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

Thomas Yurek[1,4,5] *Zhuolun Xiang*[2] *Yu Xia*[3,4] *Andrew Miller*[1,5]
[1]*University of Illinois at Urbana-Champaign,* [2]*Aptos,* [3]*MIT CSAIL,* [4]*NTT Research* [5]*IC3*
yurek2@illinois.edu, xiangzhuolun@gmail.com, 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 [46] 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 [24,42,45,52] 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 [45,52] have been designed until very recently [47,49]. However, these protocols either incur a high communication cost ($O(n^4)$ to reshare a secret), or lose liveness under asynchrony [47], or compromise for non-optimal fault tolerance [49]. 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 achieves 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 survey and discuss numerous applications of our DPSS schemes, such as in confidential blockchains and MPC. We additionally introduce a new application, "BMR escape hatches", in which DPSS-persisted precomputation can be leveraged to allow for the speedy execution of MPC programs (in our example, an MPC Automated Market Maker) in periods of high network activity.

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. [34], 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. [24] 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. [48] proposed a verifiable DPSS solution extending the work of Desmedt et al. [24]. 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. [8] provides statistical (rather than cryptographic) security but with a non-optimal resilience threshold $t < (1/2-\varepsilon)n$. They can achieve $O(n^3)$ communication 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 [42]). Additionally, the secrets in this scheme are all packed into the same polynomial and can not be used independently.

CHURP [42] 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. [31] 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 [11] for the batch setting. Benhamouda et al. [12] also designed a DPSS scheme with a guarantee called player replaceability that ensures the committee is anonymous until it performs any action. As a result, the DPSS protocol can be run in small committees and has a communication cost that is polynomial in the security parameter and independent of the total number of nodes. They consider a fully mobile adversary and thus the solution tolerates only $(1-\sqrt{0.5})n$ corruptions.

Partially synchronous DPSS schemes. The only partially synchronous DPSS schemes we are aware of are Schultz-MPSS [45] and the very recent work COBRA [47]. Schultz-MPSS [45] follows the primary-based approach where every iteration a primary node will determine a proposal containing the blinding polynomials (Herzberg et al. [34]). Practical Byzantine Fault Tolerance (PBFT) [19] 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 [47] 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. [16] 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 Validated Byzantine Agreement (VBA). Their scheme inspires our first DPSS construction. Zhou et al. [52] proposed the first dynamic-committee 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 [49]. 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 [35] polynomial commitments.

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. [34]	Sync	$n/2$	✗	✗	$O(n^3)$	$O(n^3)$	$O(n^3)$	—	✓
Desmedt et al. [24] [1]	Sync	$n/2$	✓	✗	$O(n^2)$	—	—	—	✓
Wong et al. [48]	Sync	$n/2$	✓	✗	$exp(n)$	$exp(n)$	$exp(n)$	—	✓
Baron et al. [8]	Sync	$(1/2-\varepsilon)n$	✓	✗	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(1)$ [2]	✓
CHURP [42]	Sync	$n/2$	✓	✓ [3]	$O(n^2)$	$O(n^3)$	$O(n^3)$	—	✗
Benhamouda [5]	Sync	$(1-\sqrt{0.5})n$	✓	✗	$poly(\kappa)$	$poly(\kappa)$	$poly(\kappa)$	—	✓
Goyal et al. [31]	Sync	$n/2$	✓	✗	$O(n^2)$	$O(n^3)$	$O(n^3)$	$O(n^2)$	✗
Schultz-MPSS [45]	P. Sync [4]	$n/3$	✓	✗	$O(n^4)$	$O(n^4)$	$O(n^4)$	—	✓
COBRA [47]	P. Sync	$n/3$	✓	✗	$O(n^3)$	$O(n^4)$	$O(n^4)$	—	✗ [5]
Cachin et al. [16]	Async	$n/3$	✗	✗	$O(n^4)$	$O(n^4)$	$O(n^4)$	—	✗
Zhou et al. [52]	Async	$n/3$	✓	✗	$exp(n)$	$exp(n)$	$exp(n)$	—	✓
Shanrang [49]	Async	$n/4$	✓	✗	$O(n^3 \log n)$	$O(n^3 \log n)$	$O(n^4)$	—	✗
This work	Async	$n/3$	✓	✓	$O(n^3)$	$O(n^3)$	$O(n^3)$	$O(n^2)$	✓
This work	Async	$n/3$	✓	✗	—	—	—	$O(n)$	✗

[1] Desmedt et al. [24] is not verifiable, and assumes passive adversary.

[2] Requires impractically-large committee sizes.

[3] CHURP [42] only supports dual-threshold.

[4] Schultz-MPSS [45] claims asynchrony but their protocol uses PBFT [19] and requires eventual synchrony for liveness.

[5] COBRA [47] uses KZG commitments [35], but it is possible to use other commitment schemes with no trusted setup, while keeping the same asymptotic cost.

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 which guarantees eventual correct transmission. 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 $t' < |C'|/3$.

The new committee can contain any number of the same members as the old committee (however, they must use new public keys) and $\mathcal{A}$ can choose which nodes to corrupt in each. In the case of static committees, this is 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 [45] with the important distinction that a node corrupted at the beginning of Resharing in epoch i is considered corrupted in epoch $i+1$. More details can be found in Appendix A.

Notation. Let g and h be independent generators of a prime order cyclic group $\mathbb{G}$ with order p in which the discrete log problem is believed to be hard and let $\mathbb{Z}_p$ be a finite field of order p. For a given secret $s \in \mathbb{Z}_p$, 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 ˆ symbol generally to refer to a blinding object, such as a blinding polynomial $\hat{\phi}(x)$. Any object with the ˆ symbol above it is assumed to be sampled uniformly randomly.

Additionally, we may use parentheses around an operation to clarify that only the output is public. For example, in the Pedersen commitment $(g^s h^{\hat{s}})$, g^s and $h^{\hat{s}}$ are not known individually. Similarly $(s+r)$ indicates that only the sum of s and r is known.

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 $[\hat{s}']_d^j$ 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, $t \leq d \leq n-t-1$. Compared to ordinary Shamir Secret Sharing, an ACSS adds a *completeness* property which guarantees that if the ACSS protocol terminates, then every honest party will eventually receive a correct share of s. Moreover, this will be the case even if network messages can be arbitrarily delayed. An ACSS scheme consists of the following subprotocols:

- $\mathsf{Share}(C,t,d,s) \rightarrow \langle \{[s]_d^i\}_{P_i \in [C]}, aux\rangle$: 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.

- $\mathsf{Rec}(C,t,d,\{[s]_d^i\}_{P_i \in [C]}, aux) \rightarrow \langle s\rangle$: 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 $|C|-t-1$. Thus, the protocol can tolerate t byzantine corruptions and an additional $d-t$ honest-but curious corruptions.

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:

- $\mathsf{Reshare}(C,C',t,t',d,d',\{[s]_d^i\}_{P_i \in [C]}, aux) \rightarrow \langle \{[s]_d^j\}_{P_j' \in [C']}, aux'\rangle$: The old committee C creates a new 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 C 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) [17] 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 $f(v) : \{0,1\}^{|v|} \rightarrow \{0,1\}$ 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 uses the reliable broadcast protocol of Das et al. [22], which only assumes collision-resistant hash functions of output size κ and has a communication complexity $O(n|M| + \kappa n^2)$ to broadcast 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. [23] introduced a high-threshold ACSS scheme with a total network bandwidth of $O(n^2)$. To summarize, for each share $[s]$, the dealer uses ReliableBroadcast to send a discrete log commitment $g^{[s]}$, a Paillier encryption $\mathrm{Enc}([s])$ 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 (detailed in Appendix F) to relate this value to the Paillier-encrypted shares. We present our modified protocol in Algorithms 1 and 2 along with a proof sketch that these realize an ACSS algorithm in Appendix B.

