APSS: Proactive Secret Sharing in Asynchronous Systems

LIDONG ZHOU
Microsoft Research Silicon Valley
FRED B. SCHNEIDER and ROBBERT VAN RENESSE
Cornell University

APSS, a proactive secret sharing (PSS) protocol for asynchronous systems, is explained and proved correct. The protocol enables a set of secret shares to be periodically refreshed with a new, independent set, thereby thwarting mobile-adversary attacks. Protocols for asynchronous systems are inherently less vulnerable to denial-of-service attacks, which slow processor execution or delay message delivery. So APSS tolerates certain attacks that PSS protocols for synchronous systems cannot.

Categories and Subject Descriptors: C.2.0 [**Computer–Communication Networks**]: Security and Protection; C.2.4 [**Distributed Systems**]: Client/Server; D.4.6 [**Security and Protection**]: Cryptographic Protocols; E.3 [**Data Encryption**]: Public Key Cryptosystems

General Terms: Algorithms, Design, Security, Reliability, Theory

Additional Key Words and Phrases: Threshold cryptography, proactive secret sharing, denial of service, asynchronous system

1. INTRODUCTION

An $(n, t+1)$ *secret sharing* [Blakley 1979; Shamir 1979] 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

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.

Authors' addresses: Lidong Zhou, Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043; email: lidongz@microsoft.com; Fred B. Schneider and Robbert van Renesse, Department of Computer Science, Upson Hall, Cornell University, Ithaca, New York 14853; email: {fbs, rvr}@cs.cornell.edu.

and the secret remains confidential unless more than t servers have been compromised.

Secret sharing alone does not defend against *mobile adversaries* [Ostrovsky and Yung 1991], which attack, compromise, and control one server for a limited period before moving to another. Given enough 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, 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.

Secret sharing with share refreshing is known as *proactive secret sharing* (PSS) [Herzberg et al. 1995; Jarecki 1995]. 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 two consecutive executions of share refreshing.

Prior work on PSS protocols [Herzberg et al. 1995; Jarecki 1995] assumes a *synchronous system*, which implies bounds are known for message delivery delays and processor execution speeds. 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 assumptions 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 eliminate a vulnerability. Besides implementing secret sharing, APSS can be used for *threshold cryptography* [Boyd 1989; Desmedt 1988], where servers store shares of a private key and perform cryptographic operations using these shares (without ever materializing the entire private key). The particular secret sharing scheme we employ for APSS has the number of shares grow exponentially with t and is thus practical only if t is small (e.g., 1, 2, or 3); such small values of t are common when building distributed services.

The paper is organized as follows. In Section 2, the system model and the correctness requirements for APSS are specified. Then Section 3 discusses share refreshing and identifies challenges asynchronous systems bring to share refreshing. Those challenges are addressed in Sections 4 and 5, where the APSS protocol is presented. Various ways in which the protocol can be optimized and extended are explored in Section 6, followed by a discussion of related work in Section 7. A correctness proof for the APSS protocol appears as an appendix.

2. SYSTEM MODEL AND CORRECTNESS

Consider a system comprising a set of n *servers* that hold shares of a secret and communicate through a network. At any time, a server is either *correct* or *compromised*. A compromised server might stop executing, deviate arbitrarily from its specified protocols (i.e., Byzantine failure), and/or disclose or change information stored locally. A compromised server can be *recovered* and become correct after the following actions are taken.

- Reset hardware and system configurations
- Reload the code (thereby eliminating Trojan horses)
- Reconstitute the state of each server (which might have been corrupted)
- Obsolete any confidential information an attacker might have obtained from compromised servers.

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

2.1 Secret Sharing, Share Refreshing, and Secret Reconstruction

Servers store shares that constitute an $(n, t+1)$ secret sharing for s. The initial shares are generated by a trusted party and are assumed to have been delivered securely to all participating servers. Thereafter, servers replace old shares with new shares by periodically invoking share refreshing. *Secret reconstruction* is used to reassemble a secret from the shares when needed.[1]

Complications arise when executions of share refreshing is concurrent with secret reconstruction, because share refreshing might delete (old) shares that secret reconstruction needs. This is an instance of the well-known readers/writers problem [Courtois et al. 1971], with secret reconstruction the reader and share refreshing the writer. One solution, in the spirit of Lamport's [1977] concurrent reading while writing, is to use version numbers in conjunction with verifiable secret sharing [Chor et al. 1985]: execution of secret reconstruction detects whether shares it reads constitute a sharing of s and iterates if they do not.[2]

2.2 APSS Correctness Requirements

The objective of APSS is to provide a protocol for share refreshing so that the following properties hold.

APSS SECRECY: An adversary learns nothing about secret s. ☒

APSS INTEGRITY: At any time, with high probability, secret reconstruction returns s when it terminates. ☒

APSS AVAILABILITY: If messages sent during an execution of secret reconstruction are delivered before a subsequent execution of share refreshing starts, then that execution of secret reconstruction terminates. ☒

[1] The notion of share reconstruction can be easily extended to schemes that perform other operations on the secret (e.g., digital signing in cases where the secret is a private key).

[2] Cornell On-Line Certification Authority (COCA) [Zhou et al. 2002] employs this, using APSS in conjunction with client protocols that read and update digital certificates stored by quorums of servers. The read and update client protocols (which read but do not write secret shares) and COCA's share refreshing protocol (which writes secret shares) each executes in under a few seconds with four servers in a LAN or a WAN, so very few protocol restarts are observed when share refreshing runs are hourly.

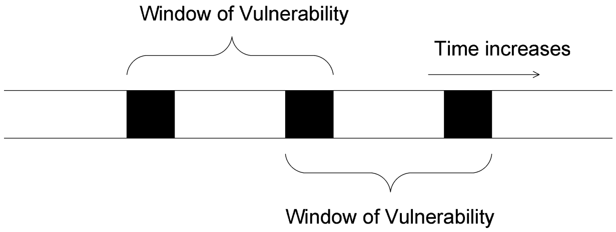

Fig. 1. Window of vulnerability definition. Each black box represents an execution of share refreshing.

APSS Progress: Execution of share refreshing terminates on all servers that are correct during that execution of share refreshing; at termination, correct servers have deleted old shares and any related information. ☒

These correctness requirements must hold in a given assumed system model and adversary model. We next elaborate on these.

2.3 Asynchrony and Window-of-Vulnerability Definition

Any assumption about execution timing or message delivery delay leads to vulnerabilities that can be exploited by launching attacks (e.g., denial-of-service attacks) that invalidate that assumption. Therefore, we make no such synchronous assumptions for APSS:

Asynchronous System: There is no bound on message delivery delay or server execution speed. ☒

The Asynchronous System assumption might seem unnecessarily pessimistic because bounds on message delivery delay and server execution speed invariably do exist. However, the Asynchrony System assumption allows us to sidestep the hard problem of determining those bounds and the consequences of making an error in that determination—namely, rendering the protocol vulnerable to denial-of-service attacks (by making an optimistic choice) or producing an extremely slow protocol due to time-outs that must delay long enough to accommodate the worst-case scenario (by making a pessimistic choice).

Safety properties for protocols designed to work in asynchronous system are necessarily independent of assumptions about timing. Furthermore, the performance of such protocols depends only on actual message delivery delays and server execution speeds—not on bounds for a worst-case scenario. However, there is a price: protocols for asynchronous systems often suffer from reduced fault-tolerance in comparison to their synchronous counterparts. For example, our APSS protocol tolerates fewer than one-third compromised servers, while PSS [Herzberg et al. 1995; Jarecki 1995] tolerates up to one-half compromised servers.

With APSS, operation of the system is partitioned into phases, where each phase is separated from the next by an execution of share refreshing. Correct servers delete information related to old shares when execution of share refreshing completes. Thus, a window of vulnerability for the system extends from the start of one execution of share refreshing to the end of the next. Figure 1 illustrates.

For a system with periodic executions of share refreshing, given the design requirement that the length of a window of vulnerability cannot exceed $\mathcal{W}$, one important design parameter is time interval $\mathcal{I}$ for triggering the periodic executions of share refreshing. Choosing $\mathcal{I}$ is easy in synchronous systems, where there is a bound $\delta \ll \mathcal{W}$ on the duration for an execution of share refreshing: we simply set $\mathcal{I}$ to be $\mathcal{W} - \delta$.

For asynchronous systems, we set $\mathcal{I}$ to be $\mathcal{W} - \Delta$, where Δ is a conservative estimate on the duration for an execution of share refreshing.[3] Note that, although Δ is used for triggering execution of share refreshing, the value of neither $\mathcal{I}$ nor Δ is used within that protocol. Thus, the execution time of share refreshing is unaffected by the choice of either parameter and, even if the interval between successive executions of share refreshing exceeds Δ, unlike a (synchronous) PSS protocol, all safety properties of APSS continue to hold.

Should estimate Δ ever be exceeded, drastic measures can be taken to limit the window of vulnerability and prevent too many servers from being compromised: servers can be disconnected from the network and administrators notified. We can make the probability of Δ being exceeded negligibly low by setting Δ conservatively.

