

# Long Live The Honey Badger: Robust Asynchronous DPSS and its Applications

Thomas Yurek<sup>1,4</sup> Zhuolun Xiang<sup>2</sup> Yu Xia<sup>3,4</sup> Andrew Miller<sup>1</sup>

<sup>1</sup>University of Illinois at Urbana-Champaign, <sup>2</sup>Carnegie Mellon University, <sup>3</sup>MIT CSAIL, <sup>4</sup>NTT Research  
yurek2@illinois.edu, zhuolunx@andrew.cmu.edu, yuxia@mit.edu, soc1024@illinois.edu

## Abstract

Secret sharing is an essential tool for many distributed applications, including distributed key generation and multiparty computation. For many practical applications, we would like to tolerate network churn, meaning participants can *dynamically* enter and leave the pool of protocol participants as they please. Such protocols, called Dynamic-committee Proactive Secret Sharing (DPSS) have recently been studied; however, existing DPSS protocols do not gracefully handle faults: the presence of even one unexpectedly slow node can often slow down the whole protocol by a factor of  $O(n)$ .

In this work, we explore optimally fault-tolerant *asynchronous* DPSS that is not slowed down by crash faults and even handles byzantine faults while maintaining the same performance. We first introduce the first *high-threshold* DPSS, which offers favorable characteristics relative to prior non-synchronous works in the presence of faults while simultaneously supporting higher privacy thresholds. We then batch-amortize this scheme along with a parallel non-high-threshold scheme which achieves optimal bandwidth characteristics. We implement our schemes and demonstrate that they can compete with prior work in best-case performance while outperforming it in non-optimal settings.

## 1 Introduction

Secret sharing [37] is an essential primitive in many fault-tolerant distributed applications, where a committee of nodes each hold a share of a secret and the secret can only be recovered once a threshold of the nodes reveal their shares. Secret shared data can also be used as input to a confidential computation using secure multiparty computation (MPC) without having to reveal the secret data at all.

For many long-running applications where the secret shared data persists over a long period of time, we need to consider practical aspects such as network churn, where the committee membership needs to change periodically due to nodes going offline. Additionally we may consider stronger adversary models, like a mobile adversary that may gradually corrupt

even more than the threshold number of nodes. Ordinary secret sharing schemes are no longer secure under these settings. To overcome these difficulties, previous works [19, 33, 36, 43] propose and study a generalization of secret sharing called *Dynamic-committee Proactive Secret Sharing (DPSS)*, where the secret shares can be refreshed among a possibly different set of committee nodes, while keeping the secret unchanged.

A limitation of most previous works is that they assume a perfectly synchronous network, e.g., a synchronous broadcast primitive or a blockchain. The consequence of this assumption is that these protocols are *unsafe under asynchrony*. A node that experiences a temporary network outage must be ejected after a timeout and deducted from the fault tolerance threshold; the protocol can be rerun without the ejected node, but now with a lower fault tolerance. For partially synchronous or asynchronous settings, very few DPSS protocols [36, 43] have been designed until very recently [38, 40]. However, these protocols either incur a high communication cost ( $O(n^4)$  to reshare a secret), or lose liveness under asynchrony [38], or compromise for non-optimal fault tolerance [40]. In contrast, this work aims to build protocols which are not only concretely efficient, but also highly *robust*, meaning that they perform well even in worst-case scenarios.

### Our contributions.

- We design the first asynchronous DPSS protocol which achieves an  $O(n^3)$  network bandwidth complexity and simultaneously achieve optimal fault tolerance and maintain our performance even under byzantine faults. This protocol additionally functions as the first realization of high-threshold resharing in a DPSS protocol.
- We additionally provide a batch-amortized version of our high threshold scheme which achieves a network complexity of  $O(n^2)$  and a third scheme which no longer supports high-threshold secrets but achieves an optimal  $O(n)$  amortized network bandwidth even under byzantine faults. All three schemes are implemented and the source code is made available.
- We provide a security analysis of our protocols under

the UC framework. We also survey and discuss numerous applications of our DPSS schemes, such as in confidential blockchains and MPC.

## 1.1 Related Work

We begin with a survey (summarized in Table 1) of prior DPSS works under a variety of fault and network assumptions. While all of these works offer an asymptotic best case performance value, few actually analyze how their protocols perform in the presence of multiple crash faults (let alone byzantine faults). Consequently, many of the asymptotic performances of prior works under faults in Table 1 are the result of our own estimates.

**Synchronous DPSS schemes.** The problem of proactive secret sharing was first introduced by Herzberg et al. [26], where a mobile adversary which gradually corrupted different members of a static committee holding some shared secret could be defended against via periodic share refreshes, each at communication cost of  $O(n^3)$ . Desmedt et al. [19] initiated the study of dynamic proactive secret sharing under synchrony, however, their protocol only considers a passive adversary which merely observes the protocol, but does not attempt to interfere with it. Wong et al. [39] proposed a verifiable DPSS solution extending the work of Desmedt et al. [19]. However, their solution requires all new committee members are honest, and has an exponential communication cost in the worst case.

The work of Baron et al. [6] provides statistical (rather than cryptographic) security but with a non-optimal resilience threshold  $t < (1/2 - \epsilon)n$ . They can achieve  $O(n^3)$  for the single-secret setting, and  $O(1)$  for the batch setting thanks to the use of virtualization techniques. However, these virtualization techniques are impractical in actual implementations as they require extraordinarily-large groups to function (to use an epsilon value small enough to achieve a  $t < n/3$  fault tolerance would require 576 committees of 576 nodes each running maliciously secure MPC. More discussion of this can be found in [33]). Additionally, the secrets in this scheme are all packed into the same polynomial and can not be used independently.

CHURP [33] uses asymmetric bivariate polynomials to refresh a secret with cost  $O(n^2)$  in the best case (when there are no faults), and cost  $O(n^3)$  in the worst case. Goyal et al. [23] recently proposed the state-of-the-art synchronous DPSS scheme that improves the cost of CHURP by a factor of  $O(n)$  in the batch setting. Similar to CHURP, their protocol optimizes the optimistic-case cost to  $O(n^2)$ , but has worst-case cost  $O(n^3)$  for the single-secret setting or  $O(n^2)$  for the batch setting. Similar to our scheme, they use a randomness extraction technique [8] for the batch setting.

**Partially synchronous DPSS schemes.** The only partially synchronous DPSS schemes we are aware of are Schultz-MPSS [36] and the very recent work of COBRA [38]. Schultz-MPSS [36] follows the primary-based approach where every iteration a primary node will determine a

proposal consisting the blinding polynomials (Herzberg et al. [26]). Practical Byzantine Fault Tolerance (PBFT) [16] is used to ensure agreement among all nodes. Similar to PBFT, malicious primaries need to be replaced via view-change, and the protocol only makes progress during periods of synchrony.

COBRA [38] uses Verifiable Secret Sharing (VSS) to generate blinding polynomials to facilitate resharing. Notably, as VSS does not guarantee that all honest parties receive shares, COBRA implements a share recovery mechanism in which for each player  $P_i$  requesting a missing share, a random polynomial  $R(\cdot)$  is generated where  $R(i) = 0$  and shares of  $\phi(\cdot) + R(\cdot)$  are sent to  $P_i$ . However, this recovery protocol costs  $O(n^3)$  network communication per recovered share and  $t$  honest parties may need to run it if some dealers crashed during the resharing phase. By using Asynchronous Complete Secret Sharing (ACSS), which guarantees that if one honest party outputs successfully then eventually all will, we are able to avoid this issue completely.

**Asynchronous DPSS schemes.** Cachin et al. [13] initiated the study of PSS under asynchronous networks, with a  $O(n^4)$  cost solution based on resharing the shares of the secret via AVSS and agreeing on the resharing via VBA. Their scheme inspires our first DPSS construction. Zhou et al. [43] proposed the first dynamic PSS scheme but with exponential cost. Then, the communication cost of asynchronous DPSS was improved to  $O(n^3 \log n)$  very recently by Shanrang [40]. However, Shanrang has non-optimal resilience of  $t < n/4$  and defaults to a synchronous fallback in the presence of byzantine behaviour.

Moreover, all existing asynchronous DPSS schemes do not consider high-threshold secrets, and do not attempt to achieve better amortized cost for the batch setting. In contrast, our work improves the communication cost of the state-of-the-art asynchronous DPSS protocol, while providing many desirable features such as optimal resilience, no trusted setup (i.e. no Structured Reference String is required to use the protocol), high-threshold reconstruction and batch amortization. For low-threshold secrets, our protocol can even achieve amortized linear cost assuming the trusted setup of KZG commitments.

## 2 Preliminaries

**System Model** We assume an asynchronous network of interconnected nodes, such that each pair of nodes can communicate over a reliable authenticated channel. We assume a static Probabilistic Polynomial Time adversary  $\mathcal{A}$  which can arbitrarily delay any message but can not read messages sent between honest nodes nor prevent them from eventually arriving. The adversary also controls  $t$  nodes in the old committee  $C$  and  $t'$  nodes in the new committee  $C'$  such that  $t < |C|/3$  and  $t' < |C'|/3$ . We also use  $n$  to denote the committee size when there is not ambiguity.

The new committee can contain any number of the same members as the old committee (including none) and  $\mathcal{A}$  can

Table 1: Comparison to Existing DPSS Schemes. For “Reshare Amortized”, the faults are Byzantine, and “—” means the cost is the same as “Reshare Byzantine”. For other places, “—” means not applicable.

Scheme	Network	Fault Tolerance	Dynamic	High-Threshold	Reshare Best-case	Reshare Crash	Reshare Byzantine	Reshare Amortized	No Trusted Setup
Herzberg et al. [26]	Sync	$n/2$	✗	✗	$O(n^3)$	$O(n^3)$	$O(n^3)$	—	✓
Desmedt et al. [19] <sup>1</sup>	Sync	$n/2$	✓	✗	$O(n^2)$	—	—	—	✓
Wong et al. [39]	Sync	$n/2$	✓	✗	$exp(n)$	$exp(n)$	$exp(n)$	—	✓
Baron et al. [6]	Sync	$(1/2-\epsilon)n$	✓	✗	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(1)$ <sup>2</sup>	✓
CHURP [33]	Sync	$n/2$	✓	✓ <sup>3</sup>	$O(n^2)$	$O(n^3)$	$O(n^3)$	—	✗
Goyal et al. [23]	Sync	$n/2$	✓	✗	$O(n^2)$	$O(n^3)$	$O(n^3)$	$O(n^2)$	✗
Schultz-MPSS [36]	P. Sync <sup>4</sup>	$n/3$	✓	✗	$O(n^4)$	$O(n^4)$	$O(n^4)$	—	✓
COBRA [38]	P. Sync	$n/3$	✓	✗	$O(n^3)$	$O(n^4)$	$O(n^4)$	—	✗ <sup>5</sup>
Cachin et al. [13]	Async	$n/3$	✗	✗	$O(n^4)$	$O(n^4)$	$O(n^4)$	—	✗
Zhou et al. [43]	Async	$n/3$	✓	✗	$exp(n)$	$exp(n)$	$exp(n)$	—	✓
Shanrang [40]	Async	$n/4$	✓	✗	$O(n^3 \log n)$	$O(n^3 \log n)$	$O(n^4)$	—	✗
<b>This work</b>	Async	$n/3$	✓	✓	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(n^2)$	✓
<b>This work</b>	Async	$n/3$	✓	✗	—	—	—	$O(n)$	✗

<sup>1</sup> Desmedt et al. [19] is not verifiable, and assumes passive adversary.

