Optimistic Asynchronous Dynamic-committee Proactive Secret Sharing

Bin Hu[◇,§] Jianwei Liu[◇,§] Zhenliang Lu[⋆,§] Qiang Tang[†,§] Zhuolun Xiang[•,§] Zongyang Zhang[◇,§]
[◇]*Beihang University* [⋆]*City University of Hong Kong* [†]*The University of Sydney* [•]*Aptos Labs*
{hubin0205, liujianwei, zongyangzhang}@buaa.edu.cn, zhenliang.lu@cityu.edu.hk
qiang.tang@sydney.edu.au, xiangzhuolun@gmail.com

***Abstract*—Dynamic-committee Proactive Secret Sharing (DPSS) has gained increased attention for its ability to dynamically update the shareholder committees and refresh secret shares, even against adversaries that gradually corrupt all nodes. However, existing state-of-the-art asynchronous DPSS protocols suffer from significant $\mathcal{O}(n^3)$ message complexity and $\mathcal{O}(\lambda n^3)$ communication complexity, where λ denotes the security parameter and n is the committee size.**

In this paper, we distinguish optimistic-case and worst-case scenarios based on node behaviors and network conditions, thus reducing the redundant communication overhead of asynchronous DPSS. Under the trusted setup assumption, we achieved an $\mathcal{O}(n^2)$ message complexity in all scenarios. Additionally, our protocol has an $\mathcal{O}(\lambda n^2)$ communication complexity in the optimistic case, where all nodes are honest and the network is synchronous, and $\mathcal{O}(\lambda n^3)$ communication complexity in the worst case. We also propose two strategies to eliminate the strong trusted setup assumptions, and the asymptotic performance still surpasses the state-of-the-art protocols. For committee sizes of 4 to 400, the estimated concrete communication cost of our DPSS is 19–100x (resp., 8–14x) smaller in the optimistic case (resp., worst case) compared to LongLive (USENIX Security '23). Experiments in AWS show that our DPSS achieves a latency of 1.9–8 seconds for committee sizes from 4 to 64. Single-machine benchmarks reveal a (computational) runtime reduction of up to 44%.

1. Introduction

Secret sharing is a well-established cryptographic primitive that has been extensively researched. The concept of secret sharing involves a dealer who possesses a secret s. The dealer's objective is to distribute secret shares among multiple nodes in such a way that a threshold number of shares is sufficient to reconstruct the original secret, whereas any subset of nodes less than the threshold gains no information about the secret. Secret sharing has found extensive applications in various domains, including Asynchronous Distributed Key Generation (ADKG) [1], [2], Multi-Party Computation (MPC) [3], Byzantine Fault Tolerant (BFT) protocols [4], [5], and many others.

However, in the context of long-running applications, standard secret sharing faces the following challenges: (1) There may exist a strengthened adversary that is able to gradually compromise nodes, thus eventually accumulating sufficient shares to reconstruct the secret; (2) A long-lived system may encounter requests for membership alternation.

Recent works [6], [7], [8], [9] have been trying to tackle the above issues with asynchronous Dynamic-committee Proactive Secret Sharing (DPSS), where time is divided into multiple epochs, with shareholder committees potentially varying across the epochs, and secret shares being refreshed periodically. Since many state-of-the-art BFT protocols [4], [5] rely on threshold cryptosystems, asynchronous DPSS may gain its advantage in long-running BFT-based blockchain systems due to its robustness and ability to reconfigure shareholder committees, even in the presence of a strengthened adversary, node churn, and network fluctuation.

Although significant progress has been made on asynchronous DPSS, the recent state-of-the-art works "Long Live The Honey Badger" [9] (referred to as LongLive hereafter) and DyCAPS [8] still incur $\mathcal{O}(\lambda n^3)$ communication complexity and $\mathcal{O}(n^3)$ message complexity, where λ denotes the security parameter and n is the committee size. Here, communication complexity is the expected total number of bits transmitted by honest nodes, while message complexity refers to the expected total number of messages exchanged among honest nodes.

We observe that LongLive and DyCAPS are suboptimal in terms of message complexity: Rambaud and Uban's work [7] (referred to as Y-VSS) achieves $\mathcal{O}(n^2)$ message complexity, despite its higher communication complexity. Besides, almost all existing protocols (except Shanrang [6], see below) do not differentiate between optimistic and worst cases. That is, when all nodes are honest and the network is synchronous[1], such that all necessary messages are received before executing the next step, the performance remains the same as in the worst case, where Byzantine faults exist. Regarding Shanrang, despite its introduction of a two-path strategy, it experiences suboptimal fault tolerance of $n/4$ and exhibits $\mathcal{O}(\lambda n^3 \log n)$ optimistic-case communication com-

§ This is the full version of the conference paper published in the proceedings of IEEE S&P 2026. Authors are listed alphabetically, Bin Hu & Zhenliang Lu led the effort with equal contributions. Zongyang Zhang is the corresponding author.

1. For example, in permissioned systems with well-provisioned cloud environments, nodes are typically honest and network conditions are stable.

TABLE 1: Performance metrics of asynchronous DPSS

Protocol	Tolerance	Dynamic	Communication Complexity		Message Complexity		Time Complexity⋆	Trusted Setup¶
			Optimistic†	Worst‡	Optimistic†	Worst‡		
Cachin et al. [10]	$t < n/3$	×	$\mathcal{O}(\lambda n^4)$	$\mathcal{O}(\lambda n^4)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	✓
Zhou et al. [11]	$t < n/3$	✓	$\mathcal{O}(\exp(\lambda n))$	$\mathcal{O}(\exp(\lambda n))$	$\mathcal{O}(\exp(n))$	$\mathcal{O}(\exp(n))$	$\mathcal{O}(1)$	×
Shanrang [6]	$t < n/4$	✓	$\mathcal{O}(\lambda n^3 \log n)$	$\mathcal{O}(\lambda n^4)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(\log n)$	✓
Y-VSS [7]	$t < n/2$	✓	$\mathcal{O}(\lambda n^4)$	$\mathcal{O}(\lambda n^4)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	×
DyCAPS [8]	$t < n/3$	✓	$\mathcal{O}(\lambda n^3)$	$\mathcal{O}(\lambda n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	✓
LongLive (hbACSS) [9]	$t < n/3$	✓	$\mathcal{O}(\lambda n^3)$	$\mathcal{O}(\lambda n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	✓
Our DPSS (§ 7)	$t < n/3$	✓	$\mathcal{O}(\lambda n^2)$	$\mathcal{O}(\lambda n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	✓
LongLive (DC-ACSS) [9]	$t < n/3$	✓	$\mathcal{O}(\lambda n^3)$	$\mathcal{O}(\lambda n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	×
Our DPSS (§ 9)	$t < n/3$	✓	$\mathcal{O}(\lambda n^2 \log n)$	$\mathcal{O}(\lambda n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	×

† Optimistic case: all nodes are honest and the network is synchronous. ‡ Worst case: up to t nodes are Byzantine faulty and the network is asynchronous.
⋆ Time complexity: the expected number of asynchronous communication rounds of the protocol.
¶ LongLive demonstrates that employing ADKG protocols can eliminate the need for trusted setups for threshold signatures, without affecting the asymptotic performance of its DC-ACSS-based version. However, the hbACSS-based LongLive still requires trusted setups to generate the powers of tau (t-SDH parameters) for KZG commitments.

plexity. Hence, there is redundancy in the optimistic case for all existing asynchronous DPSS protocols. In addition, both LongLive and DyCAPS rely on a trusted setup to initialize the underlying cryptosystems for each epoch. Although LongLive proposes an approach to circumvent this trusted setup by employing an ADKG protocol, it makes DPSS have at least the same complexity as ADKG, which has a cost of $\mathcal{O}(\lambda n^3)$ in communication complexity and $\mathcal{O}(n^3)$ in message complexity. The above observations naturally lead to the following question:

Can we decrease the message complexity of asynchronous DPSS, while also reducing the communication complexity in the optimistic case? Furthermore, is it possible to weaken the requirement for a trusted setup in each epoch?

Our contributions. We address the aforementioned question by introducing an optimistic asynchronous DPSS protocol. Here is a summary of our contributions:

- We reduce the message complexity of asynchronous DPSS from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^2)$ in both optimistic and worst cases, with a trusted setup assumption. Without a trusted setup, the optimistic-case message complexity remains $\mathcal{O}(n^2)$, while the worst-case message complexity increases to $\mathcal{O}(n^3)$, which still aligns with the state-of-the-art [9].
- With a trusted setup, we reduce the communication complexity to $\mathcal{O}(\lambda n^2)$ in the optimistic case, whereas the worst-case communication complexity remains $\mathcal{O}(\lambda n^3)$. For a committee size range of 4 to 400, the estimated concrete optimistic-case (resp., worst-case) communication cost is about 19–100x (resp., 8–14x) smaller than the state-of-the-art [9].
- We present two strategies to eliminate the strong assumption of per-epoch trusted setup. The first approach achieves a one-time trusted setup with a read-only bulletin board. The second approach eliminates the need for any trusted setup. The two approaches achieve communication complexities of $\mathcal{O}(\lambda n^2)$ and $\mathcal{O}(\lambda n^2 \log n)$ in the optimistic case, respectively, and their worst-case communication complexity is $\mathcal{O}(\lambda n^3)$. The estimated concrete communication cost of our no-trusted-setup DPSS outperforms the state-of-the-art [9] (with per-epoch trusted setup) as early as $n > 13$ in the optimistic scenario.
- We implemented our DPSS protocol and deployed it on Amazon EC2 t2.medium instances across the globe.[2] Experimental results show that when the committee size n grows from 4 to 127, the transfer of shares between two committees of equal size takes around 1.9–20 seconds. Especially, for $n \leq 64$, our DPSS completes in less than 8 seconds. In single-machine benchmarks, our protocol saves up to 44% computation time when compared with the state-of-the-art [9], given $n \leq 52$.

Remark 1. The above improvements in message and optimistic-case communication complexities do not compromise the other asymptotic performance matrices of asynchronous DPSS.

Remark 2. Improving the efficiency of DPSS protocols directly benefits the applications with frequent committee churn. However, even in systems where epochs are long, reducing the message and communication complexities of DPSS is still promising. It could mitigate the congestion during reconfiguration, which must be completed before the next committee can take effect.

2. Technical Overview

An asynchronous DPSS protocol comprises three sub-protocols: sharing, handoff, and reconstruction, referred to as DPSS.Share, DPSS.Handoff, and DPSS.Recon, respectively. Time is divided into epochs. In the initial epoch, DPSS.Share is executed, wherein a dealer distributes shares

2. Available at https://github.com/opDPSSTeam/Implementation.

of a secret s among the nodes in the first committee $\mathcal{U}^1$. The DPSS.Recon sub-protocol is responsible for reconstructing the secret s whenever invoked. The DPSS.Handoff sub-protocol is the key component of DPSS. Its primary objective is to ensure that a new committee obtains refreshed shares from an old committee while maintaining the confidentiality of the secret.

To simplify the expression, in epoch $e > 1$, we refer to the nodes in the old committee $\mathcal{U}^{e-1}$ and new committee $\mathcal{U}^e$ as "old nodes" and "new nodes", respectively. We use n_e to represent the size of $\mathcal{U}^e$ and t_e is the maximum number of faulty nodes that $\mathcal{U}^e$ can tolerate.[3] Given $n_e = 3t_e + 1$, a typical DPSS.Handoff protocol in epoch e consists of the following three phases:

- *Reshare:* Each old node reshares its old share with the new committee, ensuring that every new node outputs a valid sub-share of the old share.
- *Select:* The new committee collectively selects at least $t_{e-1} + 1$ valid resharing instances.
- *Compute:* Each new node computes its new share using the sub-shares obtained from the selected resharing instances.

Existing asynchronous DPSS.Handoff protocols, such as those proposed by Hu et al. [8] and Yurek et al. [9], employ $\mathcal{O}(n)$ parallel Asynchronous Complete Secret Sharing (ACSS) instances for the *Reshare* phase and a single Multi-valued Validated Byzantine Agreement (MVBA) instance for the *Select* phase. The global predicate P_{MVBA} in MVBA ensures that, for each selected ACSS instance, at least one honest node outputs a valid share. Moreover, the completeness property of ACSS guarantees that if one honest node outputs a valid share, then all honest nodes will eventually output valid shares. Therefore, all honest new nodes eventually obtain valid shares from all the selected ACSS instances. Finally, in the *Compute* phase, the honest new nodes locally compute the refreshed shares using the outputs of selected ACSS instances.

Bottlenecks. We observe that ACSS, while being a robust component in DPSS.Handoff, has been a dominant bottleneck in message and communication complexities. Specifically, many ACSS protocols [12], [13] use exactly the same communication pattern as Bracha's reliable broadcast [14], consuming at least $\mathcal{O}(n^2)$ messages due to the Dolev-Reischuk bound [15]. Hence, $\mathcal{O}(n)$ parallel ACSS instances incur $\mathcal{O}(n^3)$ message complexity and $\mathcal{O}(\lambda n^3)$ bits of communication. These costs align with the performance of the state-of-the-art [8], [9]. Following the identified bottlenecks, our approach to designing a more efficient asynchronous handoff protocol is as follows.

Reducing message complexity by provable sharing and batched recovery. We replace ACSS with a Weak Provable ACSS (wpACSS) protocol, consisting of a provable sharing sub-protocol and an optional recovery sub-protocol. The key insight is that we only need to call the recovery sub-protocol in an on-demand manner, ensuring that every honest node has the outputs of all selected resharing instances to compute the refreshed shares. The recovery sub-protocol could be further batched, thereby reducing the overall message complexity of DPSS.Handoff to $\mathcal{O}(n^2)$.

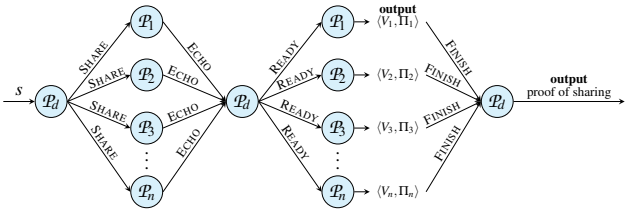

Figure 1: Message flow of provable sharing wpACSS.Share.

The provable sharing is inspired by provable broadcast [5], [16], a primitive in which a sender disseminates a value (in plaintext rather than secret) and outputs a proof certifying that a sufficient number of parties have accepted it. For our provable sharing, the dealer shares a secret and generates a proof of sharing, that guarantees the receipt of valid shares by at least $n-2t$ honest nodes, where $n = 3t+1$. The key difference between the two primitives is that the former ensures that sufficiently many recipients accept the same value, whereas the latter guarantees that each accepting recipient obtains a distinct share, which individually reveals no information about the secret. As illustrated in Figure 1, the dealer shares its secret via SHARE messages. Upon receiving a valid secret share from the dealer, each node responds with an acknowledgment of receipt. The dealer aggregates the responses from $2t + 1$ nodes and forms an intermediate proof, which demonstrates that at least $t + 1$ honest nodes have received valid shares. The proof is then multicast to all nodes. When a node receives this proof, it accepts the contents of the SHARE message as its output. Meanwhile, it sends back a confirmation to the dealer, indicating that it has output in the provable sharing. Finally, the dealer aggregates the confirmations from $2t + 1$ nodes and forms a proof of sharing. Such a proof ensures that at least $t + 1$ honest nodes have output. Since each round of communication involves n messages, the provable sharing process maintains an efficient $\mathcal{O}(n)$ message complexity.

The optional recovery sub-protocol enables a node to recover its missing share, as long as a valid proof of sharing exists. When a node multicasts a recovery request, at least $t+1$ honest nodes respond with HELP messages, leading to an $\mathcal{O}(n)$ message complexity. For each sharing instance, there are at most t honest nodes missing their deserved shares. As n parallel provable sharing instances will be employed during handoff, if we do the recovery naively, there will be at most $\mathcal{O}(n^2)$ recovery instances in the worst case, resulting in a cubic message complexity. To obtain a handoff protocol with $\mathcal{O}(n^2)$ message complexity, the key observation is that we can batch the recovery requests and responses. Namely, we let each node aggregate the indexes of missing shares into one request, with each index referring to a dealer. Similarly, every honest node responds with a single message to help the recovery. In this way, each batched recovery instance helps a node to recover $\mathcal{O}(n)$

3. We consider $\mathcal{O}(n_e) = \mathcal{O}(t_e) = \mathcal{O}(n)$ for any $e \in \mathbb{N}^*$.

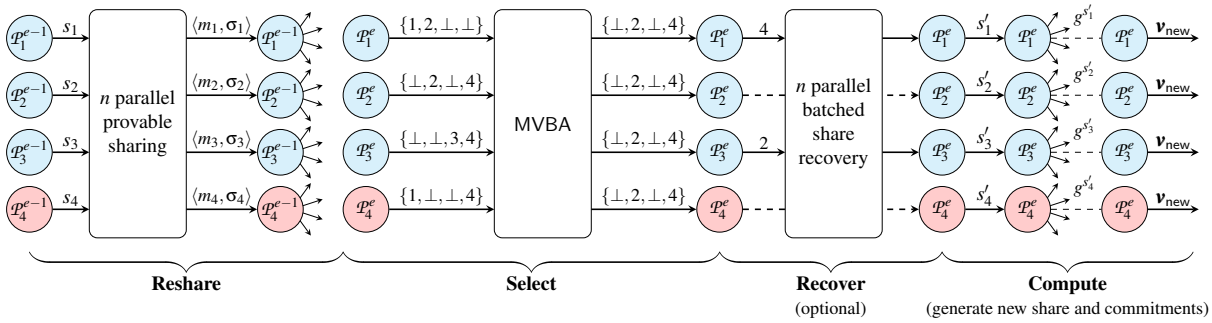

Figure 2: Overview of our DPSS.Handoff between two committees with 4 nodes, where nodes $\mathcal{P}_4^{e-1}$ and $\mathcal{P}_4^e$ are corrupted.

missing shares at the cost of $\mathcal{O}(n)$ message complexity. Hence, even if $\mathcal{O}(n)$ batched recovery instances are invoked, the message complexity is still $\mathcal{O}(n^2)$. Remarkably, the batching in wpACSS also reduces the total encoded content by minimizing the per-message encapsulation overhead.

