An extended abstract of this paper appears in *Proc. 9th ACM Conference on Computer and Communications Security (CCS-9)*, Washington DC, USA, 2002.

Asynchronous Verifiable Secret Sharing and Proactive Cryptosystems

Christian Cachin* Klaus Kursawe* Anna Lysyanskaya† Reto Strobl*

29 August 2002

Abstract

Verifiable secret sharing is an important primitive in distributed cryptography. With the growing interest in the deployment of threshold cryptosystems in practice, the traditional assumption of a synchronous network has to be reconsidered and generalized to an asynchronous model. This paper proposes the first *practical* verifiable secret sharing protocol for asynchronous networks. The protocol creates a discrete logarithm-based sharing and uses only a quadratic number of messages in the number of participating servers. It yields the first asynchronous Byzantine agreement protocol in the standard model whose efficiency makes it suitable for use in practice. Proactive cryptosystems are another important application of verifiable secret sharing. The second part of this paper introduces proactive cryptosystems in asynchronous networks and presents an efficient protocol for refreshing the shares of a secret key for discrete logarithm-based sharings.

1 Introduction

The idea of *threshold cryptography* is to distribute the power of a cryptosystem in a fault-tolerant way [12]. The cryptographic operation is not performed by a single server but by a group of n servers, such that an adversary who corrupts up to t servers and observes their secret key shares can neither break the cryptosystem nor prevent the system as a whole from correctly performing the operation.

However, when a threshold cryptosystem operates over a longer time period, it may not be realistic to assume that an adversary corrupts only t servers during the entire lifetime of the system. *Proactive cryptosystems* address this problem by operating in *phases*; they can tolerate the corruption of up to t different servers during every phase [18]. They rely on the assumption that servers may *erase* data and on a special reboot procedure to remove the adversary from a corrupted server. The idea is to proactively reboot all servers at the beginning of every phase, and to subsequently *refresh* the secret key shares such that in any phase, knowledge of shares from previous phases does not give the adversary an advantage. Thus, proactive cryptosystems tolerate a *mobile adversary* [20], which may move from server to server and eventually corrupt every server in the system.

Since refreshing is a distributed protocol, the network model determines how to make a cryptosystem proactively secure. For synchronous networks, where the delay of messages is bounded, many proactive cryptosystems are known (see [6] and references therein). However, for asynchronous networks, no proactive cryptosystem is known so far. Because of the absence of a common clock and the arbitrary delay of messages, several problems arise: First, it is not clear how to define a proactive phase when the servers have no common notion of time. Second, even if the notion of a common phase is somehow imposed by external means, a message of the refresh protocol might be delayed arbitrarily across phase boundaries, which poses additional problems. And last but not least, one needs an asynchronous share refreshing protocol.

*IBM Research, Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland. {cca,kku,rts}@zurich.ibm.com.

†Brown University, Providence, RI 02912, USA. anna@cs.brown.edu. Work done at IBM Zurich.

The distributed share refreshing protocols of all proactive cryptosystems rely on *verifiable secret sharing*. Verifiable secret sharing is a fundamental primitive in distributed cryptography [11] that has found numerous applications to secure multi-party computation, Byzantine agreement, and threshold cryptosystems. A verifiable secret sharing protocol allows a distinguished server, called the *dealer*, to distribute shares of a secret among a group of servers such that only a qualified subgroup of the servers may reconstruct the secret and the corrupted servers do not learn any information about the secret. Furthermore, the servers need to reach agreement on the success of a sharing in case the dealer might be faulty.

Asynchronous verifiable secret sharing protocols have been proposed previously [1, 9, 5]. However, all existing solutions are prohibitively expensive to be suitable for practical use: the best one has message complexity $O(n^5)$ and communication complexity $O(n^6 \log n)$. This is perhaps not surprising because they achieve *unconditional* security. In contrast, we consider a *computational* setting and obtain a much more efficient protocol. Our protocol achieves message complexity $O(n^2)$ and communication complexity $O(\kappa n^3)$, where κ is a security parameter, and optimal resilience $n > 3t$.

Specifically, we assume hardness of the discrete-logarithm problem. Our protocol is reminiscent of Pedersen's scheme [22], but the dealer creates a two-dimensional polynomial sharing of the secret. Then the servers exchange two asynchronous rounds of messages to reach agreement on the success of the sharing, analogous to the deterministic reliable broadcast protocol of Bracha [2].

Combining our verifiable secret sharing scheme with the protocol of Canetti and Rabin [9], we obtain the first asynchronous Byzantine agreement protocol that is provably secure in the standard model *and* whose efficiency makes it suitable for use in practice.

With respect to asynchronous proactive cryptosystems, our contributions are twofold. On a conceptual level, we propose a formal model for cryptosystems in asynchronous proactive networks, and on a technical level, we present an efficient protocol for proactively refreshing discrete logarithm-based shares of a secret key.

Our model of an *asynchronous proactive network* extends an asynchronous network by an abstract *timer* that is accessible to every server. The timer is scheduled by the adversary and defines the phase of a server *locally*. We assume that the adversary corrupts at most t servers who are in the same *local phase*. Uncorrupted servers who are in the same local phase may communicate via private authenticated channels. Such a channel must guarantee that every message is delayed no longer than the local phase lasts and that it is lost otherwise.

A proactive cryptosystem refreshes the sharing of the secret key at the beginning of every phase (i.e., when sufficiently many servers enter the same local phase). Our model implies that liveness for the cryptosystem is only guaranteed to the extent that the adversary does not delay the messages of the refresh protocol for longer than the phase lasts. Otherwise, the secret key may become unaccessible. Despite this danger, we believe that our model achieves a good coverage for real-world loosely synchronized networks, such as the Internet, since a phase typically lasts much longer than the maximal delay of a message in the network.

Finally, we propose an efficient proactive refresh protocol for discrete logarithm-based sharings. It builds on our verifiable secret sharing protocol and on a randomized asynchronous multi-valued Byzantine agreement primitive [3]. The refresh protocol achieves optimal resilience $n > 3t$ and has expected message complexity $O(n^3)$ and communication complexity $O(\kappa n^5)$.

1.1 Organization of the Paper

In the next section we introduce our system model and recall the definition of asynchronous multi-valued Byzantine agreement with external validity. Section 3 defines asynchronous verifiable secret sharing and presents an efficient protocol for creating discrete logarithm-based sharings of a secret. In Section 4, we extend the asynchronous system model to a *proactive* network, and in Section 5 we describe how to asynchronously refresh shares of a secret key for discrete logarithm-based cryptosystems.

2 Preliminaries

2.1 Asynchronous System Model

We adopt the basic system model from [4, 3], which describe an asynchronous network of servers with a computationally bounded adversary.

Our computational model is parameterized by a security parameter κ; a function $\epsilon(\kappa)$ is called *negligible* if for all $c > 0$ there exists a κ_0 such that $\epsilon(\kappa) < \frac{1}{\kappa^c}$ for all $\kappa > \kappa_0$.

Network. The network consists of n servers $P_1, \ldots, P_n$, which are probabilistic interactive Turing machines (PITM) [15] that run in polynomial time (in κ). There is an adversary, which is a PITM that runs in polynomial time in κ. Some servers are controlled by the adversary and called *corrupted*; the remaining servers are called *honest*. An adversary that corrupts at most t servers is called *t-limited*. There is also an initialization algorithm, which is run by a trusted party before the system starts. On input κ, n, t, and further parameters, it generates the state information used to initialize the servers, which may be thought of as a read-only tape.

We assume that every pair of servers is linked by a *secure asynchronous channel* that provides privacy and authenticity with scheduling determined by the adversary. (This is in contrast to [3], where the adversary observes all network traffic.) Formally, we model such a network as follows. All communication is driven by the adversary. There exists a global set of messages $\mathcal{M}$, whose elements are identified by a *label* (s, r, l) denoting the sender s, the receiver r, and the length l of the message. The adversary sees the labels of all messages in $\mathcal{M}$, but not their contents. $\mathcal{M}$ is initially empty. The system proceeds in steps. At each step, the adversary performs some computation, chooses an honest server P_i, and selects some message $m \in \mathcal{M}$ with label (s, i, l). P_i is then *activated* with m on its communication input tape. When activated, P_i reads the contents of its communication input tape, performs some computation, and generates one or more response messages, which it writes to its communication output tape. A response message m may contain a destination address, which is the index j of a server. Such an m is added to $\mathcal{M}$ with label $(i, j, |m|)$ if P_j is honest; if P_j is corrupted, m is given to the adversary. In any case, control returns to the adversary. This step is repeated arbitrarily often until the adversary halts.

These steps define a sequence of events, which we view as logical time. We sometimes use the phrase "at a certain point in time" to refer to an event like this.

We assume an *adaptive* adversary that may corrupt a server P_i at any point in time instead of activating it on an input message. In that case, all messages $m \in \mathcal{M}$ with label $(\cdot, i, |m|)$ are removed from $\mathcal{M}$ and given to the adversary. She gains complete control over P_i, obtains the entire *view* of P_i up to this point, and may now send messages with label $(i, \cdot, |m|)$. The *view* of a server consists of its initialization data, all messages it has received, and the random choices it made so far.

Termination. We define *termination* of a protocol instance only to the extent that the adversary chooses to deliver messages among the honest servers [4]. Technically, termination of a protocol follows from a bound on the number of messages that honest servers generate on behalf of a protocol, which must be independent of the adversary.

We say that a message is *associated* to a particular protocol instance if it was generated by any server that is honest throughout the protocol execution on behalf of the protocol.

The *message complexity* of a protocol is defined as the number of associated messages (generated by honest servers). It is a random variable that depends on the adversary and on κ.

Similarly, the *communication complexity* of a protocol is defined as the bit length of all associated messages (generated by honest servers). It is a random variable that depends on the adversary and on κ.

Recall that the adversary runs in time polynomial in κ. We assume that the parameter n is bounded by a fixed polynomial in κ, independent of the adversary, and that the same holds for all messages in the protocol, i.e., larger messages are ignored.

For a particular protocol, a *protocol statistic* X is a family of real-valued, non-negative random variables $\{X_A(\kappa)\}$, parameterized by adversary A and security parameter κ, where each $X_A(\kappa)$ is a random variable induced by running the system with A. (Message complexity is an example of such a statistic.) We restrict ourselves to protocol statistics that are bounded by a polynomial in the adversary's running time.

We say that a protocol statistic X is *uniformly bounded* if there exists a fixed polynomial $p(\kappa)$ such that for all adversaries A, there is a negligible function ϵ_A, such that for all $\kappa \geq 0$,

$$\Pr[X_A(\kappa) > p(\kappa)] \leq \epsilon_A(\kappa).$$

A protocol statistic X is called *probabilistically uniformly bounded* if there exists a fixed polynomial $p(\kappa)$ and a fixed negligible function δ such that for all adversaries A, there is a negligible function ϵ_A, such that for all $l \geq 0$ and $\kappa \geq 0$,

$$\Pr[X_A(\kappa) > lp(\kappa)] \leq \delta(l) + \epsilon_A(\kappa).$$

If X is probabilistically uniformly bounded by p, then for all adversaries A, we have $\mathrm{E}[X_A(\kappa)] = O(p(\kappa))$, with a hidden constant that is independent of A. Additionally, if Y is probabilistically uniformly bounded by q, then $X \cdot Y$ is probabilistically uniformly bounded by $p \cdot q$, and $X + Y$ is probabilistically uniformly bounded by $p + q$. Thus, (probabilistically) uniformly bounded statistics are closed under polynomial composition, which is their main benefit for analyzing the composition of randomized protocols [3].

Protocol execution and notation. We now introduce our notation for writing asynchronous protocols. Recall that a server is always activated with an input message; this message is added to an internal input buffer upon activation.

In our model, protocols are invoked by the adversary. Every protocol *instance* is identified by a unique string ID, also called the *tag*, which is chosen by the adversary when it invokes the instance.

There may be several threads of execution for a given server, but no more than one is active concurrently. When a server is activated, all threads are in *wait states*. A wait state specifies a condition defined on the received messages contained in the input buffer and other local state variables. If one or more threads are in a wait state whose condition is satisfied, one such thread is scheduled arbitrarily, and this thread runs until it reaches another wait state. This process continues until no more threads are in a wait state whose condition is satisfied. Then, the activation of the server is terminated, and control returns to the adversary.