<sup>2</sup> Requires impractically-large committee sizes.

<sup>3</sup> CHURP [33] only supports dual-threshold.

<sup>4</sup> Schultz-MPSS [36] claims asynchrony but their protocol uses PBFT [16] and requires eventual synchrony for liveness.

<sup>5</sup> COBRA [38] uses KZG commitments [27], but it is possible to use other commitment schemes with no trusted setup, while keeping the same asymptotic cost.

choose which nodes to corrupt in each. In the case of static committees, this can be considered equivalent to a mobile adversary who can over the course of several refresh periods corrupt every node (though no more than  $t$  in one epoch). Our epoch definition and the corresponding constraint on the adversary follows MPSS [36], and more details can be found in Appendix A.

**Notation** For a given secret  $s$ , we use  $[s]$  to refer to a secret share of  $s$ . Additionally we may use  $[s]_d$  to specify a  $d$ -sharing of  $s$ , meaning that  $d + 1$  shares are needed to reconstruct it. Lastly,  $[s]_d^i$  refers to the specific  $d$ -sharing of  $s$  held by player  $P_i$ .

In order to achieve our secrecy properties, it is often necessary to pair a given secret  $s$  with a *blinding secret*  $\hat{s}$ , such as the case where the Pedersen commitment  $g^s h^{\hat{s}}$  is visible to the adversary. We use the  $\hat{\cdot}$  symbol generally to refer to a blinding object, such as a blinding polynomial  $\hat{\phi}(x)$ . Any object with the  $\hat{\cdot}$  symbol above it is assumed to be sampled uniformly randomly.

When defining a polynomial, we may use  $x$  and  $y$  as *free variables* and  $i$  and  $j$  as *indices*. For example,  $\phi(x)$  is a polynomial,  $\phi(i)$  is a point,  $\phi(x,y)$  is a bivariate polynomial, and  $\phi(x,j)$  is a univariate polynomial.

Lastly we use  $C$  to refer to the old committee of nodes which is set to transfer their shares to a new committee  $C'$ . More generally, we use  $'$  when an object is held by one or more members of  $C'$ :  $\phi'(x)$  is a polynomial held by  $C'$ , and  $[s']_d^j$  is the share of the  $d$ -shared blinding secret of  $s$  held by  $P_j'$ , the  $j$ 'th member of  $C'$ .

## 2.1 Asynchronous Complete Secret Sharing

Asynchronous Complete Secret Sharing (ACSS) is a protocol in which a dealer distributes shares of some secret  $s$ , such that any  $d + 1$  correct shares can be combined to recreate  $s$ .  $\mathcal{A}$  controls  $t$  nodes and in the general case  $d = t$ , but in the high threshold setting,  $t \leq d \leq n - t - 1$ . If the ACSS protocol terminates, then every honest party is guaranteed to eventually receive a correct share of  $s$ . An ACSS scheme consists of the following subprotocols:

- $\text{Share}(C, t, d, s) \rightarrow \langle \{[s]_d^i\}_{P_i \in C}, aux \rangle$ : A dealer  $D$  shares some secret  $s$  to a committee  $C$  with  $t$  corrupt nodes, such that  $d + 1$  shares will be needed to reconstruct  $s$ . Some auxiliary information  $aux$  may also be output and used to guarantee the success of Rec.
- $\text{Rec}(C, t, d, \{[s]_d^i\}_{P_i \in C}, aux) \rightarrow \langle s \rangle$ : Each party  $P_i \in C$  uses their share  $[s]_d^i$  (and possibly some auxiliary information  $aux$ ) to publicly reconstruct  $s$ .

Secret Sharing protocols are often defined in terms of the properties they achieve. In this work, we describe properties for our schemes to achieve along with an ideal functionality which realizes them. For a protocol with a Share and Rec interface to provide ACSS, it must have the following properties

- **Correctness:** If  $D$  is correct, then Share will result in correct parties eventually outputting  $[s]_d^i$ . Once Share is complete, if all honest parties perform Rec, they will output  $s$  as long as at most  $t$  players are corrupt.
- **Secrecy:** If  $D$  is correct, then for any non-uniform PPT adversary  $\mathcal{A}$  controlling up to  $t$  members of  $C$ , there

exists a PPT simulator  $\mathcal{S}$  such that the output of  $\mathcal{S}$  and  $\mathcal{A}$ 's view in the real-world protocol are computationally indistinguishable.

- **Agreement:** If *any* correct party outputs in Share, then there exists a canonical secret  $\tilde{s}$  such that each correct party  $P_i$  eventually outputs  $\langle [\tilde{s}]_d^i, aux \rangle$  and  $\tilde{s}$  is guaranteed to be correctly reconstructed in Rec. Moreover, if  $D$  is honest,  $\tilde{s} = s$ .

A *High-Threshold* ACSS scheme additionally has the following property:

- **High Threshold:** The privacy threshold  $d$  can be different from the correctness threshold  $t$ . Specifically,  $d$  can be between  $t$  and  $|C| - t - 1$ .

## 2.2 Dynamic-committee Proactive Secret Sharing

We next describe a protocol to transfer an already-shared secret from one committee to another. Previous work originally defined *Proactive Secret Sharing* as a mechanism by which a committee holding shares of some secret  $s$  could *refresh* the shares, i.e. generate a new set of random shares that reconstruct to the same secret. This was done to defend against a mobile adversary who could eventually compromise all nodes, but never more than a fixed percentage at a time. Later work added a *dynamic-committee* property in which the committee holding the new set of shares could contain a different set of nodes than the old committee, optionally with some overlap.

We define Dynamic-Committee Proactive Secret Sharing (DPSS) as an ACSS protocol with an additional Reshare function:

- $\text{Reshare}(C, C', t, t', d, d', \{[s]_d^i\}_{P_i \in C}, aux) \rightarrow \langle \{[s]_d^i\}_{P_i \in C'}, aux' \rangle$ : The old committee  $C$  creates a new sharing  $d'$ -sharing of  $s$  for the new committee  $C'$

This Reshare function should have the following properties:

- **Correctness:**  $C'$  will receive a sharing  $[s']_{d'}$  such that invoking Rec will reveal that  $s' = s$ .
- **Secrecy:** For every non-uniform PPT adversary  $\mathcal{A}$  controlling  $t$  members of  $C$  and  $t'$  members of  $C'$ , there exists a PPT simulator  $\mathcal{S}$  such that the output of  $\mathcal{S}$  and  $\mathcal{A}$ 's view in the real-world protocol are computationally indistinguishable.
- **Liveness:** If a byzantine PPT adversary  $\mathcal{A}$  controls up to  $t$  parties in  $C$  and  $t'$  parties in  $C'$ , and additionally controls all message ordering,  $\mathcal{A}$  can not prevent Reshare from completing.

A DPSS scheme can additionally be *resizable*:

- **Resizability:**  $|C|$  and  $|C'|$  can be different as long as  $t' < |C'|/3$  and  $d' = t'$  in the normal setting or  $t' \leq d' \leq |C'| - t' - 1$  in the High Threshold setting.

We additionally define a functionality  $\mathcal{F}_{\text{DPSS}}$  in Appendix B which realizes these properties and which we use to prove the secrecy of our scheme. As it is often useful for different applications, our  $\mathcal{F}_{\text{DPSS}}$  provides an interface by which to homomorphically combine shares from different Share instances (as an arbitrary linear combination) and either reshare or reconstruct the result. We will elaborate more in Section 3.4.

## 2.3 Multi-valued Validated Byzantine Agreement

Multi-valued validated Byzantine agreement (MVBA) [14] is a Byzantine fault-tolerant agreement protocol where a set of protocol nodes each with an input value can agree on the same value satisfying a predefined external predicate  $f(v) : \{0, 1\}^{|\mathcal{V}|} \rightarrow \{0, 1\}$  globally known to all the nodes. An MVBA protocol with predicate  $f(\cdot)$  should provide the following guarantees except for negligible probability.

- **Agreement:** All honest nodes output the same value.
- **External Validity:** If an honest node outputs  $v$ , then  $f(v) = 1$ .
- **Termination:** If all honest nodes input a value satisfying the predicate, all honest nodes eventually output.

Our protocol uses an MVBA with slightly strong validity requirement, where the predicate  $f(v, e)$  additionally can have some variable  $e$  depending on the execution state of the node as the input. We will explain more details in Section 3.2.

## 2.4 Reliable Broadcast

A protocol for a set of nodes where a designated broadcaster holds an input  $M$ , is a reliable broadcast protocol, if the following properties hold:

- **Agreement:** If an honest node outputs a message  $M'$  and another honest node outputs  $M''$ , then  $M' = M''$ .
- **Validity:** If the broadcaster is honest, all honest nodes eventually output the message  $M$ .
- **Totality:** If an honest node outputs a message, then every honest node eventually outputs a message.

Our High-threshold ACSS protocol of Section 3.1 will use the reliable broadcast protocol of Das et al. [17], which only assumes collision-resistant hash functions of output size  $\kappa$  and has a communication complexity  $O(n|M| + \kappa n^2)$  for broadcasting a message  $M$



### 3 High-Threshold Share Transfer

We first introduce a DPSS protocol which functions with high-threshold shares, meaning it can support privacy thresholds between  $t + 1$  and  $n - t$ . To construct it, we need both a high-threshold ACSS protocol and a Multi-valued Validated Byzantine Agreement (MVBA) protocol.

#### 3.1 High-Threshold ACSS

The recent work of Das et al [18] introduced a high-threshold ACSS scheme with a total network bandwidth of  $O(n^2)$ . To summarize, for each share  $[s]$ , the dealer broadcasts a discrete log commitment  $g^{[s]}$ , a Paillier encryption  $\text{Enc}([s])$  under the intended receiver's public key, and a Zero Knowledge Proof of Knowledge (ZKPoK) of a value which is both the discrete log of the commitment and the result of decrypting the ciphertext. Receivers then check if every proof is valid and that the discrete log commitments correspond to a degree  $\leq d$  polynomial. If so, they can decrypt their share and output.