As shown in Figure 2, our DPSS.Handoff comprises multiple components: n parallel instances of provable sharing, an instantiation of MVBA, n parallel batched recovery instances (optional), and the computation phase for the new shares and commitments. As mentioned above, every provable sharing and batched recovery instance has $\mathcal{O}(n)$ message complexity. Besides, we employ Lu et al.'s MVBA [17] with $\mathcal{O}(n^2)$ message complexity. By our design, the final computation phase also incurs $\mathcal{O}(n^2)$ message complexity. Hence, we reduce the overall message complexity to $\mathcal{O}(n^2)$.

Reducing optimistic-case communication complexity by minimizing message lengths. Although we have made the share recovery optional, it does not help reduce the communication complexity of handoff: if each SHARE message has a length of $\mathcal{O}(\lambda n)$ bits, n provable sharing instances still cost $\mathcal{O}(\lambda n^3)$ communication complexity.

It is noteworthy that the SHARE messages in most existing ACSS protocols are typically of linear size. For example, in AVZ21 [13], a SHARE message contains one secret share and n recovery elements. The latter serve as masks during the recovery, ensuring that a node can only recover its own share and learns no knowledge of the other shares. The ACSS protocol in [1] and its variant in LongLive [9] also have $\mathcal{O}(\lambda n)$-sized SHARE messages, as they gather the encryptions of n shares into one message. Hence, a naive provable sharing protocol derived from these ACSS protocols will also have linear-sized SHARE messages.

To improve the optimistic-case communication complexity, we propose a new design of provable sharing with $\mathcal{O}(\lambda)$-sized SHARE messages. We are inspired by the Verifiable Secret Sharing with Recovery (VSSR) protocol proposed by Basu et al. [18]. The main idea of VSSR is to interpolate Distributed Pseudorandom Function (DPRF) evaluations into $\ell = \lceil n/t \rceil$ recovery polynomials, instead of sampling n recovery polynomials as in [13]. The evaluations of recovery polynomials serve as the masks for the share recovery. Now, a SHARE message contains one secret share and ℓ recovery elements, where ℓ is constant. Furthermore, every recovery element is accompanied by a constant-sized verifiable proof.

However, applying the idea of VSSR in our provable sharing is not straightforward. In the original VSSR protocol [18], the distribution of public information of DPRF among the nodes is not explicitly specified. If we let the dealer directly multicast the $\mathcal{O}(\lambda n)$-sized DPRF public keys $\{dpk_i\}_{i\in[n]}$ (used to verify DPRF contributions) to all nodes, it would result in a communication complexity of at least $\mathcal{O}(\lambda n^2)$ for each provable sharing. To address this issue, we carefully design our provable sharing protocol to incorporate the DPRF keys within the SHARE messages, keeping each message $\mathcal{O}(\lambda)$-sized. Since $\mathcal{O}(n)$ messages are consumed, we achieve a provable sharing protocol with $\mathcal{O}(\lambda n)$ communication complexity.

Now that we have an efficient provable sharing protocol, the next task is to avoid old nodes resharing incorrect shares. We let the new nodes use an $\mathcal{O}(\lambda n)$-sized commitment vector to validate the proofs of sharing. The vector consists of the commitments to all old shares. It is initially disseminated through a reliable broadcast instance during DPSS.Share, and is re-generated at the end of each handoff. However, re-generating the commitment vector is non-trivial, as we have to maintain an $\mathcal{O}(\lambda n^2)$ optimistic-case communication complexity. We propose a two-path strategy. In the optimistic path, each new node multicasts the commitment to its refreshed share and collects commitments from others, incurring $\mathcal{O}(\lambda n^2)$ bits of communication. A new node outputs a new vector if the verification of new commitments passes. Otherwise, it fallbacks into a pessimistic but robust path, where nodes exchange additional verifiable information, at the cost of $\mathcal{O}(\lambda n^3)$ communication complexity.

3. Related Work

In a long-running system based on secret sharing, there may exist a strengthened adversary that gradually corrupts the nodes. Such an adversary could eventually accumulate the necessary number of shares to reconstruct the secret, thus compromising confidentiality. To address this risk, Herzberg et al. [19] introduce the concept of Proactive Secret Sharing (PSS), which involves periodic share refreshes to constrain the power of adversary. That is, to obtain the secret value, the adversary has to corrupt more than a threshold of nodes within a limited time. Desmedt and Jajodia [20] extend PSS to the dynamic committee setting, where nodes are allowed

to join and leave dynamically. They introduce the concept of DPSS, aimed to adapt to changes in committee composition and ensure the continuous security of long-running systems.

Non-asynchronous DPSS. CHURP [21] is a synchronous protocol that achieves a communication complexity of $\mathcal{O}(\lambda n^2)$ bits and a message complexity of $\mathcal{O}(n^2)$ in the all-honest case. However, when faced with crash or Byzantine faults, its communication and message complexity increase to $\mathcal{O}(\lambda n^3)$ and $\mathcal{O}(n^3)$, respectively. Goyal et al. [22] propose a synchronous protocol that enhances CHURP by a factor of $\mathcal{O}(n)$ in the batched setting. In the single-secret setting, their protocol has the same communication cost as CHURP. COBRA [23] is a state-of-the-art partially synchronous DPSS protocol that achieves an improvement over Schultz-MPSS [24]. Schultz-MPSS has a communication complexity of $\mathcal{O}(\lambda n^4)$ and a message complexity of $\mathcal{O}(n^3)$ across all cases. In contrast, COBRA achieves a communication complexity of $\mathcal{O}(\lambda n^3)$ and a message complexity of $\mathcal{O}(n^2)$ in the optimistic case.

Asynchronous DPSS. The asynchronous setting has garnered significant interest in recent years. Table 1 shows the performance metrics of related asynchronous DPSS protocols. The earliest related work on asynchronous PSS is proposed by Cachin et al. [10], which only supports the static committee setting. Zhou et al. [11] introduce the first DPSS protocol, but it has an exponential communication cost. Recently, Shanrang [6] has made remarkable improvements to the asynchronous DPSS protocol. In the case of all honest participants, its communication complexity is reduced to $\mathcal{O}(\lambda n^3 \log n)$, and when facing adversaries, its communication complexity grows to $\mathcal{O}(\lambda n^4)$. However, this improvement comes at the expense of suboptimal resilience, i.e., $t < n/4$. A recent work Y-VSS [7] tolerates up to $1/2$ corrupted nodes in an asynchronous network. However, it assumes one of the following conditions: (1) the dealer is honest, (2) the initial network functions synchronously, or (3) the first committee has at most $1/3$ corrupted nodes. Additionally, it requires non-overlapping old and new committees, which imposes additional conditions compared to the assumptions in our paper.

The current state-of-the-art asynchronous DPSS protocols are LongLive [9] and DyCAPS [8], both achieving $\mathcal{O}(\lambda n^3)$ communication complexity and $\mathcal{O}(n^3)$ message complexity. LongLive follows the reshare-then-agree approach by Cachin et al. [10], and it simultaneously supports high-threshold and batched DPSS. On the other hand, DyCAPS utilizes bivariate polynomials to achieve high-threshold DPSS. However, DyCAPS loses the support for batched DPSS, even though it has a lower concrete communication overhead in the non-batched setting. As mentioned in Section 1, both LongLive and DyCAPS do not distinguish the performance between optimistic and pessimistic cases, and their message complexity is suboptimal when compared with Y-VSS [7].

The recent work by Lu et al. [25] present a framework to handle large-sized data using DPSS. They introduce dynamic-committee proactive information dispersal (DPID), which can be viewed as a relaxation of DPSS that omits secrecy. They encrypt the large-sized data and use DPID to refresh the ciphertext shares, and invoke a parallel DPSS instance to refresh the encryption key shares. While they focus on handling large payloads, we optimize the efficiency of the DPSS primitive itself. Our protocol is complementary to their framework. By integrating our construction, the λn^3 term in their optimistic-case communication complexity can be improved to λn^2 or $\lambda \kappa n^2$, where κ is the statistical security parameter.

4. Model and Design Goal

4.1. System Model

Established identities. The i-th node within committee $\mathcal{U}^e$ is assigned a distinct identity $\mathcal{P}_i^e$. An honest node is considered active in epoch e if it is a member of committee $\mathcal{U}^e$. If a node is active in two different epochs, it will have two distinct identities. The relationship between two committees can range from overlap to complete disjointedness.

Trusted setup. For each epoch, a trusted setup is required for initializing the cryptosystems.[4] This entails generating threshold key shares for the new committee, which are used to generate threshold signatures. The trusted setup is also responsible for generating the public parameters (e.g., powers of tau [26]) for the commitment schemes. We assume that public parameters are available to all nodes, and every node in epoch e' has access to all public parameters pertaining to all previous epochs $e < e'$.

Adversary model. We assume a static probabilistic polynomial time (PPT) adversary. It can corrupt up to $t_e < n_e/3$ nodes in each committee $\mathcal{U}^e$ at the beginning of epoch e, where n_e is the number of nodes in $\mathcal{U}^e$. A corrupted node is Byzantine faulty [27], which means it may misbehave arbitrarily. Throughout this paper, we assume $n_e = 3t_e + 1$.

If a node is honest in epoch $e-1$ ($e > 1$) and gets corrupted in epoch e, then the adversary can only access its private information assigned for epoch e, and the private information within epoch $e-1$ remains unknown (e.g., encrypted). Nonetheless, the adversary can delete previous information held by the corrupted node. Consequently, a corrupted node may lose its old share and thus fail to participate in the handoff protocol. Hence, if there is an overlap between committees $\mathcal{U}^{e-1}$ and $\mathcal{U}^e$, we assume that at least $t_{e-1}+1$ nodes in $\mathcal{U}^{e-1}$ still have their old shares, i.e., not corrupted in both epochs $e-1$ and e.

We adopt the approach outlined in Schultz-MPSS [24] for defining epochs and imposing constraints on the adversary, which helps overcome the impossibility result presented by Alexandru et al. [28]. For a more comprehensive understanding of the model, additional information is available in Appendix A of the paper [9].

Network model. We assume authenticated and point-to-point channels, ensuring that recipients can verify senders'

4. We will discuss in Section 9 both the necessity of this assumption and how it can be relaxed to one-time trusted setup or transparent setup.

identities. The adversary can control the order of messages but cannot observe or tamper with the contents.

Additionally, we consider that our protocol operates under the same network conditions as described in [29], encompassing the optimistic case (referred to as the common case in [29]) and the worst case. For epoch e, the optimistic case signifies that all nodes are honest and the network is synchronous, while the worst case denotes the presence of up to t_e corrupted nodes and the network is asynchronous. Here, synchrony means that messages among honest nodes are delivered within a known time interval Δ, whereas asynchrony implies that all messages will be delivered without any time constraints. The optimistic case is realistic in many practical deployments, such as permissioned environments with well-provisioned cloud or datacenter settings. In such scenarios, our protocol is expected to operate in the optimistic mode for the majority of executions, with fallback mechanisms only triggered under exceptional conditions.

Furthermore, in both cases, once a node completes the handoff, it waits for a specified period (measured by its local clock) before joining the next epoch as part of the old committee. It is important to note that nodes may begin the next epoch at different times if their completion times differ. When we refer to the optimistic case, we explicitly assume all nodes have received necessary messages before proceeding to the next step, and we omit the parameter Δ.

4.2. Design Goal

Our goal is to design an efficient asynchronous DPSS protocol, which consists of three sub-protocols: sharing, handoff, and reconstruction. The sharing sub-protocol DPSS.Share is only invoked in epoch $e = 1$, whereas the reconstruction sub-protocol DPSS.Recon can be invoked at any time after sharing. Between epochs $e - 1$ and e ($e > 1$), the handoff sub-protocol DPSS.Handoff is executed to facilitate the transfer of shares from the old committee to the new committee. Our main emphasis is on the efficiency of the handoff protocol, which is executed periodically and has a significant message and communication cost. The details of DPSS.Share and DPSS.Recon are deferred to Appendix C. Besides, we note that committee selection method is orthogonal to the design of DPSS protocols. For example, committees can be selected by a trustworthy party, or via publicly verifiable mechanisms (e.g., randomness beacons).

Formally, an asynchronous DPSS.Handoff protocol with epoch e satisfies the following properties with all but negligible probability.

- *Termination.* If all honest nodes in committees $\mathcal{U}^{e-1}$ and $\mathcal{U}^e$ invoke DPSS.Handoff, then all honest nodes in $\mathcal{U}^{e-1}$ eventually halt, and all honest nodes in $\mathcal{U}^e$ eventually have outputs.
- *Correctness.* The secret s stays invariant across the executions of DPSS.Handoff.
- *Secrecy.* Given a uniformly random secret s, a static PPT adversary does not gain any additional knowledge about s from DPSS.Handoff beyond g^s.

In this paper, we make the assumption that all honest nodes within the first committee $\mathcal{U}^1$ have already produced valid shares and commitments from DPSS.Share. We sometimes use the handoff sub-protocol DPSS.Handoff to represent the overall DPSS protocol when the context is unambiguous.

Throughout this paper, we assume that the secret s is uniformly random. It is worth noting that many typical applications of DPSS are not constrained by this uniformly random requirement, including: (1) Threshold cryptosystems, where secret keys are inherently uniformly random. (2) The "BMR Escape Hatch" described in [9], which relies on uniformly random values as secrets. (3) Cloud storage of confidential files, such as in COBRA [23], where files can be encrypted, and the uniformly random encryption keys can be protected using DPSS. We will give more details on how to protect a non-uniform secret using DPSS in Section 9.

5. Notations and Preliminaries

5.1. Notations

Let $\mathbb{G}$ denote a multiplicative cyclic group of prime order q, where the discrete logarithm problem is assumed to be hard, and g is a generator of $\mathbb{G}$. The secret value is referred to as $s \in \mathbb{Z}_q$. For an integer $a \in \mathbb{N}^*$, $[a]$ denotes the ordered set $\{1, 2, \cdots, a\}$. Throughout this paper, we use $|\mathcal{S}|$ to capture the size of a set $\mathcal{S}$. The cryptographic security parameter is denoted as λ, which captures the length of a (threshold) signature or a hash value.The epoch number is denoted as e, where $e \in \mathbb{N}^*$. The committee in the e-th epoch is represented as $\mathcal{U}^e = \{\mathcal{P}_i^e\}_{i\in[n_e]}$, where $\mathcal{P}_i^e$ is the i-th node and $n_e = |\mathcal{U}^e|$. When the context is unambiguous, we sometimes use $\mathcal{P}_i$ and $\mathcal{P}_i'$ to refer to nodes $\mathcal{P}_i^{e-1}$ and $\mathcal{P}_i^e$, respectively. Similarly, we will use $\mathcal{U}$ to represent the old committee and $\mathcal{U}'$ to represent the new committee. Besides, t_e is the maximum number of corrupted nodes in epoch e. The threshold full signature of a message m is denoted as σ_m, and $\sigma_{m,i}^*$ is a signature share (also called partial signature) produced by $\mathcal{P}_i$. We use PC (resp., VC) to denote a polynomial (resp., vector) commitment, and w (resp., π) refers to a witness of a polynomial evaluation (resp., opening of a vector element). The detailed symbols used in this paper are listed in Table 2.

5.2. Preliminaries

Our design employs several cryptographic primitives. Here, we only provide the abstractions of these protocols, and the formal definitions can be found in Appendix A.