There are two types of messages that protocols process and generate: The first type contains *input actions*, which represent a local activation and carry input to a protocol, and *output actions*, which signal termination and potentially carry output of a protocol; such messages are called *local events*. The second message type is an ordinary point-to-point network message, which is to be delivered to the peer protocol instance running on another server; such messages are also called *protocol messages*.

All messages are denoted by a tuple $(ID, \ldots)$; the tag ID denotes the protocol instance to which this message is *associated*. Input actions are of the form $(ID, \texttt{in}, type, \ldots)$, and output actions are of the form $(ID, \texttt{out}, type, \ldots)$, with *type* defined by the protocol specification. All other messages of the form $(ID, type, \ldots)$ are protocol messages, where *type* is defined by the protocol implementation.

We describe protocols in a modular way: A protocol instance may invoke another protocol instance by sending it a suitable input action and obtain its output via an output action of the sub-protocol. This is realized by a server-internal mechanism, which, for any message generated by the calling protocol that contains an input action for a sub-protocol, creates the corresponding protocol instance (if not already running) and delivers the input action; furthermore, it passes all output actions of the sub-protocol to the calling protocol by adding them to the input buffer.

The pseudo-code notation used for describing our protocols is as follows. To enter a wait state, a thread may execute a command of the form **wait for** *condition*, where *condition* is an ordinary predicate on the input buffer and other state variables. Upon executing this command, a thread enters a wait state with the given *condition*.

We specify a *condition* in the form of *receiving messages* or *events*. In this case, *messages* describes a set of one or more protocol messages and *events* describes a set of local events (e.g., outputs from a sub-protocol) satisfying a certain predicate, possibly involving other state variables. Upon executing this command, a thread enters a wait state, waiting for the arrival of messages satisfying the given predicate; moreover, when this predicate becomes satisfied, the matching messages are *moved* out of the input buffer into local state variables. If there is more than one set of matching messages, one is chosen arbitrarily.

We also may specify a *condition* of the form of *detecting messages*. The semantics of this are the same as for *receiving messages*, except that the matching messages are *copied* from the input buffer into local state variables.

There is a global implicit **wait for** statement that every protocol instance repeatedly executes; it matches any of the *conditions* given in the clauses of the form **upon** *condition block*. Every time a *condition* is satisfied, the corresponding *block* is executed. If there is more than one satisfied *condition*, all corresponding *blocks* are executed in an arbitrary order.

We use the terminology **unless** *condition* **do** *block* to denote that *block* is executed as long as the specified condition does *not* hold. If the thread enters a wait state during *block*, and another activation of the server changes its internal state such that the specified condition holds, the execution of *block* is aborted.

2.2 Cryptographic Assumptions

Let p and q be two large primes satisfying $q|(p-1)$, and $q > n$. Let G denote a multiplicative subgroup of order q of $\mathbb{Z}_p$, and let g and h be two generators of G chosen by an initialization algorithm such that no server knows $\log_g h$.

The *discrete-logarithm problem* is to compute $\log_g u$ given a description of G, a generator g of G, and an element $u \in G$. We assume that this problem is hard to solve in G, which means that any probabilistic polynomial-time algorithm solves this problem at most with negligible probability.

2.3 Multi-valued Validated Byzantine Agreement

Byzantine agreement is a fundamental problem in distributed computation [21]. In asynchronous networks, it is impossible to solve by deterministic protocols [13], which means that one must resort to randomized protocols. The first polynomial-time solution to this problem was given by Canetti and Rabin [9, 5]. The standard notion of Byzantine agreement implements only a binary decision in asynchronous networks. It can guarantee a particular outcome only if *all* honest servers propose the same value. *Validated Byzantine agreement* [3] extends this to arbitrary domains by means of a so-called *external* validity condition. It is based on a global, polynomial-time computable predicate Q_{ID} known to all servers, which is determined by an external application. Each server may propose a value that perhaps contains validation information. The agreement ensures that the decision value satisfies Q_{ID}, and that it has been proposed by at least one server.

When a server P_i starts a validated Byzantine agreement (VBA) protocol with a tag ID and input $v \in \{0,1\}^*$, we say P_i *proposes* v *for* ID. W.l.o.g. the honest servers propose values that satisfy Q_{ID}. When a server terminates a validated Byzantine agreement protocol with tag ID and outputs a value v, we say P_i *decides* v *for* ID.

Definition 1. (Validated Byzantine Agreement) A protocol for *validated Byzantine agreement* with predicate Q_{ID} satisfies the following conditions for every t-limited adversary, except with negligible probability:

External Validity: Every honest server that terminates decides v for ID such that $Q_{ID}(v)$ holds.

Agreement: If some honest server decides v for ID, then any honest server that terminates decides v for ID.

Liveness: If all honest servers have been activated on ID and all associated messages have been delivered, then all honest servers have decided for ID.

Integrity: If all servers follow the protocol, and if some server decides v for ID, then some server proposed v for ID.

Efficiency: For every ID, the communication complexity of instance ID is probabilistically uniformly bounded.

The protocol of Cachin et al. [3] for multi-valued validated Byzantine agreement is based on a so-called consistent broadcast protocol and on a protocol for binary Byzantine agreement, which rely on threshold signatures and on a threshold coin-tossing protocol [4]. Both sub-protocols can be implemented efficiently in the random oracle model. With these primitives, the expected message complexity of multi-valued validated agreement is $O(n^2)$, and the expected communication complexity is $O(n^3 + n^2(K + |v|))$, where v is the longest value proposed by any server and K is the length of a threshold signature. These protocols have been proven secure only against static adversaries [3].

As we show in this paper, binary asynchronous Byzantine agreement can also be implemented efficiently in the standard model and with adaptive security based on verifiable secret sharing. This solution incurs a larger communication complexity than the one in [3], however.

3 Asynchronous Verifiable Secret Sharing

In this section we define asynchronous verifiable secret sharing (AVSS) and propose a novel efficient AVSS protocol based on the discrete-logarithm problem.

3.1 Definition

We consider *dual-threshold sharings*, which generalize the standard notion of secret sharing by allowing the reconstruction threshold to exceed the number of corrupted servers by more than one [23]. In an (n, k, t) dual-threshold sharing, there are n servers holding shares of a secret, of which up to t may be corrupted by an adversary, and any group of k or more servers may reconstruct the secret ($n - t \geq k > t$). Such dual-threshold sharings are an important primitive for distributed computation and agreement problems [4].

A protocol with a tag $ID.d$ to share a secret $s \in \mathbb{Z}_q$ consists of a *sharing* stage and a *reconstruction* stage as follows.

Sharing stage. The sharing stage starts when a server initializes the protocol. In this case, we say the server *initializes a sharing $ID.d$*. There is a special server P_d, called the *dealer*, which is activated additionally on an input message of the form $(ID.d, \texttt{in}, \texttt{share}, s)$. If this occurs, we say *P_d shares s using $ID.d$* among the group. A server is said to *complete the sharing $ID.d$* when it generates an output of the form $(ID.d, \texttt{out}, \texttt{shared})$.

Reconstruction stage. After a server has completed the sharing, it may be activated on a message $(ID.d, \texttt{in}, \texttt{reconstruct})$. In this case, we say the server *starts the reconstruction for $ID.d$.* At the end of the reconstruction stage, every server should output the shared secret. A server P_i terminates the reconstruction stage by generating an output of the form $(ID.d, \texttt{out}, \texttt{reconstructed}, z_i)$. In this case, we say *$P_i$ reconstructs z_i for $ID.d$.* This terminates the protocol.

The definition of asynchronous verifiable secret sharing is the same as in synchronous networks, except that some extra care is required to ensure that all servers agree on the fact that a valid sharing has been established. Our definition provides computational correctness and unconditional privacy.

Definition 2. A protocol for *asynchronous verifiable dual-threshold secret sharing* satisfies the following conditions for any t-limited adversary:

Liveness: If the adversary initializes all honest servers on a sharing $ID.d$, delivers all associated messages, and the dealer P_d is honest throughout the sharing stage, then all honest servers complete the sharing, except with negligible probability.

Agreement: Provided the adversary initializes all honest servers on a sharing $ID.d$ and delivers all associated messages, the following holds: If some honest server completes the sharing $ID.d$, then all honest servers complete the sharing $ID.d$ and if all honest servers subsequently start the reconstruction for $ID.d$, then every honest server P_i reconstructs some z_i for $ID.d$, except with negligible probability.

Correctness: Once k honest servers have completed the sharing $ID.d$, there exists a fixed value $z \in \mathbb{Z}_q$ such that the following holds except with negligible probability:

1. If the dealer has shared s using $ID.d$ and is honest throughout the sharing stage, then $z = s$.
2. If an honest server P_i reconstructs z_i for $ID.d$, then $z_i = z$.

Privacy: If an honest dealer has shared s using $ID.d$ and less than $k - t$ honest servers have started the reconstruction for $ID.d$, the adversary has no information about s.

Efficiency: For every $ID.d$, the communication complexity is uniformly bounded.

The first two conditions are liveness conditions. They imply the same form of termination and agreement as required by the *Byzantine generals problem* [19], which implements a *reliable broadcast* with Byzantine faults [17, 3] from a distinguished server to all others. The servers must terminate the protocol only if the distinguished server is honest, but they agree on the termination of the protocol such that either none or all honest servers terminate the protocol and generate some output.

This definition is analogous to the definition of AVSS in the information-theoretical model by Canetti and Rabin [9].

3.2 Implementation

This section describes a novel verifiable secret sharing protocol for an asynchronous network with computational security. Our protocol creates a discrete logarithm-based sharing of the kind introduced by Pedersen [22], and it is much more efficient than the previous VSS protocols for asynchronous networks [1, 9, 5] (which were proposed in the information-theoretic model). Our protocol uses exactly the same communication pattern as the asynchronous broadcast primitive proposed by Bracha [2], which implements the Byzantine generals problem in an asynchronous network.

Protocol AVSS creates an (n, k, t) dual-threshold sharing for any $n - 2t \geq k > t$. The sharing stage works as follows (assume $k \geq \lceil \frac{n+t+1}{2} \rceil$ for the moment).

1. The dealer computes a two-dimensional sharing of the secret by choosing a random bivariate polynomial $f \in \mathbb{Z}_q[x, y]$ of degree at most $k - 1$ with $f(0, 0) = s$. It commits to $f(x, y) = \sum_{j,l=0}^{k-1} f_{jl} x^j y^l$ using a second random polynomial $f' \in \mathbb{Z}_q[x, y]$ of degree at most $k - 1$ by computing a matrix $\mathbf{C} = \{C_{jl}\}$ with $C_{jl} = g^{f_{jl}} h^{f'_{jl}}$ for $j, l \in [0, k - 1]$. Then the dealer sends to every server P_i a message containing the commitment matrix $\mathbf{C}$ as well as two *share polynomials* $a_i(y) := f(i, y)$ and $a'_i(y) := f'(i, y)$ and two *sub-share polynomials* $b_i(x) := f(x, i)$ and $b'_i(x) := f'(x, i)$, respectively.

2. When they receive the `send` message from the dealer, the servers *echo* the points in which their share and sub-share polynomials overlap to each other. To this effect, P_i sends an `echo` message containing $\mathbf{C}$, $a_i(j)$, $a'_i(j)$, $b_i(j)$, and $b'_i(j)$ to every server P_j.

3. Upon receiving k `echo` messages that agree on $\mathbf{C}$ and contain valid points, every server P_i interpolates its own share and sub-share polynomials $\bar{a}_i, \bar{a}'_i, \bar{b}_i$, and $\bar{b}'_i$ from the received points using standard Lagrange interpolation. (In case the dealer is honest, the resulting polynomials are the same as those in the `send` message.) Then P_i sends a `ready` message containing $\mathbf{C}, \bar{a}_i(j), \bar{a}'_i(j)$, $\bar{b}_i(j)$, and $\bar{b}'_i(j)$ to every server P_j.

   It is also possible that a server receives k valid `ready` messages that agree on $\mathbf{C}$ and contain valid points, but has not yet received k valid `echo` messages. In this case, the server interpolates its share and sub-share polynomials from the `ready` messages and sends its own `ready` message to all servers as above.

4. Once a server receives a total of $k + t$ `ready` messages that agree on $\mathbf{C}$, it *completes* the sharing. Its share of the secret is $(s_i, s'_i) = (\bar{a}_i(0), \bar{a}'_i(0))$.