This protocol, while simple, has very desirable completeness and bandwidth-overhead properties. However, it assumes that the secret  $s$  is uniformly random (otherwise,  $g^s$  would reveal information about  $s$ ) and therefore is somewhat limited in its uses. We propose a modified version which can be thought of as the "Pedersen" version of this scheme: Essentially, we add a second blinding value  $\hat{s}$  for the dealer to share alongside  $s$ , replace  $g^{[s]}$  with  $g^{[s]}h^{[\hat{s}]}$  and create a new ZKPoK to relate this value to the Paillier-encrypted shares. We present our modified protocol in Algorithms 1 and 2.

#### 3.2 MVBA

Since first proposed by Cachin et al. [14], several recent improvements have been made for MVBA [4, 25]. The state-of-the-art MVBA protocol is sMVBA [25], which has  $O(\kappa n^2)$  bit complexity and 12 asynchronous rounds as the expected worst-case round complexity. As mentioned in section 2.3, our protocols uses MVBA with slightly strengthened validity requirement, defined by the state-aware predicate below.

**Definition 1** (State-aware Predicate). *A state-aware predicate function is  $f(v, e) : \{0, 1\}^{|v|} \times \{0, 1\}^{|e|} \rightarrow \{0, 1\}$  where  $v$  is the input value and  $e$  is some variable dependent on the execution state, satisfying that once  $f(v, e) = 1$  for some execution state at a node, it remains 1 for any future execution state.*

Compared to the standard MVBA definition, it uses a state-aware predicate that can also input some execution state dependent variables. We will first explain the predicate used in our protocol, and then show how to use existing MVBA protocols for our purpose. Finally, we will discuss setup assumptions and efficiency aspects of MVBA.

In our protocols, each node  $i$  locally maintains a set  $T_i$  to record the indexes of terminated ACSS instances, i.e.,

---

#### Algorithm 1 High Threshold ACSS Share

---

Public Inputs:  $g, h, C, d, \{PK_i\}_{P_i \in C}$

Private Inputs:  $D$  holds a secret  $s$

Public Outputs:  $\{g^{[s]_d} h^{[\hat{s}]_d}\}_{i \in [n]}$

Private Outputs:  $P_i$  holds  $[s]_d^i, [\hat{s}]_d^i$

---

Share (as  $D$ ):

101: Sample two random degree  $d$  polynomials,  $\phi(x), \hat{\phi}(x)$  and set  $\phi(0) = s$

102: **for**  $i \in [n]$  **do**

103:  $v_i \leftarrow \text{Enc}_{PK_i}(\phi(i)), \hat{v}_i \leftarrow \text{Enc}_{PK_i}(\hat{\phi}(i)), c_i \leftarrow g^{\phi(i)} h^{\hat{\phi}(i)}$

104:  $\pi_i \leftarrow \text{ZK}\{(\phi(i), \hat{\phi}(i)) : v_i = \text{Enc}_{PK_i}(\phi(i)) \wedge \hat{v}_i = \text{Enc}_{PK_i}(\hat{\phi}(i)) \wedge c_i = g^{\phi(i)} h^{\hat{\phi}(i)}\}$

105: **ReliableBroadcast**( $\{v_i, \hat{v}_i, c_i, \pi_i\}_{i \in [n]}$ )

---

Share (as  $P_i$ ):

201: **upon** receiving  $\{v_i, \hat{v}_i, c_i, \pi_i\}_{i \in [n]}$  from **ReliableBroadcast** **do**

202: **if** **DegreeCheck**( $\{c_i\}_{i \in [n]}$ )  $\neq 1$  **then**

203: **Abort**

204: **for**  $i \in [n]$  **do**

205: **if** **Verify**( $v_i, \hat{v}_i, c_i, PK_i, \pi_i$ )  $\neq 1$  **then**

206: **Abort**

207:  $[s]_d^i \leftarrow \text{Decrypt}_{SK_i}(v_i), [\hat{s}]_d^i \leftarrow \text{Decrypt}_{SK_i}(\hat{v}_i),$

208: **Output**  $[s]_d^i, [\hat{s}]_d^i, \{g^{[s]_d^i} h^{[\hat{s}]_d^i}\}_{i \in [n]}$

---

$T_i \leftarrow T_i \cup \{j\}$  whenever  $j$ -th ACSS with valid commitment outputs. When  $d' + 1$  ACSS terminates, node  $i$  inputs the above set to the MVBA, with the state-aware predicate function that also includes  $T_i$  as the input. As shown in Algorithm 3, for any other node  $j$ 's input  $T_j'$ , the predicate  $f(T_j', T_i)$  immediately returns 0 if  $|T_j'| \neq d' + 1$ , and returns 1 once  $T_j' \subseteq T_i$ , meaning that the terminated  $d' + 1$  ACSS instances proposed by node  $j$  are also terminated at node  $i$ . Hence, the predicate may not return immediately. Instead, when the set of ACSS instances are not yet all terminated at  $i$ , node  $i$  will hold the predicate check and re-evaluate whenever its  $T_i$  grows. Once the condition is satisfied, the predicate returns 1. Note that it is possible that the predicate never returns for a value from Byzantine node, by proposing ACSS instances that are never terminated; but for any honest nodes  $i$  and  $j$ , due to the agreement property of ACSS, eventually  $T_j' \subseteq T_i$  and thus  $f(T_j', T_i) = 1$ .

Our protocols can directly use existing MVBA protocols in a black-box manner, by plugging in the state-aware predicate as defined in Algorithm 3. The obtained MVBA satisfies the validity property that if an honest node output  $v$  at time  $T$ , then  $f(v, e) = 1$  for at least one honest node at time  $T$ . Now we briefly argue why the agreement, termination and validity properties of MVBA holds. The validity property holds due to the external validity of the underlying MVBA. For agreement, the strengthening of the validity predicate has no effect on the safety argument. For termination, note that the predicates at all honest nodes eventually return 1 for any input from honest nodes. For an input from a Byzantine node, the predicate

---

**Algorithm 2** High Threshold ACSS Reconstruct

---

Public Inputs:  $g, h, C, d, \{g^{[s]_d^i} h^{[\hat{s}]_d^i}\}_{i \in [n]}$ Private Inputs:  $P_i$  holds  $[s]_d^i, [\hat{s}]_d^i$ Public Outputs:  $s$ 

---

Reconstruct (as  $P_i$ ):

```
301: Multicast  $[s]_d^i, [\hat{s}]_d^i$  to all parties
302: upon Receiving  $m_j, \hat{m}_j$  from  $P_j$  do
303:   if  $g^{m_j} h^{\hat{m}_j} = g^{[s]_d^i} h^{[\hat{s}]_d^i}$  then
304:     Set  $[s]_d^j = m_j$ 
305: upon Receiving  $d+1$  valid shares do
306:   Interpolate and output  $s$ 
```

---

may not return, and it is equivalent as the Byzantine node never inputs to MVBA, so the termination is also preserved.

The state-of-the-art MVBA protocol, sMVBA [25], (along with many other MVBA protocols) requires a high-threshold non-interactive threshold signature setup to reduce communication and perform leader election [15]. To setup these threshold signatures, we can either assume a trusted dealer that equips all the committees with such setup, or use existing *asynchronous distributed key generation (ADKG)* protocols [3, 17, 18, 21, 29] to lift the trusted dealer assumption. The (special-purpose) ADKG protocols of Gao et al. [21] and Das et al. [17] achieves  $O(\kappa n^3)$  cost and  $O(1)$  expected worst-case asynchronous rounds, but generates a secret key that is a group element rather than a field element. To be compatible with existing threshold signature schemes [11], the (general-purpose) ADKG protocol of Das et al. [18] generates a field element as the secret key, at the same cost of  $O(\kappa n^3)$  but  $O(\log n)$  expected worst-case rounds (and  $O(1)$  expected rounds in common-case when there are no faults and network is synchronous). Theoretically, it is possible to obtain a worst-case expected constant-round general-purpose ADKG protocol, by replacing the  $n$  instances of parallel ABA's of Das et al. [18] with one instance of MVBA (which has constant rounds), and bootstrapping its shared randomness using special-purpose ADKG protocols such as [17, 21]. Then, the total network cost will remain cubic and the latency can be reduced to constant. However, such a construction may not be concretely efficient, and in the common-case when there are no faults and the network is synchronous, may perform worse than Das et al. [18].

### 3.3 High-Threshold Share Transfer

We present our High-Threshold DPSS protocol in Algorithm 3. Relative to previous works it is the first to achieve optimal fault tolerance in asynchrony with a polynomial bit complexity, and does so without the need for trusted setup (which needed for the KZG polynomial commitments [27] used in many other DPSS works). We additionally note that this protocol is a more general version of DKG transfer: if  $\hat{s} = 0$ , then this reduces to a scheme with discrete-log

---

**Algorithm 3** High-Threshold Asynchronous DPSS

---

Private Inputs:  $P_i$  holds  $[s]_d^i, [\hat{s}]_d^i$ Public Inputs:  $\{g^{[s]_d^i} h^{[\hat{s}]_d^i}\}_{i \in [n]}$ Private Outputs:  $P_i'$  holds  $[s]_{d'}^i, [\hat{s}]_{d'}^i$ Public Outputs:  $\{g^{[s]_{d'}^i} h^{[\hat{s}]_{d'}^i}\}_{i \in [n']}$ 

---

OLD COMMITTEE PORTION (as  $P_i$ ):

```
101: Sample two degree- $d'$  polynomials  $\{\chi_i(x), \hat{\chi}_i(x)\}$  s.t.
     $\chi_i(0) = [s]_{d'}^i, \hat{\chi}_i(0) = [\hat{s}]_{d'}^i$ 
102: Use ACSS to share these polynomials with the new committee
```

---

NEW COMMITTEE PORTION (as  $P_i'$ ):

```
201:  $T_i \leftarrow \{\}$ 
202: upon outputting in  $j$ -th ACSS sessions where
     $g^{\chi_j(0)} h^{\hat{\chi}_j(0)} = g^{[s]_{d'}^i} h^{[\hat{s}]_{d'}^i}$  do
203:    $T_i \leftarrow T_i \cup \{j\}$ 
204:   if  $|T_i| = d' + 1$  then
205:      $T_i' \leftarrow T_i$ 
206:     Invoke MVBA( $T_i'$ ) with predicate  $f(T_j', T_i) // T_j'$  is the
    input value of some node  $j$ ,  $T_i$  is  $i$ 's local variable defined above.
     $f(T_j', T_i)$  is defined below.
207: upon MVBA outputting  $T$  do
    Let  $B(x, y)$  be a degree  $d, d'$  bivariate where  $B(0, 0) = s$  and
     $B(j, y) = \chi_j(y)$  for  $\forall j \in T$ 
    Let  $\hat{B}(x, y)$  be a degree  $d, d'$  bivariate where  $\hat{B}(0, 0) = \hat{s}$  and
     $\hat{B}(j, y) = \hat{\chi}_j(y)$  for  $\forall j \in T$ 
208:   Interpolate  $[s]_{d'}^i = B(0, i), [s]_{d'}^j = \hat{B}(0, i)$  from the shares in
    the subset
209:   Similarly, interpolate  $\{g^{[s]_{d'}^j} h^{[\hat{s}]_{d'}^j}\}_{j \in [n]}$ 
210:   Output Private:  $\{[s]_{d'}^i, [\hat{s}]_{d'}^i\}$ , Public:  $\{g^{[s]_{d'}^j} h^{[\hat{s}]_{d'}^j}\}_{j \in [n]}$ 
```

---

Predicate  $f(T_j', T_i)$  for MVBA (as  $P_i'$ ):

```
301: if  $|T_j'| \neq d' + 1$  then
302:   return 0
303: upon  $T_j' \subseteq T_i$  do
304:   return 1
```

---

commitments which are used to facilitate threshold signing with signature schemes such as BLS [12].