2.4 Adversary Model

We model an adversary by a probabilistic polynomial Turing machine. We also assume that computing discrete logarithms in Z_{p} for a large prime p is infeasible and assume that digital signatures are existentially unforgeable under adaptively chosen message attacks.

Under the Asynchronous System assumption, an adversary can schedule when message deliveries occur. However, we do assume that communication links between correct servers cannot be compromised by an adversary:[4]

SECURE LINKS: Communication links between two correct servers implement confidentiality, integrity, and reliable delivery of messages. ☒

This is a reasonable assumption because, to implement confidentiality and integrity of communication links, each server can maintain a unique public/private key pair with the public key known to all other servers. A session key can then be negotiated after mutual authentication, with messages encrypted under that key. The session key must be refreshed with each execution of share refreshing.[5]

[3] The value of Δ must take into account clock differences among the correct servers. (A clock synchronization protocol [Lamport and Melliar-Smith 1984; Schneider 1987] can be used to keep the local clocks on correct servers loosely synchronized.) Assuming that σ is a conservative bound on clock differences, share refreshing can be invoked on a server at the interval of $\mathcal{W} - \Delta - \sigma$ according to its local clock.

[4] As shown in [Zhou 2001; Zhou et al. 2002a], APSS can be easily extended to work with links that do not ensure reliable delivery of every message.

[5] We here assume private keys are stored in a tamper-proof cryptography coprocessor which performs all operations involving these private keys. The use of a tamper-proof cryptography coprocessor will not necessarily prevent a compromised server from performing cryptographic operations for the adversary. The adversary might, for example, cause the server to generate signed or

An adversary is assumed to gain full control over a compromised server; that is, a compromised server exhibits Byzantine failures. We restrict the fraction of the servers compromised in each window of vulnerability and assume a static adversary [Canetti et al. 1996].

t-LIMIT SERVER COMPROMISE: At most t servers are compromised within each window of vulnerability, where $3t + 1 \leq n$ holds. ⊠

For the rest of the paper, assume that $n = 3t + 1$ holds. Generalization to other failure scenarios is discussed in Section 6.3.

3. SHARE REFRESHING REVISITED

Given an $(n, t + 1)$ sharing $S = \{s_i \mid 1 \leq i \leq n\}$ of a secret s, execution of share refreshing computes a new, independent $(n, t + 1)$ sharing $S' = \{s'_j \mid 1 \leq j \leq n\}$ of s without reconstructing s first.

Let operation split generate a set of n random shares from a given secret and operation reconstruct compute the corresponding secret from a given set of $t + 1$ shares. Then, share refreshing can be described as follows:[6]

(1) Select a set $T \subset S$ of shares, where $|T| = t + 1$. For each share $s_i \in T$, invoke function split to generate a *subsharing* consisting of a set of *subshares* $\{s_{ij} \mid 1 \leq j \leq n\}$.

(2) Construct new sharing $S' = \{s'_j \mid 1 \leq j \leq n\}$ for s, where each s'_j is computed from $\{s_{ij} \mid i \in T\}$ using reconstruct.

Challenges emerge when set S of shares is distributed over n servers, each server storing a share. Because some of the n servers might be compromised, it is no longer feasible to predetermine set T in step (1) above while still guaranteeing that subsharings will be generated from shares in T. This is because we do not know beforehand which servers are going to be compromised; compromised servers might generate bogus subsharings or no subsharings. To ensure that subsharings are generated from at least $t + 1$ shares, at least $t + (t + 1) = 2t + 1$

encrypted messages for later use in attacks. A defense here is to maintain an integer counter in stable memory (so the counter's value will persist across failures and restarts) that is part of the special-purpose cryptographic hardware. This counter is incremented every time a new window of vulnerability starts; and the current counter value is included in every message that is encrypted or signed using the tamper-proof hardware. A server can now ignore any message it receives that has a counter value too low for the current window of vulnerability. In the absence of tamper-proof coprocessors, private keys must be refreshed at the same time that shares are. One simple approach has trusted administrators for each server 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 servers); 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. An automated rekeying scheme is discussed in [Canetti 1995].

[6] We write names split and reconstruct without giving specific realizations to allow different share refreshing implementations. Realizations of the two operations must ensure that the resulting new sharing constitutes an $(n, t + 1)$ secret sharing for the same secret as the old sharing. We can often use the secret splitting and secret reconstruction operations from a secret sharing scheme (e.g., the ones for Shamir's secret sharing.)

subsharings must be generated from different shares on different servers because up to t servers might be compromised. Two problems must then be solved to make share refreshing work.

DISSEMINATION PROBLEM. Because a compromised server might not follow the protocol to generate and propagate a subsharing, there must be a mechanism (i) to determine the validity of subshares and (ii) to ensure that correct servers eventually get enough subshares to compute the new shares. That is, servers must verify that a subsharing $\{s_{ij} \mid 1 \leq j \leq n\}$ is generated correctly using split from s_i and that each subshare s_{ij} has been received by server p_j if server p_j is not compromised. ☒

CONSISTENCY PROBLEM. With at least $2t + 1$ subsharings generated, subshares from up to $2t + 1 > t + 1$ subsharings might have been correctly disseminated to servers. All correct servers must select and use subshares of the same set of subsharings in order to generate a new sharing for the same secret as the old one. ☒

The Asynchronous System assumption makes both of these problems difficult to solve.

For the Dissemination Problem, verifiable secret sharing (VSS) can be used so that servers can verify subshares they receive. However, it is generally impossible to tell whether all correct servers have received and verified their respective subshares, because under the Asynchronous System assumption, a compromised and nonresponsive server cannot be distinguished from a correct, but slow, one. Fortunately, shares in an $(n, t + 1)$ secret sharing have intrinsic redundancy so that the secret can be reconstructed from any $t + 1$ shares. The same holds for each subsharing generated in share refreshing. In Section 4, we show how to expose and leverage such redundancies in order to solve the Dissemination Problem.

It might seem that solving the Consistency Problem requires consensus in an asynchronous system—known to be impossible [Fischer et al. 1985] to solve with any deterministic protocol. Fortunately, implementing consensus is unnecessary, and Section 5 shows how we do solve the Consistency Problem.

4. SOLVING THE DISSEMINATION PROBLEM

Consider an execution of share refreshing and the window of vulnerability that starts from this execution of share refreshing. To solve the Dissemination Problem, it suffices that every subsharing selected for constructing new shares be correctly disseminated. That is, correct servers must obtain their respective subshares from each subsharing.

A server disseminating subshares cannot be expected to wait for acknowledgments of subshare receipt from more than $n - t$ servers, because up to t servers might be compromised and never respond. Moreover, compromised servers might respond before correct servers do, so only $(n - t) - t = n - 2t$ of the $n - t$ subshare receipt acknowledgments are guaranteed to come from correct servers.

In theory, the subshare receipt acknowledgments from $n - 2t$ correct servers should be adequate for correct servers to reconstruct all subshares in a subsharing as long as $n - 2t \geq t + 1$ holds, since $t + 1$ subshares suffice for constructing a share.

In practice, a correct server not receiving a needed subshare should have a way to retrieve that subshare from the $t + 1$ correct servers that do receive their subshares. One such scheme was proposed in [Jarecki 1995], but it requires coordination/agreement among a set of at least $t + 1$ servers; something hard to achieve when the Asynchronous System assumption holds. APSS must therefore employ another scheme.

4.1 (l, l) Secret Sharing and Index Sets

The solution employed for APSS exploits redundancy among shares held by different servers, as follows. In an (l, l) secret sharing for s, all l shares are needed to reconstruct s, and schemes exist [e.g., Ito et al. 1987] that assign such shares to n servers in a way that shares stored by any $t + 1$ servers suffice to reconstruct s, but shares held by fewer servers are not. So APSS assigns a set of shares from an (l, l) sharing of s to each server, employing the following construction of Ito et al. [1987].

Given n servers, where up to t servers might be compromised, compute $l := \binom{n}{t}$ and generate an (l, l) secret sharing. Assign indexes $\{i \mid 1 \leq i \leq l\}$ to shares of the (l, l) secret sharing as follows:

(1) Create $l = \binom{n}{t}$ different sets $P_1, \ldots, P_l$ of servers, each containing a unique set of t of the n servers. Servers in any of these sets jointly do not have enough shares to reconstruct secret s provided servers in any set P_i do not together have all l shares.

(2) For each server p, define *index set* I_p to be $\{i \mid 1 \leq i \leq l, p \notin P_i\}$; this set identifies shares that are stored at server p. By not including i in I_p for each server p in P_i, we ensure that servers in P_i do not have the share with index i, and thus do not have all l shares needed to reconstruct s. Servers in any set Q of $t + 1$ servers will jointly have all indexes in their index sets. This is because, for each $1 \leq i \leq l$, there exists $p \in Q - P_i$, hence, $i \in I_p$.