The reconstruction stage is straightforward. Every server P_i reveals its share (s_i, s'_i) to every other server, and waits for k such shares from other servers that are consistent with the commitments $\mathbf{C}$. Then it interpolates the secret $f(0, 0)$ from the shares.

For smaller values of k, in particular for $t < k < \lceil \frac{n+t+1}{2} \rceil$, the protocol has to be modified to receive $\lceil \frac{n+t+1}{2} \rceil$ `echo` messages in step 3. This guarantees the uniqueness of the shared value.

A detailed description of the protocol is given in Figures 1 and 2. In the protocol description, the following predicates are used:

verify-poly$(\mathbf{C}, i, a, a', b, b')$, where a, a', b, and b' are polynomials of degree $k - 1$, i.e.,

$$a(y) = \sum_{l=0}^{k-1} a_l y^l, \qquad a'(y) = \sum_{l=0}^{k-1} a'_l y^l, \qquad b(x) = \sum_{j=0}^{k-1} b_j x^j, \quad \text{and} \quad b'(x) = \sum_{j=0}^{k-1} b'_j x^j;$$

the predicate verifies that the given polynomials are share and sub-share polynomials for P_i consistent with $\mathbf{C}$; it is true if and only if for all $l \in [0, k-1]$, it holds $g^{a_l} h^{a'_l} = \prod_{j=0}^{k-1} (C_{jl})^{i^j}$, and for all $j \in [0, k-1]$, it holds $g^{b_j} h^{b'_j} = \prod_{l=0}^{k-1} (C_{jl})^{i^l}$.

verify-point$(\mathbf{C}, i, m, \alpha, \alpha', \beta, \beta')$ verifies that the given values α, α', β, and β' correspond to the points $f(m, i)$, $f'(m, i)$, $f(i, m)$, and $f'(i, m)$, respectively, committed to in $\mathbf{C}$, which P_i supposedly receives from P_m; it is true if and only if $g^{\alpha} h^{\alpha'} = \prod_{j,l=0}^{k-1} (C_{jl})^{m^j i^l}$ and $g^{\beta} h^{\beta'} = \prod_{j,l=0}^{k-1} (C_{jl})^{i^j m^l}$.

verify-share$(\mathbf{C}, m, \sigma, \sigma')$ verifies that the pair (σ, σ') forms a valid share of P_m with respect to $\mathbf{C}$; it is true if and only if $g^{\sigma} h^{\sigma'} = \prod_{j=0}^{k-1} (C_{j0})^{m^j}$.

The servers may need to interpolate a polynomial a of degree at most $k - 1$ over $\mathbb{Z}_q$ from a set $\mathcal{A}$ of k points $\{(m_1, \alpha_{m_1}), \ldots, (m_k, \alpha_{m_k})\}$ such that $a(m_j) = \alpha_{m_j}$ for $j \in [1, k]$. This can be done using standard Lagrange interpolation. We abbreviate this by saying a server *interpolates* a from $\mathcal{A}$; should $\mathcal{A}$ contain more than k elements, an arbitrary subset of k elements is used for interpolation.

In the protocol description, the variables e and r count the number of `echo` and `ready` messages, respectively. They are instantiated separately only for values of $\mathbf{C}$ that have actually been received in incoming messages.

Intuitively, protocol AVSS performs a reliable broadcast of $\mathbf{C}$ using the protocol of Bracha [2], where every `echo` and `ready` message between two servers P_i and P_j additionally contains the values $f(i, j)$, $f(j, i)$, $f'(i, j)$, and $f'(j, i)$, which they have in common.

The protocol uses $O(n^2)$ messages and has communication complexity $O(\kappa n^4)$. The size of the messages is dominated by $\mathbf{C}$; it can be reduced by a factor of n as shown in Section 3.4.

Protocol AVSS **for server** P_i **and tag** $ID.d$ **(sharing stage)**

upon initialization:

  for all $\mathbf{C}$ **do**

    $e_{\mathbf{C}} \leftarrow 0; r_{\mathbf{C}} \leftarrow 0$

    $\mathcal{A}_{\mathbf{C}} \leftarrow \emptyset; \mathcal{A}'_{\mathbf{C}} \leftarrow \emptyset; \mathcal{B}_{\mathbf{C}} \leftarrow \emptyset; \mathcal{B}'_{\mathbf{C}} \leftarrow \emptyset$

upon *receiving* a message $(ID.d, \texttt{in}, \texttt{share}, s)$:   /* only P_d */

  choose two random bivariate polynomials $f, f' \in \mathbb{Z}_q[x,y]$ of degree $k-1$ with $f(0,0) = f_{00} = s$, i.e.,

$$f(x,y) = \sum_{j,l=0}^{k-1} f_{jl} x^j y^l \qquad \text{and} \qquad f'(x,y) = \sum_{j,l=0}^{k-1} f'_{jl} x^j y^l$$

  $\mathbf{C} \leftarrow \{C_{jl}\}$, where $C_{jl} = g^{f_{jl}} h^{f'_{jl}}$ for $j, l \in [0, k-1]$

  for $j \in [1,n]$ **do**

    $a_j(y) \leftarrow f(j,y); a'_j(y) \leftarrow f'(j,y); b_j(x) \leftarrow f(x,j); b'_j(x) \leftarrow f'(x,j)$

    send the message $(ID.d, \texttt{send}, \mathbf{C}, a_j, a'_j, b_j, b'_j)$ to P_j

upon *receiving* a message $(ID.d, \texttt{send}, \mathbf{C}, a, a', b, b')$ from P_d for the first time:

  if $\mathsf{verify\text{-}poly}(\mathbf{C}, i, a, a', b, b')$ **then**

    for $j \in [1,n]$ **do** send the message $(ID.d, \texttt{echo}, \mathbf{C}, a(j), a'(j), b(j), b'(j))$ to P_j

upon *receiving* a message $(ID.d, \texttt{echo}, \mathbf{C}, \alpha, \alpha', \beta, \beta')$ from P_m for the first time:

  if $\mathsf{verify\text{-}point}(\mathbf{C}, i, m, \alpha, \alpha', \beta, \beta')$ **then**

    $\mathcal{A}_{\mathbf{C}} \leftarrow \mathcal{A}_{\mathbf{C}} \cup \{(m,\alpha)\}; \mathcal{A}'_{\mathbf{C}} \leftarrow \mathcal{A}'_{\mathbf{C}} \cup \{(m,\alpha')\}$

    $\mathcal{B}_{\mathbf{C}} \leftarrow \mathcal{B}_{\mathbf{C}} \cup \{(m,\beta)\}; \mathcal{B}'_{\mathbf{C}} \leftarrow \mathcal{B}'_{\mathbf{C}} \cup \{(m,\beta')\}$

    $e_{\mathbf{C}} \leftarrow e_{\mathbf{C}} + 1$

    if $e_{\mathbf{C}} = \max\{\lceil \frac{n+t+1}{2} \rceil, k\}$ **and** $r_{\mathbf{C}} < k$ **then**

      interpolate $\bar{a}$, $\bar{a}'$, $\bar{b}$, and $\bar{b}'$ from $\mathcal{A}_{\mathbf{C}}$, $\mathcal{A}'_{\mathbf{C}}$, $\mathcal{B}_{\mathbf{C}}$, and $\mathcal{B}'_{\mathbf{C}}$, respectively

      for $j \in [1,n]$ **do** send the message $(ID.d, \texttt{ready}, \mathbf{C}, \bar{a}(j), \bar{a}'(j), \bar{b}(j), \bar{b}'(j))$ to P_j

upon *receiving* a message $(ID.d, \texttt{ready}, \mathbf{C}, \alpha, \alpha', \beta, \beta')$ from P_m for the first time:

  if $\mathsf{verify\text{-}point}(\mathbf{C}, i, m, \alpha, \alpha', \beta, \beta')$ **then**

    $\mathcal{A}_{\mathbf{C}} \leftarrow \mathcal{A}_{\mathbf{C}} \cup \{(m,\alpha)\}; \mathcal{A}'_{\mathbf{C}} \leftarrow \mathcal{A}'_{\mathbf{C}} \cup \{(m,\alpha')\}$

    $\mathcal{B}_{\mathbf{C}} \leftarrow \mathcal{B}_{\mathbf{C}} \cup \{(m,\beta)\}; \mathcal{B}'_{\mathbf{C}} \leftarrow \mathcal{B}'_{\mathbf{C}} \cup \{(m,\beta')\}$

    $r_{\mathbf{C}} \leftarrow r_{\mathbf{C}} + 1$

    if $r_{\mathbf{C}} = k$ **and** $e_{\mathbf{C}} < \max\{\lceil \frac{n+t+1}{2} \rceil, k\}$ **then**

      interpolate $\bar{a}$, $\bar{a}'$, $\bar{b}$, and $\bar{b}'$ from $\mathcal{A}_{\mathbf{C}}$, $\mathcal{A}'_{\mathbf{C}}$, $\mathcal{B}_{\mathbf{C}}$, and $\mathcal{B}'_{\mathbf{C}}$, respectively

      for $j \in [1,n]$ **do** send the message $(ID.d, \texttt{ready}, \mathbf{C}, \bar{a}(j), \bar{a}'(j), \bar{b}(j), \bar{b}'(j))$ to P_j

    else if $r_{\mathbf{C}} = k + t$ **then**

      $\bar{\mathbf{C}} \leftarrow \mathbf{C}$

      $(s_i, s'_i) \leftarrow (\bar{a}(0), \bar{a}'(0))$ /* (s_i, s'_i) is the share of P_i */

      output $(ID.d, \texttt{out}, \texttt{shared})$

Figure 1: Protocol AVSS for asynchronous verifiable secret sharing (sharing stage).

Protocol AVSS **for server** P_i **and tag** $ID.d$ **(reconstruction stage)**

upon *receiving* a message $(ID.d, \texttt{in}, \texttt{reconstruct})$:

$c \leftarrow 0; \mathcal{S} \leftarrow \emptyset$

for $j \in [1, n]$ **do** send the message $(ID.d, \texttt{reconstruct-share}, s_i, s'_i)$ to P_j

upon *receiving* a message $(ID.d, \texttt{reconstruct-share}, \sigma, \sigma')$ from P_m:

if verify-share$(\bar{\mathbf{C}}, m, \sigma, \sigma')$ **then**

$\mathcal{S} \leftarrow \mathcal{S} \cup \{(m, \sigma)\}$

$c \leftarrow c + 1$

if $c = k$ **then**

interpolate a_0 from $\mathcal{S}$

output $(ID.d, \texttt{out}, \texttt{reconstructed}, a_0(0))$

halt

Figure 2: Protocol AVSS for asynchronous verifiable secret sharing (reconstruction stage).

Note that protocol AVSS creates an ordinary $(n, t+1, t)$-sharing with optimal resilience $n > 3t$, and an $(n, 2t+1, t)$-sharing with resilience $n > 4t$. It is an open problem to develop an AVSS protocol with comparable efficiency that creates arbitrary dual-threshold sharings (or even sharings with $k = 2t + 1$) with optimal resilience.

We prove the following theorem in the next section.

Theorem 1. *Assuming the hardness of the discrete-logarithm problem, protocol* AVSS *implements asynchronous verifiable dual-threshold secret sharing for* $n - 2t \geq k > t$.

3.3 Analysis

We have to show that protocol AVSS satisfies *liveness*, *agreement*, *correctness*, *privacy*, and *efficiency* according to Definition 2. The proof relies on the following lemma.

Lemma 2 ([2]). *Suppose an honest servers* P_i *sends a* `ready` *message containing* $\mathbf{C}_i$ *and a distinct honest server* P_j *sends a* `ready` *message containing* $\mathbf{C}_j$. *Then* $\mathbf{C}_i = \mathbf{C}_j$.

Proof. We prove the lemma by contradiction. Suppose $\mathbf{C}_i \neq \mathbf{C}_j$. P_i generates the `ready` message for $\mathbf{C}_i$ only if it has received at least $\lceil \frac{n+t+1}{2} \rceil$ `echo` messages containing $\mathbf{C}_i$ or k `ready` messages containing $\mathbf{C}_i$. In the second case, at least one honest server has sent a `ready` message containing $\mathbf{C}_i$ upon receiving at least $\lceil \frac{n+t+1}{2} \rceil$ `echo` messages; we may as well assume that this is P_i to simplify the rest of the argument. Thus, P_i has received $\lceil \frac{n+t+1}{2} \rceil$ `echo` messages containing $\mathbf{C}_i$, of which up to t are from corrupted servers.