Polynomial commitment (PCom) [30] allows a node to commit to a polynomial and give witnesses of its evaluations. It consists of four algorithms: PCom.Setup, PCom.Commit, PCom.Witness, and PCom.VerifyEval. PCom.Setup generates public parameters pp, which is sometimes omitted for simplicity. PCom.Commit generates a commitment PC_ϕ for the polynomial $\phi(x) \in \mathbb{Z}_q[x]$. For $x = i$, PCom.Witness outputs an evaluation $\phi(i)$ and a

TABLE 2: Notations used in this paper

Notation	Description
λ	Security parameter
$\mathbb{G}$	Multiplicative cyclic group of prime order q
g	Generator of group $\mathbb{G}$
s	Secret value
g^s	Commitment to secret s
e	Epoch number
$\mathcal{U}^e$	Shareholder committee in epoch e (sometimes simplified as $\mathcal{U}$ or $\mathcal{U}'$)
$\mathcal{P}_i^e$	i-th node in $\mathcal{U}^e$, sometimes simplified as $\mathcal{P}_i$ or $\mathcal{P}_i'$
s_i, s_i'	Secret shares of $\mathcal{P}_i$ and $\mathcal{P}_i'$
n_e	Size of committee $\mathcal{U}^e$, sometimes simplified as n or n'
t_e	Threshold in epoch e, sometimes simplified as t or t'
$\boldsymbol{v}$	Vector
$VC_{\boldsymbol{v}}$	Vector commitment to a vector $\boldsymbol{v}$
$\pi_{\boldsymbol{v}}^i$	Opening proof of the i-th element in vector $\boldsymbol{v}$
PC_ϕ	Polynomial commitment to a polynomial ϕ
w^i	Witness of the i-th evaluation of polynomial ϕ
$\mathcal{H}$	Hash function
ID	Session identifier
$\Pi[\mathsf{ID}](x)$	Instance of Π identified by ID with input x
tsk, tpk	Secret and public keys in threshold signature (TS)
tsk_i, tpk_i	Node $\mathcal{P}_i$'s secret and public key shares in TS
$\sigma_{m,i}^*$	Signature share on message m produced by $\mathcal{P}_i$
σ_m	Threshold full signature of message m
dsk, dpk	Secret and public keys in distributed pseudorandom function (DPRF)
dsk_i, dpk_i	Node $\mathcal{P}_i$'s secret and public key shares in DPRF
$\boldsymbol{v}_{\mathsf{old}}, \boldsymbol{v}_{\mathsf{new}}$	Commitment vectors to the old and new shares

witness w^i. PCom.VerifyEval is then used to verify the correctness of the evaluation. In this paper, we utilize the KZG scheme [30] for PCom because of its constant commitment and witness sizes.

Verifiable secret sharing (VSS) [31] enables a dealer to distribute a secret among n nodes, allowing each node to independently verify the correctness of its share. It involves four algorithms: VSS.Share, VSS.Vrfy, VSS.Recon, and VSS.MakeSecret. Given a threshold t, a committee size n, a secret s, and a security parameter λ, VSS.Share produces a polynomial commitment PC_s and n share-witness pairs $\langle s_i, w_s^i \rangle_{i\in[n]}$. VSS.Vrfy verifies the shares. VSS.Recon reconstructs the secret from a sufficient number of shares. Lastly, taking as inputs a threshold t, an index set I, and a set of values $\{y_i\}_{i\in I}$, where $|I| \leq t$, VSS.MakeSecret produces a degree-t polynomial $\phi(x)$ by locally sampling $t+1-|I|$ points, such that $\phi(i) = y_i$ for every $i \in I$. The above algorithms are all non-interactive, making them oblivious to the network model.

Vector commitment (VCom) [32] allows a node to commit to a vector and later reveal the vector elements for a specific set of positions. It comprises four algorithms: VCom.Setup, VCom.Commit, VCom.Open, and VCom.Vrfy. Given a vector size n and a security parameter λ, VCom.Setup generates public parameters pp (sometimes omitted for simplicity). VCom.Commit produces a commitment $VC_{\boldsymbol{v}}$ to vector $\boldsymbol{v}$. VCom.Open yields the i-th element v_i in $\boldsymbol{v}$ and an opening proof $\pi_{\boldsymbol{v}}^i$. Taking as inputs a commitment $VC_{\boldsymbol{v}}$, an index i, an element v_i, and a proof $\pi_{\boldsymbol{v}}^i$, VCom.Vrfy outputs 1 iff v_i is the i-th element of $\boldsymbol{v}$. In this paper, we employ Pointproofs [32] for VCom, due to its constant commitment and proof sizes.

Threshold signature (TS) [33], [34] enables a quorum of nodes to collaboratively sign a message. It includes five algorithms: TS.KeyGen, TS.SignShare, TS.VrfyShare, TS.Combine, and TS.Vrfy. Taking as inputs a threshold t and a committee size n, TS.KeyGen generates threshold signing key pairs $\{tpk, \langle tpk_i, tsk_i\rangle_{i\in[n]}\}$. $\mathsf{TS.SignShare}_t$ creates a signature share $\sigma_{m,i}^*$ for a message m using $\mathcal{P}_i$'s secret key share tsk_i, while $\mathsf{TS.VrfyShare}_t$ verifies a signature share by $\mathcal{P}_i$'s public key share tpk_i and the message m. $\mathsf{TS.Combine}_t$ generates a full signature from at least $t+1$ valid signature shares, and $\mathsf{TS.Vrfy}_t$ is employed to verify a full signature using tpk. We sometimes omit the key pairs for simplicity.

Distributed pseudo-random function (DPRF) [35] enables a quorum of nodes to jointly evaluate a pseudo-random function. It has six algorithms: DPRF.Init, DPRF.Contrib, DPRF.VrfyContrib, DPRF.Combine, DPRF.Eval, and DPRF.Vrfy. Taking as inputs a threshold t, a committee size n, and a security parameter λ, DPRF.Init not only initializes the DPRF key pairs $\langle dsk, dpk, \{dsk_i, dpk_i\}_{i\in[n]}\rangle$, but it also generates a polynomial commitment PC_{dsk}, a vector commitment VC_{dpk}, and n tuples $\langle w_{dsk}^i, \pi_{dpk}^i\rangle_{i\in[n]}$. Taking a value x, a DPRF key pair $\langle dsk_i, dpk_i\rangle$, and an opening proof π_{dpk}^i (for dpk_i) as inputs, DPRF.Contrib yields a DPRF contribution $F_i(x)$ and a contribution proof $\mathsf{Proof}_{\mathsf{dprf}}^i$. DPRF.VrfyContrib verifies whether a DPRF contribution is correctly generated. DPRF.Combine combines $t+1$ valid contributions into a DPRF evaluation $y = F(x)$, whereas DPRF.Eval directly outputs $y = F(x)$ when provided with a value x and a DPRF secret key dsk. DPRF.Vrfy verifies DPRF evaluations. We provide a DPRF instantiation in Algorithm 6, Appendix B, adapted from Figure 6 in [35].

Multi-valued validated Byzantine agreement (MVBA**)** protocol [16], [17], [36] enables a group of nodes to submit individual proposals and reach an agreement on an output that satisfies a global predicate P_{MVBA}. The MVBA protocol satisfies the following properties except with negligible probability.

- *Termination.* If all honest nodes input proposals satisfying P_{MVBA}, then each honest node outputs a value.
- *Agreement.* If two honest nodes output in the same instance, then the output values are the same.
- *External-validity.* If an honest node outputs a value v, then v is external valid, i.e., $P_{\mathsf{MVBA}}(v) = 1$.

In this paper, we utilize the MVBA protocol by Lu et al. [17]. This protocol offers $\mathcal{O}(n|m| + \lambda n^2)$ communication complexity, $\mathcal{O}(n^2)$ message complexity, and $\mathcal{O}(1)$ time complexity, where $|m|$ represents the input size. We remark that the MVBA protocol in [17] relies on a common coin sub-protocol to generate common randomness. In Section 9, we will introduce a new MVBA protocol that incorporates an additional *good-case-no-coin* property. This property ensures that no randomness is required in the optimistic

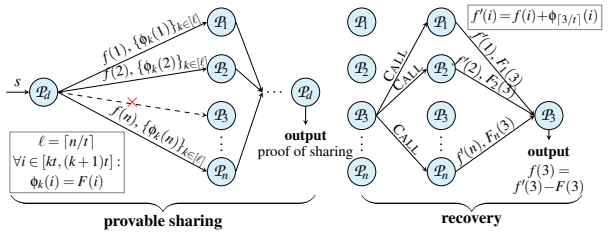

Figure 3: Overview of our wpACSS protocol.

case, which can effectively eliminate the need for common coins in such scenarios without impacting the performance metrics. We refer to this enhanced MVBA protocol as *good-case-no-coin* MVBA.

6. Weak Provable ACSS (wpACSS)

As mentioned earlier, we aim to avoid directly invoking ACSS in DPSS.Handoff to mitigate the high message complexity. Additionally, we desire to achieve a constant message size during the resharing phase. Furthermore, every honest node should be able to recover its missing share without significantly increasing the overall communication complexity. In this section, we introduce our *weak provable ACSS* (wpACSS) protocol, which consists of two sub-protocols:

- **Provable sharing:** it allows a dealer $\mathcal{P}_d$ to share a uniformly random secret s with n nodes. If the dealer outputs a proof of sharing, then at least $t+1$ honest nodes have output valid shares.
- **Recovery:** it allows a node to recover its share with the help of other nodes. If there exists a valid proof of sharing, an honest node is guaranteed to output a valid share through recovery.

Formally, a wpACSS protocol satisfies the following properties except with negligible probability.

- *Termination.* If the dealer $\mathcal{P}_d$ is honest, then all honest nodes eventually output valid shares in the provable sharing sub-protocol, and $\mathcal{P}_d$ outputs a proof of sharing.
- *Provability.* If the dealer $\mathcal{P}_d$ outputs a valid proof of sharing, then at least $t+1$ honest nodes output valid shares in the provable sharing sub-protocol. These honest nodes are able to help others to recover shares in the recovery sub-protocol, so that all honest nodes eventually output valid shares.
- *Correctness.* If a dealer $\mathcal{P}_d$ inputs a secret s in the provable sharing sub-protocol and outputs a valid proof of sharing, then there exists a fixed secret $\hat{s}$, such that any $t+1$ shares from honest nodes can reconstruct $\hat{s}$. If the dealer is honest, then $s = \hat{s}$.
- *Secrecy.* If the dealer $\mathcal{P}_d$ is honest and it shares a uniformly random secret s in wpACSS, then a computationally bounded adversary learns no extra information about the secret beyond g^s.

We remark that the key difference between our wpACSS and other ACSS protocols is that, our wpACSS exhibits a "weaker" completeness property. As mentioned in Section 2, the completeness property of ACSS ensures that if one honest node outputs a valid share, all honest nodes output valid shares. In contrast, our wpACSS only guarantees that if the dealer produces a proof of sharing, at least $t+1$ honest nodes output valid shares. The remaining honest nodes are guaranteed to recover their shares through share recovery.

6.1. Overview of our wpACSS Protocol

Our provable sharing sub-protocol wpACSS.Share has an $\mathcal{O}(n)$ message complexity. To further achieve constant-sized SHARE messages, we draw inspiration from VSSR [18]. At a high level, a dealer $\mathcal{P}_d$ generates one sharing polynomial $f(x)$ and ℓ recovery polynomials $\{\phi_k(x)\}_{k\in[\ell]}$, where $\ell = \lceil n/t \rceil$ is a constant. The sharing polynomial encodes the secret into shares, while every recovery polynomial interpolates t random masks (i.e., DPRF evaluations) for potential share recovery. As shown in Figure 3, $\mathcal{P}_d$ sends to each node $\mathcal{P}_i$ a secret share $f(i)$ and ℓ evaluation points $\{\phi_k(i)\}_{k\in[\ell]}$ via a SHARE message. Each value is accompanied by a constant-sized proof. Additionally, $\mathcal{P}_d$ distributes the DPRF key shares $\langle dsk_i, dpk_i\rangle_{i\in[n]}$ among the nodes, without asymptotically increasing the length of SHARE messages. The proofs and DPRF key shares are omitted in Figure 3.

When an honest node $\mathcal{P}_i$ observes a valid proof of sharing but has not output a valid share from wpACSS.Share, it invokes the recovery sub-protocol. Every other honest node $\mathcal{P}_j$, where $j \in [n]$, responds with a masked share $f'(j) = f(j)+\phi_{\lceil i/t \rceil}(j)$ and a DPRF contribution $F_j(i)$. $\mathcal{P}_i$ interpolates the masked polynomial $f'(x) = f(x)+\phi_{\lceil i/t \rceil}(x)$ using $t+1$ masked shares and combines $t+1$ DPRF contributions into a DPRF evaluation $F(i)$. Since the recovery polynomial $\phi_{\lceil i/t \rceil}(x)$ is constructed subject to $\phi_{\lceil i/t \rceil}(i) = F(i)$, $\mathcal{P}_i$ can recover its share as $f(i) = f'(i) - F(i)$.

Moreover, to reduce the message complexity when $\mathcal{O}(n)$ parallel share recovery instances are invoked, the messages can be batched. Specifically, a node may request share recovery for multiple shares within a single message, and every other node responds with the masked shares and DPRF evaluations in a single message as well.

6.2. Details of our wpACSS Protocol

Provable sharing. The details of wpACSS.Share are shown in Algorithm 1, which comprises four logical phases. In this sub-protocol, we refer to the dealer and committee as $\mathcal{P}_d$ and $\mathcal{U}$, respectively, and the session identifier is ID.

1. Share. In this first phase, the dealer $\mathcal{P}_d$ sends a SHARE message to each node $\mathcal{P}_i \in \mathcal{U}$ (lines 1–15). Specifically, $\mathcal{P}_d$ first generates secret shares $\{s_i\}_{i\in[n]}$ of the secret s using VSS.Share, along with their evaluation proofs. The sharing polynomial is denoted as $f(x)$, satisfying $f(0) = s$ and $f(i) = s_i$ for every $i \in [n]$. Although VSS.Share guarantees

Algorithm 1 wpACSS.Share$[\langle \mathsf{ID}, d\rangle]$, with $\mathcal{P}_d$ as the dealer

Let $\mathcal{H}: \mathbb{G} \to \mathbb{Z}_q$ be a hash function; $\ell \leftarrow \lceil n/t \rceil$

As a dealer $\mathcal{P}_d$:

1: **upon** receiving $(\mathcal{U}, t, n, s)$ as input **do**
2: $S_{\mathsf{sig},1} \leftarrow \emptyset$; $S_{\mathsf{sig},2} \leftarrow \emptyset$
3: $\{PC_s, \langle s_i, w_s^i\rangle_{i\in[n]}\} \leftarrow \mathsf{VSS.Share}(t, n, s)$
4: let $f(x)$ be the sharing polynomial, s.t. $f(0) = s$ and $f(i) = s_i$, $\forall i \in [n]$
5: $z(x) \leftarrow f(x) - s$
6: $PC_z \leftarrow \mathsf{PCom.Commit}(z(x))$
7: $\langle z(0), w_z^0\rangle \leftarrow \mathsf{PCom.Witness}(z(x), 0)$
8: $\langle dsk_i, \{\phi_k(i)\}_{k\in[\ell]}, \mathsf{Proof}_{\mathsf{rec}}^i\rangle_{i\in[n]} \leftarrow \mathsf{GenRecPoly}[\langle \mathsf{ID}, d\rangle](t, n)$
9: $\boldsymbol{v}_s \leftarrow \{g^{s_i}\}_{i\in[n]}$
10: $VC_s \leftarrow \mathsf{VCom.Commit}(\boldsymbol{v}_s)$
11: **for** each $\mathcal{P}_i \in \mathcal{U}$ **do**
12: $\langle g^{s_i}, \pi_s^i\rangle \leftarrow \mathsf{VCom.Open}(\boldsymbol{v}_s, i)$
13: $V_i \leftarrow \langle s_i, dsk_i, \{\phi_k(i)\}_{k\in[\ell]}\rangle$
14: $\Pi_i \leftarrow \langle g^s, PC_s, w_s^i, PC_z, w_z^0, VC_s, \pi_s^i, \mathsf{Proof}_{\mathsf{rec}}^i\rangle$
15: send $\langle \textsc{Share}, \mathsf{ID}, V_i, \Pi_i\rangle$ to $\mathcal{P}_i$ ▷ share
16: **upon** receiving $\langle \textsc{Echo}, \mathsf{ID}, \sigma_{\mathsf{r},d,i}^*\rangle$ from $\mathcal{P}_i \in \mathcal{U}$ for the first time **do**
17: $m_d \leftarrow \mathsf{ID}||d||g^s||PC_s||VC_s||PC_{dsk}||VC_{dpk}||\{PC_\phi^k\}_{k\in[\ell]}$
18: **if** $\mathsf{TS.VrfyShare}_{n-t}(\textsc{ready}||m_d, i, \sigma_{\mathsf{r},d,i}^*, tpk_i) = 1$ **then**
19: $S_{\mathsf{sig},1} \leftarrow S_{\mathsf{sig},1} \cup \langle i, \sigma_{\mathsf{r},d,i}^*\rangle$
20: **if** $|S_{\mathsf{sig},1}| = n-t$ **then**
21: $\sigma_{\mathsf{r},d} \leftarrow \mathsf{TS.Combine}_{n-t}(\textsc{ready}||m_d, S_{\mathsf{sig},1})$ ▷ ready
22: multicast $\langle \textsc{Ready}, \mathsf{ID}, m_d, \sigma_{\mathsf{r},d}\rangle$
23: **upon** receiving $\langle \textsc{Finish}, \mathsf{ID}, \sigma_{\mathsf{f},d,i}^*\rangle$ from $\mathcal{P}_i \in \mathcal{U}$ for the first time **do**
24: **if** $\mathsf{TS.VrfyShare}_{n-t}(\textsc{finish}||m_d, i, \sigma_{\mathsf{f},d,i}^*, tpk_i) = 1$ **then**
25: $S_{\mathsf{sig},2} \leftarrow S_{\mathsf{sig},2} \cup \langle i, \sigma_{\mathsf{f},d,i}^*\rangle$
26: **if** $|S_{\mathsf{sig},2}| = n-t$ **then**
27: $\sigma_{\mathsf{f},d} \leftarrow \mathsf{TS.Combine}_{n-t}(\textsc{finish}||m_d, S_{\mathsf{sig},2})$ ▷ finish
28: **output** $\langle m_d, \sigma_{\mathsf{f},d}\rangle$ ▷ output a proof of sharing, only for dealer