Algorithm 1 High-Threshold ACSS Share

Public Inputs: $g, h, C, d, \{PK_i\}_{P_i \in C}$
Private Inputs: The dealer D holds a secret s
Public Outputs: $\{(g^{[s]^i_d} h^{[\hat{s}]^i_d})\}_{i\in[n]}$
Private Outputs: P_i holds $[s]^i_d, [\hat{s}]^i_d$

$\mathsf{SHARE}(s,d)$ (as D):

Sample two random degree d polynomials, $\phi(\cdot), \hat{\phi}(\cdot)$ and set $\phi(0) = s$
for $i \in [n]$ **do**
    $v_i \leftarrow \mathsf{Enc}_{PK_i}(\phi(i)), \hat{v}_i \leftarrow \mathsf{Enc}_{PK_i}(\hat{\phi}(i)), c_i \leftarrow g^{\phi(i)} h^{\hat{\phi}(i)}$
    $\pi_i \leftarrow ZK\{(\phi(i), \hat{\phi}(i)) : v_i = \mathsf{Enc}_{PK_i}(\phi(i)) \wedge \hat{v}_i = \mathsf{Enc}_{PK_i}(\hat{\phi}(i)) \wedge c_i = g^{\phi(i)} h^{\hat{\phi}(i)}\}$
$\mathsf{ReliableBroadcast}(\{v_i, \hat{v}_i, c_i, \pi_i\}_{i\in n})$

$\mathsf{SHARE} \rightarrow ([s]^i_d, [\hat{s}]^i_d, c)$ (as P_i):

upon receiving $\{v_j, \hat{v}_j, c_j, \pi_j\}_{j\in n}$ from $\mathsf{ReliableBroadcast}$ **do**
    if $\mathsf{DegreeCheck}(\{c_j\}_{j\in[n]}) \neq 1$ **then**
        Abort
    for $j \in [n]$ **do**
        if $\mathsf{Verify}(v_j, \hat{v}_j, c_j, PK_j, \pi_j) \neq 1$ **then**
            Abort
    $[s]^i_d \leftarrow \mathsf{Decrypt}_{SK_i}(v_i), [\hat{s}]^i_d \leftarrow \mathsf{Decrypt}_{SK_i}(\hat{v}_i)$,
    Output $[s]^i_d, [\hat{s}]^i_d, c : \{(g^{[s]^j_d} h^{[\hat{h}]^j_d})\}_{j\in[n]}$

3.2 MVBA

Since first proposed by Cachin et al. [17], several recent improvements have been made for MVBA [5, 33]. The state-of-the-art MVBA protocol is sMVBA [33], 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., $T_i \leftarrow T_i \cup \{j\}$ 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 $f(T'_j, T_i) = 1$.

Algorithm 2 High-Threshold ACSS Reconstruct

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

REC $\rightarrow s$ (as P_i):
Multicast $([s]^i_d, [\hat{s}]^i_d)$ to all parties
upon Receiving $m_j, \hat{m}_j$ from P_j **do**
  if $g^{m_j} h^{\hat{m}_j} = (g^{[s]^i_d} h^{[\hat{s}]^i_d})$ **then**
    Set $[s]^j_d = m_j$
upon Receiving $d+1$ valid shares **do**
  Interpolate and output s

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 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 [33], (along with many other MVBA protocols) requires a high-threshold non-interactive threshold signature setup to reduce communication and perform leader election [18]. 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 [4,22,23,29,37] to lift the trusted dealer assumption. The (special-purpose) ADKG protocols of Gao et al. [29] and Das et al. [22] achieve $O(\kappa n^3)$ 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 [14], the (general-purpose) ADKG protocol of Das et al. [23] 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. [23] with one instance of MVBA (which has constant rounds), and bootstrapping its shared randomness using special-purpose ADKG protocols such as [22,29]. 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. [23].

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 is needed for the KZG polynomial commitments [35] 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 commitments which are used to facilitate threshold signing with signature schemes such as BLS [15].

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 rerandomized 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 and modifying the protocol accordingly, we arrive at a full derivation of Algorithm 3.

How Do We Ensure All Parties in C' Get A Share? If we

Algorithm 3 High-Threshold Asynchronous DPSS

Private Inputs: P_i holds $[s]^i_d, [\hat{s}]^i_d$
Public Inputs: $c : \{(g^{[s]^i_d} h^{[\hat{s}]^i_d}\})$ for $i \in [n]$
Private Outputs: P'_i holds $[s]^i_{d'}, [\hat{s}]^i_{d'}$
Public Outputs: $c' : \{(g^{[s]^i_{d'}} h^{[\hat{s}]^i_{d'}})\}$ for $i \in [n']$

//Old Committee Portion
$\mathsf{RESHARE}([s]^i_d, [\hat{s}]^i_d, d')$ (as P_i):
Sample two degree-d' polynomials $\{\chi_i(x),\ \hat{\chi}_i(x)\}$ s.t. $\chi_i(0) = [s]^i_d, \hat{\chi}_i(0) = [\hat{s}]^i_d$
Use ACSS to share these polynomials with the new committee

//New Committee Portion
$\mathsf{RESHARE} \rightarrow ([s]^i_{d'}, [\hat{s}]^i_{d'}, c')$ (as P'_i):
$T_i \leftarrow \{\}$
upon outputting in j-th ACSS sessions where $(g^{\chi_j(0)} h^{\hat{\chi}_j(0)}) = (g^{[s]^j_d} h^{[\hat{s}]^j_d})$ **do**
  $T_i \leftarrow T_i \cup \{j\}$
  if $|T_i| = d' + 1$ **then**
    $T'_i \leftarrow T_i$
    Invoke $\mathsf{MBVA}(T'_i)$ 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.*
upon MBVA 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 $\forall j \in T$
  Interpolate $[s]^i_{d'} = B(0,i), [s]^i_{d'} = \hat{B}(0,i)$ from the shares in the subset
  Similarly, interpolate $\{(g^{[s]^j_{d'}} h^{[\hat{s}]^j_{d'}})\}$ for $j \in [n]$
  Output Private: $\{[s]^i_{d'}, [\hat{s}]^i_{d'}\}$, Public: $c' : \{(g^{[s]^j_{d'}} h^{[\hat{s}]^j_{d'}})\}$ for $j \in [n']$

Predicate $f(T'_j, T_i)$ for MBVA (as P'_i):
if $|T'_j| \neq d' + 1$ **then**
  return 0
upon $T'_j \subseteq T_i$ **do**
  return 1

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 using Verifiable Secret Sharing, in which termination of the Sharing phase of the protocol guarantees that the secret will be reconstructable. 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 [9, 47].

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 $\{g^{[s]^1}, g^{[s]^2}, \ldots g^{[s]^n}\}$ that correspond to each privately held share. Then we utilize an ACSS scheme, such as the one by Das et al. [23], 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 [28] 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 than 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.

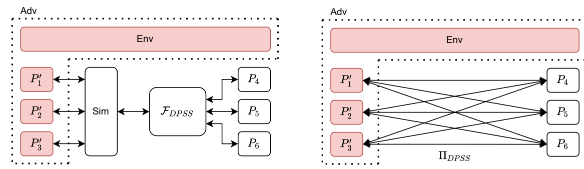
How Does C' Decide Which ACSS Instances To Use? Before the honest nodes in the new committee C' can interpolate the new shares for the secret, the protocol needs to provide two guarantees: (i) honest nodes in C' agree on the same set of ACSS instances for interpolation, and (ii) the above set of ACSS 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 MVBAs 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 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 $\mathcal{F}_{\mathrm{DPSS_{HT}}}$ functionality (c.f. Appendix C), assuming the ACSS protocol is secret and the Pedersen commitments are hiding. In the main body, 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. For simplicity, we omit some interactions that can be inferred from the context, and we assume authenticated asynchronous channels between the entities.

In principle the sequence of resharing and reconstruction commands that are run could be chosen by any process, such as a consensus protocol or a smart contract. All that matters for our protocol is that honest parties agree on this sequence. To simplify our formal model we designate a specific party called the coordinator to decide each command. When the coordinator is honest, this gives the environment full ability to adaptively choose the commands. To ensure all honest parties agree even when the coordinator is corrupt, we precede each command with an instance of reliable broadcast.



(a) Ideal (b) Real

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.