The index sets thus satisfy

$$\left(\bigcup_{p \in P} I_p \right) = \{i \mid 1 \leq i \leq l\} \qquad \text{if } |P| \geq t + 1 \tag{4.1}$$

$$\left(\bigcup_{p \in P} I_p \right) \subset \{i \mid 1 \leq i \leq l\} \qquad \text{if } |P| \leq t \tag{4.2}$$

Because each server now holds a set of (l, l) shares, recovery of the set of shares for a server is straightforward: a server p fetches from $t + 1$ server q all shares (subshares) with indexes in $I_p \bigcap I_q$.

Figure 2 illustrates how to assign four shares of a (4, 4) sharing to four servers and tolerate one compromised server. (Note that $\binom{4}{1} = 4$ holds.) The index set

Server (p)	*Index set* I_p
p_1	$\{2,3,4\}$
p_2	$\{1,3,4\}$
p_3	$\{1,2,4\}$
p_4	$\{1,2,3\}$

Fig. 2. Assignment of shares ($n = 4$ and $t = 1$).

Gen1. Given a secret $s \in Z_{\mathsf{p}}$, construct l-element vector S, where the l elements constitute an (l, l) secret sharing for s with the reconstruct function defined to be modular addition; that is, $\Sigma_{i=1}^{l} S[i] = s$ holds.

Gen2. Construct an l-element vector R, where $R[i]$ for each $1 \le i \le l$ is randomly chosen from Z_{q} and $r = \Sigma_{i=1}^{l} R[i]$ holds.

Gen3. Compute $g^s h^r$ and $g^{S[i]} h^{R[i]}$ for each $1 \le i \le l$ and make them public. For convenience, define a new vector Λ, such that $\Lambda[0] = g^s h^r$ and $\Lambda[i] = g^{S[i]} h^{R[i]}$ for each $1 \le i \le l$.

Fig. 3. VSS_Generate(s, r), verifiable sharing generation protocol.

for each server p_i consists of all indexes except i. No single index set contains all four indexes, so a single compromised server is unable to obtain all four shares needed to recover the secret.

4.2 Verifiable Secret Sharing

Recipients of subshares from a server must verify the validity of those subshares. A solution here is verifiable secret sharing. In APSS, we employ Pedersen's [1992] noninteractive verifiable secret-sharing scheme.[7] Here is how that works.

Let p and q be large prime numbers satisfying $\mathsf{p} = 2\mathsf{q} + 1$. Let g and h be two public elements of Z_{p}^{*} of order q, where no server knows $\log_g h \mod \mathsf{p}$. We assume that, given g and h, computing $\log_g h \mod \mathsf{p}$ is infeasible. Henceforth, we omit "mod q" (for operations on verifiable shares or exponents) and "mod p" (for operations on exponentiations and discrete logarithms.)

Consider a *dealer* responsible for creating and distributing an (l, l) secret sharing for a secret s. To create a verifiable secret sharing, the dealer randomly picks $r \in Z_{\mathsf{q}}$ and performs the protocol in Figure 3, generating:

$$(S, R, \Lambda) := \text{VSS\_Generate}(s, r)$$

We call (S, R, Λ) a *verifiable sharing* and $(i, S[i], R[i])$ the ith *verifiable share*; the ith verifiable share is private to each server p where $i \in I_p$ holds. Λ is the public part of a verifiable sharing, and it is used as the *label* of that verifiable sharing.

[7] We could instead use Feldman's [1987] verifiable secret sharing, with different security guarantees. See Jarecki [1995] for a comparison between Feldman's and Pedersen's verifiable secret-sharing schemes.

Given public information p, q, g, and h, validity of a verifiable sharing (S, R, Λ) is checked as follows.

verify-vec(Λ) checks validity of Λ; the check passes if and only if $\Lambda[0] = \prod_{i=1}^{l} \Lambda[i]$ holds.

verify-share$(\Lambda, i, S[i], R[i])$ checks validity of verifiable share $(i, S[i], R[i])$ with respect to Λ; the check passes if and only if $\Lambda[i] = g^{S[i]}h^{R[i]}$ holds.

The following properties of verifiable secret sharing are proved in Appendix B. These proofs instantiate results in [Pedersen 1992] for the (l, l) secret sharing scheme we use in VSS_Generate(s, r) of Figure 3.

VSS-Completeness: For a verifiable sharing generated from the protocol in Figure 3, if (S, R, Λ) := VSS_Generate(s, r), then verify-vec(Λ) holds and verify-share$(\Lambda, i, S[i], R[i])$ holds for each $1 \leq i \leq l$. ☒

VSS-Soundness: Assume that computing discrete logarithms in Z_{p} is hard and that an adversary is unable to compute $\log_g h$. Given, p q, g, h, and verifiable sharing (S, R, Λ), where $\Lambda[0]$ is known to be $g^s h^r$ for some $r \in Z_{\mathrm{q}}$. If verify-vec(Λ) holds and verify-share$(\Lambda, i, S[i], R[i])$ holds for each $1 \leq i \leq l$, then with high probability elements in S constitute an (l, l) sharing of secret s. ☒

VSS-Confidentiality: Given public information p, q, g, h, and a verifiable sharing (S, R, Λ). An adversary learns nothing about s from Λ and from $\{(i, S[i], R[i]) \mid 1 \leq i \leq l, i \neq j\}$ for some $1 \leq j \leq l$. ☒

4.3 Subsharing Certification

In APSS, a server generating and propagating a subsharing plays the role of a dealer and uses verifiable secret sharing to prove the authenticity of those subshares, since in APSS a subsharing for a share $S[i]$ is, in fact, a sharing for $S[i]$. Assume that servers hold shares of a verifiable secret sharing (S, R, Λ) prior to execution of share refreshing. Each server d, for each verifiable share $(i, S[i], R[i])$, where $i \in I_d$, performs a *subsharing certification protocol* to generate and certify a verifiable sharing (S_i, R_i, λ_i). The subsharing certification protocol is shown in Figure 4, where we use $\mathcal{E}(x)$ to represent a ciphertext of x, we use $\langle Y \rangle_p$ to denote a message Y digitally signed by server p, and recipients ignore messages bearing invalid digital signatures.[8]

A subsharing generated in step Cert1 by server d is considered *certified* when server d gathers verified messages from $2t + 1$ servers in step Cert3. Triple (Λ, i, λ_i) serves as the label for the certified subsharing and elucidates how the subsharing relates to the old sharing.

The subsharing certification protocol ensures the following properties.

Lemma 4.1. *Execution of the subsharing certification protocol initiated by a correct server d terminates.*

[8] Such signature checking step is omitted in the protocol description throughout the paper.

Cert1. Invoke the verifiable sharing generation protocol in Fig. 3 and compute $(S_i, R_i, \lambda_i) := \mathsf{VSS\_Generate}(S[i], R[i])$. Server d then disseminates these subshares in verify messages to other servers p.

$$\forall p : d \longrightarrow p : \langle \mathsf{verify}, d, p, \Lambda, i, \lambda_i, \mathcal{E}(\{(j, S_i[j], R_i[j]) \mid j \in I_p\})\rangle_d$$

Cert2. A server p, upon receiving a $\langle \mathsf{verify}, d, p, \Lambda, i, \lambda_i, \mathcal{E}(X)\rangle_d$ message, checks validity by checking the following conditions.

(a) $\mathsf{verify\text{-}vec}(\Lambda)$ holds,

(b) $\mathsf{verify\text{-}vec}(\lambda_i)$ and $\Lambda[i] = \lambda_i[0]$ hold,

(c) for each $j \in I_p$, X contains an element (j, s_{ij}, r_{ij}) and $\mathsf{verify\text{-}share}(\lambda_i, j, s_{ij}, r_{ij})$ holds.

If conditions Cert2a to Cert2c are satisfied, then server p replies to d with a $\mathsf{verified}$ message.

$$p \longrightarrow d : \langle \mathsf{verified}, p, d, \Lambda, i, \lambda_i\rangle_p$$

Cert3. Once server d collects a set M of correctly signed $\mathsf{verified}$ messages containing d, Λ, i, and λ_i from a set Q comprising $2t + 1$ servers, the subsharing is considered *certified.*

Fig. 4. Subsharing certification protocol.