As a node $\mathcal{P}_i \in \mathcal{U}$:

29: **upon** receiving $\langle \textsc{Share}, \mathsf{ID}, V_i, \Pi_i\rangle$ from $\mathcal{P}_d$ for the first time **do**
30: parse V_i as $\langle s_i, dsk_i, \{\phi_k(i)\}_{k\in[\ell]}\rangle$
31: parse Π_i as $\langle g^s, PC_s, w_s^i, PC_z, w_z^0, VC_s, \pi_s^i, \mathsf{Proof}_{\mathsf{rec}}^i\rangle$
32: **if** $PC_s = PC_z g^s \wedge \mathsf{PCom.VrfyEval}(PC_z, 0, 0, w_z^0) = 1$ **then**
33: **if** $\mathsf{VCom.Vrfy}(VC_s, i, g^{s_i}, \pi_s^i) = 1$ **then**
34: **if** all $\ell + 2$ elements in V_i are validated by Π_i **then**
35: extract PC_{dsk}, VC_{dpk} and $\{PC_\phi^k\}_{k\in[\ell]}$ from $\mathsf{Proof}_{\mathsf{rec}}^i$
36: $m_d \leftarrow \mathsf{ID}||d||g^s||PC_s||VC_s||PC_{dsk}||VC_{dpk}||\{PC_\phi^k\}_{k\in[\ell]}$
37: $\sigma_{\mathsf{r},d,i}^* \leftarrow \mathsf{TS.SignShare}_{n-t}(\textsc{ready}||m_d, tsk_i)$
38: send $\langle \textsc{Echo}, \mathsf{ID}, \sigma_{\mathsf{r},d,i}^*\rangle$ to $\mathcal{P}_d$ ▷ echo
39: **wait** until receive $\langle \textsc{Ready}, \mathsf{ID}, m_d', \sigma_{\mathsf{r},d}'\rangle$ from $\mathcal{P}_d$
40: **if** $m_d = m_d' \wedge \mathsf{TS.Vrfy}_{n-t}(\textsc{ready}||m_d', \sigma_{\mathsf{r},d}')$ **then**
41: $\sigma_{\mathsf{f},d,i}^* \leftarrow \mathsf{TS.SignShare}_{n-t}(\textsc{finish}||m_d, tsk_i)$
42: send $\langle \textsc{Finish}, \mathsf{ID}, \sigma_{\mathsf{f},d,i}^*\rangle$ to $\mathcal{P}_d$ ▷ finish
43: **output** $\langle V_i, \Pi_i\rangle$ ▷ output, for all nodes

44: **function** $\mathsf{GenRecPoly}[\langle \mathsf{ID}, d\rangle](t, n)$ ▷ cf. Alg. 6 for DPRF
45: init $\langle dsk, dpk, PC_{dsk}, VC_{dpk}, \{dsk_i, dpk_i, w_{dsk}^i, \pi_{dpk}^i\}_{i\in[n]}\rangle$ for DPRF
46: compute $y_i \leftarrow \mathcal{H}(\mathsf{DPRF.Eval}(\mathsf{ID}||d||i, dsk))$ for each $i \in [n]$
47: **for** $k \in [\ell]$ **do**
48: $I \leftarrow [(k-1)t+1, kt]$
49: $\phi_k(x) \leftarrow \mathsf{VSS.MakeSecret}(t, I, \langle y_i\rangle_{i\in I})$
50: $PC_\phi^k \leftarrow \mathsf{PCom.Commit}(\phi_k(x))$
51: **for** $i \in [n]$ **do**
52: $\langle \phi_k(i), w_\phi^{k,i}\rangle \leftarrow \mathsf{PCom.Witness}(\phi_k(x), i)$ for each $k \in [\ell]$
53: $\mathsf{Proof}_{\mathsf{rec}}^i \leftarrow \langle dpk_i, PC_{dsk}, VC_{dpk}, w_{dsk}^i, \pi_{dpk}^i, \{PC_\phi^k, w_\phi^{k,i}\}_{k\in[\ell]}\rangle$
54: **return** $\langle dsk_i, \{\phi_k(i)\}_{k\in[\ell]}, \mathsf{Proof}_{\mathsf{rec}}^i\rangle_{i\in[n]}$

the consistency of shares, it does not prevent the dealer from sharing an invalid input. To address this problem, we require $\mathcal{P}_d$ to commit to a zero polynomial $z(x) = f(x) - s$ (lines 5–7), ensuring that the dealer inputs the correct share instead of a garbage value. This step is crucial for the DPSS.Handoff procedure (cf. Algorithm 3). Next, $\mathcal{P}_d$ invokes GenRecPoly (line 8) to initialize DPRF key pairs (line 45) and $\ell = \lceil n/t \rceil$ recovery polynomials $\{\phi_k(x)\}_{k\in[\ell]}$ (lines 46–49), where each recovery polynomial goes through t DPRF evaluations. The DPRF key pairs and the recovery polynomials are accompanied by commitments, witnesses, and opening proofs, included in $\mathsf{Proof}_{\mathsf{rec}}^i$ (line 53). Then, $\mathcal{P}_d$ constructs a vector $\boldsymbol{v}_s$ and commits to it (lines 9–10). Finally, $\mathcal{P}_d$ generates a SHARE message for each $\mathcal{P}_i \in \mathcal{U}$, which includes a value set V_i containing $\ell+2$ values: a secret share s_i, a DPRF secret key share dsk_i, and ℓ polynomial evaluations $\{\phi_k(i)\}_{k\in[\ell]}$, along with a proof set Π_i certifying the validity of V_i (lines 13–14).

2. Echo. The second phase is triggered by $\mathcal{P}_i \in \mathcal{U}$ (lines 29–38). Upon receiving $\langle \textsc{Share}, \mathsf{ID}, V_i, \Pi_i\rangle$ from $\mathcal{P}_d$, node $\mathcal{P}_i$ first verifies this message, which includes verifying the correctness of sharing polynomial (line 32) and the opening proof of commitment vector $\boldsymbol{v}_s$ (line 33). The $\ell+2$ elements in V_i are implicitly validated by Π_i in line 34. If the message is valid, $\mathcal{P}_i$ replies to $\mathcal{P}_d$ with a signature share $\sigma_{\mathsf{r},d,i}^*$ on $\textsc{ready}||m_d$ via an ECHO message (lines 35–38), where m_d contains the commitments disseminated by the dealer (line 36). This signature share prevents $\mathcal{P}_d$ from sending inconsistent messages to different nodes.

3. Ready. In the third phase, $\mathcal{P}_d$ waits to collect $n - t$ valid signature shares (ECHO messages) from the nodes in $\mathcal{U}$, and then combines them into a full signature $\sigma_{\mathsf{r},d}$ (lines 16–21). Afterward, it multicasts the signature to all nodes via a READY message (line 22).

4. Finish. In this final phase, upon receiving a valid READY message from $\mathcal{P}_d$ (lines 39–40), node $\mathcal{P}_i$ replies to $\mathcal{P}_d$ with a signature share $\sigma_{\mathsf{f},d,i}^*$ via a FINISH message (line 42) and simultaneously outputs a value-proof tuple $\langle V_i, \Pi_i\rangle$. The dealer $\mathcal{P}_d$ waits to collect $n-t$ valid FINISH messages and combines them into a full signature $\sigma_{\mathsf{f},d}$ (lines 23–27). Finally, $\mathcal{P}_d$ outputs a proof of sharing $\langle m_d, \sigma_{\mathsf{f},d}\rangle$.

Recovery. If node $\mathcal{P}_i$ detects the absence of its deserved share, it invokes this sub-protocol to request a share recovery. The absence of shares may be caused by network delay or dealer's misbehavior. Algorithm 2 provides the procedures for a batched recovery, where a node $\mathcal{P}_i$ recovers a set of shares generated by different dealers. When discussing the share recovery, we assume that a proof of sharing exists. The detailed procedures are as below.

1. Call help. Node $\mathcal{P}_i$ first identifies a help set S_{help}^i, which includes the indexes of dealers from whom $\mathcal{P}_i$ has not received its deserved shares. $\mathcal{P}_i$ then multicasts the help set (lines 1–2). When S_{help}^i contains only one index d, Algorithm 2 reduces to a non-batched version.

2. Help. When node $\mathcal{P}_i$ receives the help set S_{help}^j from $\mathcal{P}_j$, it checks whether it has output the value-proof tuple $\langle V_{d,i}, \Pi_{d,i}\rangle$ from $\mathsf{wpACSS.Share}[\langle \mathsf{ID}, d\rangle]$ for each $d \in S_{\mathsf{help}}^j$ (lines 4–5). If so, it computes the response content $\mathsf{cont}_{d,i}$ using the RecContrib function (line 6). Otherwise, $\mathcal{P}_i$ sets

Algorithm 2 wpACSS.BatchedRecover$[\langle \mathsf{ID}, i\rangle]$

Let $\mathcal{H} : \mathbb{G} \to \mathbb{Z}_q$ be a hash function

1: **upon** receiving S^i_{help} as input **do** ▷ every $\mathcal{P}_i$ holds a help set S^i_{help}
2: multicast $\langle \textsc{Call}, \langle \mathsf{ID}, i\rangle, S^i_{\mathsf{help}}\rangle$ ▷ call help
3: **upon** receiving $\langle \textsc{Call}, \langle \mathsf{ID}, j\rangle, S^j_{\mathsf{help}}\rangle$ from $\mathcal{P}_j$ for the first time **do**
4: **for** each $d \in S^j_{\mathsf{help}}$ **do**
5: **if** have output $\langle V_{d,i}, \Pi_{d,i}\rangle$ from wpACSS.Share$[\langle \mathsf{ID}, d\rangle]$ **then**
6: $\mathsf{cont}_{d,i} \leftarrow \mathsf{RecContrib}[\langle \mathsf{ID}, d\rangle](j, V_{d,i}, \Pi_{d,i})$
7: $\mathsf{res}_i \leftarrow \mathsf{res}_i \cup \langle d, \mathsf{cont}_{d,i}\rangle$
8: **else**
9: $\mathsf{res}_i \leftarrow \mathsf{res}_i \cup \langle d, \perp\rangle$
10: send $\langle \textsc{Help}, \langle \mathsf{ID}, j\rangle, \mathsf{res}_i\rangle$ to $\mathcal{P}_j$ ▷ help
11: **upon** receiving $\langle \textsc{Help}, \langle \mathsf{ID}, i\rangle, \mathsf{res}_j\rangle$ from $\mathcal{P}_j$ for the first time **do**
12: parse res_j as $\langle d, \mathsf{cont}_{d,j}\rangle_{d\in S^i_{\mathsf{help}}}$ and verify $\langle s^m_{d,j}, F_{d,j}(i)\rangle$ in each non-empty $\mathsf{cont}_{d,j}$ is correctly generated
13: **for** each $d \in S^i_{\mathsf{help}}$ **do**
14: **wait** $t{+}1$ valid $\mathsf{cont}_{d,j}$ for distinct j with same $\langle VC^d_{dpk}, PC^d_{\mathsf{mask}}\rangle$
15: denote the index set as I_d
16: extract $\langle s^m_{d,j}, F_{d,j}(i), \mathsf{Proof}^{d,j}_{\mathsf{dprf}}\rangle$ from $\mathsf{cont}_{d,j}$, where $j \in I_d$
17: interpolate $s^m_{d,i}$ from $\{s^m_{d,j}\}_{j\in I_d}$
18: $F_d(i) \leftarrow \mathcal{H}(\mathsf{DPRF.Combine}(t, \mathsf{ID}||d||i, \{F_{d,j}(i), \mathsf{Proof}^{d,j}_{\mathsf{dprf}}\}_{j\in I_d}))$
19: $s_{d,i} \leftarrow s^m_{d,i} - F_d(i)$ ▷ recover
20: **output** $\{s_{d,i}\}_{d\in S^i_{\mathsf{help}}}$

21: **function** RecContrib$[\langle \mathsf{ID}, d\rangle](j, V_i, \Pi_i)$ ▷ cf. Alg. 6 for DPRF
22: parse V_i as $\langle s_i, dsk_i, \{\phi_k(i)\}_{k\in[\ell]}\rangle$
23: parse Π_i as $\langle g^s, PC_s, w^i_s, PC_z, w^0_z, VC_s, \pi^i_s, \mathsf{Proof}^i_{\mathsf{rec}}\rangle$
24: parse $\mathsf{Proof}^i_{\mathsf{rec}}$ as $\langle dpk_i, PC_{dsk}, w^i_{dsk}, \pi^i_{dpk}, \{PC^k_\phi, w^{k,i}_\phi\}_{k\in[\ell]}\rangle$
25: $s^m_i \leftarrow s_i + \phi_{\lceil j/t\rceil}(i)$
26: $w^i_{\mathsf{mask}} \leftarrow w^i_s w^{\lceil j/t\rceil,i}_\phi$
27: $PC_{\mathsf{mask}} \leftarrow PC_s PC^{\lceil j/t\rceil}_\phi$
28: $\langle F_i(j), \mathsf{Proof}^i_{\mathsf{dprf}}\rangle \leftarrow \mathsf{DPRF.Contrib}(\mathsf{ID}||d||j, dsk_i, dpk_i, \pi^i_{dpk})$
29: $\mathsf{Proof}^i_{\mathsf{help}} \leftarrow \langle g^s, VC_{dpk}, PC_{\mathsf{mask}}, w^i_{\mathsf{mask}}, \mathsf{Proof}^i_{\mathsf{dprf}}\rangle$
30: **return** $\mathsf{cont}_i \leftarrow \langle s^m_i, F_i(j), \mathsf{Proof}^i_{\mathsf{help}}\rangle$

$\mathsf{cont}_{d,i}$ as $\perp$ (line 9). After gathering all $\mathsf{cont}_{d,i}$ values into res_i, $\mathcal{P}_i$ sends res_i to $\mathcal{P}_j$ within a HELP message (line 10).

3. Recover. When the caller $\mathcal{P}_i$ receives a HELP message from $\mathcal{P}_j$ (line 11), it verifies each non-empty content $\mathsf{cont}_{d,j} \in \mathsf{res}_j$, where $d \in S^i_{\mathsf{help}}$ (line 12). Specifically, $\mathcal{P}_i$ uses the proof tuple $\mathsf{Proof}^{d,j}_{\mathsf{help}} \in \mathsf{cont}_{d,j}$ to verify whether $\langle s^m_{d,j}, F_{d,j}(i)\rangle$ is correctly generated. Then, for each d, $\mathcal{P}_i$ waits for $t+1$ valid contents containing the same commitments $\langle VC^d_{dpk}, PC^d_{\mathsf{mask}}\rangle$ (lines 13–14). This ensures that the dealer $\mathcal{P}_d$ has sent consistent DPRF key shares and evaluations of recovery polynomials in wpACSS.Share$[\langle \mathsf{ID}, d\rangle]$. Then, it interpolates the i-th masked share $s^m_{d,i}$ and computes the DPRF evaluation $F_d(i)$ (lines 17–18). Finally, $\mathcal{P}_i$ recovers its share $s_{d,i} = s^m_{d,i} - F_d(i)$ (line 19).

6.3. Analysis of our wpACSS Protocol

Performance of provable sharing. In Algorithm 1, each message typically includes a constant number of evaluations and proofs, so the size of each message is $\mathcal{O}(\lambda + |\pi|)$ bits, where $|\pi|$ represents the length of a proof. If we use KZG [30] and Pointproofs [32] for PCom and VCom, respectively, we have $|\pi| = \mathcal{O}(\lambda)$. In this way, the communication complexity of our wpACSS.Share is $\mathcal{O}(n(\lambda+|\pi|)) = \mathcal{O}(\lambda n)$. Besides, as depicted by Figure 1, a provable sharing instance consumes four rounds of communication.

TABLE 3: Performance metrics of wpACSS and ACSS

Protocol	Message	Communication† (bits)	Round‡
wpACSS (optimistic)	$4n$	$18,032n$	4
wpACSS (worst)	$\approx 4n^2/9 + 34n/9$	$\approx 659n^2 + 18,691n$	6
LongLive's ACSS [9]	$2n^2 + n$	$\approx 582,272n^2$	3–4
hbACSS [12]	$8n^2 + 2n$	$10,960n^2 + 2688n$	9–10

† Communication cost is computed based on curve bls12-381.
‡ LongLive's ACSS and hbACSS adopts Bracha's reliable broadcast, which incurs 3–4 communication rounds.