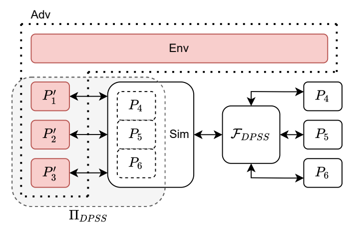


Figure 2: **Proof Idea** — Run Π_{DPSS} with simulated honest parties and use fake encryptions and fake correctness proofs for honest party data which the simulator does not know.

In the UC model, we say a protocol π 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 a trusted setup generating the PKI keys, the Pedersen setup (random curve elements g and h), the NIZK setup, and an MVBA protocol, the protocol in Algorithm 3 UC-realizes the functionality $\mathcal{F}_{DPSS_{HT}}$ if Algorithm 1 and Algorithm 2 satisfy the ACSS properties.*

We illustrate the high-level proof idea in Figure 2. The simulator samples its own PKI and creates a simulated honest party in mind for each real honest party and lets the simulated honest parties play with the external corrupted parties, handling faults in a similar manner to the real world protocol. If the ACSS dealer is corrupt, the simulator can decrypt all of the shares and save the full polynomials. Otherwise if the dealer is honest, the simulator sees only the corrupt party shares, but is able to use commitments provided by the functionality, fake encryptions, and fake NIZK proofs via a programmable random oracle to create an indistinguishable view. In reconstruction, the functionality provides full polynomials that match the commitments that the simulator used earlier. Even with this information, the environment can not tell whether or not the encryptions and NIZK proofs were fake.

Resharing uses similar techniques but allows the adversary a small degree of control over the rerandomization process, similar to what it has in a real world protocol where it can influence the output of MVBA by choosing message arrival orderings. We include a more detailed proof in Appendix D including relations between the functionality and property-based definitions introduced earlier.

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 $r \leftarrow \mathbb{Z}_{p^*}$

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

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 [11]. 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 Batch Reconstruction 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 [31] offers a solution, but it requires the use of an 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 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 some $\bar{com}$ from the other committee, we know that at least one must have come from an honest party, implying that all honest parties in the other committee will eventually receive this same commitment per the Agreement property of ACSS. Provided

Algorithm 4 Dual-Committee ACSS Share

Public Inputs: C,C',d,d'
Private Inputs: D holds a secret s
Private Outputs: P_i holds $([s]^i_d,[\hat{s}]^i_d)$, P'_j holds $([s]^j_{d'},[\hat{s}]^j_{d'})$
Public Outputs: com, com', $g^s h^{\hat{s}}$

$\mathsf{SHARE}(s,d,d')$ (as D):
// Select an ACSS scheme which produces a commitment com to the secret and proves that $\mathsf{Decommit}(com) = s$
// com may additionally include commitments to dealt shares.
Sample a random $\hat{s}$ to use for both ACSS sessions.
$\mathsf{ACSS}(s,\hat{s}) \to C$, $\mathsf{ACSS}(s,\hat{s}) \to C'$

$\mathsf{SHARE} \to ([s]^i,[\hat{s}]^i,c,c')$ (as either P_i or P'_i):
upon outputting $([s]^i,[\hat{s}]^i,com)$ in the local copy of ACSS **do**
  $\mathsf{Multicast}$ com to all parties in the other committee
  Store com locally
  upon Receiving $\bar{com}$ from the other committee t+1 times **do**
    Derive $g^s h^{\hat{s}}$ from both com and $\bar{com}$
    If the values match, Output $([s]^i,[\hat{s}]^i,com,\bar{com})$

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 $\{[s_j]^i_d,[\hat{s}_j]^i_d\}$ for $j \in [B]$
Public Inputs: $\{\{(g^{[s_j]^k_d} h^{[\hat{s}_j]^k_d})\}$ for $k \in [n]\}$ for $j \in [B]$
Private Outputs: P'_i holds $\{[s'_j]^i_{d'},[\hat{s}'_j]^i_{d'}\}$ for $j \in [B]$
Public Outputs: $\{\{(g^{[s'_j]^k_{d'}} h^{[\hat{s}'_j]^k_{d'}})\}$ for $k \in [n']\}$ for $j \in [B]$

//Old Committee Portion
$\mathsf{RESHARE}(\{[s_j]^i_d,[\hat{s}_j]^i_d\}$ for $j \in [B],d')$ (as P_i):
Sample $B/(n-2t)$ 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'.
Use MVBA to agree on $n-t$ players for whom all DC-ACSS instances terminated successfully.
Use a hyperinvertible matrix to extract B globally random sharings from the subset
Use $\mathsf{BatchReconstruct}$ 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
$\mathsf{RESHARE} \to (\{[s'_j]^i_{d'},[\hat{s}'_j]^i_{d'}\}$ for $j \in [B],c')$ (as P'_i):
upon receiving $\{(s_j+r_j),(\hat{s}_j+\hat{r}_j)\}$ for $j \in [B]$ from C **do**
  for $j \in [B]$ **do**
    $[s'_j]^i_{d'} = (s_j+r_j) - [r'_j]^i_{d'}, [\hat{s}'_j]^i_{d'} = (\hat{s}_j+\hat{r}_j) - [\hat{r}'_j]^i_{d'}$
    for $k \in [n']$ **do**
      $(g^{[s'_j]^k} h^{[\hat{s}'_j]^k}) = (g^{(s_j+r_j)} h^{(\hat{s}_j+\hat{r}_j)})/(g^{[r'_j]^k} h^{[\hat{r}'_j]^k})$
Output $\{[s'_j]^i_{d'},[\hat{s}'_j]^i_{d'}\}$ for $j \in [B]$, $\{\{(g^{[s'_j]^k_{d'}} h^{[\hat{s}'_j]^k_{d'}})\}$ for $k \in [n']\}$ for $j \in [B]$

the commitments are to the same s (and $\hat{s}$ if applicable), 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 amortizing, Algorithm 5 still requires $O(n)$ 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 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 [51] secret sharing scheme which achieves a linear network overhead per secret when instantiated with the KZG [35] 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 [28], 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 $\{g, g^{\alpha}, g^{\alpha^2}, \ldots g^{\alpha^t}\}$ where α 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 nonrandom secrets, 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 Vassantlal et al. [47] 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 [13], Belisarius [44] and Basu et al. [9] building upon PBFT [19] under partial synchrony, and Secure Store [38], CODEX [43] building upon Byzantine Quorum Systems [41] under asynchrony.

For dynamic committees, the recent works of Goyal et al. [31] and Benhamouda et al. [12] design new synchronous DPSS schemes for storing secrets on blockchains, while COBRA builds upon HotStuff [50] under partial synchrony. CALYPSO [36] 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 [19], 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. [31] 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* [30]. 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. [31] 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. [31] 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 some function of private user inputs. This can be divided 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 [40] utilized a *non-robust* offline phase in which precomputation attempts could fail but would

Table 2: Asymptotic and Concrete Costs of BMR Escape Hatch vs Gate-by-Gate MPC for AMM application

	Rounds	Mults	Rounds	Mults
Gate by Gate	O(d)	O(C)	50	2025
BMR Offline	O(1)	O(C)	258	$1.1 \cdot 10^8$
BMR Online	O(1)	O(I+O)	18	1281

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. The use of DPSS extends this successful termination guarantee to applications with network churn or mobile adversaries.

BMR Escape Hatch. The ability to proactively reshare offers the potential of a tradeoff where an expensive preprocessing phase that is not needed in the typical case can be generated when network utilization is low and persisted for a long duration. For example, in an MPC-based automated market application, an "escape hatch" may be used only to accelerate transactions in the online phase during periods of anomalously high usage, such as during a flash crash or price spike.

The BMR [10] MPC protocol is a multi-party variant of Yao's Garbled Circuits in which players jointly compute and evaluate circuits. An ordinary MPC that is used to generate wire labels and garbled gates is run in parallel for each gate in the boolean circuit, resulting in a constant round complexity independent of the circuit and committee size.

Once generated, these garbled circuits can be evaluated locally at a low cost. An MPC function evaluation can thus be split into a relatively-high cost offline precomputation which generates the garbled circuit and a low-cost online phase in which inputs are mapped to input wires, circuits are evaluated locally, and output wire labels are mapped to results.