Using the same argumentation, P_j must have received at least $\lceil \frac{n+t+1}{2} \rceil$ `echo` messages containing $\mathbf{C}_j$.

Then there are at least $2\lceil \frac{n+t+1}{2} \rceil = n+t+1$ `echo` messages received by P_i and P_j together, among them at least $n - t + 1$ from honest servers. But no honest server generates more than one such message by the protocol. □

Liveness. If the dealer P_d is honest, it follows directly by inspection of the protocol that all honest servers complete the sharing $ID.d$, provided all servers initialize the sharing $ID.d$ and the adversary delivers all associated messages.

Agreement. We first show that if some honest server completes the sharing $ID.d$, then all honest servers complete the sharing $ID.d$, provided all servers initialize the sharing $ID.d$ and the adversary delivers all associated messages.

Suppose an honest server has completed the sharing. Then it has received $k+t$ valid `ready` messages that agree on some $\bar{\mathbf{C}}$. Of these, at least k have been sent by honest servers. A *valid* `echo` or `ready` message is one that satisfies verify-point, and it is easy to see from the definition of verify-poly and verify-point that honest servers send only valid `ready` messages.

Since an honest server sends its `ready` message to all servers, every honest server receives at least k valid `ready` messages with the same $\bar{\mathbf{C}}$ by Lemma 2 and sends a `ready` message containing $\bar{\mathbf{C}}$. Hence, by the assumption of the theorem, any honest server receives $n-t \geq k+t$ valid `ready` messages containing $\bar{\mathbf{C}}$ and completes the sharing.

As for the reconstruction part, it follows from Lemma 2 that every honest server P_i computes the same $\bar{\mathbf{C}}$. Moreover, P_i has received enough valid `echo` or `ready` messages with respect to $\bar{\mathbf{C}}$ so that it computes valid `ready` messages and a *valid* share s_i, s'_i with respect to $\bar{\mathbf{C}}$ (a share such that verify-share$(\bar{\mathbf{C}}, i, s_i, s'_i)$ holds). Thus, if all honest servers subsequently start the reconstruction stage, then every server receives enough valid shares to reconstruct some value, provided the adversary delivers all associated messages.

Correctness. Let $\mathcal{J}$ be the index set of the k honest servers P_j that have completed the sharing, and let (s_j, s'_j) be their shares. Let $\lambda_j^{\mathcal{J}}$ for $j \in \mathcal{J}$ denote the appropriate Lagrange interpolation coefficients for the set $\mathcal{J}$ and position 0. Define

$$z = \sum_{j \in \mathcal{G}} \lambda_j^{\mathcal{J}} s_j.$$

To prove the first part, suppose the dealer has shared s and is honest throughout the sharing stage. Towards a contradiction assume $z \neq s$. Because the dealer is honest, it is easy to see that every `echo` message sent from an honest P_i to P_j contains $\mathbf{C}$, $f(i,j)$, $f'(i,j)$, $f(j,i)$, and $f'(j,i)$ as computed by the dealer. Furthermore, if the servers in $\mathcal{J}$ computed their shares only from these `echo` messages, then $s_j = \bar{a}_j(0) = f(j,0)$. But since $z \neq s$, at least one honest server P_i computed a polynomial $\bar{a}_i(y) \neq f(i,y)$; this must be because P_i accepted an `echo` or `ready` message from some corrupted P_m containing $\alpha \neq f(m,i)$.

Since P_i has evaluated verify-point to true, we have

$$g^{\alpha} h^{\alpha'} = \prod_{j,l=0}^{k-1} (\bar{C}_{jl})^{m^j i^l}. \tag{1}$$

On the other hand, the dealer has sent polynomials a_m and a'_m to P_m satisfying

$$g^{a_m(i)} h^{a'_m(i)} = \prod_{j,l=0}^{k-1} (C_{jl})^{i^j m^l} \tag{2}$$

and $(a_m(i), a'_m(i) = (f(m,i), f'(m,i))$. It is easy to see from Lemma 2 and from the fact that the dealer is honest that $\mathbf{C}$ used by the dealer and $\bar{\mathbf{C}}$ used by P_i are equal. Thus, $g^{\alpha} h^{\alpha'} = g^{a_m(i)} h^{a'_m(i)}$ from (1) and (2). Together with $\alpha \neq f(m,i) = a_m(i)$ from above, this implies also

$$\alpha' \neq f'(m,i) = a'_m(i). \tag{3}$$

Rewriting this using $h = g^{\log_g h}$ and comparing exponents yields $\alpha + (\log_g h)\alpha' = a_m(i) + (\log_g h) a'_m(i)$. Because of (3), one can compute $\log_g h = (\alpha - a_m(i))/(a'_m(i) - \alpha')$, a contradiction.

To prove the second part, assume that two distinct honest servers P_i and P_j reconstruct values z_i and z_j such that $z_i \neq z_j$. This means that they have received two distinct sets $\mathcal{S}_i = \{(l, s_l^{(i)}, s_l'^{(i)})\}$ and $\mathcal{S}_j = \{(l, s_l^{(j)}, s_l'^{(j)})\}$ of k shares each, which are valid with respect to the unique commitment matrix $\bar{\mathbf{C}}$ used by P_i and P_j (the uniqueness of $\bar{\mathbf{C}}$ follows from Lemma 2).

According to the protocol, z_i and z_j are interpolated from the sets $\{(l, s_l^{(i)})\}$ and $\{(l, s_l^{(j)})\}$ obtained from $\mathcal{S}_i$ and $\mathcal{S}_j$, respectively. Let z_i' and z_j' be interpolated analogously from $\{(l, s_l'^{(i)})\}$ and $\{(l, s_l'^{(j)})\}$. Since the shares in $\mathcal{S}_i$ and $\mathcal{S}_j$ are valid, it is easy to see that $g^{z_i} h^{z_i'} = \bar{C}_{00} = g^{z_j} h^{z_j'}$. But then one can rewrite this using $h = g^{\log_g h}$ and compute $\log_g h = (z_i - z_j)/(z_j' - z_i')$.

Privacy. Fix any point in time, and let $\mathcal{B}$ be the index set of servers that are either corrupted or have already started the reconstruction for $ID.d$. W.l.o.g. assume $|\mathcal{B}| = k - 1$ and that the adversary's view consists of the polynomials $f(x, i)$, $f'(x, i)$, $f(i, y)$, and $f'(i, y)$ for $i \in \mathcal{B}$ and the commitments $\mathbf{C}$ as computed by the dealer.

We have to show that for every value $\tilde{s} \in \mathbb{Z}_q$, there exist two polynomials $\tilde{f}, \tilde{f}' \in \mathbb{Z}_q[x, y]$ of degree at most $k - 1$ that are consistent with the adversary's view and such that $\tilde{f}(0, 0) = \tilde{s}$.

Note that there is a unique value $\tilde{s}' \in \mathbb{Z}_q$ such that $C_{00} = g^{\tilde{s}} h^{\tilde{s}'}$. The values $\tilde{s}'$ and $\tilde{s}$ together with the polynomials $f(x, i)$, $f'(x, i)$, $f(i, y)$, and $f'(i, y)$ in the view of the adversary define uniquely two polynomials $\tilde{f}, \tilde{f}' \in \mathbb{Z}_q[x, y]$ of degree at most $k - 1$ such that $\tilde{f}(0, 0) = \tilde{s}$ and $\tilde{f}'(0, 0) = \tilde{s}'$, as well as

$$f(x, i) = \tilde{f}(x, i), \quad f'(x, i) = \tilde{f}'(x, i), \quad f(i, y) = \tilde{f}(i, y), \quad \text{and} \quad f'(i, y) = \tilde{f}'(i, y) \tag{4}$$

for $i \in \mathcal{B}$. It remains to show that $C_{lm} = g^{\tilde{f}_{lm}} h^{\tilde{f}'_{lm}}$ for $l, m \in [0, k - 1]$.

Define $e(x, y) = f(x, y) + \ell f'(x, y)$, where $\ell = log_g h$. Then we have $g^{e_{lm}} = C_{lm}$ for $l, m \in [0, k - 1]$. If we analogously define $\tilde{e}(x, y) = \tilde{f}(x, y) + \ell \tilde{f}'(x, y)$, all we have to show is $e(x, y) = \tilde{e}(x, y)$.

Recall that $e(x, i) = f(x, i) + \ell f'(x, i)$ and $e(i, y) = f(i, y) + \ell f'(i, y)$ for $i \in \mathcal{B}$ as well as $e(0, 0) = f(0, 0) + \ell f'(0, 0)$ by construction. Inserting (4) into the definition of $\tilde{e}$, we have $\tilde{e}(x, i) = e(x, i)$ and $\tilde{e}(i, y) = e(i, y)$ for $i \in \mathcal{B}$. In addition, we know that $g^{e(0,0)} = C_{00} = g^{\tilde{s}} h^{\tilde{s}'} = g^{\tilde{s} + \ell \tilde{s}'} = g^{\tilde{e}(0,0)}$, from the definitions of $\tilde{s}'$, of ℓ, and of $\tilde{f}$, $\tilde{f}'$, and $\tilde{e}$ (in this order). Thus, the polynomials e and $\tilde{e}$ are equal.

Efficiency. Every honest server sends at most one `echo`, `ready`, and `reconstruct-share` message to every other server, which yields a total of $O(n^2)$ messages. Since the size of all messages is bounded by $O(\kappa n^2)$, it follows easily that the communication complexity is uniformly bounded.

3.4 Reducing Message Sizes

In the sharing stage of the protocol AVSS described above, every server P_i resends the commitment matrix $\mathbf{C}$ with every message it sends. Intuitively, this is needed for two reasons: first, to allow the honest servers to agree on the value that is a commitment to the secret being shared, and second, to allow the servers to verify that the secret shares they receive correspond to this commitment. We show in this section how to guarantee these two ends without having the servers resend so much data.

The new protocol relies on a collision-resistant hash function H. This is not an extra assumption because it is well-known that the hardness of the discrete-logarithm problem implies efficient collision-resistant hash functions. In practice, hash functions can be implemented at very little cost.

Recall from Section 3.2 that to create a secret sharing, the dealer selects two bivariate polynomials f and f'. Also, recall the notation a_i, a_i', b_i, b_i' from the description in Section 3.2. Let $A^{(i)} = (A_0^{(i)}, A_1^{(i)}, \ldots, A_n^{(i)})$ denote the $(n + 1)$-element list formed by setting $A_j^{(i)} = g^{a_i(j)} h^{a_i'(j)}$ for $j \in [0, n]$. Let $B^{(i)}$ be derived analogously from b_i and b_i'. Define lists $A^{(0)}$ and $B^{(0)}$ analogously with $A_j^{(0)} = g^{f(0,j)} h^{f'(0,j)}$ and $B_j^{(0)} = g^{f(j,0)} h^{f'(j,0)}$ for $j \in [0, n]$.

Modifications to the dealer's part of the sharing protocol. Instead of sending $\mathbf{C}$ to each server, P_d adds the following values, which we will denote by $\mathbf{D}$, to every `send` message:

1. $A^{(0)}$ and $B^{(0)}$;

2. $h_a = (h_{a,0}, \ldots, h_{a,n})$ and $h_b = (h_{b,0}, \ldots, h_{b,n})$, where $h_{a,j} = H(A^{(j)})$ and $h_{b,j} = H(B^{(j)})$.

In addition, the dealer sends the polynomials a_i, a_i', b_i and b_i' to each server P_i as before. Note that as a result, the dealer sends n messages of length $O(\kappa n)$ each.

Modifications to P_i's part of the sharing protocol. In the modified protocol, P_i computes the lists $A^{(i)}$ and $B^{(i)}$ from the received data and adds them to every `echo` or `ready` message, together with the public $\mathbf{D}$ from the dealer's message. This allows every server to perform the same checks as before, but reduces the length of every message to $O(\kappa n)$. Furthermore, messages are counted separately with respect to $\mathbf{D}$ instead of $\mathbf{C}$.

