptology, Proceedings of CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 231–241, Berlin, Germany, 1985. Springer-Verlag.

[25] L. Lamport. Concurrent reading while writing. *Communications of the ACM*, 20(11):806–811, 1977.

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

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

[28] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 51–59, 1991.

[29] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[30] T. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In J. Feigenbaum, editor, *Advances in Cryptology — Crypto'91 (Lecture Notes in Computer Science 576)*, pages 129–140, Santa Barbara, California, U.S.A., August 1992. Springer-Verlag.

[31] T. Rabin. A simplified approach to threshold and proactive RSA. In H. Krawczyk, editor, *Advances in Cryptology — Crypto'98 (Lecture Notes in Computer Science 1462)*, pages 89–104, Santa Barbara, California, U.S.A., August 1998. Springer-Verlag.

[32] M. K. Reiter. The Rampart toolkit for building high-integrity services. In K. P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems, International Workshop, Selected Papers*, volume 938 of *Lecture Notes in Computer Science*, pages 99–110, Berlin, Germany, 1995. Springer-Verlag.

[33] M. K. Reiter. Distributing trust with the Rampart toolkit. *Communications of the ACM*, 39(4):71–74, April 1996.

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

[35] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliççöte, and P. K. Khosla. Survivable information storage system. *IEEE Computer*, 33(8):61–68, August 2000.

[36] L. Zhou. *Towards Building Secure and Fault-tolerant On-line Services*. PhD thesis, Computer Science Department, Cornell University, Ithaca, NY USA, May 2001.

[37] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, To appear.