Typically, to avoid a blow-up associated with emulating encryption using arithmetic circuits, BMR protocols use a distributed encryption due to Damgård and Ishai [20, 21]. On the other hand, for large committees like what we consider, it is eventually more efficient (and in any case simpler) to use MPC-friendly symmetric encryption and accept this cost. The pseudocode for the later approach is described in Algorithm 6. Here $\mathsf{COND}(s, a \mapsto x, b \mapsto y)$ is implemented as $(s-b)/(a-b) \cdot x + (s-a)/(b-a) \cdot y$ where any of these values may be public or secret shared. Concretely, if we take MiMC as the MPC-friendly PRF (parameterized with $k = 64$ rounds), the overhead is approximately ≈ 6400 multiplications per gate in the program circuit. Note that this does not depend on the number of parties n.

In Table 2 we give a comparison based on an MPC automated market making task (specifically, we reimplemented* the Trade function from HoneyBadgerSwap [39]). For a gate-by-gate algorithm, we wrote a version of the program which uses built-in arithmetic routines from MP-SPDZ in the malicious-secure Shamir sharing mode. This takes 50 rounds and requires 2025 total multiplications with Beaver triples, mainly due to the need to split a finite-field element into bits in order to perform division. We also used MP-SPDZ to implement the BMR Escape Hatch program from Algorithm 6 and used this to garble a boolean circuit which implements the same Trade function. For the asymptotic analysis, we consider a boolean circuit with I input bits, O output bits, C total gates, and depth d. The online cost savings would be even greater for a circuit that is larger relative to the input/output size.

Algorithm 6 BMR Escape Hatch

Garbling Phase (BMR Offline)

Public inputs: Circuit C

Public outputs: $\{e_{g,x,y}\}_{x,y \in \{0,1\}}$ for each gate g

Secret shared outputs: $[m_i], \{[w_{i,x}]\}_{x \in \{0,1\}^2}$ for each input i, $[m_o], \{[w_{o,x}]_{x \in \{0,1\}}$ for each output o

Garble (as P_i):

For each wire j, sample $[w_{j,0}], [w_{j,1}] \xleftarrow{\$} \mathbb{F}$, $[m_j] \xleftarrow{\$} \{0,1\}$
For each NAND gate g with input wires a,b and output c,
For each $x,y \in \{0,1\}^2$,
$[z_{g,x,y}] := \mathsf{NAND}(x \oplus [m_a], y \oplus [m_b]) \oplus [m_c]$
$[w_{g,x,y}] := \mathsf{COND}([z_{g,x,y}], 0 \mapsto [w_{c,0}], 1 \mapsto [w_{c,1}])$
$[e_{g,x,y}] := \mathsf{DualEnc}([w_{a,x}], [w_{b,y}], [w_{g,x,y}])$
Reveal and output $e_{g,x,y}$
Output $[m_i], \{[w_{i,x}]\}_{x \in \{0,1\}^2}$ for each input wire i,
Output $[m_o], \{[w_{o,x}]\}_{x \in \{0,1\}^2}$ for each output wire o

Input Mapping Phase (BMR Online)

Secret shared inputs: $[v_i \in \{0,1\}]$, $[m_i], \{[w_{i,x}]\}_{x \in \{0,1\}}$ for each input wire i

Public outputs: $w_{i,v_i \oplus m_i}$ for each input wire i

InputMap (as P_i):

For each input wire i
$[w_{i,v_i \oplus m_i}] := \mathsf{COND}([v_i] \oplus [m_i], 0 \mapsto w_{i,0}, 1 \mapsto w_{i,1})$
Reveal and output $w_{i,v_i \oplus m_i}$

Local Evaluation Phase (BMR Online)

Public inputs: $w_{i,v_i \oplus m_i}$ for each input wire i, $\{e_{g,x,y}\}_{x,y \in \{0,1\}^2}$ for each gate g

Public outputs: $w_{o,v_o \oplus m_o}$ for each output wire o

Evaluate (as P_i):

For each input wire i, $k[i] := w_{i,v_i \oplus m_i}$
For each gate g with input wires a, b, output c,
For each $x,y \in \{0,1\}^2$,
if $w \leftarrow \mathsf{DualDec}(k[a], k[b], e_{g,x,y}) \neq \perp$
then $k[c] := w$
Output $k[o]$ for each output o

Output Mapping Phase (BMR Online)

Public inputs: $w_{o,v_o \oplus m_o}$ for each output wire o

Secret shared inputs: $[m_o], \{[w_{o,x}]\}_{x \in \{0,1\}}$ for each output wire o

Secret shared outputs: $[v_o \in \{0,1\}]$ for each output wire o

OutputMap (as P_i):

For each output wire o
$[v_o] := [m_o] \oplus \mathsf{COND}(w_{o,v_o}, [w_{o,0}] \mapsto 0, [w_{o,1}] \mapsto 1)$
Output $[v_o]$

*Code available at https://github.com/tyurek/bmr-escape-demo

6 Evaluation

We implemented[†] all of our asynchronous DPSS protocols and characterize their performance in this section. For a concrete example, we evaluate the cost of resharing the "escape hatch" for our MPC Automated Market Maker application.

6.1 Experimental Setup

Our implementations were done primarily in python (forking from the codebase of [23]), while core cryptographic operations rely on libraries written in rust. In particular, we used the ristretto group implementation of curve25519_dalek [1] and a Paillier modulus of 2048 bits (corresponding to 112 bits of security per NIST guidlines [7]) to instantiate our high-threshold DPSS protocols and we used ZCash's bls12-381 library [32] 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.

Additionally, we remark that although our high-threshold constructions are UC secure and rely on a NIZK, our simulator proof (Appendix D) does not rely on extracting any values from the NIZK and so we can utilize the Fiat-Shamir heuristic [25] rather then a Fichlin transformation [26].

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 asyncio tasks sending serialized messages.

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). Given the latency across different AWS regions is typically at most $300-400$ms [2], and our protocols have constant round complexity with small constants (less than 20), the running time caused by network latency is several orders of magnitude smaller than the computation time (as in Table 3) for a moderate committee size.

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 [3], 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.

Table 3: 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

n	Low-Threshold No Crashes / t Crashes	High-Threshold 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	-

6.3 Experimental Results

Our primary results in Table 3 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. This is likely because in our crash-fault setup, nodes crash instantly and consequently, honest nodes do not waste time participating in ACSS instances which do not end up in the final subset.

We additionally evaluate the concrete costs of proactivizing the precomputation for our AMM escape hatch. The program in question has two player inputs (desired amount of Token A, slippage allowance), four system inputs (user and pool balances of both tokens in the trading pair), and four outputs (the

[†]Repo available at https://github.com/tyurek/dpss

Table 4: Computation time (in seconds) required for a user to refresh their Escape Hatch with a new committee of size n in the fault-free setting

n	4	10	19	31
time	25.50	46.81	72.79	118.20

updated user and pool balances). Each input/output is 64 bits in length, and every input/output bit corresponds to two secret shared wire labels and a secret shared mask bit, meaning that resharing this computation requires resharing 1920 different shared secrets, the costs of which are given in Table 4.

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 [45]. 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 multiexponentations, and optimizing polynomial operations, this would yield misleading results: Both our scheme and COBRA (as well as several others [31,42,49]) 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 DPSS 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 recalled prior applications which used DPSS and show how they how they can be better equipped to handle more adversarial environments. We additionally used batch-amortized DPSS to refresh and transfer precomputed data in a novel "BMR escape hatch". We hope that these advancements allow future practitioners to build awesome resilient applications for use on a decentralized internet.

Acknowledgements. We thank Matthieu Rambaud, Antoine Urban, Garry Ferrando Cooper Ranee, Mario Alessandro Barbara, Aniruhd Venu, and our anonymous shepherd for technical discussions related to this paper. This work was funded in part by NSF award #1943499 and by IC3 industry partners.