The modified protocol uses the following predicates (in each, $\mathbf{D} = (A^{(0)}, B^{(0)}, h_a, h_b)$ as described above):

$\textsf{check-poly}(\mathbf{D}, i, A, B)$, where A and B are $(n+1)$-element lists, is satisfied if $A_i^{(0)} = B_0$, $B_i^{(0)} = A_0$, $h_{a,i} = H(A)$, and $h_{b,i} = H(B)$.

$\textsf{check-point}(C, \gamma, \gamma')$ checks that C is a commitment to γ and γ'; it is satisfied if and only if $C = g^\gamma h^{\gamma'}$.

$\textsf{verify-poly}(\mathbf{D}, i, a, a', b, b')$, where a, a', b, and b' are polynomials of degree $k-1$, is satisfied if and only if $\textsf{check-poly}(\mathbf{D}, i, A, B)$ for the lists $A = (A_0, \ldots, A_n)$ and $B = (B_0, \ldots, B_n)$ formed by setting $A_j = g^{a(j)} h^{a'(j)}$ and $B_j = g^{b(j)} h^{b'(j)}$, respectively.

$\textsf{verify-point}(\mathbf{D}, i, m, A, B, \alpha, \alpha', \beta, \beta')$, where A and B are the $(n+1)$-element lists received from P_m, verifies that the given values α, α', β, and β' correspond to the points $f(m,i)$, $f'(m,i)$, $f(i,m)$, and $f'(i,m)$, respectively, committed to in $\mathbf{D}$; it is true if and only if

$$\textsf{check-poly}(\mathbf{D}, m, A, B) \ \wedge\ \textsf{check-point}(A_i, \alpha, \alpha') \ \wedge\ \textsf{check-point}(B_i, \beta, \beta').$$

$\textsf{verify-share}(\mathbf{D}, m, \sigma, \sigma')$ verifies that the pair (σ, σ') forms a valid share of P_m with respect to $\mathbf{D}$; it is true if and only if $g^\sigma h^{\sigma'} = A_m^{(0)}$.

The remaining details of the modified protocol can now easily be filled in. The part for reconstructing the secret remains the same, except for the new definition of the verify-share predicate.

It is clear that the message complexity of the revised protocol is the same as the message complexity of the protocol in Section 3.2. It is also clear that the communication complexity is reduced to $O(\kappa n^3)$ because every single message sent out by the new protocol includes $\mathbf{D}$, which is of size $O(\kappa n)$, instead of $\mathbf{C}$, which is of size $O(\kappa n^2)$.

Analysis. We must now argue why the resulting protocol retains the properties of an asynchronous verifiable secret sharing protocol. The *liveness*, *agreement*, and *privacy* properties follow in exactly the same way as in Section 3.3. The *correctness* property is the only one that needs to be elaborated on.

First, observe that Lemma 2 holds as well for this protocol with the obvious modifications (replacing $\mathbf{C}$ by $\mathbf{D}$). Let $\mathcal{J}$ and z be as in the proof of correctness in Section 3.3.

Suppose the dealer is honest and has shared s and yet $z \neq s$. As before, because the dealer is honest, it is easy to see that every `echo` message sent from an honest P_i to P_j contains $\mathbf{D}$, $A^{(i)}$, $B^{(i)}$, $f(i,j)$, $f'(i,j)$, $f(j,i)$, and $f'(j,i)$ as computed by the dealer. Furthermore, if the servers in $\mathcal{J}$ computed their shares only from these `echo` messages, then $s_j = \bar{a}_j(0) = f(j,0)$. But since

$z \neq s$, at least one honest server P_i computed a polynomial $\bar{a}_i(y) \neq f(i, y)$; this must be because P_i accepted an `echo` or `ready` message from some corrupted P_m containing $\alpha \neq f(m, i)$. Since P_i has evaluated $\mathsf{verify\text{-}point}(\mathbf{D}, i, m, A, B, \alpha, \alpha', \beta, \beta')$ to true, we know that $\mathsf{check\text{-}poly}(\mathbf{D}, m, A, B)$ and $\mathsf{check\text{-}point}(A_i, \alpha, \alpha')$ hold. Suppose $A^{(i)} \neq A$. Then we have broken the hash function, since check-poly would fail if $h_{a,i} = H(A^{(i)}) \neq H(A)$. So $A^{(i)} = A$. But then we break the discrete-logarithm assumption as before.

Suppose the dealer is corrupted and two distinct honest servers P_i and P_j reconstruct values z_i and z_j such that $z_i \neq z_j$. By Lemma 2, they both accepted the unique commitment $\bar{\mathbf{D}}$, which includes the value $A^{(0)}$. This means that they have received two distinct sets $\mathcal{S}_i = \{(l, s_l^{(i)}, s_l'^{(i)})\}$ and $\mathcal{S}_j = \{(l, s_l^{(j)}, s_l'^{(j)})\}$ of k shares each, which are valid with respect to $A^{(0)}$. According to the protocol, z_i and z_j are interpolated from the sets $\{(l, s_l^{(i)})\}$ and $\{(l, s_l^{(j)})\}$ obtained from $\mathcal{S}_i$ and $\mathcal{S}_j$, respectively. Let z_i' and z_j' be interpolated analogously from $\{(l, s_l'^{(i)})\}$ and $\{(l, s_l'^{(j)})\}$. Since the shares in $\mathcal{S}_i$ and $\mathcal{S}_j$ pass the verify-share test, but interpolate to distinct values, we obtain two ways of opening the commitment contained in $A^{(0)}$, which contradicts the discrete-logarithm assumption. The details are similar as before and left to the reader.

Further Improvements. Suppose instead of using just the two generators g and h of the group G, we use generators $g_1, \ldots, g_N$, and h. Then, in order to share N secrets $s_1, \ldots, s_N$, the dealer computes $N + 1$ bivariate polynomials $f_1, \ldots, f_N$, and f', and forms the entries of the verification matrix $\mathbf{C}$ as $C_{jl} = g_1^{f_1(j,l)} g_2^{f_2(j,l)} \cdots g_n^{f_n(j,l)} h^{f'(j,l)}$. The rest of the protocol is carried out analogously to the protocol described above. As a result, we can have a dealer share N secrets at the cost of $O(n^2)$ messages and $O(\kappa n^2(n + N))$ communication.

3.5 Application to Asynchronous Byzantine Agreement

Byzantine agreement is a fundamental problem in distributed computation [21]. In asynchronous networks, it is impossible to solve by deterministic protocols [13], which means that one must resort to randomized protocols. The first polynomial-time solution to this problem was given by Canetti and Rabin [9, 5]. However, this result is a *proof of concept* and not a practical solution because the complexity of their protocol is rather high: the message complexity is $O(n^6)$ and the communication complexity is $O(n^8 \log n)$.

The cost of this protocol is dominated by their asynchronous verifiable secret sharing protocol for sharing n secrets. Our protocol for the same task from the previous section is $\Theta(n^3)$ times more efficient for message complexity, and approximately $\Theta(n^4)$ times more efficient for communication complexity. We propose to plug our AVSS protocol directly into the Byzantine agreement protocol of Canetti and Rabin [9] (an excellent exposition of how AVSS is used in asynchronous Byzantine agreement is given in [5]). As a result, assuming the hardness of the discrete-logarithm problem, the complexity of asynchronous Byzantine agreement is reduced to $O(n^3)$ message complexity and $O(\kappa n^4)$ communication complexity.

We stress that this works in the *computational* setting, whereas Canetti and Rabin [9] use an unconditional model. We also mention that in the so-called random-oracle model, a more efficient protocol exists, which is secure against a static adversary [4]. However, the random-oracle model makes an idealizing assumption about cryptographic hash functions, which involves certain problems [7], and a proof in this model falls short from a proof in the real world. Hence, our AVSS protocol yields the first asynchronous Byzantine agreement protocol that is provably secure in the standard model *and* whose efficiency makes it suitable for use in practice.

4 Asynchronous Proactive Model

Proactive cryptosystems combine distribution with a periodic refresh operation in order to protect the secret key against a *mobile* adversary, who can move from one server to another and corrupt all servers during the lifetime of the system [20, 18]. In this section, we propose an extension of the asynchronous system model given in Section 2 for proactive cryptosystems. We argue that such an extension is necessary and that our proposal is minimal. An asynchronous proactive refresh protocol for shared secrets, which forms the core of every proactive cryptosystem, is presented in the next section.

Motivation. A proactive cryptosystem is a threshold cryptosystem that tolerates an adversary who can gradually break into any number of servers. To protect against leaking the secret key, it operates in a sequence of *phases* and the servers periodically *refresh* their shares between two phases. The new set of shares is independent of the previous one and the old shares are erased. Thus, the adversary may corrupt up to t different servers in any phase without learning anything about the secret key.

The underlying assumption is that breaking into a server requires a certain amount of time, which occurs for every server that is corrupted, independent from other corruptions. It must also be possible to remove the adversary by rebooting a server in a trusted way (e.g., from a read-only device) and to erase information on a server permanently.

This concept maps onto a synchronous network in a straightforward way. In an asynchronous network, however, the following two issues regarding phases and secure channels arise.

First, the notion of a common phase is not readily available because there is no common clock. Since refreshing requires a distributed protocol, in which all servers should participate, at least some synchronization primitive is needed to define the length of a phase in a meaningful way. It turns out that a single time signal or *clock tick*, which defines the start of every phase locally, is enough. In our formal model, we leave the scheduling of this signal up to the network, i.e., the adversary. In practice, this might be an impulse from an external clock, say every day at 0:00 UTC. Hence, phases are defined locally to every server. The adversary may corrupt up to t servers who are in the same local phase.

Second, the channels that link the servers have to be adapted to this model. Recall that all servers are linked by secure channels (i.e., private and authenticated links), which are scheduled by the adversary. Given only locally defined phases and purely asynchronous scheduling, however, it would be possible for the adversary to break the secure channels assumption as follows. Suppose all servers are in the same local phase and the adversary has corrupted t of them. In order to read any message sent between two honest servers, the adversary may delay the message until the receiver enters the next phase and some of the previously corrupted servers are again honest. Then she corrupts the receiver and observes the message, which gives her access to private information from the previous phase of more than t servers.

Therefore, we assume that secure channels in the proactive model guarantee that messages are delivered in the same local phase in which they are sent. More precisely, a message sent in some local phase of the sender arrives when the receiver is in the same local phase or it is invariably lost. Under these restrictions, we leave all scheduling up to the adversary. In practice, such proactive secure channels might be implemented by re-keying every point-to-point link when a phase change occurs, as discussed below.

We now proceed to the formal description of the model.

Formal Model. A server is a PITM as before, which can now also *erase* information. We define erasing in terms of restricting a server's view. To erase information means to exclude the corresponding values from the server's view.

As before, the adversary may corrupt a server at any point in time, but now it can now also be removed from a corrupted server by a *reboot* procedure. In this case, the server is restarted with correct initialization data, and the proactive protocols running before the corruption are invoked again (how these protocols are determined is outside our model). The internal state of the server may have been modified arbitrarily by the adversary.

Every server operates in a sequence of local phases, which are defined with respect to a trivial protocol *timer*. Every honest server continuously runs one instance of this protocol, which starts when the server is initialized. Upon initialization, the protocol sends a timer message called a *clock tick* to itself. Whenever the server receives a clock tick, the server resends the message to itself over the network. The *local phase* of an uncorrupted server P_i is defined as the number of clock ticks that it has received so far. If the adversary corrupts a server during some phase τ, we define the corrupted server to remain in local phase τ until it is rebooted and the adversary is removed. We assume that after a reboot, a server is automatically activated on a clock tick and continues to operate in the subsequent phase. Hence every server is honest at the point in time when it enters the next local phase. However, the adversary can cause a server to appear corrupted during multiple subsequent phases (and across the phase changes) by corrupting it again immediately after the phase change.

Since the set of honest servers may change from one phase to another, we also define the set of *associated messages* accordingly.

An adversary in the proactive network model is called t*-limited* if for every phase index $\tau \geq 0$, it corrupts at most t servers in local phase τ. Recall that activations are atomic and cannot be interrupted by a corruption. This allows an honest server to perform some actions, like erasing critical data, at the very beginning of a phase (**upon** *detecting* a clock tick) *before* it can be corrupted by the adversary during this phase.