Performance of batched recovery. In Algorithm 2, when a node requests to recover $|S_{\mathsf{help}}|$ shares, there is one CALL message and n HELP messages, leading to an $\mathcal{O}(n)$ message complexity. The CALL message is of $\mathcal{O}(|S_{\mathsf{help}}|)$ bits, and every HELP message consumes $\mathcal{O}(\lambda|S_{\mathsf{help}}|)$ bits, if KZG and Pointproofs schemes are employed. Therefore, the overall communication complexity is $\mathcal{O}(\lambda n|S_{\mathsf{help}}|)$. Besides, each recovery instance consumes two communication rounds.

Comparison with ACSS. The performance metrics of our wpACSS and ACSS protocols are shown in Table 3. In the worst case, only $t+1$ honest nodes output a valid share through provable sharing, and the other t honest nodes invoke the recovery sub-protocol. Our wpACSS exhibits lower concrete message and communication complexities.

Security analysis. In the following, we provide a security proof sketch of our wpACSS. Detailed security proofs are delayed to Appendix D.

Termination. If the dealer $\mathcal{P}_d$ is honest, then it will generate a valid value-proof tuple $\langle V_i, \Pi_i\rangle$ for each node $\mathcal{P}_i \in \mathcal{U}$. Hence, every honest node $\mathcal{P}_i$ receives such a tuple containing its deserved valid share, all honest nodes will output valid shares. Besides, before the honest nodes output, they will each send a FINISH message to $\mathcal{P}_d$. Each FINISH message contains a valid signature share, so $\mathcal{P}_d$ can formulate a full signature and output a valid proof of sharing.

Provability. In our model, the threshold key pairs are initialized by a trusted setup. Hence, the adversary cannot generate a proof of sharing by itself. If the dealer $\mathcal{P}_d$ outputs a valid proof of sharing from wpACSS.Share, then $\mathcal{P}_d$ has received at least $n-t$ FINISH messages from distinct nodes, among which $n - 2t \geq t+1$ nodes are honest. These honest nodes will output value-proof tuples, each tuple contains a secret share. Additionally, for each caller $\mathcal{P}_j$, the honest nodes $\{\mathcal{P}_i\}$ who have valid shares can generate valid responses. Hence, $\mathcal{P}_j$ can recover its share through wpACSS.BatchedRecover.

Correctness. Given there are at most t corrupted nodes and some honest nodes output a valid share from wpACSS.Share. If there are two valid distinct message-signature pairs $\langle m_d, \sigma_{\mathsf{r},d}\rangle$ and $\langle m'_d, \sigma'_{\mathsf{r},d}\rangle$, then at least one honest node have signed to both m_d and m'_d, leading to a contradiction. Hence, it implies that the tuples output by these honest nodes are bound with the same m_d, while the

m_d contains the polynomial commitment PC_s, so the shares output by these honest nodes are bound with the same PC_s. Besides, according to the Algorithm 2, an honest node $\mathcal{P}_j$ has to wait for $t+1$ valid HELP responses with the same commitments $(VC^d_{dpk}, PC^d_{\mathsf{mask}})$ before recovering its share. As there are at most t corrupted nodes, at least one response comes from honest nodes, which ensures the commitments are correctly computed. Hence, the recovered share is also bounded by PC_s. Since we have proved that all shares output by honest nodes are bound with the same polynomial commitment PC_s. Hence, any $t+1$ shares from honest nodes can reconstruct the secret $\hat{s}$. Moreover, if the dealer is honest, then $\hat{f}(x) = f(x)$, $s = \hat{s}$.

Secrecy. In wpACSS.Share, only the SHARE messages contain private information about the secret, the adversary has access to at most t value-proof tuples $\langle v_i, \pi_i \rangle$. Each value in v_i lies on a degree-t random polynomial, where t points obtained by the adversary reveals no information about the other evaluations on it, let alone the secret. Hence, wpACSS.Share reveals no information about the secret. As for wpACSS.BatchedRecover, the adversary can invoke up to t share recovery instances, and he is able to unmask t shares using t DPRF evaluations. These t DPRF evaluations do not help the adversary to interpolate the degree-t recovery polynomials. So the adversary obtains no further knowledge about the secret in wpACSS.BatchedRecover.

7. Our DPSS Protocol

We focus on the design of the DPSS.Handoff protocol (Algorithm 3), which is built upon wpACSS and invokes Algorithm 4 as a sub-protocol. For simplicity, we denote $\mathcal{P}_i^{e-1}$ as $\mathcal{P}_i$ and $\mathcal{P}_i^e$ as $\mathcal{P}'_i$. The i-th old and new shares are referred to as s_i and s'_i, respectively. We also let $t_{e-1} = t$, $t_e = t'$, $n_{e-1} = n$, $n_e = n'$, and $\mathsf{ID} = e$.

High-level overview. As shown in Figure 2, DPSS.Handoff begins with n parallel wpACSS.Share instances, where each old node reshares its share with the new committee. Once the resharing is completed, the old node outputs a valid proof of sharing, which is then multicast to the new committee. Due to the presence of Byzantine nodes, it is possible that some honest new nodes receive shares while others do not. As a result, honest new nodes may disagree on which sharing instances can be used to compute the new share. To address this issue, they run an MVBA instance, in which each node inputs $t+1$ proofs of sharing. Then, the MVBA component selects $t+1$ valid resharing instances as output, and the new nodes compute their shares from the outputs of these instances.

If an honest node has not yet output from any selected resharing instances, it invokes wpACSS.BatchedRecover to recover the missing shares. The missing of share may be caused by malicious dealers or network delay. To prevent old nodes from invoking wpACSS.Share with invalid old shares, the old committee transfers a commitment vector to the new committee before resharing, where the vector contains the commitments to all old shares. This vector is generated in a two-path sub-protocol GetNewCom (Algorithm 4) at the end of handoff, after the new nodes have obtained new shares.

7.1. Details of our DPSS Protocol

Our DPSS.Handoff protocol, as shown in Algorithm 3, comprises five logical phases: *Transfer commitments*, *Reshare*, *Select*, *Recover*, and *Compute*, where the *Recover* phase is optional. The specific procedures are as follows.

1. Transfer commitments. Initially, each honest old node $\mathcal{P}_i$ checks whether it has an old share s_i and $t+1$ valid signatures on the commitment vector $\boldsymbol{v}_{\mathsf{old}} = \{g^{s_1}, ..., g^{s_n}\}$ containing the commitments to all old shares. If so, $\mathcal{P}_i$ commits to this vector by VC_{old} and generates an opening proof π^i_{old} for the i-th element g^{s_i}. Subsequently, $\mathcal{P}_i$ multicasts $\langle VC_{\mathsf{old}}, g^{s_i}, \pi^i_{\mathsf{old}} \rangle$ to the new committee $\mathcal{U}'$ using a COM message (lines 1–5). Each new node $\mathcal{P}'_i$ waits for $t+1$ valid COM messages from distinct old nodes, all with the same commitment VC_{old} (line 9). Then, $\mathcal{P}'_i$ uses the received elements to interpolate g^s and the vector $\boldsymbol{v}_{\mathsf{old}} = \{g^{s_1}, ..., g^{s_n}\}$ (line 10), where g^s is the commitment to the secret s. This is possible due to the multiplicative homomorphism of exponentiation in cyclic groups.

2. Reshare. Each old node $\mathcal{P}_i$ reshares its old share s_i to the new committee $\mathcal{U}'$ by wpACSS.Share$[\langle \mathsf{ID}, i \rangle]$, from which $\mathcal{P}_i$ outputs a proof of sharing $\langle m_i, \sigma_i \rangle$ (line 6). $\mathcal{P}_i$ multicasts this proof to $\mathcal{U}'$ via PROOF messages (line 7). Afterward, $\mathcal{P}_i$ deletes all private information and halts.

3. Select. Each new node $\mathcal{P}'_i$ awaits $t+1$ valid PROOF messages from distinct old nodes and records the $t+1$ proofs of sharing in W_i (lines 11–14). Then, $\mathcal{P}'_i$ inputs W_i into MVBA[ID] (lines 15–16) and waits for the output $\overline{W}$ (line 17). The predicate of MVBA requires at least $t+1$ valid signatures are included, so that the selected provable sharing instances cannot all originate from corrupted nodes. Otherwise, the adversary could reconstruct the secret and compromise secrecy. The index set I of selected resharing instances is computed from $\overline{W}$ (line 18), where $|I| = t+1$.

4. Recover (optional). Upon obtaining the index set I, each new node $\mathcal{P}'_i$ identifies a help set S^i_{help} containing the indexes of all missing shares (lines 19–21). In case the help set is non-empty, i.e., there is any missing share, $\mathcal{P}'_i$ invokes wpACSS.BatchedRecover$[\langle \mathsf{ID}, i \rangle]$ (line 23). This phase is not executed in the optimistic case.

5. Compute. When $\mathcal{P}'_i$ obtains the shares $s_{j,i}$ for all $j \in I$, it interpolates the refreshed share s'_i from $\{(j, s_{j,i})\}_{j \in I}$ (line 24). Subsequently, each $\mathcal{P}'_i$ invokes GetNewCom[ID] (Algorithm 4) to obtain the new commitment vector $\boldsymbol{v}_{\mathsf{new}} = \{g^{s'_1}, ..., g^{s'_{n'}}\}$ (line 25). Then, $\mathcal{P}'_i$ signs $\boldsymbol{v}_{\mathsf{new}}$ and multicast the signature. Finally, $\mathcal{P}'_i$ waits for $t'+1$ valid and consistent signatures on $\boldsymbol{v}_{\mathsf{new}}$ and outputs $\langle s'_i, \boldsymbol{v}_{\mathsf{new}} \rangle$ (lines 26–28).

The GetNewCom[ID] sub-protocol consists of two paths: the optimistic and pessimistic paths. Every new node initially follows the optimistic path to generate the new commitment vector. If any misbehavior is detected, it multicasts an ERROR message and transitions to the pessimistic path. An

Algorithm 3 DPSS.Handoff[ID] between $\mathcal{U}$ and $\mathcal{U}'$

Let $\mathcal{U}$ be the old committee, with n nodes and t corrupted nodes, and $\mathcal{U}'$ be the new committee, with n' nodes and t' corrupted nodes
Let $\boldsymbol{v}_{\mathsf{old}} = \{g^{s_1}, ..., g^{s_n}\}$ include the commitments to all old shares, and g^s be the commitment to the secret s
Let the predicate of MVBA be: $P_{\mathsf{MVBA}}[\{\langle m_1, \sigma_1\rangle, \cdots, \langle m_n, \sigma_n\rangle\}]$ $\equiv$ (there are $t+1$ valid signatures, and for each g^{s_j} extracted from m_j, it holds that $g^{s_j} = \boldsymbol{v}_{\mathsf{old}}[j]$)

As an old node $\mathcal{P}_i \in \mathcal{U}$:
upon receiving START **do**
  if $\mathcal{P}_i$ have s_i and $t+1$ valid signatures on $\boldsymbol{v}_{\mathsf{old}}$ **then**
    $VC_{\mathsf{old}} \leftarrow \mathsf{VCom.Commit}(\boldsymbol{v}_{\mathsf{old}})$
    $\langle g^{s_i}, \pi^i_{\mathsf{old}}\rangle \leftarrow \mathsf{VCom.Open}(\boldsymbol{v}_{\mathsf{old}}, i)$
    multicast $\langle \text{COM}, \mathsf{ID}, VC_{\mathsf{old}}, g^{s_i}, \pi^i_{\mathsf{old}}\rangle$ to $\mathcal{U}'$ ▷ transfer commitments
    $\langle m_i, \sigma_i\rangle \leftarrow \mathsf{wpACSS.Share}[\langle \mathsf{ID}, i\rangle](\mathcal{U}', t', n', s_i)$ ▷ reshare
    multicast $\langle \text{PROOF}, \mathsf{ID}, m_i, \sigma_i\rangle$ to $\mathcal{U}'$
  delete all private information and halt ▷ halt (old node)

As a new node $\mathcal{P}'_i \in \mathcal{U}'$:
Init: start $\mathsf{wpACSS.Share}[\langle \mathsf{ID}, j\rangle]$ for all $j \in [n]$; ▷ reshare
$W_i \leftarrow \{\langle m_j, \sigma_j\rangle_{j\in[n]}\}$, where $\langle m_j, \sigma_j\rangle \leftarrow \langle \perp, \perp\rangle$; $S^i_{\mathsf{help}} \leftarrow \emptyset$;
▷ transfer commitments
upon receiving $t+1$ valid $\langle \text{COM}, \mathsf{ID}, VC_{\mathsf{old}}, g^{s_j}, \pi^j_{\mathsf{old}}\rangle$ with VC_{old} **do**
  interpolate g^s and $\boldsymbol{v}_{\mathsf{old}} = \{g^{s_1}, ..., g^{s_n}\}$

upon receiving $\langle \text{PROOF}, \mathsf{ID}, m'_j, \sigma'_j\rangle$ from $\mathcal{P}_j$ for the first time **do**
  extract g^{s_j} from m'_j
  if $\mathsf{TS.Vrfy}_{n'-t'}(\text{FINISH}||m'_j, \sigma'_j) = 1 \wedge g^{s_j} = \boldsymbol{v}_{\mathsf{old}}[j]$ **then**
    $\langle m_j, \sigma_j\rangle \leftarrow \langle m'_j, \sigma'_j\rangle$, where $\langle m_j, \sigma_j\rangle \in W_i$
    if there are $t+1$ non-empty elements in W_i **then**
      input W_i to $\mathsf{MVBA}[\mathsf{ID}]$ ▷ select

upon receiving $\overline{W} = \{\langle \bar{m}_j, \bar{\sigma}_j\rangle_{j\in[n]}\}$ from $\mathsf{MVBA}[\mathsf{ID}]$ **do**
  $I \leftarrow \{j \mid \langle \bar{m}_j, \bar{\sigma}_j\rangle \in \overline{W} \wedge \langle \bar{m}_j, \bar{\sigma}_j\rangle \neq \langle \perp, \perp\rangle\}$
  for $j \in I$ **do**
    if not yet receive $s_{j,i}$ from $\mathsf{wpACSS.Share}[\langle \mathsf{ID}, j\rangle]$ **then**
      $S^i_{\mathsf{help}} \leftarrow S^i_{\mathsf{help}} \cup \{j\}$
  if $|S^i_{\mathsf{help}}| > 0$ **then** ▷ recover (optional)
    $\{s_{j,i}\}_{j\in S^i_{\mathsf{help}}} \leftarrow \mathsf{wpACSS.BatchedRecover}[\langle \mathsf{ID}, i\rangle](S^i_{\mathsf{help}})$
  interpolate s'_i from $\{(j, s_{j,i})\}_{j\in I}$ ▷ compute
  $\boldsymbol{v}_{\mathsf{new}} \leftarrow \mathsf{GetNewCom}[\mathsf{ID}]$ ▷ cf. Alg. 4
  sign $\boldsymbol{v}_{\mathsf{new}}$ and multicast the signature
  wait for $t'+1$ valid and consistent signatures on $\boldsymbol{v}_{\mathsf{new}}$
    output $\langle s'_i, \boldsymbol{v}_{\mathsf{new}}\rangle$ ▷ output (new node)

honest node will also switch to the pessimistic path upon receiving an ERROR message. In the pessimistic path, every honest node that multicasts ERROR attempts to collect the commitments to subshares and interpolates them to obtain new commitments to the refreshed shares. When there are $t'+1$ new commitments, it interpolates the whole commitment vector $\boldsymbol{v}_{\mathsf{new}}$. To ensure correctness, the subshare commitments in our protocol are verifiable, and the proofs are extracted from the output of the wpACSS.Share sub-protocol. The step-by-step procedures for the two paths are outlined below (cf. Algorithm 4).