Proof Sketch: The subsharing certification protocol terminates if server d receives $\mathsf{verified}$ messages from $2t + 1$ servers. Receipt of these messages is guaranteed, if there exists at least $2t + 1$ correct servers that receive verify messages satisfying conditions Cert2a to Cert2c (Figure 4). According to Cert1, all servers receive verify messages; by t-limit Server Compromise assumption, at least $2t + 1$ of those are correct. Due to VSS-Completeness, conditions Cert2a to Cert2c will be satisfied at those servers, so the $2t + 1$ $\mathsf{verified}$ messages being sought will be received by d. ⊠

Lemma 4.2. *If an (l, l) verifiable subsharing with label (Λ, i, λ_i) is certified by the protocol in Figure 4, then, with high probability, there exists a set of l subshares, each on a correct server, that constitute a subsharing for the ith share of sharing Λ.*

Proof Sketch: According to Cert3, for a subsharing to be considered certified, there must exist a set Q of $2t+1$ servers that have sent the $\mathsf{verified}$ message to d for that subsharing. Since by the t-limit Server Compromise assumption at most t servers are compromised, there exists a subset $Q' \subseteq Q$ of at least $t + 1$ correct servers. Due to Eq. (4.1), $\bigcup_{p \in Q'} I_p = \{j \mid 1 \le j \le l\}$ holds. According to Cert2c, there exists l verifiable subshares $\{(j, s_{ij}, r_{ij}) \mid 1 \le j \le l\}$, such that $\mathsf{verify\text{-}share}(\lambda_i, j, s_{ij}, r_{ij})$ holds for each $1 \le j \le l$. Given that $\lambda_i[0] = \Lambda[i] = g^{S[i]}h^{R[i]}$ holds (Cert2b) and that $\mathsf{verify\text{-}vec}(\Lambda)$ holds (Cert2a), due to VSS-Soundness, $\{s_{ij} \mid 1 \le j \le l\}$ constitutes an (l, l) sharing of share $S[i]$ with high probability.

Rec1. To recover subshares corresponding to (Λ, i, λ_i), a server p sends a recover message to all servers:

$$\forall q : p \longrightarrow q : \langle \textsf{recover}, p, \Lambda, i, \lambda_i \rangle_p$$

Rec2. A server q, upon receiving a recover message from server p for subsharing (Λ, i, λ_i), checks whether it has subshares of the requested subsharing. If it does, let S_i and R_i be the vectors for the verifiable subsharing, it sends a recovered message to p.

$$q \longrightarrow p : \langle \textsf{recovered}, q, p, \Lambda, i, \lambda_i, \mathcal{E}(\{(j, S_i[j], R_i[j]) \mid j \in I_p \bigcap I_q\}) \rangle_q$$

Rec3. Server p, upon receiving a recovered message from q, decrypts set X of verifiable subshares in the message, and checks the following conditions:

- X contains an entry (j, s_{ij}, r_{ij}) for each $j \in I_p \bigcap I_q$.
- For each $(j, s_{ij}, r_{ij}) \in X$, verify-share$(\lambda_i, j, s_{ij}, r_{ij})$ holds.

If the conditions hold, then p stores all the verifiable subshares received. Server p awaits receipt of all its subshares $\{(j, S_i[j], R_i[j]) \mid j \in I_p\}$ in this manner.

Fig. 5. Subsharing recovery protocol.

Furthermore, by construction, Q' consists of at least $t + 1$ correct servers. So, subshare s_{ij} for each $1 \leq j \leq l$ is stored on at least one of the servers in Q', and servers in Q' are all correct. ⊠

4.4 Subsharing Recovery

The protocol in Figure 4 ensures that at least $t+1$ correct servers have received and verified their subshares, but not that all correct servers receive their subshares. A subsharing recovery protocol (see Figure 5) allows retrieval of those subshares.

Using that protocol, if subsharing (Λ, i, λ_i) is certified, then a correct server p will receive all its subshares (i.e., subshares with indexes in I_p).[9]

LEMMA 4.3. *Execution of the subsharing recovery protocol terminates if invoked by a correct server p for a certified subsharing.*

PROOF SKETCH: According to Cert3, for a certified subsharing, there must exist a set Q of $2t + 1$ servers that have sent the verified message to server d—executing Cert1—for that subsharing. Because at most t servers are compromised, there exists a subset $Q' \subseteq Q$ of at least $t + 1$ correct servers. So, by Eq. (4.1), $\bigcup_{q \in Q'} I_q = \{j \mid 1 \leq j \leq l\}$ holds. Therefore, $\bigcup_{q \in Q'} (I_q \bigcap I_p) = \{j \mid 1 \leq j \leq l\} \bigcap I_p = I_p$ holds. The recovered messages from servers in Q' (step Rec2) will contain all subshares that p needs to complete the subsharing recovery protocol. Server p will always consider recovered messages from servers

[9] Here, we assume old shares and subshares are not deleted.

in Q' valid, because these servers are correct and because the verification in step Rec3 is subsumed in the verification that servers in Q' perform in step Cert2. ☒

5. SOLVING THE CONSISTENCY PROBLEM

Certified sharings, one for each original share, are used in APSS for constructing new shares. Multiple certified subsharings might exist, though. Achieving agreement on which to use might seem to require consensus in an asynchronous system, for which impossibility results [Fischer et al. 1985] are known. For APSS, we instead exploit the observation that having a single set of new shares generated is not really necessary. Multiple new sharings could be produced by an execution of share refreshing and we allow APSS to generate up to n new sharings. Execution of share refreshing that starts with n old sharings (as a result of a previous execution) will generate up to n sharings (rather than n^2), thereby avoiding exponential explosion in the number of sharings created in subsequent windows of vulnerability.

Solving the Consistency Problem requires that, for any new sharing generated by execution of share refreshing, servers use the same set of subsharings to construct their new shares. This would be easy to implement if there were a *coordinator* that picks an old sharing (among possibly n old sharings) for refreshing and selects a set of certified subsharings (from among the multiple certified subsharings produced by the holders of each share) for each share in the old sharing. Share refreshing could then be carried out as shown in Figure 6. There, a version number v indicates the vth execution of share refreshing and is used to prevent replay attacks. A coordinator identifier is also included in each message to distinguish the different instances of share refreshing initiated by different coordinators.

Any server can be a coordinator. A compromised coordinator might try to cause servers to generate inconsistent shares, but inconsistent shares have different labels and will not be used together in reconstructing the secret. A compromised coordinator instead might fail to complete execution of the protocol in Figure 6. However, this is not a problem provided some correct server also is a coordinator. APSS therefore has at least $t + 1$ servers function as coordinators—for simplicity, the protocol has all n servers act as coordinators.

Deletion of Old Shares and Subshares

Correct servers delete old shares and subshares when new shares are generated. However, deletion must wait until those shares and subshares are no longer needed—that is, until $t + 1$ correct servers have constructed new shares for the sharing, since, with $t + 1$ correct servers, there exists a correct server that holds each new share, and other correct servers can use a protocol similar to the subsharing recovery protocol to recover their shares.

A coordinator C can determine that $t + 1$ correct servers have constructed new shares by collecting digitally signed completed messages from $2t+1$ servers and using these as the evidence that $(2t + 1) - t = t + 1$ correct servers have obtained the requisite new shares. These completed messages are attached to

SS1. Coordinator C starts the vth execution of share refreshing by sending an init message with the label Λ for an old verifiable sharing to all servers p.

$$\forall p : C \longrightarrow p : \langle \mathsf{init}, v, \Lambda \rangle_C$$

SS2. Each server p, upon receipt of $\langle \mathsf{init}, v, \Lambda \rangle_C$ from C, performs the following steps for each verifiable share $(i, S[i], R[i])$ it stores for sharing Λ and version number v.

(a) Generate and propagate a verifiable subsharing $(\Lambda, i, \lambda_i^p)$ by invoking the subsharing certification protocol in Fig. 4 to obtain a set M of digitally signed verified messages from $2t + 1$ servers.

(b) Send a certified message to C.

$$p \longrightarrow C : \langle \mathsf{certified}, v, \Lambda, i, \lambda_i^p, M \rangle_p$$

SS3. Coordinator C awaits receipt of a valid certified message with $(\Lambda, i, \lambda_i^{p_i}, M_i)$ for each $1 \leq i \leq l$, where a certified message is considered *valid* if M_i contains correctly signed verified messages from $2t + 1$ servers, each containing $(\Lambda, i, \lambda_i^{p_i})$. Let M' be the set of certified messages. Coordinator C sends a select message to all servers p.

$$\forall p : C \longrightarrow p : \langle \mathsf{select}, v, \Lambda, \{(i, \lambda_i^{p_i}) \mid 1 \leq i \leq l\}, M' \rangle_C$$

SS4. A server p, upon receiving a valid select message from C, checks whether it has received its subshares for the selected subsharings and invokes the subsharing recovery protocol if it has not. A select message is valid if it contains valid certified messages, one for each subsharing $(\Lambda, i, \lambda_i^{p_i})$ with $1 \leq i \leq l$.

Let $(\Lambda, i, \lambda_i^{p_i})$ be the label for the ith verifiable subsharing selected, and let S_i and R_i be the vectors for corresponding verifiable subshares. Server p constructs a new verifiable sharing Λ', stores the computed verifiable shares with the label, and sends a completed message to C:

- $\Lambda'[0] := \Lambda[0]$ and $\Lambda'[j] := \prod_{k=1}^{l} \lambda_k^{p_k}[j]$ for each $1 \leq j \leq l$,
- $S'[j] := \sum_{i=1}^{l} S_i[j]$ and $R'[j] := \sum_{i=1}^{l} R_i[j]$ for each $j \in I_p$.

$$p \longrightarrow C : \langle \mathsf{completed}, v, C, \Lambda, \Lambda' \rangle_p$$

Fig. 6. Share refreshing with coordinator C.

done messages sent to all servers by adding step SS5 (Figure 7) to the end of the protocol in Figure 6. The receipt of a valid done message on a server leads a server to delete old subshares and shares, as well as echoing that done messages to all servers (step SS6 in Figure 7).