References

[1] curve25519-dalek: A pure-rust implementation of group operations on ristretto and curve25519, 2021. https://github.com/dalek-cryptography/curve25519-dalek.

[2] CloudPing - AWS Latency Monitoring. https://www.cloudping.co/grid, Dec 2022.

[3] Speedtest Global Index. https://www.speedtest.net/global-index, April 2022.

[4] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *ACM PODC*, 2021.

[5] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *ACM PODC*, 2019.

[6] Andreea B Alexandru, Erica Blum, Jonathan Katz, and Julian Loss. State machine replication under changing network conditions. *Cryptology ePrint Archive*, 2022.

[7] Elaine Barker. Nist special publication 800-57 part 1, rev 5. *NIST, Tech. Rep*, 2020.

[8] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. Communication-optimal proactive secret sharing for dynamic groups. In *ACNS*, 2015.

[9] 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 *ACM CCS*, 2019.

[10] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *ACM STOC*, 1990.

[11] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In *TCC*, 2008.

[12] 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 *TCC*, 2020.

[13] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. Depspace: a byzantine fault-tolerant coordination service. In *ACM EuroSys*, 2008.

[14] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *IACR PKC*, 2003.

[15] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *ASIACRYPT*, 2001.

[16] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *ACM CCS*, 2002.

[17] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, 2001.

[18] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 2005.

[19] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 2002.

[20] Sandro Coretti, Juan Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multi-party computation based on one-way functions. In *ASIACRYPT*, 2016.

[21] Ivan Damgård and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In *CRYPTO*, 2005.

[22] Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *ACM CCS*, 2021.

[23] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *IEEE Symposium on Security and Privacy (SP)*, 2022.

[24] Yvo Desmedt and Sushil Jajodia. Redistributing secret shares to new access structures and its applications. Technical report, Citeseer, 1997.

[25] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *EUROCRYPT*, 1986.

[26] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In *CRYPTO*, 2005.

[27] Pierre-Alain Fouque and Jacques Stern. One round threshold discrete-log key generation without private channels. In *International Workshop on Public Key Cryptography*, 2001.

[28] Shuhong Gao. A new algorithm for decoding reed-solomon codes. In *Communications, information and network security*. 2003.

[29] 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.

[30] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In *ACM STOC*, 2013.

[31] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. In *IACR PKC*, 2022.

[32] Jack Grigg and Sean Bowe. zkcrypto/pairing. https://github.com/zkcrypto/pairing.

[33] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous bft closer to practice. In *NDSS*, 2022.

[34] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO*, 1995.

[35] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, 2010.

[36] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Private data management for decentralized ledgers. In *Proc. VLDB Endow.*, 2020.

[37] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *ACM CCS*, 2020.

[38] Subramanian Lakshmanan, Mustaque Ahamad, and H Venkateswaran. Responsive security for stored data. *IEEE TPDS*, 2003.

[39] Yunqi Li. Honeybadgerswap: Making mpc as a sidechain, 2021.

[40] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *ACM CCS*, 2019.

[41] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 1998.

[42] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. Churp: dynamic-committee proactive secret sharing. In *ACM CCS*, 2019.

[43] Michael A Marsh and Fred B Schneider. Codex: A robust and secure secret distribution system. *IEEE TDSC*, 2004.

[44] Ricardo Padilha and Fernando Pedone. Belisarius: Bft storage with confidentiality. In *2011 IEEE 10th International Symposium on Network Computing and Applications*. IEEE, 2011.

[45] David A Schultz, Barbara Liskov, and Moses Liskov. Mobile proactive secret sharing. In *ACM PODC*, 2008.

[46] Adi Shamir. How to share a secret. *Communications of the ACM*, 1979.

[47] Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira, and Alysson Bessani. Cobra: Dynamic proactive secret sharing for confidential bft services. In *IEEE Symposium on Security and Privacy (SP)*, 2022.

[48] Theodore M Wong, Chenxi Wang, and Jeannette M Wing. Verifiable secret redistribution for archive systems. In *IEEE SISW*, 2002.

[49] Yunzhou Yan, Yu Xia, and Srinivas Devadas. Shanrang: Fully asynchronous proactive secret sharing with dynamic committees. *Cryptology ePrint Archive*, 2022.

[50] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *ACM PODC*, 2019.

[51] Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. hbacss: How to robustly share many secrets. In *NDSS*, 2022.

[52] Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. Apss: Proactive secret sharing in asynchronous systems. *ACM TISSEC*, 2005.

A Epoch Definition

As in MPSS [45], 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. [6] (which has a slightly different model where the adverary 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 [45] for completeness. We add (in italics) the necessary additional requirement that a node participating in resharing is considered to be in both epochs at the same time (and thus, an adversary corrupting during resharing spends its corruption budget for both epochs).

- Local Epochs. A node is in epoch e if it has secret shares (*including the reshares of any secret share during the Resharing*) 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 so that it can no longer be recovered and additionally discards its reshares if it had one. *A node with shares in epoch e that begins resharing is considered to be in both e and* $e+1$ *at the same time until resharing materials have been received from all honest parties and both the prior shares and resharing materials are securely deleted.*

- System Epochs. Let G_e denote the set of nodes responsible for the secret in epoch e. Nodes not in G_e advance to the next epoch by advancing their secret decryption key; nodes in G_e additionally discard their shares and transfer the new information to nodes in G_{e+1}. 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 Proof Sketch for ACSS

As mentioned, our ACSS protocol is a modified version of the high-threshold ACSS scheme in Das et al. [23]. Due to the similarity, the proof of our ACSS is also analogous to the one of Das et al. [23]. For completeness, we provide a proof sketch of our ACSS protocol here.

Correctness. If the dealer is correct, by the Validity property of the reliable broadcast, and the Completeness property of the commitment scheme and non-interactive zero-knowledge scheme, all correct parties will output the share of dealer's secret after the Sharing phase, and are able to reconstruct the secret during the Reconstruction phase.

Secrecy. We argue that there exists a PPT simulator that simulates the view of any static PPT adversary $\mathcal{A}$. Without loss of generality, suppose $\mathcal{A}$ corrupts the first d nodes. Given the shares $s_i, \hat{s}_i$ for $i \in [d]$ and the commitments $g^{s_i}h^{\hat{s}_i}$ for $i \in [n]$, the simulator encrypts the shares of node $i \in [d]$, and uses encryption of 0 and 1 as the encryption of shares s_i and $\hat{s}_i$ for node $j \in [d+1,n]$, respectively. Due to CPA security of the encryption scheme, the encryptions are indistinguishable from the encryptions of the actual shares. Then the simulator uses the zero-knowledge simulator of the NIZK scheme to construct the proofs for each $j \in [d+1,n]$. The view of the adversary $\mathcal{A}$ in the interaction with the simulator is computationally indistinguishable from the view in actual execution.

Agreement. If any correct party outputs in the Sharing phase, due to the Totality property of the reliable broadcast and the Soundness property of the commitment scheme and NIZK scheme, all correct parties eventually output a share of a canonical secret.

C High Threshold DPSS Functionality

This section presents the high-threshold DPSS functionality $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$, which serves as the primary security specification for our main construction. To simplify our model, we have a designated party, the coordinator, issue the signals for the Reconstructing and Resharing events. This party ensures unanimous agreement amongst honest parties that a given protocol has started, as otherwise some properties of our DPSS would not hold. In a real deployment, such coordination would be implemented using a bulletin board or broadcast protocol.

In our analysis, the coordinator must use ReliableBroadcast to notify all parties of new protocol-beginning events and is not restricted in how it decides which events to run. This ensures that if any honest party receives the broadcast successfully, then all honest parties eventually will do so as well.

We further assume that the session identifier sid contains the identity of the corresponding dealer.

Output Modeling. When designing $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$, we make the explicit choice for the functionality to give secret shares to honest parties, rather than simply giving them the result of reconstruction. We justify this decision by presenting two different modeling choices.

- *Option A* (what we do): Honest parties receive shares directly from the functionality after Share and Reshare, and receive the secret from Reconstruct.
- *Option B* ($\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$ doesn't send honest parties shares): Honest parties receive only the message "OK" following Share and Reshare, but still receive the secret from Reconstruct.