6. Optimistic path. Upon entering the GetNewCom[ID] sub-protocol, each new node $\mathcal{P}'_i$ initializes an empty index set I' and a flag $flg = 0$, and it multicasts the commitment $g^{s'_i}$ using NEWCOM messages. Then, $\mathcal{P}'_i$ awaits $t'+1$ distinct commitments and use them to interpolate $g^{s'_0}$ and $\boldsymbol{v}_{\mathsf{new}} = \{g^{s'_1}, ..., g^{s'_{n'}}\}$. If the interpolated $g^{s'_0}$ equals g^s, which is

Algorithm 4 GetNewCom[ID], cf. Alg. 3 as main protocol

Init: $I' \leftarrow \emptyset$; $flg \leftarrow 0$; multicast $\langle \text{NEWCOM}, \mathsf{ID}, g^{s'_i}\rangle$ to $\mathcal{U}'$

upon receiving $t'+1$ distinct $\langle \text{NEWCOM}, \mathsf{ID}, g^{s'_j}\rangle$ **do** ▷ optimistic
  interpolate $g^{s'_0}$ and $\boldsymbol{v}_{\mathsf{new}} = \{g^{s'_1}, ..., g^{s'_{n'}}\}$
  if $g^{s'_0} = g^s$ **then** ▷ g^s is obtained in line 10, Alg. 3
    return $\boldsymbol{v}_{\mathsf{new}}$ ▷ output
  else $flg \leftarrow 1$; multicast $\langle \text{ERROR}, \mathsf{ID}\rangle$ to $\mathcal{U}'$

upon receiving $\langle \text{ERROR}, \mathsf{ID}\rangle$ from $\mathcal{P}'_j$ for the first time **do** ▷ pessimistic
  $I_i \leftarrow I \backslash S^i_{\mathsf{help}}$ ▷ I and S^i_{help} are from line 18 and 21, Alg. 3
  for $k \in I_i$ **do**
    denote the output of $\mathsf{wpACSS.Share}[\langle \mathsf{ID}, k\rangle]$ as $\langle V_{k,i}, \Pi_{k,i}\rangle$
    extract $\langle g^{s_{k,i}}, VC^k_s, \pi^{k,i}_s\rangle$ from $\Pi_{k,i}$
  send $\langle \text{AUX}, \mathsf{ID}, \{k, g^{s_{k,i}}, VC^k_s, \pi^{k,i}_s\}_{k\in I_i}\rangle$ to $\mathcal{P}'_j$

upon receiving $\langle \text{AUX}, \mathsf{ID}, \{k, g^{s_{k,j}}, VC^k_s, \pi^{k,j}_s\}_{k\in I_j}\rangle$ from $\mathcal{P}'_j$ **do**
  if $flg = 1 \wedge \mathsf{VCom.Vrfy}(VC^k_s, k, g^{s_{k,j}}, \pi^{k,j}_s) = 1, \forall k \in I_j$ **then**
    record $\langle k, g^{s_{k,j}}\rangle_{k\in I_j}$
    if $I_j \neq I$ **then**
      for $k \in I \backslash I_j$ **do** ▷ $I \backslash I_j = \mathcal{S}^j_{\mathsf{help}}$
        wait $t'+1$ AUX messages containing $g^{s_{k,*}}$ to interpolate $g^{s_{k,j}}$
        record $\langle k, g^{s_{k,j}}\rangle$
      interpolate $g^{s'_j} = g^{s_{0,j}}$ from $\langle k, g^{s_{k,j}}\rangle_{k\in I}$
    $I' \leftarrow I' \cup \{j\}$
    if $|I'| = t'+1$ **then**
      interpolate $\boldsymbol{v}_{\mathsf{new}} = \{g^{s'_1}, ..., g^{s'_{n'}}\}$ from $\{g^{s'_j}\}_{j\in I'}$
      return $\boldsymbol{v}_{\mathsf{new}}$ ▷ output

the commitment to the original share obtained in line 10 of Algorithm 3, $\mathcal{P}'_i$ returns the new commitment vector $\boldsymbol{v}_{\mathsf{new}}$. Otherwise, $\mathcal{P}'_i$ enters the pessimistic path.

7. Pessimistic path. When node $\mathcal{P}'_i$ receives an ERROR message from $\mathcal{P}'_j$, it replies with an AUX message containing the commitments to subshares and detailed verifiable information, which is extracted from the provable sharing instances. Specifically, let $I_i = I \backslash S^i_{\mathsf{help}}$ be an index set, such that for each $k \in I_i$, $\mathcal{P}'_i$ has output $\langle V_{k,i}, \Pi_{k,i}\rangle$ from $\mathsf{wpACSS.Share}[\langle \mathsf{ID}, k\rangle]$. $\mathcal{P}'_i$ extracts the useful elements from $\{\Pi_{k,i}\}_{k\in I_i}$ and gathers them into the AUX message. On the other side, if $\mathcal{P}'_i$ has previously sent an ERROR message, then upon receiving an AUX message from $\mathcal{P}'_j$, $\mathcal{P}'_i$ verifies the elements. If all verifications return true, $\mathcal{P}'_i$ records $\langle k, g^{s_{k,j}}\rangle_{k\in I_j}$. If $I_j \neq I$, which means $\mathcal{P}'_j$ fails to contribute some of the required commitments, then for each $k \in I \backslash I_j$, $\mathcal{P}'_i$ waits for $t'+1$ commitments $\{g^{s_{k,*}}\}$ from other AUX messages to interpolate the missing $g^{s_{k,j}}$. Once $\mathcal{P}'_i$ obtains $\{g^{s_{k,j}}\}_{k\in I}$, it interpolates the new commitment $g^{s'_j}$. When there are $t'+1$ such new commitments, $\mathcal{P}'_i$ interpolates and returns $\boldsymbol{v}_{\mathsf{new}}$.

7.2. Analysis of our DPSS Protocol

Message complexity. In the *Transfer commitments* phase, each old node sends n' COM messages, resulting in a message complexity of $\mathcal{O}(n^2)$. In the *Reshare* phase, the message complexity of each wpACSS.Share instance is $\mathcal{O}(n)$, and every old node multicasts a PROOF message to all new nodes, leading to a message complexity of $\mathcal{O}(n^2)$. The *Select* phase only involves one MVBA instance. Since

we use the MVBA protocol [17] with $\mathcal{O}(n^2)$ message complexity, the message complexity of this phase is also $\mathcal{O}(n^2)$. In the optional *Recover* phase, the nodes may invoke up to $\mathcal{O}(n)$ wpACSS.BatchedRecover instances, incurring $\mathcal{O}(n^2)$ message complexity. In the *Compute* phase, either path in GetNewCom has an $\mathcal{O}(n^2)$ message complexity. Overall, the message complexity of DPSS.Handoff is $\mathcal{O}(n^2)$.

Communication complexity. The communication complexity of DPSS.Handoff can be broken down into several steps. Each old node first multicasts an $\mathcal{O}(\lambda)$-bit COM message to the new committee. Then, as the dealer, it invokes a wpACSS.Share instance, consuming $\mathcal{O}(\lambda n)$ bits. Next, the new committee runs an MVBA instance, with $\mathcal{O}(\lambda n)$-sized inputs, resulting in an $\mathcal{O}(\lambda n^2)$ communication cost. The subsequent steps are divided into two cases.

Optimistic case: In this situation, no share recovery is needed, and the new committee invokes GetNewCom to obtain a new commitment vector. In the optimistic path of GetNewCom, each node only needs to multicast an $\mathcal{O}(\lambda)$-bit NEWCOM message and then locally compute the new vector. Hence, the overall optimistic-case communication complexity of DPSS.Handoff is $\mathcal{O}(\lambda n^2)$.

Worst case: In the worst situation, there are two bottlenecks that incur $\mathcal{O}(\lambda n^3)$ communication cost. Firstly, at most $\mathcal{O}(n)$ batched recovery instances are invoked, and each new node needs to batch-recover $\mathcal{O}(n)$ shares, resulting in an $\mathcal{O}(\lambda n^3)$ communication complexity. Secondly, in the pessimistic path of GetNewCom, each node multicasts an $\mathcal{O}(\lambda)$-bit ERROR message and an $\mathcal{O}(\lambda n)$-bit AUX message, so the communication complexity of GetNewCom in the worst case also reaches $\mathcal{O}(\lambda n^3)$. In total, the worst-case communication complexity of DPSS.Handoff is $\mathcal{O}(\lambda n^3)$.

Time complexity. The *Select* phase costs $\mathcal{O}(1)$ time complexity due to the MVBA instance [17], and the other phases consume constant rounds of communication. Therefore, the overall time complexity of DPSS.Handoff is $\mathcal{O}(1)$.

Security analysis. In the following, we provide a security proof sketch of our DPSS.Handoff. Detailed security proofs are delayed to Appendix E.

Termination. Firstly, we prove that each honest old node terminates. By definition, each honest old node holds an old share and a commitment vector. Hence, they can send the COM messages and invoke wpACSS.Share. By the termination of wpACSS, every honest old node is guaranteed to output a proof of sharing, which is used to formulate the PROOF message. Hence, all honest old nodes will multicast the necessary messages and halt. Secondly, we prove that all honest new nodes terminates. According to our adversary model, there are at least $t+1$ honest old nodes in each epoch. They have the same commitment vector $\boldsymbol{v}_{\mathsf{old}}$, so each honest new node $\mathcal{P}'_i$ is guaranteed to receive $t+1$ valid COM messages with the same VC_{old} and interpolate the commitments. Besides, the honest new nodes will receive at least $t+1$ valid PROOF messages, so they will input valid proposals to MVBA. By the termination of MVBA, they obtain the same output and index set I. For each $j \in I$, there exists a valid proof of sharing. Hence, by the provability of wpACSS, each $\mathcal{P}'_i$ can recover the missing shares (if any). Similarly, each $\mathcal{P}'_i$ can generate the new commitment vector $\boldsymbol{v}_{\mathsf{new}}$ from GetNewCom, leading to the termination.

Correctness. During DPSS.Handoff, the new nodes obtain a common index set I, where $|I| = t+1$. For each $j \in I$, the old node $\mathcal{P}_j$ reshares its share s_j using a degree-t' sharing polynomial $f_j(x)$, such that $f_j(0) = s_j$. The commitment vector $\boldsymbol{v}_{\mathsf{old}}$ ensures that the input to each wpACSS.Share instance is the correct old share. By the external validity of MVBA, the proofs of sharing from $\{\mathcal{P}_j\}_{j\in I}$ are all valid. The $t+1$ polynomials $\{f_j(x)\}_{j\in I}$ form a degree-$\langle t', t\rangle$ bivariate polynomial $B(x,y)$, such that $B(x,j) = f_j(x)$. The secret value $s = B(0,0)$ can be reconstructed from $t+1$ old shares or $t'+1$ new shares, so the secret value s remains invariant.

Secrecy. During DPSS.Handoff, only the provable sharing and batched recovery instances involve sensitive information about the secret s. By the secrecy of wpACSS, the adversary learns no information about the old shares if the dealers are honest. For each epoch e, let $B_e(x,y)$ denote the degree-$\langle t', t\rangle$ bivariate polynomial. Without loss of generality, we assume the adversary has t old shares $\{B(0,i)\}_{i\in[t]}$ and t' new shares $\{B(i,0)\}_{i\in[t']}$. Since $B(x,y)$ has $(t+1)(t'+1)$ unknown coefficients, these $t+t'$ evaluations reveal no information about the secret $s = B(0,0)$.

8. Extensions

The DPSS protocol in Section 7 only supports refreshing the low-threshold shares of a single secret. In this section, we briefly introduce two variations of our DPSS, batched DPSS and high-threshold DPSS. Both variations employ the hyperinvertible matrix [37], referred to as H matrix hereafter, to derive batched randomness. Given n shares, at most t of which are determined by the adversary, the H matrix allows a node to extract $n-t$ uniformly random shares.

8.1. Batched DPSS

Batched DPSS allows a batch of shares to be refreshed simultaneously within a single execution of DPSS, amortizing the communication cost. Our batched DPSS (bDPSS) adopts the strategy of LongLive [9]. From a high level, given a batch size $B = \mathcal{O}(n)$, we let the old committee jointly generate a batch of uniformly random values $\{r_k\}_{k\in[B]}$, such that each old node $\mathcal{P}_i$ and new node $\mathcal{P}'_i$ hold $\{r_{k,i}\}_{k\in[B]}$ and $\{r'_{k,i}\}_{k\in[B]}$, respectively, with $r_{k,i}$ and $r'_{k,i}$ being the i-th shares of the same r_k. Then, we let the old committee reconstruct $\{s_k + r_k\}_{k\in[B]}$ and send them to the new committee, where s_k is the k-th secret. Each new node $\mathcal{P}'_i$ locally computes its new share $s'_{k,i} = s_k + r_k - r'_{k,i}$.

More specifically, each old node locally samples $B/(n-2t)$ random values and shares them by wpACSS.Share. Then, $n-t$ nodes are selected by MVBA, such that for each selected node $\mathcal{P}_j$, every honest node $\mathcal{P}_i$ outputs the shares of the $B/(n-2t)$ random values sampled by $\mathcal{P}_j$. Assuming $B = b(n-2t)$, $\mathcal{P}_i$ receives b sets of shares,

each set containing $n-t$ shares. Using H matrices, $\mathcal{P}_i$ extracts $n-2t$ uniformly random shares from every $n-t$ non-uniform shares. Therefore, the total number of random shares derived by $\mathcal{P}_i$ is $b(n-2t)=B$.

We highlight the primary differences between DPSS and bDPSS in Figure 8, Appendix F. The details of bDPSS are presented in Algorithm 9, Appendix F.

Performance. For a batch size B, the communication complexity of bDPSS is $\mathcal{O}(\lambda nB)$ in the optimistic case, which implies an amortized communication complexity of $\mathcal{O}(\lambda n)$ per secret. The message complexity is $\mathcal{O}(nB)$, and the time complexity is $\mathcal{O}(1)$.

8.2. High-Threshold DPSS

In some threshold cryptosystems, $2t+1$ secret shares are required to reconstruct the secret or generate a threshold full signature, where at most t nodes are corrupted. Such high-threshold schemes are widely used in BFT protocols [4], [38], [39]. To support dynamic membership in these applications, a high-threshold DPSS is needed.

In the following, we describe how to extend our DPSS to the high-threshold setting using a high-threshold asynchronous distributed key generation (hADKG) protocol. This technique has been mentioned by Das et al. [2].

At a high level, the old committee inputs the low-threshold old shares $\{s_{\mathsf{low},i}\}_{i\in[n]}$ to the handoff protocol, so that every new node $\mathcal{P}'_i$ acquires a low-threshold new share $s'_{\mathsf{low},i}$. Additionally, $\mathcal{P}'_i$ outputs a high-threshold zero-share $z(i)$ from hADKG, where $z(x)$ is a random polynomial of degree $2t'$ subject to $z(0)=0$. The refreshed high-threshold share for $\mathcal{P}'_i$ is calculated as $s'_{\mathsf{high},i}=s'_{\mathsf{low},i}+z(i)$. The high-threshold shares $\{s'_{\mathsf{high},i}\}_{i\in[n']}$ are used for DPSS.Recon, whereas the input to the subsequent handoff is $\{s'_{\mathsf{low},i}\}_{i\in[n']}$.

Performance. The hADKG protocol by Das et al. [2] utilizes n parallel ACSS instances and n parallel asynchronous Byzantine binary agreement (ABA) instances. To reduce the overall message and communication complexities, we may replace the ACSS instances with our wpACSS. Besides, we also replace the ABA instances with an MVBA instance of $\mathcal{O}(n|m|+\lambda n^2)$ communication complexity and $\mathcal{O}(1)$ time complexity [17]. In this way, we achieve a high-threshold DPSS with $\mathcal{O}(n^2)$ message complexity, $\mathcal{O}(\lambda n^2)$ optimistic-case communication complexity, and $\mathcal{O}(1)$ time complexity.

9. Discussion

9.1. Trusted Setup

Our DPSS protocol relies on a trusted setup in every epoch to initialize (1) the public parameters for the underlying commitment schemes (e.g., powers of tau for the KZG and Pointproofs schemes) and (2) threshold signing keys for each new committee (as required by MVBA [17]). This assumption is also essential in state-of-the-art protocols [8], [9]. Although the KZG parameters can be reused across epochs, the public parameters for Pointproofs must be regenerated whenever the committee size changes, since their length is tied to the size of the committee. Furthermore, while the initialization of threshold signing keys can be achieved by an ADKG protocol, this approach introduces significant overhead.The state-of-the-art ADKG protocol [1] incurs $\mathcal{O}(\lambda n^3)$ communication complexity, aligning with both LongLive [9] and DyCAPS [8]. However, the time complexity in the worst case grows to $\mathcal{O}(\log n)$.

Although in theory, the time complexity of ADKG [1] can be reduced to $\mathcal{O}(1)$, the authors of LongLive [9] caution that such a construction may not be practically efficient. Regarding our DPSS, even if $\mathcal{O}(1)$ time complexity is achieved, employing an ADKG protocol with $\mathcal{O}(\lambda n^3)$ communication complexity still harms our optimistic-case performance. Below, we carefully design two strategies to eliminate the strong per-epoch trusted setup assumption.

Achieving one-time trusted setup. Formally, a one-time trusted setup requires all secret trapdoors (e.g., the value of tau) to be generated only once during initialization, and the resulting public parameters are reused across all epochs. We first address the setup of public parameters for commitment schemes. By introducing a read-only bulletin board, we can employ the one-time trusted setup to produce the public parameters for all $n\in[N]$, where $N=\mathcal{O}(n)$ is the maximum allowed committee size. Prior to each handoff, both old and new committees retrieve $\mathcal{O}(\lambda n)$-sized public parameters from the bulletin board, incurring $\mathcal{O}(\lambda n^2)$ communication cost. Regarding the threshold keys, they serve two purposes in our DPSS: (1) Creating proofs of sharing through threshold signatures in wpACSS; (2) Generating common coins in MVBA. To avoid using threshold keys in the optimistic case, we take the following steps.

In wpACSS, we implement the approach by Das et al. [40] to substitute threshold signatures with multi-signatures, accompanied by succinct non-interactive arguments of knowledge (SNARK). Specifically, the dealer combines $2t+1$ standard digital signatures into one multi-signature and generates a SNARK proof for this aggregation. With a constant-sized universal SNARK instantiation (e.g., Plonk [41] and Vampire [42]), the communication complexity of wpACSS remains at $\mathcal{O}(\lambda n)$. Moreover, both Plonk and Vampire facilitate a one-time setup, where the size of public parameters scales linearly with the maximum committee size $N=\mathcal{O}(n)$.