SS5. Coordinator C awaits receipt of a set M'' of correctly signed completed messages containing v, C, Λ, and Λ' from $2t+1$ servers and sends a done message to all servers.

$$\forall p : C \longrightarrow p : \langle \mathsf{done}, v+1, C, \Lambda, \Lambda', M'' \rangle_C$$

SS6. At any time, a server, upon receiving a valid done message containing version number $v+1$, records $(v+1, \Lambda')$, deletes all old shares and subshares, stops being a coordinator for version v, and forwards the done messages to all other servers. A done message is valid if it contains a set of valid completed messages from $2t+1$ servers with matching parameters C, λ, Λ', and v.

Fig. 7. Old share/subshare deletion—extensions to Figure 6.

6. VARIATIONS ON A THEME

6.1 Protocol Optimizations

Our discussion so far has ignored cost. Performance optimization of the protocol is possible.

A system comprising all correct servers would not need the redundant processing that having at least $t+1$ coordinators brings. Moreover, in the absence of denial-of-service attacks, it becomes reasonable to assume optimistic bounds on message delivery delays and processor execution speeds. More efficient protocols are thus possible when those bounds do hold. So here is an opportunity to optimize the protocol by eliminating unnecessary work in settings where compromise and attack is rare—the likely case when the number of servers is small.

The key insight for implementing this optimization for cases where attacks are absent is to note that actions can be delayed arbitrarily without affecting correctness of any protocol designed for an asynchronous system. APSS is designed for an asynchronous system and, therefore, execution of all but one correct coordinator can be delayed without ill effect. Similarly, for that single coordinator, only l subsharings, one for each share of the selected old sharing, need to be certified by servers executing the subsharing certification protocol in Figure 4. Thus, an optimized version of APSS is obtained by (i) selecting a single coordinator C whose execution is not delayed; (ii) delaying other coordinators so they do not start executing until C should have finished were it correct and the system synchronous. Note that a valid done message can serve as the notification that additional coordinators are not needed for this execution of share refreshing; (iii) instructing servers to have l subsharings certified for coordinator C, while delaying the certifications for other subsharings.

Without such optimizations to APSS, each of the n coordinators will trigger executions of the subsharing certification protocol on each of the n servers for each share that server has for the selected old sharing. Because of the use of (l, l) secret sharing, the size of I_p for each server p is $O(l)$ (or more precisely, $l(n-t)/n$). Each execution of the subsharing certification protocol has message complexity of $O(n)$, communication complexity of $O(\kappa nl)$, where κ is the size of p, and computational cost of $O(nl)$ in terms of the number of modular exponentiations. It is also easy to verify that this phase dominates the message complexity, communication complexity, and computational cost of share refreshing, as described in Figures 6 and 7. Therefore, the message complexity of the protocol is $O(n^3 l)$ with communication complexity of $O(\kappa n^3 l^2)$ and computational cost of $O(n^3 l^2)$.

With the optimizations, when the optimistic system assumptions do hold, there is a single coordinator that triggers a total of l executions of the subsharing certification protocol on all servers. Therefore, the message complexity, communication complexity, and computational cost of the protocol are reduced to $O(nl)$, $O(\kappa nl^2)$, and $O(nl^2)$, respectively.

6.2 Mitigating Denial-of-Service Attacks

APSS was designed to ensure that APSS Secrecy, APSS Integrity, APSS Availability, and APSS Progress cannot be violated by the actions of t or fewer compromised servers. But launching a denial-of-service attack does allow compromised servers to slow down processing. Of specific concern to APSS are attacks that cause servers to generate unnecessarily large numbers of subsharings and/or sharings. A compromised server could launch such an attack by sending init messages containing different labels Λ in step SS1, by generating and disseminating multiple different subsharings from each share in step SS2a, and by sending select messages containing different choices of subsharing sets in step SS3.

We can eliminate these APSS vulnerabilities by requiring that correct servers never process two different messages of the same type and *pedigree*, where two messages are defined to have the same pedigree if they are sent on behalf of the same coordinator, in the same execution of share refreshing, 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 this run and then discards subsequent incoming messages of the same type and pedigree. Messages from a correct server will never be discarded, because correct servers never send different messages with the same type and pedigree. Therefore, the APSS protocol can be modified to defend against this denial-of-service attack.

6.3 General and Dynamic Adversary Structures

An *adversary structure* [Hirt and Maurer 2000] is a collection of sets of servers, where each set specifies servers that might all be compromised in the same window of vulnerability, and all subsets of each set in the collection are also in the collection. APSS was described above for *t threshold* adversary structures—collections of all sets containing t or fewer servers. Other adversary structures exist. For a given system and threat, these other adversary structures might well be better models of what sets of servers 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

- employs the construction of Ito et al. [1987] to formulate share sets and
- is built from steps that involve sets of $t+1$ servers and sets of $2t+1$ servers.

Two important properties 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* [Malkhi and Reiter 1998] with each such set of $2t+1$ servers constituting a *quorum*, and therefore:

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

Quorum Intersection: The intersection of any two quorums contains a correct server. ☒

Moreover, the quorums in APSS also satisfy

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

Thus, we refer to this collection of quorums as forming a *recoverable quorum system*.

The construction in Ito et al. [1987] 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. Provided no three sets in an adversary structure $\mathcal{A}$ cover the set U of all servers, then the collection $\mathcal{Q} = \{U - F | F \in \mathcal{A}\}$ of server sets will constitute a recoverable quorum system[10] because $\mathcal{Q}$ satisfies Quorum Availability, Quorum Intersection, and Quorum Recoverability.[11] 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.[12] Given old and new server sets and adversary structures, an execution of share refreshing must delete all old shares on correct old servers and store new shares on the correct new servers. This is accomplished if

- step Cert1 in Figure 4 is changed so that old servers now generate and propagate the subsharings to the new servers based on the new adversary structure, and
- step SS6 in Figure 6 is changed to forward done messages to both the old and new servers.

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

[11] See Malkhi and Reiter [1998] 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 $F \in \mathcal{A}$.

[12] The underlying mathematics for schemes that allow such dynamic changes of sets of servers and adversary structures were discussed in Frankel et al. [1997a] and in Desmedt and Jajodia [1997]. A protocol for verifiably redistributing secrets for a different adversary structure is presented in Wong et al. [2002].

6.4 Share-Refreshing and Subshare-Dissemination Alternatives

Various alternatives to the share refreshing and subshare dissemination employed by APSS are possible; we explore those next.

6.4.1 *Share Refreshing.* Herzberg et al. [1995] describes a share-refreshing scheme that exploits the property that share-wise addition of any sharing of 0 with a sharing of s yields another sharing of s. So given an old $(n, t+1)$ sharing $\{s_i \mid 1 \leq i \leq n\}$ of s, a new sharing for s is $\{s_i + s'_i \mid 1 \leq i \leq n\}$, where $\{s'_i \mid 1 \leq i \leq n\}$ is a randomly generated sharing of 0.

This scheme can be used in APSS.[13] In step Cert1 of Figure 4, a server will generate a distinct, random sharing of 0; shares in this sharing of 0 are then distributed in verify messages to the other servers, much like subshares are now distributed in step Cert1; once certified, a label for the sharing of 0 is sent to the coordinator using a certified message, as in step SS2b of Figure 6. The coordinator, in step SS3, then selects $t+1$ sharings of 0 (to ensure that at least one of them is generated by a correct server)[14] and informs all servers of those selections with select messages. Servers then add the shares in the selected sharings of 0 to their old shares in order to generate new shares.

6.4.2 *Subshare Dissemination.* In APSS, each subshare is stored by enough different servers so that copies remain available at the correct servers provided no more than t servers are compromised. An alternative, inspired by a scheme proposed in Cachin et al. [2002] 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 servers 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.

The scheme presented by Cachin et al. [2002] does not explicitly require a (sub)sharing recovery protocol, as we do. This is because in the protocol of Cachin et al. [2002], correct servers proactively echo subshares to other servers. Our protocol can be modified to do the same: at the end of step Cert2, server p will send a message containing the same information in the recovered message to every other server. Having recovery instead of proactive echoing produces an optimistic protocol that saves the extra echo messages in the (normal) cases

[13] Dynamically changing adversary structures discussed in Section 6.3 cannot be accommodated, because new shares here are computed from old shares, and these old shares will not necessarily exist on servers when the adversary structure is dynamic.

[14] 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 each window of vulnerability, even if all t sharings of 0 are created by compromised servers, no other servers are compromised in the new window of vulnerability or the old (recall that an execution of share refreshing belongs to both windows). Therefore, an 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.

where no subsharing recovery is needed. Such a scheme also makes it easy to adapt our APSS protocol to a system model where links are not reliable, but fair, as shown in Zhou [2001] and Zhou et al. [2002a].