Why should we prefer Option A? While it's true that some applications (like many MPC applications) can be modeled using secret inputs and public outputs, with no need for intermediate details about shares in the functionality, not all applications end in reconstruction and may instead rely on share outputs. For example, in threshold signatures, the master secret is never reconstructed, and instead the shares themselves are used to sign messages. Using Option A allows the same functionality to be used, regardless of whether or not an application needs to call Reconstruct.

Share. The Share portion of $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$ allows the dealer to distribute shares of a secret to the committee of recipients. In the case of an honest dealer, the sharing polynomial will be of the specified degree d and have uniformly random coefficients, which are sampled by the functionality. However, a dishonest dealer can, in the real world, choose any (possibly non-random) sharing polynomial provided it satisfies the protocol's degree bound, and so $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$ accounts for this.

Additionally, $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$ calculates and exposes Pedersen commitments to shares. The motivation for this is that some sort of consistent information like this is needed in order to use high-threshold shares in many applications (the privacy threshold is too high to use error correcting algorithms to account for faulty shares, so instead there needs to be something to individually check shares against).

Lastly, the functionality leaks outputs to $\mathcal{A}$ before it sends them to honest parties, so to model the control $\mathcal{A}$ is given over message ordering in our asynchronous network model.

Reconstruct. The Reconstruct portion of $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$ allows for the public revealing of shared secrets to all parties in a given committee. In many secret sharing applications (such as multiparty computation) it is desirable to also be able to reconstruct a sum (or more generally a linear combination) of secrets, rather than revealing each secret individually. To account for this, the Reconstruct portion has the coordinator specify one or more secrets to reconstruct, along with coefficients for each to use in a linear combination.

If multiple secrets are specified, no action is taken until all of the secrets are submitted to the functionality by their respective dealers. Here a dealer submitting a secret to the functionality corresponds to the real world dealer sending enough information to guarantee that the protocol terminates successfully.

Reshare. As with Reconstruct, Reshare can also be used on a linear combination of sharings, rather than resharing all of them individually. Similarly, the functionality must wait until all of the inputs are received before proceeding and should leak results to the adversary first.

Some additional complexity comes from the influence that the real world adversary has over MVBA, which determines which polynomials are used to randomize shares. To model this, we allow $\mathcal{A}$ to directly choose which polynomials are used after seeing their shares and all of the commitments.

Relation to Property-Based Definition. In Sections 2.1 and 2.2, we gave descriptions of Share, Reconstruct, and Reshare and defined properties that characterized their performance. We will now briefly relate these properties to those provided by $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$.

- **Correctness**: Correctness for Share and Reconstruct is defined such that if an honest Dealer inputs s in Share, then Reconstruct will successfully reveal s to all honest parties. This is captured by $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$ logging polynomials encoding s and sending them to all parties.
  In Reshare, Correctness implies that the new committee will receive shares that also lead to s being output in Reconstruct. Here $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$ creates new polynomials which encode the same secret and gives them to C'.
- **Secrecy**: Secrecy for both Share and Reshare is defined in terms of the existence of a simulator for our functionality.
- **Agreement**: Our Agreement property ensures that if an honest party receives output in Share, then some value has actually been shared and can be reconstructed. If the last line of Share in $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$ is reached, then there must be degree-d sharing polynomials which all parties receive shares of.
- **Liveness**: Reshare has a liveness property that guarantees that an adversary in unable to prevent the protocol from finishing as long as all honest parties agree to start it. A similar property is encoded into Correctness for Reconstruct and Share (when the Dealer is honest).
  In $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$, we capture this as the coordinator signaling the starts of Reconstruct and Reshare (all honest parties will eventually receive this message and agree to start if the message is valid). If the coordinator's message is valid, then there is no mechanism to prevent $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$ from delivering outputs for either of these subprotocols.
  For Share this is even simpler: Once the dealer gives a valid input, the functionality eventually outputs to all parties.
- **High-Threshold** and **Resizability**: We define $\mathcal{F}_{\mathrm{DPSS}_{\mathrm{HT}}}$ as allowing the coordinator to specify an arbitrary polynomial degree for both Share and Reshare. We note that our real world protocol is only simulatable when $t \leq d \leq |C| - t - 1$ and $t' \leq d' \leq |C'| - t' - 1$.

D Proof of Theorem 1

We construct the following simulator Sim with the ability to program the random oracle and thus simulate proofs for the NIZK proof used in Algorithm 3. We assume a static corruption model, namely, that at the beginning of the protocol the simulator Sim knows the identities of the t Byzantine corrupted parties and the $d-t$ additional parties which $\mathcal{A}$ can observe but not control. These parties will be collectively referred to as C_{Corrupt}.

The general simulation strategy used here is for the simulator to run local copies of all of the honest parties in the network, including the simulator. As our asynchronous network model assumes that messages can be ordered adversarially, the simulator only adds messages to the message queues of the simulated parties. The environment chooses when these messages are actually sent.

Share. During sharing, if the dealer is corrupted, the simulator will receive a 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 $[s]_d^i, [\hat{s}]_d^i$. Note that the Share function might abort due to degree check failures or verification failures. In this case, the simulator aborts. Otherwise, it forwards the sharing polynomials to the functionality.

If the dealer is honest, the simulator will receive the shares for the corrupted parties and commitments for all parties from the functionality. In this case, its job is to simulate a Dealer who will eventually distribute correctly encrypted shares to corrupted nodes via ReliableBroadcast. The ReliableBroadcast input also includes the commitments received from the functionality, along with encryptions of 0 and fake correctness proofs for all the positions occupied by honest nodes. Note here that our simulator only needs to be able to create fake NIZK proofs and does not require any online extraction.

Because the functionality outputs shares to honest parties, the environment will always know the full sharing polynomials. However, our simulator does not ever need to sample fake shares, only fake encryptions and correctness proofs, both of which can be done with no knowledge of the shares. Because the proofs are zero knowledge and the encryptions are semantically secure, the environment can not distinguish them from their real-world equivalents. Finally, the other simulated receiver nodes should output messages indicating that they accept these proofs.

Reconstruct. If the coordinator is honest, then when the simulator receives a REC message from the functionality, it needs to simulate a coordinator that would have sent the message which triggered this reconstruction. If the coordinator is corrupted, the simulator needs to run the receiving algorithm for ReliableBroadcast on each simulated honest party and send the result to the functionality if successful.

Once the functionality begins returning results, the simulator learns the full reconstruction polynomials and it can easily program the simulated honest parties to be ready to send the appropriate shares once they hear from the coordinator. From the perspective of the environment, the messages sent in the real and ideal worlds are identical.

Reshare. As with the Reconstruct protocol, the simulator runs ReliableBroadcast in the case of a corrupted coordinator and learns which protocol is being run from the functionality in the case of an honest coordinator.

Once the Reshare protocol starts, the functionality will leak shares and commitments to the simulator. The simulator then initiates ACSS Dealer sessions in simulated honest parties once they have received all of their input shares and heard from the coordinator. As before, these ACSS sessions will include encryptions of 0 and fake correctness proofs for indexes occupied by honest parties in the new committee.

The simulator also programs simulated honest parties to begin running MVBA (when appropriate) to agree on a set of polynomials to use to build the final resharing. Once MVBA outputs for one honest node, it is guaranteed that all honest nodes will receive the same output, and so the output can be fed into the functionality. At this point, the corrupted nodes have all the messages they need to output and the non-simulated ideal world honest parties will eventually receive their correct results, so no further action is needed.

E Proof Sketch for Batch DPSS Constructions

In this section, we will describe a proof sketch for the UC security of our batch-amortized DPSS constructions, both for a high-threshold and regular-threshold cases. Both batch-amortized constructions are built off of the same modification to the base protocol, in which agreeing upon a set of reshared shares to use is replaced with making many reconstructions of $(s+r)$ and instead agreeing on inputs to use to construct globally-random values for r. The other major difference is that the use of ACSS in Reshare involves sharing a secret with both the old and new committees at the same time.