We will now describe the operation of our protocol. The core mathematical component is that if some committee  $C$  holds shares of some degree  $d$  polynomial, they can create new shares for some new committee  $C'$  who wishes for shares of some degree  $d'$  polynomial by having  $d+1$  members of  $C$   $d'$ -share their shares with  $C'$ . We can then use each of these polynomials to define a degree  $d, d'$  bivariate polynomial  $B(x, y)$  where  $B(i, y)$  would be the polynomial which  $P_i \in C$  shared to  $C'$ . Note that relative to this bivariate, each party  $P_i \in C$  held  $B(i, 0)$ , meaning that  $B(0, 0) = s$ . As a result of these sharings  $P_j' \in C'$  receives  $\{B(i, j)\}_{i \in [C]}$ , from which she can derive  $B(0, j)$ , a point on a degree  $d'$  univariate polynomial which encodes the same secret  $s$ . The high level takeaway of this is that the new committee derives randomized shares of a specified degree as a linear combination of

$d+1$  instances of a member of  $C$  secret sharing their share.

A concise outline of the protocol strategy then is 1) Have all members of  $C$  secret share their shares to  $C'$ , 2) Have all members of  $C'$  agree on  $d+1$  such instances which succeeded, 3) Use the outputs of these instances to allow  $C'$  to derive new rerandomized shares.

A few questions still need to be answered to create a maliciously-secure protocol. By answering them one at a time, we arrive at a full derivation of Algorithm 3.

**How Do We Ensure All Parties in  $C'$  Get A Share?** If we used simple Shamir Sharing, we would have no guarantees about the correctness or eventual arrival of the shares that  $C'$  needs to receive from  $C$ . Previous works address this through the use of Verifiable Secret Sharing, in which an honest party outputting guarantees that there is a canonical polynomial of the correct degree which can be guaranteed to be reconstructed successfully if desired. However, this does not necessarily imply that all honest parties will receive a share and previous works which relied on this needed a more expensive fallback mechanism to recover missing shares [7, 38].

Instead, we sidestep this issue by using Asynchronous Complete Secret Sharing, in which an honest party outputting guarantees that all honest parties will do so successfully.

**How Does  $C$  Prove They Are Resharing The Correct Shares?** To prevent a malicious member of  $C$  from resharing anything besides their share, we utilize public commitments. Say that all honest members of  $C$  agree on some set of discrete log commitments  $\{g^{[s]^1}, g^{[s]^2}, \dots, g^{[s]^n}\}$  that correspond to each privately held share. Then we utilize an ACSS scheme, such as the one by Das et al [18], which includes a discrete log commitment to the secret being shared.  $C$  could transfer the set of commitments to  $C'$ , who could then use them to individually check that each member of  $C$  is sharing the correct value.

Das et al's ACSS scheme also includes discrete log commitments to each share that every other party receives. These commitment strings from each ACSS session can also be combined via the same linear operations that derive the new shares, resulting in each node being able to homomorphically calculate commitments to the new shares of all other nodes in  $C'$ . This thus completes the invariant of a committee knowing public commitments which correspond to its shares, which can be used to facilitate the next share transfer.

**How Do We Handle High Thresholds?** Most secret sharing schemes that assume  $t$  corrupt parties will also use degree  $t$  polynomials, such that  $t+1$  shares are needed to reconstruct the secret. This privacy threshold works nicely in the  $n=3t+1$  setting, as it means that during reconstruction a robust decoding algorithm such as Berlekamp-Welch or Gao's Algorithm [20] can be used to find and correct faulty shares without relying on cryptography.

However, for larger privacy thresholds, these robust decoding techniques no longer work and it becomes necessary to be able to detect faulty shares individually. The share

commitments discussed earlier are sufficient for this purpose and allow polynomials of degree up to  $n-t-1$  to be reconstructed successfully.

**How Do We Handle Non-Uniform Secrets?** The discrete log commitments discussed previously are only computationally hiding if the committed values are uniformly random. And while secret shares should in fact be uniformly random, the additive homomorphism property of discrete log commitments means that an attacker who can see  $d+1$  different share commitments can derive a commitment to the secret. If the secret is non-uniform (say, a single bit in the extreme case), an adversary can guess possible decommitments until she finds one which matches.

To avoid this, it is necessary to use a *perfectly hiding* commitment such as a Pedersen Commitment of the form  $g^s h^r$  where  $h$  is a second generator of the same cyclic group as  $g$ , but the relationship between  $g$  and  $h$  is unknown (which is necessary for the commitment to be computationally binding). Then, as long as  $r$  is uniformly random, the commitment protects the secrecy of a non-uniform  $s$ .

Lastly, we need these commitments to be openable even after being transferred and recalculated. To facilitate this, we replace  $r$  with a "blinding secret"  $\hat{s}$  shared on a polynomial of the same degree as  $s$ . For every operation performed with a share of  $s$ , a parallel one should take place with a share of  $\hat{s}$ , and any discrete log commitment  $g^{[s]^i}$  should be replaced with Pedersen commitment  $g^{[s]^i} h^{[\hat{s}]^i}$ . We modify the ACSS protocol from earlier as well as our DPSS to incorporate these changes.

**How Does  $C'$  Decide Which ACSS Instances To Use?** Before the honest nodes in the new committee  $C'$  can interpolate the new shares for the secret, the protocol needs to provide two guarantees: (i) honest nodes in  $C'$  agree on the same set of ACSS instances for interpolation, and (ii) the above set of ACSS will eventually terminate at all honest nodes, so that they can receive their shares for interpolation. For (i), our design uses a multi-valued Byzantine agreement (MVBA), where each node  $i$  can input the set  $T_i$  of finished ACSS instances to the MVBA, and MVBA ensures that all honest nodes will agree on the same set of ACSS. However, since malicious nodes can input any set of ACSS instances, including those that will never terminate, running MVBA naively does not guarantee (ii).

To ensure the agreement only on the set of ACSS that will eventually terminate at all honest nodes, in MVBA, honest nodes should only consider a set of ACSS instances valid if all the instances in the set have terminated locally. More specifically, we modify the validity predicate function of existing MVBA's to also take the node's local execution state (the set of finished ACSS instances) into account. Then, a node considers an input of ACSS set to be valid, only when all the instances in the set have terminated locally. Since the output of MVBA is valid to at least one honest node, meaning the set of ACSS have been terminated at that honest node, due

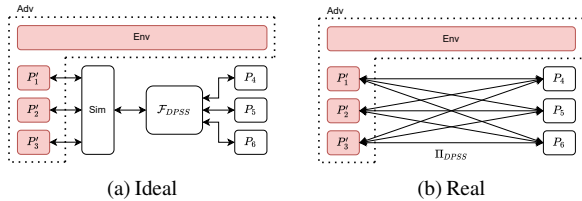


Figure 1: **UC Security** — The setup in ideal and real worlds. The adversarial entities are shaded in red. We omit the communications between the environment and the other entities to make the figure clean.

to the Agreement property of ACSS, the agreed set of ACSS instances will eventually terminate at all honest nodes as well.

### 3.4 Security Analysis

In this section, we show that the DPSS protocol in Algorithm 3 implements the  $\mathcal{F}_{DPSS}$  functionality (c.f. Appendix B), assuming the ACSS protocol is secret and the Pedersen commitments are hiding. Due to the page limit, we only present a high-level analysis. We do not explicitly model the party corruption process. We assume once the environment instructs the adversary to corrupt a party, the adversary learns the memory of the party and the party becomes a proxy of the adversary. Namely, the adversary sends and receives messages on behalf of the party. We also include a coordinator, which is an uncorruptible party for easier security analysis. The coordinator issues instructions for events, including sharing a secret, recovering a secret, and resharing the secret. We assume the coordinator notifies everyone on the beginning of an event. Namely, it listens from the environment and tells everyone if an event happened. The environment is free to choose any secret  $s$  and issue any number of Resharing events between the Sharing and Reconstruction events. For simplicity, we omit some interactions that can be inferred from the context, and we assume authenticated asynchronous channels between the entities.

In the UC model, we say a protocol  $\pi$  UC-realizes a functionality  $\mathcal{F}$  if and only if there exists a simulator such that, in the ideal and real worlds shown in Figure 1, the adversary cannot distinguish which world he/she is interacting with by sending and receiving messages.

**Theorem 1.** *Assuming the ACSS protocol is secret, the protocol in Algorithm 3 UC-realizes the functionality  $\mathcal{F}_{DPSS}$ .*

We illustrate the high-level proof idea in Figure 2. The simulator creates a simulated honest party in mind for each real honest party and lets the simulated honest parties play with the external corrupted parties by running an instance of the asynchronous DPSS protocol in Algorithm 3 with a fake secret  $s' = 0$ . Only at the time of reconstruction, the simulator secretly “overwrites” the memory of the simulated honest parties and lets them continue the reconstruction protocol. The

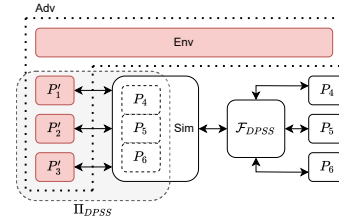


Figure 2: **Proof Idea** — Run  $\Pi_{DPSS}$  with fake secret  $s'$  when resharing; Switch true secret  $s$  back when reconstructing.

overwritten shares correspond to the real secret, which the simulator learns by: (1) hijacking the leaking message ( $\text{REC}, \phi, \hat{\phi}$ ) from the functionality  $\mathcal{F}_{DPSS}$  to the adversary when the dealer is honest, or (2) decrypting from the broadcast messages using the keys it simulated for honest parties when the dealer is corrupted. We include a more detailed proof in Appendix C.