We assume that every pair of servers is linked by a *proactive secure asynchronous channel*, which is defined as follows. Recall that in our asynchronous network model, the adversary can schedule messages in a set $\mathcal{M}$ with labels of the form (s, r, l). In the proactive network, a number τ is added to every label denoting the local phase in which P_s has sent the message. Then we restrict the scheduling as follows. If P_j enters local phase τ, all messages in $\mathcal{M}$ with labels $(\cdot, j, \cdot, \sigma)$ where $\sigma < \tau$ are removed from $\mathcal{M}$. Furthermore, the adversary may not schedule any message with label $(\cdot, j, \cdot, \tau)$ before P_j has entered its local phase τ. We say that *the adversary delivers messages within phases* to denote an adversary that delivers all messages in $\mathcal{M}$ with a label of the form $(\cdot, j, \cdot, \tau)$ to a receiver P_j when P_j is in local phase τ. If the adversary corrupts a server P_j during its local phase τ, then all messages $m \in \mathcal{M}$ with label $(\cdot, j, \cdot, \tau)$ are removed from $\mathcal{M}$ and given to the adversary, who may now send messages with label $(j, \cdot, \cdot, \tau)$.

Note that every honest server runs a separate instance of the *timer* protocol, and that we view this protocol as an integral part of the proactive system model. As such, it is not required to terminate or to satisfy a uniform bound on its communication complexity. It will simply run until the adversary halts.

Implementation. In practice, asynchronous proactive secure channels with the described properties could be implemented using secure co-processors as follows. The communication link between every pair of servers is encrypted and authenticated using a phase session key that is stored in secure hardware. A fresh session key is established in the co-processor as soon as both enter a new phase, with authentication based on data stored in secure hardware (if a public-key infrastructure is used, this may be a single root certificate). Thus, even if the adversary corrupts a server, she gains access to the phase session key only through calls to the co-processor. The external clock which triggers the phase changes must have a trusted path into the secure co-processor and an intruder must not be able to influence it.

The related problem of maintaining proactive authenticated communication in a synchronous network has been investigated by Canetti et al. [8].

Related Work. Proactive systems in asynchronous networks have been discussed by Castro and Liskov [10] and by Zhou [24]; the former aims at maintaining a common state, and the latter at maintaining a shared secret. In these works, the phases are defined with respect to proactive protocols, i.e., a phase *ends* upon the termination of the corresponding update protocol. Our approach is more general in the sense that we postulate the phases only with respect to a timeout mechanism, independent of proactive protocols. This models also systems where a refresh protocol may not terminate within a phase. Our protocols therefore postulate two types of conditions: liveness conditions, which hold only if the protocol terminates within a phase, and safety conditions, which hold in any case.

Another difference lies in our network model, which identifies the main security requirements on asynchronous proactive secure communication. While authenticity of messages in such a setting is addressed in terms of a special freshness requirement in [10], a formal treatment of these aspects is missing in [24].

Finally, from a practical point of view, our implementation of the refresh protocol is much more efficient than the one of Zhou [24]. Ours has an expected message complexity of $\mathcal{O}(n^3)$ as opposed to $\mathcal{O}(\binom{n}{t})$.

5 Asynchronous Proactive Refresh Protocol

In this section, we describe how a group of servers holding shares of a secret may refresh these shares in an asynchronous proactive network such that the adversary does not learn anything about the secret. Such protocols form the basis of any proactive cryptosystem. We define the notion of a verifiable sharing and the properties of a protocol to refresh such a sharing. Then we propose an implementation of a refresh protocol for discrete logarithm-based verifiable sharings as established by protocol AVSS from Section 3. We restrict ourselves to ordinary $(n, t+1, t)$-sharings in this section.

5.1 Definitions

Verifiable sharing. A *sharing* of a (secret) value $s_0 \in \mathbb{Z}_q$ can be seen as an encoding of s_0 into a set of *shares* S_i such that all sets of at least $t+1$ shares uniquely define s_0, whereas any other set of shares does not give any information about s_0.

Such a sharing results, for example, from the first stage of an AVSS protocol. A sharing is robust against erasures in the sense that a unique secret can also be reconstructed from a subset the shares. Missing shares of honest servers are denoted by $\perp$.

A *verifiable sharing*, or *v-sharing* for short, has the additional property that the secret is defined uniquely even if the adversary corrupts up to t servers and modifies their shares in an arbitrary way.

We define a verifiable sharing in terms of an algorithm *reconstruct* that takes as input a set of shares $\{S_i\}$ and outputs a value in $\mathbb{Z}_q$ or $\perp$.

Definition 3. We say the servers hold a *verifiable sharing of s_0 with tag ID and with respect to an algorithm reconstruct*, if every server P_i holds a share S_i such that the following conditions are satisfied:

Integrity: For any set $\{S_i\}$ of shares that contains at least $t+1$ shares of honest servers different from $\perp$, running *reconstruct* on input $\{S_i\}$ yields s_0, except with negligible probability.

Privacy: Any set $\{S_i\}$ of at most t shares contains no information about s_0.

Notice that the integrity property is computational and the privacy property unconditional.

Refreshing a verifiable sharing. The goal of a proactive refresh protocol is to protect a verifiable sharing by providing the servers with new shares for the next phase such that the adversary's knowledge of shares from the previous phase is rendered useless.

Suppose the servers hold a v-sharing $S_1, \ldots, S_n$ of a value s_0 with tag *ID* and with respect to an algorithm *reconstruct* at some point in time where all honest servers are in local phase $\tau - 1$. Then an honest server starts a *refresh protocol* with tag *ID* and input S_i as soon as it *detects* and *receives* the next clock tick (all ongoing computations are aborted as soon as the clock tick is *detected*). This also marks the end of local phase $\tau - 1$ and the begin of phase τ. The refresh protocol terminates either when the server generates an output of the form $(ID, \texttt{refreshed})$ or when it *detects* the next clock tick. In the first case, we say *the server completes the refresh of sharing* ID.

The refresh protocol must ensure that the honest servers compute a fresh v-sharing of the same value s_0 and that any t-limited adversary does not learn any information on s_0. This is captured by the following definition.

Definition 4. Suppose the servers hold a verifiable sharing of some value s_0 with tag ID and with respect to some algorithm *reconstruct*. An *asynchronous secure refresh* protocol satisfies the following conditions for any t-limited adversary:

Liveness: If the adversary activates all honest servers on a clock tick and delivers all associated messages within phases, then all honest servers complete the refresh of sharing ID, except with negligible probability.

Correctness: If at least $t + 1$ honest servers have completed the refresh of sharing ID and have not *detected* a subsequent clock tick, the servers hold a verifiable sharing of s_0 with tag ID and with respect to *reconstruct*, except with negligible probability.

Privacy: In any polynomial number of consecutive executions of the protocol, the adversary's view is statistically independent of s_0.

Efficiency: For every ID, the communication complexity of instance ID is probabilistically uniformly bounded.

Note that this definition guarantees that the servers complete the refresh only when the adversary delivers messages within phases. Otherwise, the model allows the adversary to cause the secret to be lost, in order to preserve privacy. One could also imagine a different formalization of asynchronous proactive refresh protocols that preserves correctness at the cost of privacy, i.e., where the adversary may learn the secret. Such a trade-off between privacy and correctness seems unavoidable in asynchronous networks where messages may be delayed for longer than the duration of a proactive phase; interestingly, it does not arise for proactive cryptosystems in synchronous networks.

Another difference to the synchronous case is the fact that our phases do not overlap. As a consequence of this, a server must erase the old share during the *same* activation in which it receives the clock tick (in order to guarantee privacy of the secret). This point in time corresponds to the beginning of the refresh protocol, before the server may receive messages from other servers or become corrupted in the new local phase. In contrast, two subsequent phases in synchronous proactive cryptosystems are usually assumed to overlap for the duration of the refresh protocol, and a server may delay the erasure of an old share until the end of the refresh protocol.

5.2 Implementation

This section describes protocol $\mathsf{Refresh}$ for refreshing a discrete logarithm-based verifiable $(n, t+1, t)$-sharing in an asynchronous network. Its implementation needs the multi-valued validated Byzantine agreement protocol from Section 2.3, a digital signature scheme secure against adaptive chosen-message attacks [16] for every server, and the AVSS protocol from Section 3 as building blocks. We assume that such sub-protocols have the property that the calling protocol can access and modify their internal state and *abort* them if necessary by terminating the corresponding instance and *erasing* all associated local data. A local variable x associated with sub-protocol instance ID is denoted $x^{[ID]}$.

Recall that these primitives were defined in a purely asynchronous, non-proactive network. Hence, we use them only as sub-protocols running within a single phase; if a protocol does not terminate before the end of the phase, it must be aborted by the calling protocol. The security of the keys for the digital signature scheme and for the VBA protocol in the proactive corruption model has to be guaranteed by storing them inside secure co-processors or by using a proactively secure refresh protocols. The details of this are beyond the scope of this paper.

The verifiable sharing. We investigate how to refresh a discrete logarithm-based verifiable sharing as computed by protocol AVSS from Section 3. The share of an honest server P_i is of the form $S_i = (i, s_i, s'_i, V)$, where $V = (V_0, \ldots, V_t)$ is the same for all servers and $g^{s_i} h^{s'_i} = \prod_{j=0}^{t} (V_j)^{i^j}$; in other words, there exist two polynomials $a(x) = \sum_{j=0}^{t} a_j x^j$ and $a'(x) = \sum_{j=0}^{t} a'_j x^j$ over $\mathbb{Z}_q$ such that $a(i) = s_i$ and $a'(i) = s'_i$ for all correct shares S_i, and $g^{a_j} h^{a'_j} = V_j$ for $j \in [0, t]$. (Note that $V_j = C_{j0}$ using the notation of protocol AVSS.)

Algorithm *reconstruct* works as follows. On input a set $\mathcal{S}$ of shares, it selects a value V that is found in at least $t+1$ shares and discards shares that contain a different value for V. If V is not unique or does not exist, it returns $\perp$; otherwise, it computes a set $\mathcal{G} \subseteq \mathcal{S}$ of tuples $(i, s_i, s'_i, \cdot)$ that satisfy $g^{s_i} h^{s'_i} = \prod_{j=0}^{t} (V_j)^{i^j}$. If $|\mathcal{G}| \leq t$, it returns $\perp$; otherwise it interpolates a polynomial a of degree at most t from the set $\{(i, s_i) | (i, s_i, s'_i, \cdot) \in \mathcal{G}\}$ and returns $a(0)$.

The refresh protocol. From a high-level point of view, the protocol works in three stages. First, every server P_i shares its share s_i of s_0 using an AVSS protocol. Second, the servers use multi-valued Byzantine agreement to select $t+1$ such sharings that have successfully terminated. Third, they compute a fresh share of s_0 from the set of sharings which they agreed on.

More precisely, suppose the servers hold a verifiable sharing of s_0 with tag ID as described in the previous paragraph and have set up a digital signature scheme such that every server can verify signatures issued by any other server.

Then every server executes the following steps for protocol Refresh in phase τ.

1. Server P_i initializes n verifiable $(n, t+1, t)$-sharings $ID|\texttt{avss}.j$ for $j \in [1, n]$ using an *extended* version of protocol AVSS. Then it shares s_i and s'_i using $ID|\texttt{avss}.i$, where $f'^{[ID|\texttt{avss}.i]}(0,0)$ is set to s'_i, and immediately *erases* the current share and the sharing polynomials $f^{[ID|\texttt{avss}.i]}$ and $f'^{[ID|\texttt{avss}.i]}$ in instance $ID|\texttt{avss}.i$.

   The extension of protocol AVSS is that each server adds a digital signature to every `ready` message; in AVSS instance $ID|\texttt{avss}.j$, the signature is computed on $(ID|\texttt{avss}.j, \tau, \texttt{ready})$. A list Π of $2t+1$ such signatures is output when the server completes the sharing and may serve as a *proof* for this fact.

   The server also sends its current value of $V = (V_0, \ldots, V_t)$ all other servers in a `recover` message. Then it waits for *receiving* $t+1$ identical `recover` messages and assigns the value found in them to D.

2. The server waits for completing $t+1$ sharings $ID|\texttt{avss}.j$ such that $\mathbf{C}^{[ID|\texttt{avss}.j]}$ is consistent with D, i.e., $C_{00}^{[ID|\texttt{avss}.j]} = \prod_{l=0}^{t} (D_l)^{j^l}$. Recall that the extended AVSS protocol also returns a proof Π_j for the completion of the sharing.