Both of our batch-amortized DPSS constructions also utilize a PKI, and so the simulator similarly begins by sampling secret keys to assign all of the nodes.

Batch-amortized High-Threshold DPSS With regards to the functionality and simulation used for the Share and Reconstruction, the only changes that need to be made are to replace references to a single share with references to a batch of them. This is because the real world version of these subprotocols is not any different except for the fact that they operate on multiple secrets at once.

The main significant difference comes from the different mechanism in how shares are transferred in the Reshare protocol. Instead of imposing the condition that $\phi_i(0) = \phi(i)$ and $\hat{\phi}_i(0) = \hat{\phi}(i)$, the condition is instead that if i is the index of an honest node, then $\phi_i(0)$ = and $\hat{\phi}_i(0)$ will be uniformly random. Additionally, polynomials $\phi_i'(\cdot)$ and $\hat{\phi}_i'(\cdot)$ of degree d' must be sampled by the functionality s.t. $\phi_i'(0) = \phi_i(0)$ and $\hat{\phi}_i'(0) = \hat{\phi}_i(0)$. Otherwise, the functionality and proof proceed as before, with Sim running MVBA on simulated honest nodes and submitting $\mathcal{A}$'s choice of polynomials to the functionality.

Batch-amortized Regular-Threshold DPSS The batch-amortized regular-threshold DPSS scheme uses hbACSS [51] as the underlaying ACSS scheme, but structurally it is very similar in that it also involves reliably distributing encrypted shares. However, it is no longer necessary to create fake proofs, as hbACSS does not require corrupted nodes to verify the proofs correctness of honest nodes' shares (moreover, these proofs can be encrypted).

The regular-threshold scheme's DPSS functionality retains the modification to Reshare described previously, and additionally modifies all subprotocols to no longer output share commitments, as these are no longer required to be produced by the simulator since reconstruction can instead be performed by error correction algorithms.

F Sigma Protocol for Equivalence of Committed Values and Paillier Decryptions

We define a Zero Knowledge Proof (ZKP) as a proof of the existence of a value which satisfies a given relation R. A ZKP satisfies properties of *completeness*, *soundness*, and *zero knowledge*

- *Completeness (informal)*: A honest prover who knows that a witness satisfies R will successfully convince an honest verifier of this fact.
- *Soundness (informal)*: If an honest verifier accepts the proof, then there exists a witness satisfying the relation, except with a negligable probablity κ referred to as the *soundness error* .
- *Zero knowledge (informal)*: The proof itself reveals no additional information about the witness to any verifier, beyond what was already known.

A Sigma protocol refers to a specific interactive ZKP structure (which can be made noninteractive by applying the Fiat-Shamir heuristic) in which the prover first commits to some information, the verifier responds with a challenge, and the prover responds to the challenge, completing the proof.

Fouque and Stern [27] present a sigma protocol for the existence of a witness x such that the discrete log of g^x and the decryption of $enc_{pk}(x;\, r)$ in the Paillier cryptosystem are the same. We wish to modify this proof to allow the prover to convince the verifier that there exist two field elements $(x,\hat{x})$ and randomizers $(r,\hat{r})$ such that $Y = g^x h^{\hat{x}}, c = enc_{pk}(x;r), \hat{c} = enc_{pk}(\hat{x};\hat{r})$, even if both the prover and verifier do not know the secret key corresponding to pk.

We begin by briefly recalling the Paillier cryptosystem.

- **Generate**: Generate a modulus N which is the product of two large primes p,q. Here N is the pubic key and (p,q) is the private key
- **Encrypt**: $enc(m;r) = (1+N)^m r^N \; mod\, N^2$, where $0 \le m < N$
- **Decrypt**: $dec(c) = \frac{(c^{\phi(N)} \; mod\, N^2) - 1}{N} \cdot \phi(N)^{-1} \; mod\, N$, where ϕ is Euler's totient function

We next specify our sigma protocol $\Pi_{\textbf{ComDecEq}}$ and provide a proof sketch which follows that of the original protocol.

$\Pi_{\textbf{ComDecEq}}$

- **Setup**: $g,h \in \mathbb{G}$, and a Paillier modulus N.
- **Inputs**: Common input is $(Y,c,\hat{c})$. The prover knows $x,\hat{x},r,\hat{r}$ such that $Y = g^x h^{\hat{x}}, c = enc(x;r), \hat{c} = enc(\hat{x};\hat{r})$

1. Prover samples $(u,\, \hat{u},\, s,\, \hat{s}) \leftarrow \mathbb{Z}_N^*$ and computes $T = g^u h^{\hat{u}}, e_u = enc(u;s), \hat{e}_u = enc(\hat{u};\hat{s})$
2. Verifier receives $(T,\, e_u,\, \hat{e}_u)$ and responds with a challenge e
3. Prover computes $z = u + e \cdot x, \hat{z} = \hat{u} + e \cdot \hat{x}, w = s \cdot r^e \; mod\, N^2, \hat{w} = \hat{s} \cdot \hat{r}^e \; mod\, N^2$
4. Verifier receives $(z,\, \hat{z},\, w,\, \hat{w})$ and accepts iff all the following hold
   - $T = g^z h^{\hat{z}} \cdot Y^{-e}$
   - $e_u \cdot c^e = enc(z;w) \bmod N^2$
   - $\hat{e}_u \cdot \hat{c}^e = enc(\hat{z};\hat{w}) \bmod N^2$

- *Completeness*: An honest prover who follows the protocol will produce values which will pass the checks at the end of the proof:

$$\begin{aligned} g^z h^{\hat{z}} \cdot Y^{-e} &= g^{u+ex} h^{\hat{u}+e\hat{x}} \cdot (g^x h^{\hat{x}})^{-e} = g^u h^{\hat{u}} \\ enc(z;w) &= (1+N)^z w^N = (1+N)^{u+ex} (s \cdot r^e)^N \\ &= e_u \cdot (1+N)^{ex} (r^e)^N = e_u \cdot c^e \bmod N^2 \end{aligned}$$

- *Soundness*: A proof algorithm has *special soundness* if there exists an efficient Extractor algorithm which can extract a witness w for the relation from two prover transcripts with a common first prover message. The existence of this extractor implies soundness, because it implies that a validating proof must have a corresponding (extractable) witness, save for a neglibable error probability.

  Given two accepting transcripts $(g,\, h,\, N,\, Y,\, c,\, \hat{c},\, T,\, e_u,\, \hat{e}_u,\, e,\, z,\, \hat{z},\, w,\, \hat{w})$ and $(g,h,N,Y,c,\hat{c},T,e_u,\hat{e}_u,e',z',\hat{z}',w',\hat{w}')$ where $e \neq e'$, the values of x, $\hat{x}$ can be derived by computing $x = (z - z')/(e - e')$ and $\hat{x} = (\hat{z} - \hat{z}')/(e - e')$

- *Zero Knowledge*: A proof is said to be zero knowledge if there exists a Simulator which is capable of creating a valid proof for a statement for which it does not know the witness, as this would imply the the proof itself does not leak information about the witness, as the Simulator knew no information to leak. Here, the Simulator is given the ability to choose the challenge it receives from the verifier, allowing it to easily create a fake proof which otherwise would require $1/\kappa$ attempts on average.

  Our Simulator begins by sampling $e,w,\hat{w},z,\hat{z}$ at random. It then sets $T = g^z h^{\hat{z}} \cdot Y^{-e}$, $e_u = c^{-e} enc(z;w) \bmod N^2$, and $\hat{e}_u = \hat{c}^{-e} enc(\hat{z};\, \hat{w}) \bmod N^2$. The simulator then programs the random oracle to output e when presented with the transcript $(g,h,N,Y,c,\hat{c},T,e_u,\hat{e}_u)$. Now, the simulator can present $(T,\, e_u,\, \hat{e}_u)$ to the uncorrupted verifier and proceed through the remainder of the proof.

$\mathcal{F}_{\mathbf{DPSS_{HT}}}$

Share

On receiving $(\mathsf{SHARE}, \mathsf{sid} := \{C, d\}, s)$ 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 $(\mathsf{SHARE}, \mathsf{sid} := \{C, d\}, \phi(\cdot), \hat{\phi}(\cdot))$ from a corrupt dealer (only once per sid):

- Abort if either polynomial has degree $> d$ or $d > N - t - 1$.