### 3.5 Performance Analysis

The protocol presented in Algorithm 3 has communication complexity of  $O(n^3)$ , since each node invokes an ACSS instance of cost  $O(n^2)$  (thus  $O(n^3)$  in total), and participates in one MVBA instance which has cost  $O(n^2)$  and requires  $O(n^3)$  to generate the public DKG parameters without trusted setup. The overall round complexity is constant if a constant-round MVBA is used, which can itself be instantiated by a constant-round asynchronous DKG protocol. Alternatively, if  $n$  concurrent ABA protocols are used instead, then the best case round complexity is still  $O(1)$  but the worst case is reduced to  $O(\log n)$ .

The presence of byzantine behaviour does not meaningfully affect the performance of our protocol. If a byzantine ACSS dealer provides an invalid sharing or proof, their malfeasance is immediately identified by all honest parties and their messages are simply ignored. The MVBA is guaranteed to have enough valid ACSS instance inputs to form a subset of valid instances which fully define the shares that the new committee receives. Once the subset has been defined, honest parties are guaranteed to eventually receive all shares simply by waiting for them to arrive.

Computationally, each node in the protocol is expected to perform  $O(n^2)$  work due to the need to check  $n$  different correctness proofs in each of the  $n$  ACSS instances. While a lucky node may only need to check the proofs in  $d + 1$  instances, this does not change the overall asymptotic behaviour.

## 4 Batch-Amortized Share Transfer

For many applications, such as distributed Key Value stores or more generally Multiparty Computation, it would be beneficial to be able to transfer a large number of secrets from one committee to the next in a more bandwidth-efficient manner. However, the high-threshold DPSS scheme we introduced previously relies on the availability of Pedersen



commitments to every share generated in the share-resharing process which we use to realize DPSS. Unfortunately this reliance requires all  $n$  parties to receive  $n$  commitments from each of the  $n - 1$  other parties, imposing a cubic bandwidth overhead for the whole network.

To get around this, we switch away from using share-resharing to facilitate share transfer and instead look to a classic MPC technique for inspiration:

Given three independent secret sharings  $[s]$ ,  $[r]$ ,  $[r']$  where  $r = r'$  and  $r \leftarrow \mathbb{Z}_p^*$

$$\begin{aligned} [s+r] &= [s] + [r] \\ (s+r) &\leftarrow \text{Open}([s+r]) \\ [s'] &= (s+r) - [r'] \end{aligned}$$

Essentially, if we can create some paired sharing  $([r],[r'])$  such that the old committee holds  $r$  and the new committee holds  $r'$ , we can have the old committee reconstruct  $(s+r)$  and the new committee can use this information to derive new rerandomized shares of  $s$ .

A key challenge here is that  $r$  needs to be uniformly random and not known to any party. One solution is for each party to share their own locally sampled random value  $[r_i]$  and add together a set of such values to derive a globally random  $[r] = \sum_i [r_i]$ , where the set of  $[r_i]$  values to use is determined by MVBA. The issue with this approach is that it does not result in a bandwidth savings: A cubic bandwidth is required to use ACSS to secret share the  $O(n)$  local secrets that constitute  $r$ .

Instead, we leverage a classic randomness extraction technique using hyperinvertible matrices [8]. In short, by performing a series of local linear operations to a set of  $m$  locally-random shared secrets, we can extract  $m - t$  globally-random secrets. Thus if our starting subset of  $[r_i]$  values contained  $n - t$  entries, we could extract  $n - 2t$  globally-random outputs, a linear yield in the optimally byzantine fault-tolerant asynchronous protocol setting. We can also leverage a BatchReconstruction technique from the same work to efficiently open many  $(s+r)$  values at once with an amortized network overhead of  $O(n)$  per opening.

The last major obstacle to overcome is the following: *How do we create a shared random value which is held by both the old and new committees?* The recent work of [23] offers a solution, but it requires the use of a synchronous broadcast channel to publish shares and accuse faulty nodes. Instead we introduce a general technique to turn an ACSS protocol into what we call a *dual-committee ACSS*, the goal of which is to share a secret to two committees at once (one polynomial per committee) such that one honest player outputting implies that all honest parties in both committees will eventually receive shares that will reconstruct the same secret.

We present our dual-committee ACSS modification in Algorithm 4. We remark that the construction is very straightforward. Given an ACSS scheme which produces a commitment to the secret which can be verified to be

---

#### Algorithm 4 Dual-Committee ACSS Share

---

Let  $C^*$  denote the joint committee of both  $C$  and  $C'$

Public Inputs:  $C, C', d, d'$

Private Inputs:  $D$  holds a secret  $s$

Private Outputs:  $P_i$  holds  $[s]_d^i$ ,  $P'_j$  holds  $[s]_{d'}^j$

---

Share (as  $D$ ):

Select an ACSS scheme which produces a commitment  $com$  to the secret and proves that  $\text{Decommit}(com) = s$

101:  $\text{ACSS}(s) \rightarrow C, \text{ACSS}(s) \rightarrow C'$

---

Share (as either  $P_i$  or  $P'_j$ ):

201: **upon** outputting  $([s]_d^i, com, aux)$  in the local copy of ACSS **do**

202:     Multicast  $com$  to all parties in the other committee

203:     Store  $com$  locally

204:     **upon** Receiving  $com$  from the other committee  $t+1$  times **do**

205:         Output  $([s]_d^i, aux)$

---

correct, a Dealer executes two ACSS instances (one for each Committee) in which it shares the same secret. Upon terminating their local ACSS instance, a player in one committee sends the commitment to the secret  $com$  to every player in the other committee. Upon receiving  $t + 1$  copies of this same  $com$  from the other committee, we know that at least one must have come from an honest party, implying that all honest parties will eventually receive shares of the same secret per the Agreement property of ACSS. At this point an honest node can safely output their share and use it elsewhere.

With the requisite building blocks described, we now present our Batch-Amortized High-Threshold DPSS scheme in Algorithm 5. Given a batch size  $B$ , each member of  $C$  needs to share enough locally random values that there will be enough globally random shares to open each  $(s+r)$  and  $(\hat{s} + \hat{r})$ . After sharing their random values,  $C$  runs MVBA with a similar predicate to before in order to agree on  $n - t$  players from whom to use output.  $C$  can then calculate all of the  $(s+r)$  and  $(\hat{s} + \hat{r})$  openings and send them to  $C'$ .  $C'$ , by virtue of using the Dual Committee ACSS, does not need to perform its own agreement subprotocol, as the node ids agreed upon by  $C$  should all eventually deliver shares to  $C'$  which can be used to calculate the final output.

After amortization, Algorithm 5 would still require  $O(n)$  constant-sized share commitments to be known by everybody for each secret in the batch, making the amortized network cost  $O(n^2)$ .

#### Batch-Amortized Share Transfer With Linear Network Overhead

The previous DPSS protocols we presented require a  $O(n)$ -sized set of commitments to be public in order to function, resulting in an  $O(n^2)$  network bandwidth bottleneck per secret. While this appears to be necessary in order to facilitate a high-threshold share transfer, it is not necessary when dealing with secrets which are merely

---

**Algorithm 5** Batch-Amortized High-Threshold DPSS

---

Let  $B$  be the number of degree  $d$  (secret, blind) pairs  $([s], [\hat{s}])$  to be transferred

Private Inputs:  $P_i$  holds  $\{[s_j]_d^i, [\hat{s}_j]_d^i\}$  for  $j \in [B]$

Public Inputs:  $\{g^{[s_j]_d^k} h^{[\hat{s}_j]_d^k}\}$  for  $k \in [n']$  for  $j \in [B]$

Private Outputs:  $P'_i$  holds  $\{[s'_j]_{d'}^i, [\hat{s}'_j]_{d'}^i\}$  for  $j \in [B]$

Public Outputs:  $\{g^{[s'_j]_{d'}^k} h^{[\hat{s}'_j]_{d'}^k}\}$  for  $k \in [n']$  for  $j \in [B]$

---

OLD COMMITTEE PORTION (as  $P_i$ ):

101: Sample  $B/(n-t)$  random  $(r, \hat{r})$  pairs and use a Dual-Committee ACSS to share them with a degree  $d$  polynomial for  $C$  and degree  $d'$  polynomial for  $C'$ .

102: Use MVBA to agree on  $n-t$  players for whom all DC-ACSS instances terminated successfully.

103: Use a hyperinvertible matrix to extract  $B$  globally random sharings from the subset

104: Use BatchReconstruct to open  $\{(s_j + r_j), (\hat{s}_j + \hat{r}_j)\}$  for  $j \in [B]$  (shares can be individually validated by checking against  $g^{[s+r]} h^{[\hat{s}+\hat{r}]}$ ) and send these openings to  $C'$

---

NEW COMMITTEE PORTION (as  $P'_i$ ):

201: **upon** receiving  $\{(s_j + r_j), (\hat{s}_j + \hat{r}_j)\}$  for  $j \in [B]$  from  $C$  **do**

202:     **for**  $j \in [B]$  **do**

203:          $[s'_j]_{d'}^i = (s_j + r_j) - [r_j]_{d'}^i, [\hat{s}'_j]_{d'}^i = (\hat{s}_j + \hat{r}_j) - [\hat{r}_j]_{d'}^i$

204:         **for**  $k \in [n']$  **do**

205:              $g^{[s'_j]_{d'}^k} h^{[\hat{s}'_j]_{d'}^k} = g^{(s_j+r_j)} h^{(\hat{s}_j+\hat{r}_j)} / g^{[r_j]_{d'}^k} h^{[\hat{r}_j]_{d'}^k}$

206: Output  $\{[s'_j]_{d'}^i, [\hat{s}'_j]_{d'}^i\}$  for  $j \in [B]$ ,  $\{g^{[s'_j]_{d'}^k} h^{[\hat{s}'_j]_{d'}^k}\}$  for  $k \in [n']$  for  $j \in [B]$

---

$t$ -shared. In this section, we introduce a third DPSS protocol which is not high-threshold but which achieves an amortized linear network overhead per share.