7. RELATED WORK

APSS was first described in Zhou [2001] and Zhou et al. [2002a]. This paper restructures the original protocol, presents extensions, and provides a more rigorous proof than the one given in Zhou [2001].

A second proactive secret sharing protocol for asynchronous systems, developed independently, is described in Cachin et al. [2002], where a formal model and proof are presented. This protocol differs from APSS in two significant ways:

- It employs a randomized multivalued validated Byzantine agreement protocol Cachin et al. [2001] 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 avoids the need to run an agreement protocol.
- It employs a bivariate polynomial to implement subsharing recovery. The scheme, which can be retrofitted into APSS (as discussed in Section 6.4), circumvents the exponential explosion that the (l, l) secret sharing causes for APSS;[15] the protocol of Cachin et al. [2002] has polynomial-time communication and message complexity.

  However, the solution in Cachin et al. [2002] does not apply to refreshing shares of an RSA private key because of its use of bivariate polynomial—for RSA, $\phi(n)$, the modulus, is not known to servers. In contrast, APSS can be easily adapted: because of its use of (l, l) secret sharing, APSS can use the (n, n) threshold RSA scheme and share-refreshing scheme outlined in Rabin [1998]—these schemes do not require operations with modulus $\phi(n)$.

Proactive secret sharing was suggested in [Herzberg et al. 1995; Jarecki 1995] and is an instance of *proactive security*, introduced by Ostrovsky and Yung [1991] for multiparty computations. Herzberg et al. [1997] give proactive schemes for discrete-logarithm based public key cryptography; Frankel et al. [1997a, 1997b] and T. Rabin [1998] give proactive RSA schemes. A survey appeared in [Canetti et al. 1997]. 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 [Zhou et al. 2002].

Besides COCA, other systems efforts are notable for their attempts to compose security with fault-tolerance and thus for the weak assumptions they too

[15] In practice, most distributed services involve only a relatively small number of servers, so this exponential factor is unlikely to be of concern.

make about the environment. Rampart [Reiter 1995, 1996] implements process groups in an asynchronous distributed system where compromised servers can exhibit arbitrary behavior. BFT (Byzantine Fault-Tolerance) Castro and Liskov [2002] and SINTRA (Secure Intrusion-Tolerant Replication on the Internet) Cachin and Poritz [2002] are toolkits that support asynchronous group communication primitives along with proactive security. It seems clear that 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.

APPENDIX

A. Sharing Transformation

The proofs presented in the appendix rely on transforming one verifiable sharing to another. The transformation is described here.

Let (S, R, Λ) be a verifiable sharing for secret $s \in Z_{\mathrm{q}}$, such that verify-vec(Λ) holds, and verify-share$(\Lambda, i, S[i], R[i])$ holds for each $1 \leq i \leq l$. For any secret $s' \in Z_{\mathrm{q}}$ and index j $(1 \leq j \leq l)$, function $\mathit{transform}_{s,s',j}(S, R, \Lambda)$ produces a new verifiable sharing (S', R', Λ') as follows:

Compute $r' \in Z_{\mathrm{q}}$, such that $g^s h^r = g^{s'} h^{r'}$ and compute

$$S'[i] := \begin{cases} S[i] & \text{for } i \neq j \\ s' - s + S[j] & \text{for } i = j \end{cases}$$

$$R'[i] := \begin{cases} R[i] & \text{for } i \neq j \\ r' - r + R[j] & \text{for } i = j \end{cases}$$

$$\Lambda'[i] := \begin{cases} g^{s'} h^{r'} & \text{for } i = 0 \\ g^{S'[i]} h^{R'[i]} & \text{for } 1 \leq i \leq l \end{cases}$$

It is easy to verify:

- $\Lambda' = \Lambda$ holds because $g^s h^r = g^{s'} h^{r'}$.
- R' (resp. S') differs from R (resp. S) only at the jth element.
- (S', R', Λ') is a verifiable sharing of s' because
  —verify-vec(Λ') holds because of $\Lambda' = \Lambda$.
  —verify-share$(\Lambda', i, S[i]', R[i]')$ holds for each $1 \leq i \leq l$.
  —$\sum_{i=1}^{l} S'[i] = s'$ holds.

B. VSS Proofs

VSS-Completeness: For a verifiable sharing generated from the protocol in Figure 3, if (S, R, Λ) := VSS_Generate(s, r), then verify-vec(Λ) holds and verify-share$(\Lambda, i, S[i], R[i])$ holds for each $1 \leq i \leq l$.

Proof Sketch: Condition verify-vec(Λ) holds because

$$\prod_{i=1}^{l} \Lambda[i] = \prod_{i=1}^{l} g^{S[i]} h^{R[i]} = g^{\sum_{i=1}^{l} S[i]} h^{\sum_{i=1}^{l} R[i]} = g^s h^r = \Lambda[0]$$

For each $1 \leq i \leq l$, verify-share($\Lambda, i, S[i], R[i]$) holds by definition. ☒

VSS-Soundness: Assume that discrete logarithms in Z_{p} are hard and an adversary is unable to compute $\log_g h$. Given p q, g, h, and verifiable sharing (S, R, Λ), where $\Lambda[0]$ is known to be $g^s h^r$ for some $r \in Z_{\mathrm{q}}$. If verify-vec(Λ) holds and verify-share($\Lambda, i, S[i], R[i]$) holds for each $1 \leq i \leq l$, then, with high probability, elements in S constitute an (l, l) sharing of secret s.

Proof Sketch: Proof by contradiction. Assume $\sum_{i=1}^{n} S[i] = s'$ for some $s' \neq s$ and let $r' = \sum_{i=1}^{n} R[i]$. Because by construction of (S', R', Λ') $g^s h^r = g^{s'} h^{r'}$ holds, the dealer is able to compute $log_g h = (s - s')/(r' - r)$. This contradicts the assumption on the hardness of computing $log_g h$. ☒

VSS-Confidentiality: Given public information p, q, g, h, and a verifiable sharing (S, R, Λ). An adversary learns nothing about s from Λ and $\{(i, S[i], R[i]) \mid 1 \leq i \leq l, i \neq j\}$ for some $1 \leq j \leq l$.

Proof Sketch: It suffices to show that given any secret $s' \in Z_{\mathrm{q}}$ we can construct a verifiable secret sharing that differs from the given one only at the jth verifiable share, which is the only verifiable share of (S, R, Λ) unknown to an adversary.

Compute

$$(S', R', \Lambda') := transform_{s,s',j}(S, R, \Lambda)$$

An adversary cannot distinguish (S', R', Λ') from (S, R, Λ) because an adversary has no access to the jth verifiable share. Therefore, an adversary learns nothing about s. ☒

C. APSS Correctness Proof

Proving the correctness of APSS involves demonstrating that APSS Integrity, APSS Availability, APSS Progress, and APSS Secrecy hold in the system described in Section 2, where Asynchronous System, Secure Links, and t-Limit Server Compromise hold.

C.1 APSS Integrity and APSS Availability

Assuming the existence of a secret reconstruction scheme that terminates as long as l verifiable shares of the same label exists on some correct servers and that it will reconstruct secret s as long as these l verifiable shares of the same label constitute a verifiable sharing for s,[16] we prove APSS Integrity and APSS Availability.

[16] This assumption is reasonable because an entity wishing to reconstruct the secret can simply send the request to all servers for verifiable shares stored on the servers, await verifiable shares with the same label, check their validity, and reconstruct the secret from l valid shares.

APSS Integrity: At any time, with high probability, secret reconstruction returns s when it terminates.

APSS Availability: If messages sent during an execution of secret reconstruction are delivered before a subsequent execution of share refreshing starts, then that execution of secret reconstruction terminates.

Proof Sketch: For APSS Integrity, it suffices to show that, for each execution of share refreshing and any label Λ that has been included in a valid done message, the set of l shares with label λ on correct servers constitute an (l, l) sharing of secret s. For APSS Availability, it suffices to show that old shares and related subshares are deleted only when there exists a new label Λ that has been included in a valid done message and that every one of the l shares with label Λ is stored on some correct server in the new window of vulnerability starting from this execution of share refreshing. This suffices because of our assumption on secret reconstruction—if messages for execution of secret reconstruction are delivered before the next execution of share refreshing, then shares stored on correct servers are not deleted before the execution of secret reconstruction terminates.

Consider any new label Λ' in a valid done message. A done message is valid if and only if it contains a set of valid completed messages from $2t + 1$ servers. Because of t-limit Server Compromise, at least $t + 1$ of the $2t + 1$ servers are correct in the new window of vulnerability starting from this execution of share refreshing. Each server p in the set P of those $t + 1$ servers must have executed step SS4 (Figure 6) and constructed the new shares in I_p for the new sharing Λ' from a set of l subsharings with labels $\{(\Lambda, i, \lambda_i^{p_i}) \mid 1 \leq i \leq l\}$.