In either case, compute $\mathsf{polynomials} = \{\phi(\cdot), \hat{\phi}(\cdot)\}$, $c = \{g^{\phi(i)} h^{\hat{\phi}(i)}\}_{i \in [0,N]}$, and store $(\mathsf{polynomials}, \mathsf{c}, \mathsf{sid}) \rightarrow \mathsf{storage}$.

Send $(\mathsf{SHARE}, \mathsf{sid}, \phi(i), \hat{\phi}(i), c)$ to each party $P_i \in C_{\text{Corrupt}}$ and eventually to each party $P_i \in C_{\text{Honest}}$.

Reconstruct

On receiving $(\mathsf{REC}, \mathsf{rid} := \{(\mathsf{sid}_i, \mathsf{coeff}_i), C'\})$ from the Coordinator where each pair $(\mathsf{sid}_i, \mathsf{coeff}_i)$ defines a term to use in a linear combination of outputs of Share, and C' receives the reconstruction:

- If the committee C is not the same in every sid, ignore the request.
- If any share ID sid_i is not available in the memory, wait for the share from the dealer.

After receiving all the shares referred by sid_i's in the message,

- Compute polynomials $\phi(\cdot) = \sum_i (\mathsf{coeff}_i \cdot \phi_{\mathsf{sid}_i} \leftarrow \mathsf{storage}[\mathsf{sid}_i])$, $\hat{\phi}(\cdot) = \sum_i (\mathsf{coeff}_i \cdot \hat{\phi}_{\mathsf{sid}_i} \leftarrow \mathsf{storage}[\mathsf{sid}_i])$
- Send $(\mathsf{REC}, \mathsf{rid}, \phi(\cdot), \hat{\phi}(\cdot))$ to each party $P_i \in C'_{\text{Corrupt}}$ and eventually to each party $P_i \in C'_{\text{Honest}}$.

Reshare

On receiving $(\mathsf{RESHARE}, \mathsf{rid} := \{(\mathsf{sid}_i, \mathsf{coeff}_i), d', C'\})$ from the Coordinator, where C' is the set of the new committee:

- If the committee C is not the same in every sid, ignore the request.
- If any share ID sid_i is not available in the memory, wait for the share from the dealer.

After receiving all the shares referred by sid_i's in the message,

- Compute polynomials $\phi(\cdot) = \sum_i (\mathsf{coeff}_i \cdot \phi_{\mathsf{sid}_i} \leftarrow \mathsf{storage}[\mathsf{sid}_i])$, $\hat{\phi}(\cdot) = \sum_i (\mathsf{coeff}_i \cdot \hat{\phi}_{\mathsf{sid}_i} \leftarrow \mathsf{storage}[\mathsf{sid}_i])$
- 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 $(\mathsf{LEAK}, \mathsf{rid}, \{\{(\phi_i(j), \hat{\phi}_i(j), c'_i)\}_{i \in C_{\text{Honest}}}\}_{j \in C'_{\text{Corrupt}}})$ to $\mathcal{A}$.
- 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.
- Let $B(\cdot,\cdot)$ and $\hat{B}(\cdot,\cdot)$ be degree d, d' bivariate polynomials defined by $\{B(i,\cdot) = \phi_i(\cdot)\}_{i \in S}$, $\{\hat{B}(i,\cdot) = \hat{\phi}_i(\cdot)\}_{i \in S}$
- Calculate $\phi(\cdot) = B(0,\cdot)$, $\hat{\phi}(\cdot) = \hat{B}(0,\cdot)$, $c' = \{g^{\phi'(i)} h^{\hat{\phi}'(i)}\}_{i \in [|C'|]}$.
- Eventually send $(\mathsf{RESHARE}, \mathsf{rid}, \phi'(i), \hat{\phi}'(i), c')$ to each party $P'_i \in C'$.

Simulator Sim **for** $\mathcal{F}_{\mathbf{DPSS_{HT}}}$

When node P_i enters a new local epoch, sample its private key SK_i.

Share

If the dealer is corrupt, begin by running the $\mathsf{ReliableBroadcast}$ algorithm on all simulated honest nodes.

On receiving $(\mathsf{sid}, \{v_i, \hat{v}_i, c_i, \pi_i\}_{i \in C})$ from a corrupted dealer via $\mathsf{ReliableBroadcast}$:

- Run Algorithm 1 starting at line 201 to verify that all proofs are correct and that the commitments correspond to a degree d polynomial.
- If the checks pass, decrypt $d+1$ of the pairs of shares, interpolate $\phi(\cdot), \hat{\phi}(\cdot)$, and send $(\mathsf{SHARE}, \mathsf{sid}, \phi(\cdot), \hat{\phi}(\cdot), d)$ to $\mathcal{F}_{DPSS_{HT}}$.

If the dealer is honest:

- Intercept each $(\mathsf{SHARE}, \mathsf{sid}, \phi(i), \hat{\phi}(i), c)$ destined for a corrupted P_i.
- Simulate a honest dealer running Algorithm 1. For each recipient P_i...
  - If P_i is corrupted, then use $\phi(i)$ and $\hat{\phi}(i)$ to calculate $v_i, \hat{v}_i, c_i, \pi_i$ as in Algorithm 1 lines 103-104
  - If P_i is honest, set $v_i = \mathsf{Enc}(0), \hat{v}_i = \mathsf{Enc}(0), c_i = c[i]$. For π_i, create a "fake proof" for Π_{ComDecEq} by programming the random oracle as specified by the *zero-knowledge* section of Appendix F
- The simulated honest dealer should run $\mathsf{ReliableBroadcast}$ with this payload. Upon completion, set the internal state of each honest receiver such that it accepts the payload as non-faulty.

Reconstruct

If the coordinator is honest, on receiving $(\mathsf{REC}, \mathsf{rid}, \phi(\cdot), \hat{\phi}(\cdot))$ from $\mathcal{F}$, have the simulated coordinator send $(\mathsf{REC}, \mathsf{rid})$ via $\mathsf{ReliableBroadcast}$.

If the coordinator is corrupted, on a simulated honest party receiving $(\mathsf{REC}, \mathsf{rid})$ from the coordinator's $\mathsf{ReliableBroadcast}$, send $(\mathsf{REC}, \mathsf{rid})$ to $\mathcal{F}$ and await the message $(\mathsf{REC}, \mathsf{rid}, \phi(\cdot), \hat{\phi}(\cdot))$.

When simulated honest party P_i outputs in the coordinator's $\mathsf{ReliableBroadcast}$, set its outbound message queue to include messages sending $\phi(i)$ and $\hat{\phi}(i)$ to every player (including adversary-controlled ones) as per Algorithm 2.

Reshare

If the coordinator is corrupted, on a simulated honest party receiving $(\mathsf{RESHARE}, \mathsf{rid})$ from the coordinator's $\mathsf{ReliableBroadcast}$, send $(\mathsf{REC}, \mathsf{rid})$ to the functionality.

On receiving $(\mathsf{LEAK}, \mathsf{rid}, \{\{(\phi_i(j), \hat{\phi}_i(j), c'_i)\}_{i \in C_{\text{Honest}}}\}_{j \in C'_{\text{Corrupt}}})$ from the functionality

- If the coorinator is honest, have the simulated coordinator initiate $\mathsf{ReliableBroadcast}$ with the message $(\mathsf{RESHARE}, \mathsf{rid})$
- Begin simulating honest parties in C (once they have heard from the coordinator) resharing their shares to C' via ACSS as in Algorithm 3, using fake proofs and encryptions of 0 for shares destined for honest nodes.
- Similarly, begin running MVBA on simulated honest nodes of C' once they have heard from the coordinator and have them start listening for ACSS messages from corrupted nodes, responding as they would in Algorithm 3.

On a simulated honest party in C' outputting in MVBA:

- Input any polynomials chosen by $\mathcal{A}$ which made it into the MVBA output, and then send the output set to the functionality