To achieve this, we utilize the batch amortized hbACSS [42] secret sharing scheme which achieves a linear network overhead per secret when instantiated with the KZG [27] polynomial commitment scheme, which unfortunately comes with a trusted setup assumption (though we note that KZG is used in most recent DPSS schemes).

As this third and final scheme is no longer concerned with high-threshold secrets, it is no longer necessary for there to be public commitments relating to the values being transferred. This is because opening a  $t$ -shared value in the asynchronous  $n = 3t + 1$  setting is possible using a simple error correction algorithm such as Berlekamp-Welch or Gao's algorithm [20], rather than relying on the ability to validate shares individually. Consequently, relative to Algorithm 5, the main changes needed here are to switch the ACSS scheme to hbACSS, drop the usage of public share commitments, and to use a Structured Reference String (SRS) for KZG polynomial commitments.

hbACSS utilizes polynomial commitments in order to function. Given an appropriate polynomial commitment scheme, a dealer commits to their sharing polynomial, broadcasts this commitment, and then can send (via a

verifiable communication channel) a receiver their share along with a proof that the share is a point on the committed polynomial. In the case of a malicious dealer, share recovery is also handled in a batch-amortized way which does not result in any worsened asymptotics.

The KZG PolyCommit paper presents two schemes: PolyCommitDL and PolyCommitPed. In the former, a prover commits to a polynomial  $\phi(\cdot)$  by calculating  $g^{\phi(\alpha)}$ , which itself is calculated using a SRS of the form  $\{g, g^\alpha, g^{\alpha^2}, \dots, g^{\alpha^t}\}$  where  $\alpha$  is an unknown value generated during trusted setup. In PolyCommitPed, a second blinding polynomial  $\hat{\phi}(\cdot)$  is sampled and used to calculate the commitment  $g^{\phi(\alpha)} h^{\hat{\phi}(\alpha)}$ . In order to reshare secrets which are not uniformly random, we need to use PolyCommitPed, which allows us to verify the correctness of the commitment  $g^s h^{\hat{s}}$  as required by our Dual-Committee ACSS construction.

## 5 Applications

### 5.1 An Upgrade To Previous Applications

**Confidentiality in BFT State Machine Replication.** The recent work Vasantlal et al [38] introduced COBRA as a DPSS protocol to facilitate the storage of private information in State Machine Replication (SMR) systems. The core idea is that an application like a Key-Value store can be realized by a decentralized committee which collectively maintains a public state (say the Keys in a KV store) along side a per-node private state (the secret shares which can be combined to reconstruct the Value in a KV store). Protecting data confidentiality in a replicated system has been studied for decades, but most of the works only focus on static committees, such as DepSpace [10], Belisarius [35] and Basu et al. [7] building upon PBFT [16] under partial synchrony, and Secure Store [30], CODEX [34] building upon Byzantine Quorum Systems [32] under asynchrony.

For dynamic committees, the recent works of Goyal et al. [23] and Benhamouda et al. [9] design new synchronous DPSS schemes for storing secrets on blockchains, while COBRA builds upon HotStuff [41] under partial synchrony. CALYPSO [28] also proposes a verifiable data-management framework based on blockchain and threshold encryption, for a different use case where some authorized parties can access the secret data via an access-control blockchain.

Regardless of whether the application calls for a secret-shared threshold decryption key or for the private data itself to be secret-shared (so to possibly facilitate computations over the data), the usage of DPSS to either refresh or transfer the secret information remains the same. By improving upon DPSS itself, we therefore offer two mechanisms by which our work can help improve upon state of the art applications. The first is that our asynchronous protocols can offer better performance in less-than-optimal network conditions. While relying on a synchronous DPSS will weaken the properties

of systems built on practical partially-synchronous consensus such as PBFT [16], even state of the art partially synchronous DPSS protocols like COBRA suffer asymptotic performance hits (from  $O(n^3)$  to  $O(n^4)$ ) during period of asynchrony. Alternatively, using a similar-performing asynchronous DPSS protocol (like our  $O(n^3)$  DPSS scheme) can limit the damage done by slowness or network partitions, even if other parts of the system make stronger network assumptions.

Secondly, by offering a *High-Threshold* DPSS scheme, we can improve the privacy offered by distributed KV stores over prior solutions. By encoding secrets in high-degree polynomials, a passive adversary would need to corrupt over  $2/3$  of the network at once to compromise the privacy of the stored information. While an active attacker controlling the majority of the network could stop the protocol from operating (and fundamentally this is impossible to fix), any such interference could easily be detected and a new protocol instance could be started with new nodes.

**Extractable Witness Encryption.** Goyal et al [23] also utilize the combination of DPSS and State Machine Replication, but they use it to build a primitive which is functionally equivalent to *extractable witness encryption* [22]. Roughly speaking, a witness encryption scheme for an NP language  $L$  allows a user to encrypt a message with respect to a problem instance  $x$ . The decryptor is able to decrypt the message if  $x \in L$  and the decryptor knows a witness  $w$  that  $x \in L$ . For instance, the problem instance  $x$  can be any NP search problem and  $w$  can be any valid solution to the problem. If a witness encryption scheme is extractable, then any adversary that is able to distinguish two ciphertexts encrypted to the same  $x$  is also able to provide a witness  $w$  for  $x \in L$ .

Goyal et al [23] introduce the extractable Witness Encryption on Blockchain (eWEB), where any depositor that wants to deposit a secret with some releasing condition can distribute the encoded secrets among the miners via threshold secret sharing schemes. The set of miners will be constantly changing, thus a hand-off procedure using DPSS is periodically executed by the miners to ensure the secret is properly stored and can be released. Any requester with a valid witness to the release condition of the secret can learn the secret from the miners securely via reconstruction. Our DPSS protocols can further enhance the robustness the eWEB scheme, by tolerating arbitrary network delays and adversarial schedule of message delivery. Moreover, our high-threshold DPSS scheme can provide better privacy guarantees and achieve the same single-secret cost and amortized cost without trusted setup. On the other hand, our low-threshold DPSS scheme reduces the amortized cost by a factor of  $O(n)$  compared to Goyal et al [23] under the same setup assumption.

## 5.2 Transferable MPC Computations

**MPC-as-a-Service.** In an *MPC-as-a-Service* setting, a group of  $N$  servers evaluates an arithmetic circuit in order

to output some function of private user inputs. It is often useful to divide this service into two parts: an *offline phase* in which precomputation is performed continuously, and an *online phase* which utilizes this precomputation to evaluate a circuit upon receiving client inputs. Previous works such as HoneyBadgerMPC [31] utilized a *non-robust* offline phase in which precomputation attempts could fail but would be assumed to succeed eventually. Once successful, this precomputation could be used for a *robust* online phase, which is guaranteed to terminate successfully even in the presence of byzantine faults and asynchrony.

**Precomputation Examples.** Shared secrets which are encoded with Shamir’s Secret Sharing Scheme are additively homomorphic, meaning that a circuit program consisting only of additions of different secrets could be evaluated non-interactively once instantiated (but with an interactive step at the end to reconstruct the result). However, most non-trivial programs also require the ability to perform other functions such as multiplications or coin flips. These operations may require the use of data which may be precomputed, such as Beaver triples (a set of three  $t$ -shared values  $\{[a]_t, [b]_t, [ab]_t\}$ ) to facilitate multiplication, a shared random bit  $[b]_t, b \in \{0, 1\}$  to perform a coin toss, or a shared random input mask  $[r]_t$  to efficiently allow for inputs from untrusted clients. All of these  $t$ -shared elements can be computed more efficiently by forgoing robustness (instead, aborting on detecting a failure), but crucially, once they are calculated they can be spent, transferred, or refreshed robustly.

## 6 Evaluation

We implemented\* all of our asynchronous DPSS protocols and characterize their performance in this section.

### 6.1 Experimental Setup

Our implementations were done primarily in python (forking from the codebase of [18]), while core cryptographic operations rely on libraries written in rust. In particular, we used the ristretto group implementation of curve25519\_dalek [1] to instantiate our high-threshold DPSS protocols and we used ZCash’s bls12-381 library [24] as the backend for our  $t$ -threshold DPSS. This is because our  $t$ -threshold scheme requires the use of pairings to implement KZG polynomial commitments, while our high-threshold scheme uses different cryptography which does not require pairings.

All of our programs were evaluated on a consumer-grade laptop with an Intel i5-1135G7 processor and 64GB of RAM. All benchmarks are run on a single core and players are modeled as async tasks which send serialized messages to each other.

\*Repo available at <https://github.com/tyurek/dpss>

$n$	Low Threshold	High Threshold
	No Crashes / $t$ Crashes	No Crashes / $t$ Crashes
4	13.28 / 9.31	3172.75 / 2456.61
10	24.38 / 21.25	7687.48 / 5703.75
19	37.91 / 36.26	14557.51 / 10366.71
31	61.56 / 58.83	-

Table 2: Computation time (in seconds) required for a member of a committee to receive 1000 shares from the previous committee and then transfer 1000 shares to a new committee

## 6.2 Network Considerations

Although we do not evaluate our protocols on a geo-distributed network, we argue that the primary bottleneck in evaluations should be computational, rather than related to bandwidth and network latency. We first observe that the round complexity of our protocols is not affected by the number of shares being transferred, so in the case where a sizable batch of shares are used, the throughput lost to round-trip times is vanishing. This is especially true in the case where a constant-round MVBA is used to achieve an overall constant round complexity (notably, our prototype implementation uses  $N$  concurrent ABA instances instead, which, while constant-rounded in the absence of byzantine faults, leads to a worst case  $O(\log n)$  round complexity).

We next observe that the amount of bandwidth required per secret transferred is quite low: For the  $t$ -threshold DPSS, each party needs to receive two 32-byte field elements (a share and a blinding share) and two 48-byte bls12-381 G1 elements (a KZG polycommit and witness). The distribution of these values via a batch-amortized Asynchronous Verifiable Information Dispersal algorithm adds a constant factor of roughly 6x, while the costs of randomness extraction impart another 3x overhead factor. Using speedtest.net’s global median upload speed for April 2022 of 27.06 Mbps [2], this would imply a throughput of over 1200 shares per second (or roughly an order of magnitude faster than our fastest result) if computation were not an issue.

Notably, our high-threshold DPSS protocol has an amortized network bandwidth of  $O(n^2)$  and therefore may be more susceptible to bandwidth limitations. In our implementation we measured that two Paillier ciphertexts, a Pedersen commitment, and a proof about the correctness of the ciphertexts measured roughly 10KB. Each participant in the DPSS needs to process roughly  $3n$  of these tuples per share transferred and incur a roughly 3x overhead on top of this for the reliable broadcast mechanism. Even in this case however, the computational costs of the protocol dominate by a significant margin.