Because a valid select message is necessary to trigger step SS4, each subsharing with the label included in the select message is certified according to Cert3 (Figure 4). This ensures that, for each subsharing labeled $(\Lambda, i, \lambda_i^{p_i})$, there exist S_i and R_i, such that the following hold:

$$\text{verify-vec}(\Lambda) \tag{C.3}$$

$$\text{verify-vec}(\lambda_i^{p_i}) \tag{C.4}$$

$$\Lambda[i] = \lambda_i[0] \tag{C.5}$$

$$\text{for each } 1 \leq j \leq l, \text{verify-share}(\lambda_i^{p_i}, j, S_i[j], R_i[j]) \tag{C.6}$$

Based on the construction of sharing Λ' in step SS4, the following conditions hold.

$$\Lambda[0] = \Lambda'[0] \tag{C.7}$$

$$\text{for each } 1 \leq j \leq l, \Lambda'[j] = \prod_{i=1}^{l} \lambda_i^{p_i}[j] \tag{C.8}$$

$$\text{for each } 1 \leq j \leq l, S'[j] = \sum_{i=1}^{l} S_i[j],\ R'[j] = \sum_{i=1}^{l} R_i[j] \tag{C.9}$$

- verify-vec(Λ') holds because

$$\begin{aligned}
&\prod_{j=1}^{l} \Lambda'[j] \\
&= \text{Eq. (C.8)} \\
&\prod_{j=1}^{l}\prod_{i=1}^{l} \lambda_i^{p_i}[j] \\
&= \text{(Transposition)} \\
&\prod_{i=1}^{l}\prod_{j=1}^{l} \lambda_i^{p_i}[j] \\
&= \text{Eq. (C.4)} \\
&\prod_{i=1}^{l} \lambda_i^{p_i}[0] \\
&= \text{Eq. (C.5)} \\
&\prod_{i=1}^{l} \Lambda[i] \\
&= \text{Eq. (C.3)} \\
&\Lambda[0] \\
&= \text{Eq. (C.7)} \\
&\Lambda'[0]
\end{aligned}$$

- For each $1 \leq j \leq l$, let $(j, S'[j], R'[j])$ be the jth verifiable share computed in SS4, verify-share($\Lambda', j, S'[j], R'[j]$) holds because

$$\begin{aligned}
&g^{S'[j]}h^{R'[j]} \\
&= \text{Eq. (C.9)} \\
&g^{\sum_{i=1}^{l} S_i[j]}h^{\sum_{i=1}^{l} R_i[j]} \\
&= \text{(Transformation)} \\
&\prod_{i=1}^{l} g^{S_i[j]}h^{R_i[j]} \\
&= \text{Eq. (C.6)} \\
&\prod_{i=1}^{l} \lambda_i^{p_i}[j] \\
&= \text{Eq. (C.8)} \\
&\Lambda'[j]
\end{aligned}$$

Due to VSS-Soundness, (i) $\{S'[i] \mid 1 \leq i \leq l\}$ constitutes an (l, l) sharing for s and (ii) for every share $S'[i]$ $(1 \leq i \leq l)$, there exists a correct server in P that stores that share with the same label. Property (i) establishes APSS Integrity.

For an execution of share refreshing, a correct server deletes old shares and the subshares generated from old shares in step SS6 (Figure 7) only when it receives a valid done message containing some new label Λ'. Note that properties (i) and (ii) hold for any label in a valid done message. Therefore, due to property (ii), APSS Availability holds. ☒

C.2 APSS Progress

APSS PROGRESS: Execution of share refreshing terminates on all servers that are correct during that execution of share refreshing; at termination, correct servers have deleted old shares and any related information.

PROOF SKETCH: APSS Progress requires that, for each execution of share refreshing, all correct servers in the new window of vulnerability eventually reach step SS6 (Figure 7).

Each execution of share refreshing belongs to two windows of vulnerability. Call the first the *old* window of vulnerability and the second the *new* window of vulnerability. As an induction hypothesis, we assume that every correct coordinator/server in the old window of vulnerability knows the label of an old sharing and each of its l shares can be found on a correct server in the old window of vulnerability. This induction hypothesis is initially true due to initialization by a trusted entity and it will remain true after each execution of share refreshing, as we now prove.

It suffices to consider an instance of the protocol initiated by a correct coordinator C in the old window of vulnerability. Such a coordinator must exist because there are at least $t + 1$ coordinators and at most t of them are compromised. There are two cases.

1. *Coordinator C executes step SS6 after receiving a* done *message from another server.* In this case, C will propagate the done message to all servers. This leads to termination of the execution of this share refreshing.
2. *C never receives a* done *message from another server.* However, C will attempt to execute steps SS1 through SS5. Also, no correct servers delete old shares/subshares in this case, because, if any correct server performs such deletion in step SS6 of Figure 7, all correct servers including coordinator C will receive a done message from that server—we are thus returning to the first case. Let Λ be the label, known to C, of an old sharing, whose shares are available on correct servers in the old window of vulnerability. Coordinator C includes this label in the init message it sends in step SS1.

   For each share of sharing labeled Λ, there exists one correct server that has that share and that initiates subsharing certification protocol (step SS2a) for this share. Due to Lemma 4.1, the subsharing certification protocol always terminates. Therefore, C is guaranteed to receive a valid certified message for a subsharing generated from each share of sharing Λ (sent in step SS2b). Coordinator C thus must complete step SS3 and send a select message to all servers.

   For each correct server p that receives that select message, either p has all the subshares needed to generate new shares or it invokes subsharing recovery to retrieve missing shares (SS4). All selected subsharings are certified so, due to Lemma 4.3, a correct server eventually receives all needed subshares and sends back a completed message to C (SS4). Because there are at least $2t + 1$ correct servers, coordinator C is guaranteed to receive completed messages from $2t + 1$ servers. Thus coordinator C will complete step SS5. Because C sends a done message to all servers all correct servers will receive the message and execute step SS6 due to Secure Links, thereby ensuring termination of this execution of share refreshing. Note that the label included in the done message corresponds to a verifiable sharing, whose shares are stored on at least $t + 1$ correct servers in the new window of vulnerability. The label is going to be used in the init message for the next execution of share refreshing. ⊠

C.3 APSS Secrecy

APSS SECRECY: An adversary learns nothing about secret s.

PROOF SKETCH: We show that for any given series of share refreshing executions χ_s for secret s and for any given secret $\hat{s} \in Z_q$, we can construct a series of share refreshing executions $\chi_{\hat{s}}$ that is indistinguishable from χ_s to an adversary. We also assume that encryption function $\mathcal{E}$ is semantically secure and that the adversary is static.

Assign version number v to the vth execution of share refreshing and to the window of vulnerability that spans from the start of this execution of share refreshing to the end of the next. Let the period of time from the initial start of the system to the end of the first execution of share refreshing to be window of vulnerability 0. Let F_v be the set of servers compromised in a window of vulnerability v. Due to t-limit Server Compromise, $|F_v| \leq t$ holds for $0 \leq v$. Eq. (4.2) states that there exists i_v, such that $1 \leq i_v \leq l$ and $i_v \notin I_p$ hold for any $p \in F_v$. Because i_v is not in the index set for any compromised server p, i_v is the index of a share/subshare inaccessible to an adversary. Therefore, changes to shares/subshares of index i_v are invisible to an adversary.

The construction of $\chi_{\hat{s}}$ from χ_s is done inductively as follows:

BASE CASE: For the initial sharing (S, R, Λ) in χ_s, construct for $\chi_{\hat{s}}$ a corresponding sharing

$$(\hat{S}, \hat{R}, \hat{\Lambda}) := \mathit{transform}_{s,\hat{s},i_0}(S, R, \Lambda)$$

Because (S, R, Λ) and $(\hat{S}, \hat{R}, \hat{\Lambda})$ are identical except for $S[i_0] \neq \hat{S}[i_0]$ and $R[i_0] \neq \hat{R}[i_0]$ and because i_0 is not in the index set for any compromised server in this window of vulnerability, an adversary cannot distinguish the two sharings.

INDUCTION STEP: Consider window of vulnerability v with $v > 0$.

Induction Hypothesis: For each old sharing (S, R, Λ) generated in the $(v-1)$st execution of share refreshing in χ_s,

1. There exists a corresponding sharing $(\hat{S}, \hat{R}, \hat{\Lambda})$ in $\chi_{\hat{s}}$, such that $(\hat{S}, \hat{R}, \hat{\Lambda}) = \mathit{transform}_{s,\hat{s},i_{v-1}}(S, R, \Lambda)$ holds.
2. An adversary cannot distinguish the sharings and subsharings generated in the $(v - 1)$st execution of share refreshing in χ_s and $\chi_{\hat{s}}$.