As for MVBA [17], we are inspired by Crain's ABA [43], which has the *good-case-coin-free* property. This property states that if all honest nodes input the same value to the ABA, then all honest nodes can output without invoking a common coin sub-protocol. Hence, we enhance MVBA [17] with a *good-case-no-coin* property, such that no randomness is needed in the optimistic case.

Specifically, the MVBA protocol [17] consists of three main phases: broadcast, leader election, and voting. The latter two phases involve invoking the common coin sub-protocol, with the voting phase essentially being an instance of ABA. To achieve the *good-case-no-coin* property, we make two key modifications. First, during the leader elec-

tion phase, the common coin sub-protocol is modified to output the predetermined value $(e \mod n_e)$ upon its first invocation, ensuring that the first output incurs no cost. Second, in the voting phase, we employ Crain's ABA [43]. In the optimistic case, all nodes will receive the necessary messages from the selected leader. As a result, all nodes will vote for the selected leader in the ABA protocol. According to the *good-case-coin-free* property of Crain's ABA [43], the ABA protocol ensures agreement on the selection without relying on a common coin. Consequently, the MVBA protocol outputs without requiring any common randomness in the optimistic case. This also implies that the *good-case-no-coin* MVBA does not need to employ threshold keys to implement the common coin in the optimistic case. As the ABA protocol [43] incurs a communication cost of $\mathcal{O}(\lambda n^2)$ bits, the overall communication complexity of both MVBA and our DPSS remains at $\mathcal{O}(\lambda n^2)$ in the optimistic case.

If the initial round of MVBA fails, the committee activates a hADKG instance [2] to establish threshold keys, which are utilized to generate common coins for the subsequent rounds of MVBA. This fallback results in a communication complexity of $\mathcal{O}(\lambda n^3)$ and a message complexity of $\mathcal{O}(\lambda n^3)$.

The above modification preserves the properties of MVBA. In our design, all honest nodes attempt to execute the *good-case-no-coin* MVBA using a predetermined common coin in the first round. They can achieve agreement in the optimistic case within just one round. If an agreement is not reached in the first round, the nodes turn to the operation of MVBA [17], which is guaranteed to produce an agreement. Therefore, our *good-case-no-coin* MVBA satisfies the agreement property. Besides, this protocol is ensured to terminate in either the first or subsequent rounds, thus implying the termination of MVBA. The external validity property is also preserved because our modification does not involve the global predicate.

Removing the trusted setup. In the following, we describe a variant with transparent setup. Here, all public parameters consist of publicly sampled random values, eliminating the requirement for secret trapdoors.

In the absence of a bulletin board, an alternative approach is to adopt sublinear-sized commitment schemes and multi-signatures that do not require any trusted setup. Specifically, we employ Bulletproofs [44] for PCom, and Merkle trees for VCom. In wpACSS, the threshold signatures are replaced with $\mathcal{O}(\lambda \log n)$-sized multi-signatures [45]. The *good-case-no-coin* MVBA described above is still adopted. By incorporating all these trusted-setup-free components, we achieve an asynchronous DPSS protocol with a transparent setup. The resulted protocol achieves an optimistic-case communication complexity of $\mathcal{O}(\lambda n^2 \log n)$ and a worst-case communication complexity of $\mathcal{O}(\lambda n^3)$. Similarly, the message complexity in the optimistic case is $\mathcal{O}(\lambda n^2)$, while the worst-case message complexity remains $\mathcal{O}(\lambda n^3)$. Such performance still surpasses the existing asynchronous DPSS protocols [8], [9].

9.2. Non-uniformly Random Secrets

Recall that our DPSS protocol discussed in Section 7 requires the secret to be uniformly random, while many typical applications of DPSS are not constrained by this uniformly random requirement, including: (1) Threshold cryptosystems, where secret keys are inherently uniformly random. (2) The "BMR Escape Hatch" described in [9], which uses uniformly random values as secrets. (3) Cloud storage of confidential files [23], where files can be encrypted, and the uniformly random encryption keys are protected using DPSS.

In this section, we further discuss how to extend our DPSS to protect a non-uniformly random secret s. At a high level, users can employ a threshold ElGamal encryption (TE) scheme to encrypt the non-uniform secret s while periodically refreshing the uniform encryption key using our proposed DPSS. The TE scheme is a variant of the standard ElGamal encryption algorithm, where the secret key is shared among multiple nodes. It consists of four algorithms: TE.KeyGen, TE.Enc, TE.DecShare, and TE.Dec. TE.KeyGen takes a threshold t, a committee size n, and a security parameter λ as inputs to generate a secret key esk, a public key $epk = g^{esk}$, and a set of key shares $\{esk_i, epk_i\}_{i\in[n]}$. TE.Enc encrypts a message m using epk, and the encrypted message is denoted as c. TE.DecShare computes a decryption share d_i from an encrypted message c and a secret key share esk_i. Finally, TE.Dec reconstructs the plaintext message m from at least $t+1$ valid decryption shares $\{d_i\}_{i\in I}$, where I is an index set satisfying $|I| = t+1$.

We leverage the TE primitive to protect a non-uniform secret s. In epoch $e = 1$, the owner of s (the client) invokes TS.KeyGen to uniformly sample a secret key esk, a public key $epk = g^{esk}$. The secret key is shared among the nodes in committee $\mathcal{U}^1$ using DPSS.Share. Meanwhile, the client encrypts s using epk via TE.Enc, and the resulted ciphertext c is disseminated to $\mathcal{U}^1$ via a reliable broadcast protocol [46]. Upon receiving c, each node generates a threshold signature share on c and multicasts it to $\mathcal{U}^1$. The node then collects at least t_1+1 valid signature shares to form a full signature. For every subsequent epoch $e > 1$, the shares of esk are periodically refreshed using our proposed DPSS.Handoff protocol. Simultaneously, each old node multicasts the ciphertext c to the new committee, if it possesses a valid full signature. Once a new node receives at least $t_{e-1}+1$ identical copies of c, it generates a new signature share on c with the updated threshold t_e and multicasts it to $\mathcal{U}^e$. The new node collects at least t_e+1 valid signature shares and forms a full signature on c. Note that at the end of each handoff, every new node obtains a commitment vector $\boldsymbol{v}_{\mathsf{new}}$, which exactly includes all public key shares of new nodes.

When retrieving s, there are two approaches. In the first approach, each node $\mathcal{P}_i^e$ generates a decryption share d_i via TE.DecShare. The owner (client) collects t_e+1 decryption shares and inputs them into TE.Dec, thus obtaining the decrypted secret s as output. The second approach utilizes the DPSS.Recon (cf. Algorithm 8, Appendix C.2) to reconstruct the esk. During this process, the client implicitly obtains

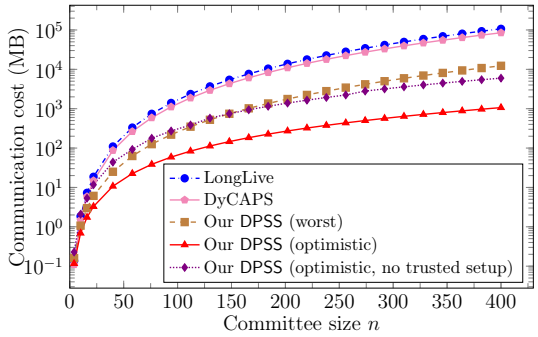

Figure 4: Comparison of estimated communication costs.

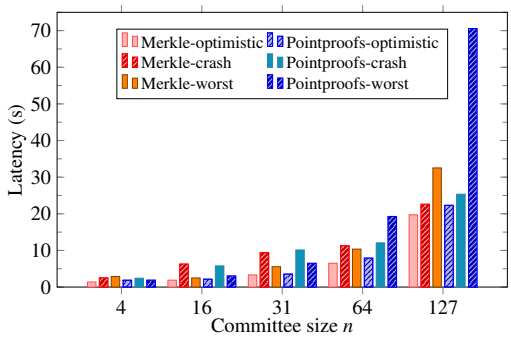

Figure 5: Latency of our DPSS in AWS.

the public key epk and verifies esk against epk. The new committee then sends the valid ciphertext m to the client. Once the client receives at least $t_e + 1$ identical copies of m, it uses esk to decrypt m and recovers s via the standard ElGamal decryption algorithm.

10. Performance Evaluation

We implemented our DPSS in Go, using kilic's bls12-381 curve library [47]. For PCom, we used the KZG scheme implemented by protolambda [48]. For VCom, we employed Pointproofs implemented by Zhang [49]. For threshold signatures, we used Boldyreva's scheme [33] implemented by drand/kyber [50]. Finally, for MVBA, we adopted the implementation of Dumbo-MVBA from [51], which is part of the Dory protocol [52].

We deployed our DPSS protocol on up to 254 Amazon EC2 t2.medium instances, each serving as a node. Each instance has 2 vCPUs and 4GB memory. The nodes were evenly distributed across eight regions: North Virginia, North California, Canada, London, Seoul, Singapore, Sydney, and Sao Paulo. During the experiments, we performed handoffs between two committees of the same size, with the committee size ranging from 4 to 127 nodes. In the following, the evaluation results are under the trusted setup assumption, unless otherwise stated.

In the following, we make a comparison with the state-of-the-art protocols, DyCAPS [8] and LongLive [9]. To ensure a fair comparison, we employ the same MVBA implementation. Besides, the ACSS in LongLive is instantiated by the KZG-based hbACSS [12].

Communication cost. We evaluate the estimated concrete communication of our DPSS for $n \leq 400$. We consider both optimistic and worst cases for our DPSS. In the optimistic case, the batched recovery is never invoked, and the new commitments are generated via the optimistic path. In the worst-case scenario, we require all nodes to execute share recovery for all dealers selected by MVBA, even if the nodes have already received shares from these dealers. Additionally, the nodes generate the new commitments through the pessimistic path.

As depicted in Figure 4, our DPSS outperforms the state-of-the-art DPSS protocols in both optimistic and worst cases. DyCAPS and LongLive exhibit similar communication costs, with DyCAPS achieving slightly lower overhead than LongLive. Since LongLive and our DPSS share the same resharing mechanism, we mainly compare our DPSS against LongLive in the following. In the optimistic case, the communication overhead of our DPSS is a mere 1.0% to 5.2% of LongLive (i.e., about 19–100x smaller), while in the worst case, it is 7.1% to 12.7% (i.e., about 8–17x smaller). The advantage in the optimistic case becomes more pronounced as n scales, owing to the reduction of a $\mathcal{O}(n)$ factor in communication complexity. When there is no trusted setup, the concrete communication cost of our DPSS outperforms LongLive (with per-epoch trusted setup) as early as $n > 13$ in the optimistic case.

Round cost. Table 4 compares the concrete round cost of our DPSS protocol with DyCAPS [8] and LongLive [9]. Here, α_1 and α_2 denote the concrete round costs (expected) of the underlying MVBA protocol in the optimistic and worst cases, respectively. If we employ the Dumbo-MVBA instance from [42], then $\alpha_1 = 9$, $\alpha_2 = 15$. Our DPSS protocol consumes slightly fewer rounds in the optimistic case, and more rounds in the worst case.

Latency. The latency is measured as the average duration between the start time of an old node and the time that a new node outputs a refreshed share. We evaluated the latency of our DPSS in the optimistic case, crash case [27], and worst case, respectively. To further reveal the influence of computational cost, we implemented another version of our DPSS with Merkle tree [53] as VCom. Compared to the Pointproofs-based DPSS, this variant reduces the computational cost but increases the communication complexity from $\mathcal{O}(\lambda n^2)$ to $\mathcal{O}(\lambda n^2 \log n)$ in the optimistic case. Furthermore, we note that this approach cannot be applied directly to LongLive, as it employs Pedersen commitments as PCom but does not use vector commitments. Replacing Pedersen commitments with vector commitments would require a significant redesign of LongLive's construction.

As shown in Figure 5, we observe that our DPSS with Merkle trees as vector commitments is 10% to 20% faster than the Pointproofs version. This difference is due to the Pointproofs scheme consuming more computational resources, and the advantage of constant-length proofs not being apparent. In the worst-case scenario, our DPSS takes about 30% to 210% more runtime than in the optimistic

TABLE 4: Comparison of concrete round cost

Case	Our DPSS	DyCAPS	LongLive
Optimistic	$8 + \alpha_1$	$10 + \alpha_1$	$9 + \alpha_1$
Worst	$12 + \alpha_2$	$10 + \alpha_2$	$9 + \alpha_2$

case. This increase is attributed to the frequent occurrence of share recovery. Especially noteworthy is the low latency when the committee consists of only 4 nodes, where the Merkle-tree version takes just 1.38 seconds and the Pointproofs version takes 1.87 seconds in the optimistic case. Even as the committee size scales up to 64, the latency for each optimistic-case handoff consistently stays below 8 seconds for both versions.

For the crash case, we designate t nodes within each committee to behave as dumb nodes at the start of the handoff, refraining from transmitting any information to others, while the remaining nodes are honest. In this situation, there is no invocation of share recovery instances. However, the honest nodes must wait longer to receive a sufficient number of messages during the handoff because the crashed nodes are not contributing to the communication. As illustrated in Figure 5, the impact of crashed nodes is more pronounced when the committee sizes are 16 and 31, where the crash-case latency is approximately 170% to 230% larger than that in the all-honest scenario. As the committee size reaches 64 and 127, the difference in latency decreases. This happens because the increasing computation time becomes the dominant factor as the committee size n grows, effectively masking the impact of network latency in the overall measurements.

Computational cost. As the LongLive source code [54] supports only local benchmarks, we compared the computational performance on a single EC2 r5.large instance with 16GB memory, assuming all nodes were honest (up to $n = 52$ nodes). To align with the benchmark setting of LongLive, we limited the number of CPU cores to one. In this experiment, the network delays are not accounted.

Considering that our DPSS and LongLive employ similar batching techniques (cf. Section 8.1), it is reasonable to focus on the single-secret scenario. The results are presented in Table 5. The local benchmarks demonstrate that both versions of our DPSS, based on Merkle trees and Pointproofs, respectively, outperform DyCAPS and LongLive in terms of runtime. As n increases, the advantage of our DPSS becomes more pronounced. Notably, our Merkle-tree-based DPSS achieves a runtime reduction of approximately 50% starting from $n = 16$ when compared with LongLive. This performance difference indicates that DyCAPS and LongLive experience a faster increase in latency as n grows.

We highlight that the benefits of our $\mathcal{O}(n^2)$ message complexity may not be fully evident in the above comparison. When deployed in a distributed environment, our DPSS is expected to exhibit even better scalability and efficiency than LongLive.

TABLE 5: Comparison of computational cost (seconds)

Scale	DyCAPS	LongLive	DPSS-Merkle-tree		DPSS-Pointproofs	
$n = 10$	7.61	6.20	3.81	$\downarrow$ 38.5%	5.82	$\downarrow$ 6.20%
$n = 16$	29.41	19.80	10.36	$\downarrow$ 47.7%	15.83	$\downarrow$ 20.0%
$n = 31$	217.76	108.82	44.45	$\downarrow$ 59.1%	68.60	$\downarrow$ 37.0%
$n = 40$	481.89	208.14	79.91	$\downarrow$ 61.6%	131.87	$\downarrow$ 36.6%
$n = 52$	—	426.21	148.74	$\downarrow$ 65.1%	239.65	$\downarrow$ 43.8%

Note: Percentages indicate the improvement relative to LongLive.

11. Conclusion

In this paper, we have designed and implemented an asynchronous DPSS protocol with $\mathcal{O}(n^2)$ message complexity. In the optimistic case, we have further reduced the redundant communication costs of DPSS, while ensuring that our DPSS protocol does not compromise on worst-case performance compared to the state-of-the-art. We first discussed our DPSS with a per-epoch trusted setup assumption and then provided two strategies to weaken it. Experimental results demonstrate a significant reduction in communication and computation costs for our DPSS in the optimistic case.

Our DPSS may benefit BFT-based blockchains by enabling the reconfiguration of shareholder committees. Besides, our weak provable ACSS could potentially enhance the optimistic-case performance of the state-of-the-art ADKG protocols [1], [2]. We leave these explorations as future work.

Acknowledgments

We sincerely thank our shepherd and anonymous reviewers for their constructive comments. The work of Bin, Jianwei, and Zongyang were supported by the National Cryptologic Science Fund of China (2025NCSF02022), the National Natural Science Foundation of China (62472015, U21B2021, 62372020). The work of Qiang was supported in part by ARC Discovery Project (DP250101739), Input Output (iog.io) as part of the Cardano Vision 2026 research initiative, and Research Awards from the Stellar Development Foundation. The work of Zhenliang was supported in part by a grant from City University of Hong Kong (Project No. 9610761).

References

[1] S. Das, T. Yurek, Z. Xiang, A. K. Miller, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous distributed key generation," in *IEEE SP*, 2022, pp. 2518–2534.

[2] S. Das, Z. Xiang, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling," in *USENIX Security*, 2023, pp. 5359–5376.

[3] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller, "HoneyBadgerMPC and asynchromix: Practical asynchronous MPC and its application to anonymous communication," in *ACM CCS*, 2019, pp. 887–903.

[4] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous BFT protocols," in *ACM CCS*, 2020, pp. 803–818.

[5] B. Guo, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Speeding Dumbo: Pushing asynchronous BFT closer to practice," in *NDSS*, 2022, pp. 1–18.