   Next, P_i proposes the set of completed sharings for validated Byzantine agreement with tag $ID|\texttt{vba}$. Its proposal is a set $\mathcal{L}_i = \{(j, \Pi_j)\}$ of $t+1$ tuples, indicating the dealer P_j of every completed sharing and containing the list Π_j of signatures on `ready` messages from the extended sharing. The predicate of the VBA protocol is set to verify-termination(), described below, which verifies that a proposal contains $t+1$ valid lists of signatures from instances of protocol AVSS.

3. After the server decides in the VBA protocol for a set $\mathcal{L}$ that indicates $t+1$ AVSS instances, it waits for these sharings to complete. Then P_i computes its new share as follows: it interpolates two polynomials over $\mathbb{Z}_q$ from the set of shares computed in the AVSS instances indicated by $\mathcal{L}$; more precisely, polynomials $\bar{a}$ and $\bar{a}'$ of degree t are interpolated from the sets

$\{(j, s_i^{[ID|\mathtt{avss}.j]})|(j, \Pi_j) \in \mathcal{L}\}$ and $\{(j, s_i'^{[ID|\mathtt{avss}.j]})|(j, \Pi_j) \in \mathcal{L}\}$, respectively. Then the server sets the shares s_i and s_i' to $\bar{a}(0)$ and $\bar{a}'(0)$, respectively. The new commitments V are computed analogously.

Finally, the server *aborts* all sub-protocols $ID|\mathtt{avss}.j$, which automatically *erases* all information of these protocol instances.

Predicate verify-termination$(ID|\mathtt{vba}, \tau, \mathcal{L})$ used in VBA instance $ID|\mathtt{vba}$ verifies that $\mathcal{L}$ contains $t+1$ tags of AVSS protocols with the proofs that these protocols will actually terminate. It is true if and only if $|\mathcal{L}| = t+1$ and for every $(j, \Pi_j) \in \mathcal{L}$, the list Π_j contains at least $2t+1$ valid signatures on the string $(ID|\mathtt{avss}.j, \tau, \mathtt{ready})$ from distinct servers.

Figure 3 shows the detailed description of the protocol. A server may *interpolate* a polynomial $a \in \mathbb{Z}_q[x]$ of degree at most t from $t+1$ points as before. Let $\lambda_j^{\mathcal{J}}$ for some $\mathcal{J} \subset \{1, \ldots, n\}$ denote the appropriate Lagrange interpolation coefficient for $j \in \mathcal{J}$, set $\mathcal{J}$, and position 0.

Protocol Refresh **for server** P_i**, phase** τ**, and tag** ID

/* local inputs: s_i, s_i', and V */
for $j \in [1, n]$ **do** initialize a sharing $ID|\mathtt{avss}.j$ using the extended protocol AVSS
if $s_i \neq \perp$ **then**
  share s_i using $ID|\mathtt{avss}.i$, where $f'^{[ID|\mathtt{avss}.i]}_{00} = s_i'$
  erase s_i, s_i', and $f^{[ID|\mathtt{avss}.i]}$ and $f'^{[ID|\mathtt{avss}.i]}$ of instance $ID|\mathtt{avss}.i$
  send the message $(ID, \mathtt{recover}, V)$ to every server
unless a clock tick is *detected* **do**
  wait for *receiving* $t+1$ identical messages $(ID, \mathtt{recover}, D)$ from distinct servers
    with $D = (D_0, \ldots, D_t)$
  wait for $t+1$ sharings with tag $ID|\mathtt{avss}.j$ to complete such that $C_{00}^{[ID|\mathtt{avss}.j]} = \prod_{l=0}^{t} (D_l)^{j^l}$
  $\mathcal{L}_i \leftarrow \{(j, \Pi_j)\}$ for all sharings $ID|\mathtt{avss}.j$ completed in the previous step
  propose $\mathcal{L}_i$ in a multi-valued validated Byzantine agreement for $ID|\mathtt{vba}$ with
    predicate verify-termination
  wait for the VBA protocol to decide some $\mathcal{L}$ for $ID|\mathtt{vba}$
  $\mathcal{J} \leftarrow \{j|(j, \Pi_j) \in \mathcal{L}\}$
  wait for all sharings $ID|\mathtt{avss}.j$ for $j \in \mathcal{J}$ to complete
  interpolate $\bar{a}$ and $\bar{a}'$ from $\{(j, s_i^{[ID|\mathtt{avss}.j]})|j \in \mathcal{J}\}$ and $\{(j, s_i'^{[ID|\mathtt{avss}.j]})|j \in \mathcal{J}\}$, resp.
  for $l \in [0, t]$ **do**
    $\bar{V}_l \leftarrow \prod_{j \in \mathcal{J}} (C_{l0}^{[ID|\mathtt{avss}.j]})^{\lambda_j^{\mathcal{J}}}$
  $(s_i, s_i', V) \leftarrow (\bar{a}(0), \bar{a}'(0), (\bar{V}_0, \ldots, \bar{V}_t))$
  output $(ID, \mathtt{refreshed})$
abort protocols $ID|\mathtt{vba}$ and $ID|\mathtt{avss}.j$ for $j \in [1, n]$
/* local outputs: s_i, s_i', and V */

Figure 3: Protocol Refresh for proactive share refreshing in an asynchronous network, started upon *receiving* the τ-th clock tick.

As mentioned before, a key point of the protocol is that every server erases its old share in the first activation before waiting for any network input. The event of *receiving* the clock tick and starting the refresh protocol defines the end of local phase $\tau - 1$. Thus, one cannot tolerate to leave share information from phase $\tau - 1$ around when entering a wait state in phase τ because at any point in time afterwards, a corruption might occur that counts towards phase τ. This is also the reason why the protocol does not follow the approach of Gennaro et al. [14], which is to establish a set of sharings of the value 0 and to *add* these shares to the shares of the secret from phase $\tau - 1$ later on. Instead, our protocol creates sharings of previous shares of the secret and uses the agreed-on set of such sharings as a polynomial sharing of the secret itself.

The purpose of the `recover` messages is to supply the verification information V of phase $\tau-1$ to those honest servers that might have been corrupted in phase $\tau-1$ and have been rebooted into phase τ.

Protocol Refresh invokes n protocols for AVSS and one VBA sub-protocol. With AVSS implemented according to Section 3.4 and VBA from [3], its expected message complexity is $O(n^3)$ and its expected communication complexity is $O(\kappa n^4)$.

We prove the following theorem in the next section.

Theorem 3. *Assuming the hardness of the discrete-logarithm problem, protocol* Refresh *is an asynchronous secure refresh protocol for* $n > 3t$.

5.3 Analysis

We have to show the verifiable sharing satisfies Definition 3 and that protocol Refresh satisfies the *liveness*, *correctness*, *privacy*, and *efficiency* properties of Definition 4.

Verifiable sharing. Recall that the shares of our verifiable sharing are of the form $S_i = (i, s_i, s_i', V)$ with $V = (V_0, \ldots, V_t)$ such that there exist two polynomials $a, a' \in \mathbb{Z}_q[x]$ of degree at most t such that $a(i) = s_i$, $a'(i) = s_i'$, and $g^{s_i} h^{s_i'} = \prod_{j=0}^{t} (V_j)^{i^j}$ for all correct shares S_i. Furthermore, $a(0)$ is equal to the shared secret s_0.

We may assume that the adversary knows the shares (j, s_j, s_j', V) of the corrupted servers.

For *integrity*, we have to show that the adversary cannot compute shares $\tilde{S}_j = (j, \tilde{s}_j, \tilde{s}_j', \tilde{V})$ of corrupted servers P_j such that running *reconstruct* with these shares and at least $t+1$ shares of honest servers different from $\perp$ yields a value different from s_0.

Towards a contradiction, suppose the adversary has computed such values. Then it must be that $\tilde{V} = V$, because the adversary corrupts at most t servers and *reconstruct* accepts only shares that contain a value V found in at least $t+1$ shares. Moreover, it must be the case that $g^{\tilde{s}_j} h^{\tilde{s}_j'} = \prod_{l=0}^{t} (V_l)^{j^l}$ because otherwise *reconstruct* ignores these shares as well. Hence, the adversary has computed a tuple $(\tilde{s}_j, \tilde{s}_j') \neq (s_j, s_j')$ such that $g^{\tilde{s}_j} h^{\tilde{s}_j'} = \prod_{l=0}^{t} (V_l)^{j^l} = g^{s_j} h^{s_j'}$. Rewriting this with $h = g^{\log_g h}$ and comparing exponents gives $\tilde{s}_j + (\log_g h)\tilde{s}_j' = s_j + (\log_g h) s_j'$. Thus, the adversary has computed $\log_g h = (\tilde{s}_j - s_j)/(s_j' - \tilde{s}_j')$.

Privacy follows directly from the privacy property of protocol AVSS, which computes this verifiable sharing.

Liveness. We have to show that the conditions of all wait states that a server enters are eventually satisfied, provided the adversary delivers all associated messages within phases.

A server waiting for $t+1$ identical `recover` messages containing a value V from the verifiable sharing above will receive them because the *integrity* property of the v-sharing implies that there are at least $t+1$ honest servers with identical values V. It guarantees also that the value D computed by honest servers is unique.

The next step is to wait for $t+1$ sharings consistent with D to complete. There are at least $n - t \geq 2t+1$ honest servers who have completed the last phase with valid shares different from $\perp$. Even if t of them are now corrupted, there are at least $t+1$ honest servers whose sharings complete and are consistent with D. Thus, no honest server is blocked here.

Because all honest servers start the VBA protocol (with valid proposals), the *liveness* and *efficiency* conditions of VBA together imply that the VBA protocol also terminates.

The last step is to wait for the agreed-on sharings in $\mathcal{J}$ to complete. By the *external validity* of the VBA protocol and by the definition of the predicate verify-termination, there are for every AVSS instance $ID|\texttt{avss}.j$ with $j \in \mathcal{J}$ at least $2t+1$ valid signatures on the `ready` message, according to the extended protocol AVSS. Thus, at least $t+1$ honest servers have sent a `ready` message in these instances, which is sufficient to guarantee termination of the agreed-on instances of protocol AVSS.

Correctness. Fix a point in time where a set $\mathcal{H}$ of at least $t+1$ honest servers has completed the refresh protocol and not yet detected the next clock tick.

We have to show that they hold a verifiable sharing of s_0. According to the properties of our discrete logarithm-based sharing, it is sufficient to show that their shares (i, s_i, s'_i, V) as computed at the end of the refresh protocol satisfy $a(i) = s_i$ and $a'(i) = s'_i$ for $i \in \mathcal{H}$, and $g^{a_j} h^{b_j} = V_j$ for $j \in [0,t]$ for polynomials $a(y) = \sum_{j=0}^{t} a_j y^j$ and $a'(y) = \sum_{j=0}^{t} a'_j y^j$ with $a(0) = s_0$, and that V is the same for all servers.

Let $(i, \hat{s}_i, \hat{s}'_i, \hat{V})$ denote the shares of the initial verifiable sharing of s_0 which the servers hold at the beginning of the refresh protocol.

We start with the last condition above. By the definition of our verifiable sharing, honest servers start with the same $\hat{V} \neq \perp$; therefore, all honest servers accept the same $D = (D_0, \ldots, D_t)$ in the `recover` messages and any two honest servers receive the same $C_{00}^{[ID|\texttt{avss}.j]}$. Together with the *agreement* property of the VBA protocol, it follows easily that all servers in $\mathcal{H}$ assign the same value to V before completing the refresh protocol.