## 6.3 Experimental Results

Our primary results in Table 2 show the amortized amount of computation required for a node to receive a share from an old committee and then transfer it to a new committee

when all committees are of size  $n$ . We observe that while our high-threshold protocol comes with a meaningful performance penalty relative to our  $t$ -threshold protocol, it also enables a new class of applications and an increase in privacy that practitioners may find worthwhile.

We evaluate our protocols in both the fault-free setting and with  $t$  nodes crashing in each committee. As expected, the difference in performance is minimal, with the  $t$ -crash case actually performing slightly better due to nodes having to handle fewer messages and not participating in ACSS instances which do not end up in the final subset.

## 6.4 Discussion

**Relative Slowness of High-Threshold Scheme.** Upon implementing our high-threshold DPSS scheme, we discovered that the vast majority (above 80%) of the computation is spent performing the modular exponentiations needed to generate Paillier encryptions as well as prove and verify their correctness. We note that unlike many other applications which utilize Paillier encryption, we do not require the ciphertexts to be additively homomorphic, and that it may be more efficient to use a different cryptosystem when proving knowledge about the correspondence between plaintexts and committed values in Pedersen commitments. However, we are not aware of an instantiation of this proof in any other cryptosystem.

**Comparison With Other Works.** The recent work of COBRA evaluates their DPSS scheme on a local network of up to ten servers and benchmarks the refreshing of 100,000 shares in a time of 743.8 seconds for the ten server case, claiming a roughly 5x speedup over the prior state of the art MPSS [36]. Though this corresponds to a 3.28x greater throughput, We argue that this difference is explainable as an artifact of the experimental setup, as their benchmarks were run on servers as an 8-threaded program, while our implementation is single threaded. While in principle we could improve our benchmarks in a number of ways including using multiple cores, implementing persistent precomputation for multiexponentiations, and optimizing polynomial operations, this would yield misleading results: Both our scheme and COBRA (as well as several others [23, 33, 40]) utilize KZG polynomial commitments as a subcomponent, which often then becomes the primary computational bottleneck.

## 7 Conclusion

In this work, we designed and implemented three asynchronous Dynamic-committee Proactive Secret Sharing schemes, each of which achieved new asymptotic bounds while also incorporating useful new properties such as supporting high privacy thresholds. Moreover, we demonstrated that asynchronous and *robust* DPSS protocols can compete with prior work in good-case scenarios and outperform them in the presence of faults. Leveraging this, we recall



prior applications which used DPSS and show how they how they can be better equipped to handle more adversarial environments. We additionally propose the usage of batch-amortized DPSS in refreshing and transferring precomputed data for use in robust MPC applications. We hope that these advancements allow future practitioners to build awesome resilient applications for use on a decentralized internet.

## References

- [1] curve25519-dalek: A pure-rust implementation of group operations on ristretto and curve25519, 2021. <https://github.com/dalek-cryptography/curve25519-dalek>.
- [2] Speedtest Global Index. <https://www.speedtest.net/global-index>, April 2022.
- [3] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 363–373, 2021.
- [4] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.
- [5] Andreea B Alexandru, Erica Blum, Jonathan Katz, and Julian Loss. State machine replication under changing network conditions. *Cryptology ePrint Archive*, 2022.
- [6] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. Communication-optimal proactive secret sharing for dynamic groups. In *International Conference on Applied Cryptography and Network Security*, pages 23–41. Springer, 2015.
- [7] Soumya Basu, Alin Tomescu, Ittai Abraham, Dahlia Malkhi, Michael K Reiter, and Emin Gün Sirer. Efficient verifiable secret sharing with share recovery in bft protocols. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2387–2402, 2019.
- [8] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In *Theory of Cryptography Conference*, pages 213–230. Springer, 2008.
- [9] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In *Theory of Cryptography Conference*, pages 260–290. Springer, 2020.
- [10] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. Depspace: a byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 163–176, 2008.
- [11] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.
- [12] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.
- [13] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strohli. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 88–97, 2002.
- [14] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.
- [15] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.
- [16] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.
- [17] Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.
- [18] Sourav Das, Tom Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. *Cryptology ePrint Archive*, 2021.
- [19] Yvo Desmedt and Sushil Jajodia. Redistributing secret shares to new access structures and its applications. Technical report, Citeseer, 1997.
- [20] Shuhong Gao. A new algorithm for decoding reed-solomon codes. In *Communications, information and network security*, pages 55–68. Springer, 2003.

- [21] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. *arXiv preprint arXiv:2106.07831*, 2021.
- [22] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 467–476, 2013.
- [23] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. In *IACR International Conference on Public-Key Cryptography*, pages 252–282. Springer, 2022.
- [24] Jack Grigg and Sean Bowe. zkcrypto/pairing. <https://github.com/zkcrypto/pairing>.
- [25] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbos: Pushing asynchronous bft closer to practice. In *Network and Distributed Systems Security Symposium*, 2022.
- [26] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *annual international cryptology conference*, pages 339–352. Springer, 1995.
- [27] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *International conference on the theory and application of cryptology and information security*, pages 177–194. Springer, 2010.
- [28] Eleftherios Kokoris-Kogias, Enis Ceyhan Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Private data management for decentralized ledgers. In *Proc. VLDB Endow.*, page 586–599, 2020.
- [29] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1751–1767, 2020.
- [30] Subramanian Lakshmanan, Mustaque Ahamad, and H Venkateswaran. Responsive security for stored data. *IEEE Transactions on Parallel and Distributed systems*, 14(9):818–828, 2003.
- [31] Donghang Lu, Thomas Yurek, Samarath Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchmix: Practical asynchronous mpc and its application to anonymous communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 887–903, 2019.
- [32] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.
- [33] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. Churp: dynamic-committee proactive secret sharing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2369–2386, 2019.
- [34] Michael A Marsh and Fred B Schneider. Codex: A robust and secure secret distribution system. *IEEE Transactions on Dependable and secure Computing*, 1(1):34–47, 2004.
- [35] Ricardo Padilha and Fernando Pedone. Belisarius: Bft storage with confidentiality. In *2011 IEEE 10th International Symposium on Network Computing and Applications*, pages 9–16. IEEE, 2011.
- [36] David A Schultz, Barbara Liskov, and Moses Liskov. Mobile proactive secret sharing. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 458–458, 2008.
- [37] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
- [38] Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira, and Alysson Bessani. Cobra: Dynamic proactive secret sharing for confidential bft services. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1528–1528. IEEE Computer Society, 2022.
- [39] Theodore M Wong, Chenxi Wang, and Jeannette M Wing. Verifiable secret redistribution for archive systems. In *First International IEEE Security in Storage Workshop, 2002. Proceedings.*, pages 94–105. IEEE, 2002.
- [40] Yunzhou Yan, Yu Xia, and Srinivas Devadas. Shanrang: Fully asynchronous proactive secret sharing with dynamic committees. *Cryptology ePrint Archive*, 2022.
- [41] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.
- [42] Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. hbaccs: How to robustly share many secrets. *Cryptology ePrint Archive*, 2021.
- [43] Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. Apss: Proactive secret sharing in asynchronous systems. *ACM transactions on information and system security (TISSEC)*, 8(3):259–286, 2005.

## A Epoch Definition

As in MPSS [36], we define *epoch* to constrain the power of the mobile adversary in the proactive secret sharing scheme, in order to circumvent the impossibility result of Alexandru et al. [5] (which has a slightly different model where the adversary can corrupt more than  $t$  nodes in one epoch as defined below). Due to the nature of asynchrony, different nodes may enter and leave different epochs at different physical times. Below we provide the definition of epoch in MPSS [36] for completeness.

- **Local Epochs.** A node is in epoch  $e$  if it has secret shares or secret keys associated with epoch  $e$ . It leaves epoch  $e$  and enters epoch  $e + 1$  when it wipes out all such information from its memory.
- **System Epochs.** Let  $G_e$  denote the set of nodes responsible for the secret in epoch  $e$ . The system is in system epoch  $e$  from the moment the first honest node in  $G_e$  enters epoch  $e$  until the moment the last honest node in  $G_e$  leaves epoch  $e$ .
- **System Compliance Constraint.** The adversary is system compliant through epoch  $e$  if, for all  $i \leq e$ , it is able to corrupt no more than  $t$  nodes belonging to  $G_i$  while the system is in any system epoch up to  $i$ .

## B Functionality of DPSS

This section presents the DPSS functionality  $\mathcal{F}_{DPSS}$ . The coordinator is a honest party that issues signals for the Sharing, Reconstructing, and Resharing events.

### $\mathcal{F}_{DPSS}$

```
//ParamGen
Sample  $g, h \in \mathbb{G}$  and make  $(g, h)$  public.

//Share
On receiving (DEAL, sid,  $s, d$ ) from an honest dealer (only once per sid):
    Sample random degree- $d$  polynomials  $\phi(\cdot), \hat{\phi}(\cdot)$  and set  $\phi(0) = s$ 
On receiving (DEAL, sid,  $\phi(\cdot), \hat{\phi}(\cdot), d$ ) from a corrupt dealer (only once per sid):
    Abort if either polynomial has degree  $> d$  or  $d > N - t - 1$ .
In either case, compute:
    • shares =  $\{\phi(i), \hat{\phi}(i)\}_{i \in [0, N]}$ ,
    •  $c = \{g^{\phi(i)} h^{\hat{\phi}(i)}\}_{i \in [0, N]}$ ,
and store (shares, sid)  $\rightarrow$  storage. Eventually send (SHARE, sid,  $\phi(i), \hat{\phi}(i), c$ ) to each party  $P_i$ .

//Reconstruct
```

On receiving (REC, sid, coeffs) from the Coordinator where coeffs defines the coefficients to use in a linear combination of outputs of Share (only once per sid):

Interpolate degree  $d$  polynomials  $\phi(x), \hat{\phi}(x)$  from the specified linear combination of shares.

Send (REC, sid,  $\phi(x), \hat{\phi}(x)$ ) to every corrupted player.

Eventually send  $s$  to every honest player.

//Reshare

On receiving (Reshare, sid, coeffs,  $d', C'$ ) from the Coordinator (only once per sid):

Send (LEAK,  $C, C', d, d'$ ) to the adversary.

Interpolate degree  $d$  polynomials  $\phi(x), \hat{\phi}(x)$  linear combination of shares.

For each  $(\phi(i), \hat{\phi}(i))$  held by an honest party in  $C$ , sample degree  $d'$  polynomials  $\phi_i(\cdot), \hat{\phi}_i(\cdot)$  where  $\phi_i(0) = \phi(i), \hat{\phi}_i(0) = \hat{\phi}(i)$ .