For convenience, we use extended label $(S_i, R_i, \Lambda, i, \lambda_i)$ to denote subsharing (Λ, i, λ_i) generated from verifiable share (i, S_i, R_i) of old sharing Λ. During the vth execution of share refreshing, for each subsharing $(S_i, R_i, \Lambda, i, \lambda_i)$ of an old sharing in χ_s, construct a corresponding subsharing $(\hat{S}_i, \hat{R}_i, \hat{\Lambda}, i, \hat{\lambda}_i)$ in $\chi_{\hat{s}}$ as follows.

$$(\hat{S}_i, \hat{R}_i, \hat{\Lambda}, i, \hat{\lambda}_i) = \begin{cases} (S_i, R_i, \Lambda, i, \lambda_i) & \text{for } i \neq i_{v-1} \\ \mathit{transform}_{S[i], \hat{S}[i], i_v}(S_i, R_i, \lambda_i) & \text{for } i = i_{v-1} \end{cases}$$

By definition of *transform*, we know that $(\hat{S}_i, \hat{R}_i, \hat{\Lambda}, i, \hat{\lambda}_i)$ constitutes a verifiable subsharing for $\hat{S}[i]$. Also, the corresponding subsharings in the vth execution of χ_s and $\chi_{\hat{s}}$ are either identical or differ only at element i_v. Note that the vth execution of share refreshing belongs to window of vulnerability v. Because we chose i_v to be the index that falls out of the index sets of any compromised server in this window of vulnerability, no compromised servers during the execution of share refreshing have access to subshares with index i_v. Therefore, an adversary cannot distinguish the corresponding subsharings in χ_s and $\chi_{\hat{s}}$.

Now consider the sharings. For each new sharing labeled Λ' generated in this execution of share refreshing of χ_s, let $\{(S_i, R_i, \Lambda, i, \lambda_i) \mid 1 \leq i \leq l\}$ be the set of subsharings from which new sharing Λ' is generated. In $\chi_{\hat{s}}$, we construct a corresponding new sharing $\hat{\Lambda}'$ from the corresponding set of subsharings $\{(\hat{S}_i, \hat{R}_i, \hat{\Lambda}, i, \hat{\lambda}_i) \mid 1 \leq i \leq l\}$.

Due to the induction hypothesis, the old sharing labeled $\hat{\Lambda}$ is a verifiable sharing for $\hat{s}$. So is the new verifiable sharing $\hat{\Lambda}'$. By construction in step SS4 (Figure 6), sharing $\hat{\Lambda}'$ differs from Λ' only at index i_v. By choice of i_v, an adversary cannot distinguish sharing $\hat{\Lambda}'$ from sharing Λ'. It is easy to verify that $(\hat{S}', \hat{R}', \hat{\Lambda}') = \mathit{transform}_{s,\hat{s},i_v}(S', R', \Lambda')$ holds. This completes the inductive construction of $\chi_{\hat{s}}$. $\boxtimes$

ACKNOWLEDGMENTS

Reviewers of an earlier version of this paper provided extremely helpful feedback. We are also grateful to Christian Cachin and Andrew Myers for discussions about exposition and content as this paper evolved.

REFERENCES

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

BOYD, C. 1989. Digital multisignatures. In *Cryptography and Coding*, H. Baker and F. Piper, Eds. Clarendon Press, pp. 241–246.

CACHIN, C. AND PORITZ, J. A. 2002. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2002)*, IEEE (June), 167–176.

CACHIN, C., KURSAWE, K., LYSYANSKAYA, A., AND STROBL, R. 2002. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM Press, New York (Nov.), 88–97.

CACHIN, C., KURSAWE, K., PETZOLD, F., AND SHOUP, V. 2001. Secure and efficient asynchronous broadcast protocols (extended abstract). In *Advances in Cryptology—Crypto'2001*, J. Kilian, Ed. Lecture Notes in Computer Science, vol. 2139. Springer-Verlag, 524–541.

CANETTI, R. 1995. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, Department of Computer Science and Applied Mathematics, The Weizmann Institute of Science, Israel (June).

CANETTI, R., FEIGE, U., GOLDREICH, O., AND NAOR, M. 1996. Adaptively secure multi-party computation. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*. ACM Press, New York, 639–648.

CANETTI, R., GENNARO, R., HERZBERG, A., AND NAOR, D. 1997. Proactive security: Long-term protection against break-ins. *CryptoBytes (The Technical Newsletter of RSA Laboratories, A Division of RSA Data Security Inc.) 3*, 1, 1–8.

CASTRO, M. AND LISKOV, B. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*, 4 (Nov.), 398–461.

CHOR, B., GOLDWASSER, S., MACALI, S., AND AWERBUCH, B. 1985. Verifiable secret sharing and achieving simultaneous broadcast. In *Proceedings of the 26th Symposium on Foundations of Computer Science*. 335–344.

COURTOIS, P., HEYMANS, F., AND PARNAS, D. 1971. Concurrent control with readers and writers. *Communications of the ACM 14*, 10 (Oct.), 667–668.

DESMEDT, Y. 1988. Society and group oriented cryptography: A new concept. In C. Pomerance, Ed. *Advances in Cryptology—Crypto'87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, CA (Aug.) 16–20, 1987*, Vol. 293 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 120–127.

DESMEDT, Y. AND JAJODIA, S. 1997. Redistributing secret shares to new access structures and its applications. Technical Report ISSE_TR-97-01, George Mason University, July.

FELDMAN, P. 1987. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the 28th IEEE Symposium on the Foundations of Computer Science*. 427–437.

FISCHER, M. J., LYNCH, N. A., AND PETERSON, M. S. 1985. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM 32*, 2 (Apr.), 374–382.

FRANKEL, Y., GEMMEL, P., MACKENZIE, P., AND YUNG, M. 1997a. Optimal resilience proactive public-key cryptosystems. In *Proceedings of the 38th Symposium on Foundations of Computer Science*, Miami Beach, FL, (Oct.) 20–22. IEEE, pp. 384–393.

FRANKEL, Y., GEMMELL, P., MACKENZIE, P., AND YUNG, M. 1997b. Proactive RSA. In *Advances in Cryptology—Crypto'97*, B. Kaliski, Ed. Lecture Notes in Computer Science vol. 1294, Santa Barbara, California (Aug.). Springer-Verlag, pp. 440–454.

HERZBERG, A., JARECKI, S., KRAWCZYK, H., AND YUNG, M. 1995. Proactive secret sharing or: How to cope with perpetual leakage. In *Advances in Cryptology—Crypto '95*, D. Coppersmith, Ed. Lecture Notes in Computer Science vol. 963, Santa Barbara, California (Aug.). Springer-Verlag, 457–469.

HERZBERG, A., JAKOBSSON, M., JARECKI, S., KRAWCZYK, H., AND YUNG, M. 1997. Proactive public-key and signature schemes. In *Proceedings of the Fourth Annual Conference on Computer Communications Security*. ACM, New York, 100–110.

HIRT, M. AND MAURER, U. 2000. Player simulation and general adversary structures in perfect multi-party computation. *Journal of Cryptology 13*, 1, 31–60.

ITO, M., SAITO, A., AND NISHIZEKI, T. 1987. Secret sharing scheme realizing general access structure. In *Proceedings of IEEE Global Communication Conference (GLOBALCOM'87)*, Tokyo, Japan (Nov.), 99–102.

JARECKI, S. 1995. Proactive secret sharing and public key cryptosystems. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (Sept.).

LAMPORT, L. 1977. Concurrent reading while writing. *Communications of the ACM 20*, 11, 806–811.

LAMPORT, L. AND MELLIAR-SMITH, P. M. 1984. Byzantine clock synchronization. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada. ACM Press, New York, 68–74.

MALKHI, D. AND REITER, M. 1998. Byzantine quorum systems. *Distributed Computing 11*, 4, 203–213.

OSTROVSKY, R. AND YUNG, M. 1991. How to withstand mobile virus attacks. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*. 51–59.

Pedersen, T. 1992. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology—Crypto'91*, J. Feigenbaum, Ed. Lecture Notes in Computer Science vol. 576, Santa Barbara, California (Aug.). Springer-Verlag, 129–140.

Rabin, T. 1998. A simplified approach to threshold and proactive RSA. In *Advances in Cryptology—Crypto'98*, H. Krawczyk Ed. Lecture Notes in Computer Science vol. 1462, Santa Barbara, California (Aug.). Springer-Verlag, 89–104.

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

Reiter, M. K. 1996. Distributing trust with the Rampart toolkit. *Communications of the ACM 39*, 4 (Apr.), 71–74.

Schneider, F. B. 1987. Understanding protocols for Byzantine clock synchronization. Technical Report TR 87-859, Computer Science Department, Cornell University, New York,

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

Wong, T. M., Wang, C., and Wing, J. M. 2002. Verifiable secret redistribution for threshold sharing schemes. Technical Report CMU-CS-02-114, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 2002.

Zhou, L. 2001. *Towards building secure and fault-tolerant on-line services*. PhD thesis, Computer Science Department, Cornell University, Ithaca, New York (May).

Zhou, L., Schneider, F. B., and van Renesse, R. 2002a. APSS: Proactive secret sharing in asynchronous systems. Technical Report TR 2002-1877, Computer Science Department (Oct.), Cornell University, Ithaca, New York.

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

Received October 2002; revised January 2005; accepted April 2005