Let $\mathcal{J}$ denote the set of agreed-on sharings computed by honest servers and let $\lambda_j^{\mathcal{J}}$ for $j \in \mathcal{J}$ denote the Lagrange interpolation coefficients for the set $\mathcal{J}$ and position 0. Recall that protocol AVSS establishes a discrete logarithm-based verifiable sharing. Hence, for $j \in \mathcal{J}$ there exist two polynomials $a^{(j)}(y)$ and $a'^{(j)}(y)$ such that every share $(s_i^{[ID|\texttt{avss}.j]}, s_i'^{[ID|\texttt{avss}.j]})$ computed by any server P_i in $\mathcal{H}$ satisfies

$$a^{(j)}(i) = s_i^{[ID|\texttt{avss}.j]}, \quad a'^{(j)}(i) = s_i'^{[ID|\texttt{avss}.j]}, \quad \text{and} \quad g^{a_l^{(j)}} h^{a_l'^{(j)}} = C_{l0}^{[ID|\texttt{avss}.j]}. \tag{5}$$

Define

$$a(y) = \sum_{j \in \mathcal{J}} \lambda_j^{\mathcal{J}} a^{(j)}(y) \qquad \text{and} \qquad a'(y) = \sum_{j \in \mathcal{J}} \lambda_j^{\mathcal{J}} a'^{(j)}(y). \tag{6}$$

We first show that $a(i) = s_i$ and $a'(i) = s'_i$ for all servers in $\mathcal{H}$. Combining (5) and (6) yields

$$a(i) = \sum_{j \in \mathcal{J}} \lambda_j^{\mathcal{J}} a^{(j)}(i) = \sum_{j \in \mathcal{J}} \lambda_j^{\mathcal{J}} s_i^{[ID|\texttt{avss}.j]} \tag{7}$$

and

$$a'(i) = \sum_{j \in \mathcal{J}} \lambda_j^{\mathcal{J}} a'^{(j)}(i) = \sum_{j \in \mathcal{J}} \lambda_j^{\mathcal{J}} s_i'^{[ID|\texttt{avss}.j]}. \tag{8}$$

Now let $\bar{a}_i$ and $\bar{a}'_i$ denote the polynomials interpolated by P_i in protocol Refresh. According to the protocol, P_i interpolates $\bar{a}_i$ from $\{(j, s_i^{[ID|\texttt{avss}.j]}) | j \in \mathcal{J}\}$ and $\bar{a}'_i$ from $\{(j, s_i'^{[ID|\texttt{avss}.j]}) | j \in \mathcal{J}\}$. Hence, we can write

$$s_i = \bar{a}_i(0) = \sum_{j \in \mathcal{J}} \lambda_j^{\mathcal{J}} s_i^{[ID|\texttt{avss}.j]} \qquad \text{and} \qquad s'_i = \bar{a}'_i(0) = \sum_{j \in \mathcal{J}} \lambda_j^{\mathcal{J}} s_i'^{[ID|\texttt{avss}.j]}. \tag{9}$$

Combining (7) and (8) with (9) gives $a(i) = s_i$ and $a'(i) = s'_i$, as required.

We proceed by showing that $g^{a_l} h^{a'_l} = V_l$ for $l \in [0,t]$. By the definition (6) of the polynomials a and a', their coefficients are a linear combination of the coefficients of $a^{(j)}$ and $a'^{(j)}$, respectively: $a_l = \sum_{j \in \mathcal{J}} \lambda_j^{\mathcal{J}} a_l^{(j)}$ and $a'_l = \sum_{j \in \mathcal{J}} \lambda_j^{\mathcal{J}} a_l'^{(j)}$. Hence,

$$g^{a_l} h^{a'_l} = \prod_{j \in \mathcal{J}} \left(g^{a_l^{(j)}} h^{a_l'^{(j)}} \right)^{\lambda_j^{\mathcal{J}}} = \prod_{j \in \mathcal{J}} \left(C_{l0}^{[ID|\texttt{avss}.j]} \right)^{\lambda_j^{\mathcal{J}}} = V_l,$$

where the second step follows from (5) and the last step from the protocol.

It remains to show that $a(0) = s_0$. Towards a contradiction, assume $a(0) \neq s_0$. By the definition of a, this can only be the case if some AVSS protocol with tag $ID|\texttt{avss}.j^*$ for $j^* \in \mathcal{J}$ resulted in a verifiable sharing defining polynomials $a^{(j^*)}$ and $a'^{(j^*)}$ such that

$$a^{(j^*)}(0) \neq \hat{s}_{j^*}. \tag{10}$$

Let $\tilde{s}'_{j^*} := a^{(j^*)}(0)$. We have

$$g^{\tilde{s}_{j^*}} h^{\tilde{s}'_{j^*}} = C_{00}^{[ID|\texttt{avss}.j^*]} = \prod_{l=0}^{t} (D_l)^{(j^*)^l} \tag{11}$$

by the properties of protocol AVSS and because $j^* \in \mathcal{J}$ in protocol $\mathsf{Refresh}$. By the properties of the initial verifiable sharing,

$$g^{\hat{s}_{j^*}} h^{\hat{s}'_{j^*}} = \prod_{l=0}^{t} (\hat{V}_l)^{(j^*)^l}. \tag{12}$$

As shown earlier, $D = (D_0, \ldots, D_t)$ as computed by the protocol is equal to $\hat{V}$. Therefore, we conclude from (11) and (12) that $g^{\hat{s}_{j^*}} h^{\hat{s}'_{j^*}} = g^{\tilde{s}_{j^*}} h^{\tilde{s}'_{j^*}}$. But then, (10) implies $\log_g h = (\tilde{s}_{j^*} - \hat{s}_{j^*})/(\hat{s}'_{j^*} - \tilde{s}'_{j^*})$.

Privacy. We show that the adversary's view in an execution of the protocol is statistically independent of s_0.

The protocol starts from a verifiable sharing of s_0. Let $\hat{\mathcal{B}}$ denote the index set of the corrupted servers for the initial verifiable sharing. In regular operation of the proactive cryptosystem, for instance, $\hat{\mathcal{B}}$ are the corrupted servers from the *previous* execution of the refresh protocol. Furthermore, the adversary may corrupt a set $\mathcal{B}$ of servers *during* the execution of the refresh protocol, but only *after* their first activation according to the proactive system model. Let $\mathcal{H} = \{1, \ldots, n\} \setminus \mathcal{B}$, and assume w.l.o.g. that $\hat{\mathcal{B}} \cap \mathcal{B} = \emptyset$.

Thus, the view of the adversary in protocol $\mathsf{Refresh}$ consists w.l.o.g. of (i) the initial shares $(i, \hat{s}_i, \hat{s}'_i, (\hat{V}_0, \ldots, \hat{V}_t))$ for $i \in \hat{\mathcal{B}}$ from past corruptions, (ii) the polynomials received by the corrupted servers in the sub-protocol AVSS from honest servers during the current protocol,

$$f^{[ID|\texttt{avss}.j]}(x,i), \quad f'^{[ID|\texttt{avss}.j]}(x,i), \quad f^{[ID|\texttt{avss}.j]}(i,y), \quad \text{and} \quad f'^{[ID|\texttt{avss}.j]}(i,y)$$

for $i \in \mathcal{B}$ and $j \in \mathcal{H}$, (iii) the commitments $\mathbf{C}^{[ID|\texttt{avss}.j]}$ for $j \in \mathcal{H}$, and (iv) the set $\mathcal{J}$ as output by the Byzantine agreement protocol. Observer that the new shares of the corrupted servers in $\mathcal{B}$ are determined by this.

We have to show that this view is consistent with every possible $\tilde{s} \in \mathbb{Z}_q$. Because the protocol starts with a verifiable sharing, it follows directly from the privacy property of the discrete logarithm-based sharing that there exist polynomials $\tilde{a}, \tilde{a}' \in \mathbb{Z}_q[x]$ such that for $i \in \hat{\mathcal{B}}$ and $l \in [0, t]$,

$$\tilde{a}(0) = \tilde{s}, \quad \tilde{a}(i) = \hat{s}_i, \quad \tilde{a}'(i) = \hat{s}'_i, \quad \text{and} \quad g^{\tilde{a}_l} h^{\tilde{a}'_l} = \hat{V}_l.$$

The polynomials $\tilde{a}$ and $\tilde{a}'$ define initial shares $\tilde{s}_j = \tilde{a}(j)$ and $\tilde{s}'_j = \tilde{a}'(j)$ for P_j with $j \in \mathcal{H}$; moreover,

$$g^{\tilde{s}_j} h^{\tilde{s}'_j} = \prod_{l=0}^{t} (\hat{V}_l)^{j^l}. \tag{13}$$

From the *privacy* property of the AVSS sub-protocol, we know that the adversary's view in every instance $ID|\texttt{avss}.j$ for honest P_j is consistent with $\tilde{s}_j$ and $\tilde{s}'_j$ if these values are consistent with $C_{00}^{[ID|\texttt{avss}.j]}$.

Thus, it remains to show that $g^{\tilde{s}_j} h^{\tilde{s}'_j} = C_{00}^{[ID|\texttt{avss}.j]}$. We obtain

$$g^{\tilde{s}_j} h^{\tilde{s}'_j} = \prod_{l=0}^{t} (\hat{V}_l)^{j^l} = \prod_{l=0}^{t} (D_l)^{j^l} = C_{00}^{[ID|\texttt{avss}.j]}$$

from (13), from the proof of *correctness* above, and from the acceptance condition of the AVSS sub-protocols in the protocol.

Efficiency (Sketch). The servers execute n AVSS protocols and one VBA sub-protocol. In addition, they send $O(n^2)$ `recover` messages in total. Because the communication complexity of protocol AVSS is uniformly bounded and the communication complexity of the VBA sub-protocol is probabilistically uniformly bounded, it follows from the argumentation of Cachin et al. [3] that the communication complexity of protocol Refresh is probabilistically uniformly bounded.

References

[1] M. Ben-Or, R. Canetti, and O. Goldreich, "Asynchronous secure computation," in *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, 1993.

[2] G. Bracha, "An asynchronous $[(n-1)/3]$-resilient consensus protocol," in *Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 154–162, 1984.

[3] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols (extended abstract)," in *Advances in Cryptology: CRYPTO 2001* (J. Kilian, ed.), vol. 2139 of *Lecture Notes in Computer Science*, pp. 524–541, Springer, 2001.

[4] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography," in *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 123–132, 2000.

[5] R. Canetti, *Studies in Secure Multiparty Computation and Applications*. PhD thesis, Weizmann Institute, 1995.

[6] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor, "Proactive security: Long-term protection against break-ins," *RSA Laboratories' CryptoBytes*, vol. 3, no. 1, 1997.

[7] R. Canetti, O. Goldreich, and S. Halevi, "The random oracle methodology, revisited," in *Proc. 30th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 209–218, 1998.

[8] R. Canetti, S. Halevi, and A. Herzberg, "Maintaining authenticated communication in the presence of break-ins," *Journal of Cryptology*, vol. 13, no. 1, pp. 61–106, 2000.

[9] R. Canetti and T. Rabin, "Fast asynchronous Byzantine agreement with optimal resilience," in *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 42–51, 1993.

[10] M. Castro and B. Liskov, "Proactive recovery in a Byzantine-fault-tolerant system," in *Proc. Fourth Symp. Operating Systems Design and Implementation (OSDI)*, 2000.

[11] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch, "Verifiable secret sharing and achieving simultaneity in the presence of faults," in *Proc. 26th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 383–395, 1985.

[12] Y. Desmedt, "Threshold cryptography," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 449–457, 1994.

[13] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, pp. 374–382, Apr. 1985.

[14] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure key generation for discrete-log based cryptosystems," in *Advances in Cryptology: EUROCRYPT '99* (J. Stern, ed.), vol. 1592 of *Lecture Notes in Computer Science*, pp. 295–310, Springer, 1999.

[15] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on Computing*, vol. 18, pp. 186–208, Feb. 1989.

[16] S. Goldwasser, S. Micali, and R. L. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM Journal on Computing*, vol. 17, pp. 281–308, Apr. 1988.

[17] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems* (S. J. Mullender, ed.), New York: ACM Press & Addison-Wesley, 1993.

[18] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or how to cope with perpetual leakage," in *Advances in Cryptology: CRYPTO '95* (D. Coppersmith, ed.), vol. 963 of *Lecture Notes in Computer Science*, pp. 339–352, Springer, 1995.

[19] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382–401, July 1982.

[20] R. Ostrovsky and M. Yung, "How to withstand mobile virus attacks," in *Proc. 10th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 51–59, 1991.

[21] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, pp. 228–234, Apr. 1980.

[22] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology: CRYPTO '91* (J. Feigenbaum, ed.), vol. 576 of *Lecture Notes in Computer Science*, pp. 129–140, Springer, 1992.

[23] V. Shoup, "Practical threshold signatures," in *Advances in Cryptology: EUROCRYPT 2000* (B. Preneel, ed.), vol. 1087 of *Lecture Notes in Computer Science*, pp. 207–220, Springer, 2000.

[24] L. Zhou, *Towards Fault-tolerant and Secure On-line Services*. PhD thesis, Cornell University, 2001.