Send (LEAK,  $\{(\phi_i(j), \hat{\phi}_i(j), c'_i)\}_{i \in C_{\text{Honest}}}$  to  $\mathcal{A}$ , where  $c'_i := \{g^{\phi_i(k)} h^{\hat{\phi}_i(k)}\}_{k \in [0, |C'|]}$   $j \in C'_{\text{Corrupted}}$ .

Allow  $\mathcal{A}$  to input degree  $d'$  polynomials  $\phi_i(\cdot), \hat{\phi}_i(\cdot)$  for any adversarial  $P_i$ , and verify  $\phi_i(0) = \phi(i), \hat{\phi}_i(0) = \hat{\phi}(i)$ .

Let  $\mathcal{A}$  choose a set  $S$  of  $d + 1$  polynomials  $\{\phi_i\}_{i \in S}$  to use to calculate the output. If  $\mathcal{A}$  does not specify, then eventually choose an arbitrary  $S$ .

Interpolate shares' =  $\{\phi'(i), \hat{\phi}'(i)\}_{i \in [0, |C'|]}$  from the specified set and calculate  $c' = \{g^{\phi'(i)} h^{\hat{\phi}'(i)}\}_{i \in [0, |C'|]}$ .

Eventually send (RESHARE, sid,  $\phi'(i), \hat{\phi}'(i), c'$ ) to each party  $P'_i \in C'$

## C Proof of Theorem 1

*Proof.* We construct the following simulator Sim with access to trapdoors of the Pedersen commitment scheme and the zero-knowledge proving scheme.

### Simulator Sim for DPSS

Initially, sample  $n$  private keys  $SK_1, SK_2, \dots, SK_n$ , and an honest dealer's public key. Let the fake secret  $s' = 0$ . Let the corrupted set  $C = \emptyset$ . Get the trapdoors of the commitment scheme and the zero-knowledge protocol per the crs model.

On Receiving (CORRUPT, sid,  $i$ ) from the adversary:

- mark the party  $P_i$  as corrupted and take over its memory and messaging channels.
- $C = C \cup \{i\}$ .

On Receiving  $(\{v_i, \hat{v}_i, c_i, \pi_i\})$  from ReliableBroadcast:

- Run Share (as  $P_i$ ) in Algorithm 3.

- Store  $[s]_d^i, [\hat{s}]_d^i$ .

On all the simulated honest parties finish running Share.

- Interpolate a degree  $d$  polynomial  $\phi$  from  $[s]_d^i$ , and a degree  $d$  polynomial  $\hat{\phi}$  from  $[\hat{s}]_d^i$ .
- store  $\phi$  and  $\hat{\phi}$ .
- Send  $(\text{DEAL}, \phi, \hat{\phi})$  to  $\mathcal{F}_{DPSS}$ .

On receiving  $(\text{SHARE}, x_i, \hat{x}_i, c, \text{sid})$  designated to  $P_i$  from  $\mathcal{F}_{DPSS}$  (once):

- compute  $\pi = \text{Sim}_{zk}(x_i, \hat{x}_i)$ ,
- and store  $(i, x_i, \hat{x}_i)$ .

After the store contains the shares of all the corrupt parties in  $C$ :

- If we previously already stored  $\phi, \phi'$ , namely, the dealer is honest, stop and do nothing.
- Otherwise, for every honest party  $P_i$ , the simulator samples uniformly  $x_i$  and  $\hat{x}_i$  s.t. all the  $x_i$  are on a polynomial  $\phi(x)$  where  $\phi(0) = s' = 0$ .
- Then, it computes for every party  $P_i$ ,  $v_i = \text{Enc}_{PK_i}(x_i), \hat{v}_i = \text{Enc}_{PK_i}(\hat{x}_i), c_i = g^{x_i} h^{\hat{x}_i}$ , and  $\pi_i \leftarrow \text{Sim}_{zk}(x_i, \hat{x}_i)$  with the zero-knowledge trapdoor.
- Call  $\text{ReliableBroadcast}(\{v_i, \hat{v}_i, c_i, \pi_i\})$  with honest parties simulated in mind. For any message sent to the corrupt party, forward it simply.

On receiving outputs from the reliable broadcast, follow the protocol  $\Pi_{ACSS}$ .

On receiving  $(\text{REC}, \tilde{\phi}(\cdot), \tilde{\hat{\phi}}(\cdot))$  from the functionality.

- Compute the real secret  $s = \tilde{\phi}(0)$ .
- Sample a random degree  $d$  polynomial  $\tilde{\phi}(\cdot)$  s.t. for all the corrupted party  $P_i$ ,  $\tilde{\phi}(i) = \phi(i)$ , and  $\tilde{\phi}(0) = s$ .
- Overwrite the simulated party  $P_i$ 's memory that  $[s]_d^i = \tilde{\phi}(i)$ .
- Use the trapdoor to compute  $[\hat{s}]_d^i$  such that the previous commitments are consistent with  $\tilde{\phi}(i)$ . Overwrite the simulated party  $P_i$ 's memory that  $[\hat{s}]_d^i = [\hat{s}]_d^i$ .
- Let the simulated honest parties follow the protocol in Algorithm 2. Forward the messages to and from the corrupt parties directly.

- Initiate the last message of  $\mathcal{F}_{DPSS}$ . Namely,  $\mathcal{F}_{DPSS}$  sends  $s$  to every honest player.

On receiving  $(\text{LEAK}, C, C', d, d')$  from the functionality:

- Store  $C, C', d, d'$ .

On receiving messages from the malicious party:

- Forward it to the simulated honest parties.

On receiving  $(\text{LEAK}, \phi_i(j), \hat{\phi}_i(j), c'_i)_{i \in C \setminus C, j \in C \cap C}$  from  $\mathcal{F}_{DPSS}$ :

- Drop the message.
- Let every simulated honest party follow the protocol to produce  $\phi_i$  and  $\hat{\phi}_i$ .
- Run ACSS between the corrupted parties and the simulated honest parties.
- For every simulated honest party  $P_i$ , initialize  $T_i = \{\}$ .

On  $j$ -th ACSS outputs and the commitment check passes:

- let  $T_i \leftarrow T_i \cup \{j\}$  for all simulated honest party  $P_i$ .
- If  $|T_i| = d' + 1$ , let  $T'_i \leftarrow T_i$ . Let the simulated party  $P_i$  invoke  $MVBA(T'_i)$ .

On MVBA outputting  $T$ :

- Input the polynomials  $\{\phi_i\}_{i \in T}$  to  $\mathcal{F}_{DPSS}$ .
- The simulated honest parties follow the protocol.

On receiving messages  $(\text{RESHARE}, \cdot)$  from  $\mathcal{F}_{DPSS}$  to corrupted parties in the new committee, simply drop them.

It suffices to argue that the simulator makes the ideal world's communication indistinguishable from the real world's communication.

During sharing, if the dealer is honest, the simulator will receive the shares for the corrupted parties from the functionality. In this case, it samples polynomials for the fake secret  $s'$  and produces the transcripts of the ciphertexts  $v_i, \hat{v}_i$ , commitments, and zero-knowledge proofs for the simulated honest parties. Then, it sends these values to the corrupted parties. Because (a) the values are encrypted, (b) the commitments are hiding, and (c) the zero-knowledge proofs are private, the corrupted parties cannot tell whether the underlying secret is  $s$  or  $s'$ . If the dealer is corrupted, the simulator will receive messages of the reliable broadcast from



the corrupted parties, which might not follow the protocol at all. The simulator lets the simulated honest parties run the Share function (as  $P_i$ ) in Algorithm 1 and store the shares  $[s]_d^i, [\hat{s}]_d^i$ . After all the simulated parties get their shares, the simulator interpolates polynomials  $\phi$  and  $\hat{\phi}$  from the shares, and sends a DEAL message to the functionality  $\mathcal{F}_{DPSS}$ .

During the reconstruction event, the simulator would receive a message from  $\mathcal{F}_{DPSS}$  which contains the real polynomials  $\phi$  and  $\hat{\phi}$ . It then samples a new polynomial  $\tilde{\phi}$  which has the same zero-point, (namely,  $\phi(0) = \tilde{\phi}(0)$ ), and makes the corrupted parties' shares consistent, (namely,  $\phi(i) = \tilde{\phi}(i)$  for corrupted parties  $P_i$ ). Note that if the dealer is corrupted, then the polynomial  $\tilde{\phi}$  already satisfies the requirements. So the simulator can let  $\tilde{\phi} = \phi$  in this case. It seems that the simulator only needs to change the memory of the simulated honest parties to get everything on the right track. However, there is one problem — the adversary will realize that the commitments do not open successfully because the simulated honest parties commitments are based on their original shares  $\phi(i)$ , which might correspond to a fake secret  $s'$ . To fix this, the simulator needs to use the commitment trapdoor to compute  $[\hat{s}]_d^i$  s.t.  $c_i = g^{\phi(i)} h^{\hat{\phi}(i)} = g^{\tilde{\phi}(i)} h^{[\hat{s}]_d^i}$  for the simulated honest parties and interpolate  $\hat{\tilde{\phi}}$  from  $[\hat{s}]_d^i$ . The simulator overwrites the memory of the simulated honest parties to let their share  $[s]_d^i \leftarrow \tilde{\phi}(i)$  and the corresponding hiding share  $[\hat{s}]_d^i \leftarrow \hat{\tilde{\phi}}(i)$ . Finally, the simulator lets the simulated honest parties to continue the reconstruction protocol in Algorithm 2. Note that if the dealer is corrupted, the polynomials  $\tilde{\phi}, \hat{\tilde{\phi}}$  equal the ground truth polynomials  $\phi, \hat{\phi}$ . If the dealer is honest, the polynomials  $\tilde{\phi}, \hat{\tilde{\phi}}$  compose a valid set of polynomials which could result from the randomness of the honest dealer in another universe with the same likelihood. Therefore, the adversary cannot distinguish the simulated world from the real world in both cases.

Now, it suffices to show that during every resharing, the messages look indistinguishable for different underlying secrets. When the old committee sends shares to the new committee, each of the party  $P_i$  performs an ACSS instance. By its secrecy property, the ACSS does not reveal anything computationally about the the honest parties' local shares as long as the adversary controls up to  $t$  parties in the new committee  $C'$ . For the new committee portion, the MVBA instance only deals with the set of indices. In other words, the messages are independent from the underlying secret. So, if the adversary learns anything about the underlying secret, it must come from the commitments. However, the Pedersen commitments are hiding. By a simple induction, we can conclude that after a polynomial number of resharing, both worlds are computationally indistinguishable. This concludes our proof.  $\square$