[6] Y. Yan, Y. Xia, and S. Devadas, "Shanrang: Fully asynchronous proactive secret sharing with dynamic committees," IACR Cryptology ePrint Archive, 2022.

[7] C. Levrat, M. Rambaud, and A. Urban, "Breaking the $t < n/3$ consensus bound: Asynchronous dynamic proactive secret sharing under honest majority," IACR Cryptology ePrint Archive, 2023.

[8] Z. Zongyang, H. Bin, C. Han, Z. You, J. Huazu, and L. Jianwei, "Dy-CAPS: Asynchronous dynamic-committee proactive secret sharing," *Chinese Journal of Electronics*, vol. 35, no. 1, pp. 307–321, 2026.

[9] T. Yurek, Z. Xiang, Y. Xia, and A. Miller, "Long live the honey badger: Robust asynchronous DPSS and its applications," in *USENIX Security*, 2023, pp. 5413–5430.

[10] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl, "Asynchronous verifiable secret sharing and proactive cryptosystems," in *ACM CCS*, 2002, pp. 88–97.

[11] L. Zhou, F. B. Schneider, and R. van Renesse, "APSS: proactive secret sharing in asynchronous systems," *ACM Trans. Inf. Syst. Secur.*, vol. 8, no. 3, pp. 259–286, 2005.

[12] T. Yurek, L. Luo, J. Fairoze, A. Kate, and A. Miller, "hbACSS: How to robustly share many secrets," in *NDSS*, 2022.

[13] N. Alhaddad, M. Varia, and H. Zhang, "High-threshold AVSS with optimal communication complexity," in *FC*, ser. LNCS, vol. 12675. Springer, 2021, pp. 479–498.

[14] G. Bracha, "Asynchronous Byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[15] D. Dolev and R. Reischuk, "Bounds on information exchange for Byzantine agreement," *JACM*, vol. 32, no. 1, pp. 191–204, 1985.

[16] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous Byzantine agreement," in *ACM PODC*, 2019, pp. 337–346.

[17] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-MVBA: Optimal multi-valued validated asynchronous Byzantine agreement, revisited," in *ACM PODC*, 2020, pp. 129–138.

[18] S. Basu, A. Tomescu, I. Abraham, D. Malkhi, M. K. Reiter, and E. G. Sirer, "Efficient verifiable secret sharing with share recovery in BFT protocols," in *ACM CCS*, 2019, pp. 2387–2402.

[19] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or: How to cope with perpetual leakage," in *CRYPTO*. Springer, 1995, pp. 339–352.

[20] Y. Desmedt and S. Jajodia, "Redistributing secret shares to new access structures and its applications," Citeseer, Tech. Rep., 1997.

[21] S. K. D. Maram, F. Zhang, L. Wang, A. Low, Y. Zhang, A. Juels, and D. Song, "CHURP: dynamic-committee proactive secret sharing," in *ACM CCS*, 2019, pp. 2369–2386.

[22] V. Goyal, A. Kothapalli, E. Masserova, B. Parno, and Y. Song, "Storing and retrieving secrets on a blockchain," in *PKC*, ser. LNCS, vol. 13177. Springer, 2022, pp. 252–282.

[23] R. Vassantlal, E. Alchieri, B. Ferreira, and A. Bessani, "COBRA: Dynamic proactive secret sharing for confidential BFT services," in *IEEE SP*, 2022, pp. 1335–1353.

[24] D. A. Schultz, B. Liskov, and M. Liskov, "Mobile proactive secret sharing," in *ACM PODC*, 2008, pp. 458–458.

[25] Z. Lu, T. Zhang, A. Fekete, K. Y. Lam, and Q. Tang, "Asynchronous dynamic committee proactive secret sharing for large data," in *IEEE ICDCS*, 2025, pp. 78–88.

[26] S. Das, Z. Xiang, and L. Ren, "Powers of tau in asynchrony," in *NDSS*, 2024, pp. 1–17.

[27] I. Abraham, D. Dolev, A. Kagan, and G. Stern, "Brief announcement: Authenticated consensus in synchronous systems with mixed faults," in *DISC*, ser. LIPIcs, vol. 246, 2022, pp. 38:1–38:3.

[28] A. B. Alexandru, E. Blum, J. Katz, and J. Loss, "State machine replication under changing network conditions," in *ASIACRYPT*, ser. LNCS, vol. 13791. Springer, 2022, pp. 681–710.

[29] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT protocols made practical," in *ACM CCS*, 2022, pp. 2705–2718.

[30] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT*, ser. LNCS, vol. 6477. Springer, 2010, pp. 177–194.

[31] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *CRYPTO*, ser. LNCS, vol. 576. Springer, 1991, pp. 129–140.

[32] S. Gorbunov, L. Reyzin, H. Wee, and Z. Zhang, "Pointproofs: Aggregating proofs for multiple vector commitments," in *ACM CCS*, 2020, pp. 2007–2023.

[33] A. Boldyreva, "Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme," in *PKC*, ser. LNCS, vol. 2567. Springer, 2003, pp. 31–46.

[34] S. Das and L. Ren, "Adaptively secure BLS threshold signatures from DDH and co-cdh," in *CRYPTO*, ser. LNCS. Springer, 2024, pp. 251–284.

[35] S. Agrawal, P. Mohassel, P. Mukherjee, and P. Rindal, "DiSE: Distributed symmetric-key encryption," in *ACM CCS*, 2018, pp. 1993–2010.

[36] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *CRYPTO*. Springer, 2001, pp. 524–541.

[37] Z. Beerliová-Trubíniová and M. Hirt, "Perfectly-secure MPC with linear communication complexity," in *TCC*, ser. LNCS, vol. 4948. Springer, 2008, pp. 213–230.

[38] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *ACM CCS*, 2016, pp. 31–42.

[39] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: practical asynchronous Byzantine agreement using cryptography," in *ACM PODC*, 2000.

[40] S. Das, P. Camacho, Z. Xiang, J. Nieto, B. Bünz, and L. Ren, "Threshold signatures from inner product argument: Succinct, weighted, and multi-threshold," in *ACM CCS*, 2023, pp. 356–370.

[41] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," IACR Cryptology ePrint Archive, 2019.

[42] H. Lipmaa, J. Siim, and M. Zajac, "Counting vampires: From univariate sumcheck to updatable ZK-SNARK," in *ASIACRYPT*, ser. LNCS, vol. 13792. Springer, 2022, pp. 249–278.

[43] T. Crain, "Experimental evaluation of asynchronous binary Byzantine consensus algorithms with $t < n/3$ and $O(n^2)$ messages and $O(1)$ round expected termination," *arXiv preprint arXiv:2002.08765*, 2020.

[44] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *IEEE SP*, 2018, pp. 315–334.

[45] T. Qiu and Q. Tang, "Predicate aggregate signatures and applications," in *ASIACRYPT*, ser. LNCS, vol. 14439. Springer, 2023, pp. 279–312.

[46] N. Alhaddad, S. Das, S. Duan, L. Ren, M. Varia, Z. Xiang, and H. Zhang, "Balanced Byzantine reliable broadcast with near-optimal communication and improved computation," in *ACM PODC*, 2022, pp. 399–417.

[47] kilic, "BLS12-381 library," 2022. [Online]. Available: https://github.com/kilic/bls12-381

[48] protolambda, "Go-KZG library," 2023. [Online]. Available: https://github.com/protolambda/go-kzg

[49] Z. Zhang, "Pointproof," 2022. [Online]. Available: https://github.com/zhenfeizhang/pointproofs

[50] Drand, "Drand/kyber library," 2023. [Online]. Available: https://github.com/drand/kyber

[51] DoryTeam, "Dory BFT consensus," 2023. [Online]. Available: https://github.com/xygdys/Dory-BFT-Consensus

[52] Z. Zhang, Y. Zhou, S. Duan, H. Zhang, B. Hu, L. Wang, and J. Liu, "Dory: Faster asynchronous BFT with reduced communication for permissioned blockchains," IACR Cryptology ePrint Archive, 2023.

[53] C. Bergoon, "Merkle tree in golang," 2022. [Online]. Available: https://github.com/cbergoon/merkletree

[54] T. Yurek, "DPSS library," 2022. [Online]. Available: https://github.com/tyurek/dpss

[55] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *FOCS*. IEEE Computer Society, 1987, pp. 427–437.

[56] D. Catalano and D. Fiore, "Vector commitments and their applications," in *PKC*, ser. LNCS, vol. 7778. Springer, 2013, pp. 55–72.

[57] B. Libert, A. Passelègue, and M. Riahinia, "Pointproofs, revisited," in *ASIACRYPT*, ser. LNCS, vol. 13794. Springer, 2022, pp. 220–246.

[58] M. Naor, B. Pinkas, and O. Reingold, "Distributed pseudo-random functions and KDCs," in *EUROCRYPT*, ser. LNCS, vol. 1592. Springer, 1999, pp. 327–346.

[59] S. Das, Z. Xiang, and L. Ren, "Asynchronous data dissemination and its applications," in *ACM CCS*, 2021, pp. 2705–2721.

Appendix A. Instantiations of Building Blocks

Polynomial commitment (PCom) [30], [55] allows a node commit to a polynomial and provide witnesses of its evaluations. We mainly use four algorithms: PCom.Setup, PCom.Commit, PCom.Witness, and PCom.VerifyEval.

- $pp \leftarrow \mathsf{PCom.Setup}(t, 1^\lambda)$: this algorithm initializes the public parameters. It takes as inputs a degree bound t and a security parameter λ in unary form. The output is public parameters pp. We sometimes omit pp for simplicity.
- $PC_\phi \leftarrow \mathsf{PCom.Commit}(\phi(x), pp)$: this algorithm commits to a polynomial. It takes as inputs a polynomial $\phi(x) \in \mathbb{Z}_p[x]$ and public parameters pp. The output is a commitment PC_ϕ.
- $\langle \phi(i), w^i \rangle \leftarrow \mathsf{PCom.Witness}(\phi(x), i, pp)$: this algorithm creates a witness for polynomial evaluation. It takes as inputs a polynomial $\phi(x)$, an index i, and public parameters pp. The output is an value $\phi(i)$ and a witness w^i.
- $0/1 \leftarrow \mathsf{PCom.VrfyEval}(PC_\phi, i, v, w^i, pp)$: this algorithm verifies a polynomial evaluation. It takes as inputs a commitment PC_ϕ, an index i, an evaluation v, a witness w^i, and public parameters pp. It outputs 1 iff $v = \phi(i)$.

We use the KZG scheme [30] for the polynomial commitment, due to its constant commitment and witness size. It has the following properties except with negligible probability.

- *Correctness.* For all honestly generated public parameters and a fixed commitment, the output of PCom.Witness always passes PCom.VrfyEval.
- *Degree Binding.* If t is the input to PCom.Setup and $t' > t$, an adversary cannot commit to a degree-t' polynomial.
- *Evaluation Binding.* Given a commitment PC_ϕ, an adversary cannot generate two witnesses, w^i and w'^i, that both pass PCom.VrfyEval and satisfy $\phi(i) \neq \phi'(i)$.
- *Hiding.* Given t evaluation-witness pairs $\langle i, \phi(i), w^i \rangle$ and the commitment PC_ϕ to a degree-t polynomial $\phi(x)$, an adversary cannot determine $\phi(i')$ for any unqueried i'.
- *Homomorphism.* The commitment to $\phi(x) = \phi_1(x) + \phi_2(x)$ can be computed as $PC_\phi = PC_{\phi_1} PC_{\phi_2}$.[5] Similarly, $w^i_\phi = w^i_{\phi_1} w^i_{\phi_2}$ holds for $\phi(i) = \phi_1(i) + \phi_2(i)$.

Verifiable secret sharing (VSS) [13], [31] allows a dealer to share a secret among n nodes, such that each node can verify the correctness of its share. We mainly use four algorithms: VSS.Share, VSS.Vrfy, VSS.Recon, and VSS.MakeSecret.

- $\{PC_s, \langle s_i, w^i_s \rangle_{i \in [n]}\} \leftarrow \mathsf{VSS.Share}(t, n, s, 1^\lambda)$: this algorithm generates secret shares. It takes as inputs a threshold t, a committee size n, a secret s, and a security parameter λ in unary form. The output is a polynomial commitment PC_s and a set of share-witness pairs.
- $0/1 \leftarrow \mathsf{VSS.Vrfy}(PC_s, i, s_i, w^i_s)$: this algorithm verifies a secret share. It takes as input a polynomial commitment PC_s, an index i, a secret share s_i and its witness w^i_s. It outputs 1 iff s_i is a valid share of s.
- $s/\perp \leftarrow \mathsf{VSS.Recon}(t, \{s_i\}_{i \in I})$: this algorithm reconstructs a secret. It takes as inputs a threshold t and a set of shares $\{s_i\}_{i \in I}$, where $|I| \geq t + 1$. The output is a secret s or an empty symbol $\perp$.
- $\phi(x) \leftarrow \mathsf{VSS.MakeSecret}(t, I, \{y_i\}_{i \in I})$: this algorithm generates a polynomial from a set of points. It takes as inputs a threshold t, an index set I, and a set of values $\{y_i\}_{i \in I}$, where $|I| \leq t$. It samples $t + 1 - |I|$ random points and interpolates a degree-t polynomial $\phi(x)$ as output, satisfying that $\phi(i) = y_i, \forall i \in I$.

The above algorithms are all non-interactive, so they are oblivious to the network model. A VSS protocol has the following properties [30] except with negligible probability.

- *Secrecy.* An adversary learns nothing about the secret, given t shares and the public information.
- *Correctness.* If the dealer is honest, the output of VSS.Recon is the same as the input to VSS.Share. If the dealer misbehaves, the output of VSS.Recon is either the same as the input to VSS.Share or $\perp$.
- *Agreement.* Whenever two distinct nodes invoke VSS.Recon, they will always output the same value.

5. Specially, in the context of the KZG [30] scheme, g^s is a polynomial commitment to $\phi_1(x) = s$. Hence, $PC_\phi = g^s PC_{\phi_2}$ can be seen as the commitment to $\phi(x) = s + \phi_2(x)$.

Vector commitment (VCom) [32], [56], [57] allows a node to commit to a vector and later open some of its elements. We mainly use four algorithms: VCom.Setup, VCom.Commit, VCom.Open, and VCom.Vrfy.

- $pp \leftarrow \mathsf{VCom.Setup}(n, 1^\lambda)$: this algorithm initializes the public parameters. It takes as inputs a vector size n and a security parameter λ in unary form. The output is public parameters pp. We sometimes omit pp for simplicity.
- $VC_{\boldsymbol{v}} \leftarrow \mathsf{VCom.Commit}(\boldsymbol{v}, pp)$: this algorithm commits to a vector. It takes as inputs a vector $\boldsymbol{v} = \{v_1, ..., v_n\}$ and public parameters pp. The output is a commitment $VC_{\boldsymbol{v}}$.
- $\langle v_i, \pi^i_{\boldsymbol{v}} \rangle \leftarrow \mathsf{VCom.Open}(\boldsymbol{v}, i, pp)$: this algorithm creates an opening proof for the i-th position in vector. It takes as inputs a vector $\boldsymbol{v}$, an index i, and public parameters pp. The output is an evaluation v_i and a proof $\pi^i_{\boldsymbol{v}}$.
- $0/1 \leftarrow \mathsf{VCom.Vrfy}(VC_{\boldsymbol{v}}, i, v_i, \pi^i_{\boldsymbol{v}}, pp)$: this algorithm verifies a opening proof. It takes as inputs a commitment $VC_{\boldsymbol{v}}$, an index i, a vector element v_i, a proof $\pi^i_{\boldsymbol{v}}$, and public parameters pp. It outputs 1 iff v_i is the i-th element of $\boldsymbol{v}$.

We use Pointproofs [32] for the vector commitment, due to its constant commitment size and proof size. It satisfies the following properties except with negligible probability.

- *Correctness.* For all honestly generated public parameters and a fixed vector commitment, the output of VCom.Open always passes VCom.Vrfy.
- *Binding.* Given a fixed vector commitment and public parameters, an adversary cannot generate two distinct vector-proof tuples that both pass VCom.Vrfy at the same position.
- *Hiding.* Given a commitment and several opening proofs of a vector, an adversary cannot obtain the information of any other elements in the vector.

Threshold signature (TS) [33] allows a quorum of nodes to jointly sign a message. It has five algorithms: TS.KeyGen, TS.SignShare, TS.VrfyShare, TS.Combine, and TS.Vrfy.

- $\{tpk, \langle tpk_i, tsk_i \rangle_{i\in[n]}\} \leftarrow \mathsf{TS.KeyGen}(t, n, 1^\lambda)$: this algorithm generates threshold signing key pairs. It takes as inputs a threshold t, a committee size n, and a security parameter λ in unary form. The output is a threshold public key tpk, a threshold public key share set $\{tpk_i\}_{i\in[n]}$, and a threshold secret key share set $\{tsk_i\}_{i\in[n]}$. Each node $\mathcal{P}_i$ is assigned with $\langle tpk, \{tpk_i\}_{i\in[n]}, tsk_i \rangle$. We sometimes omit tpk, tsk_i, and tpk_i for simplicity.
- $\sigma^*_{m,i} \leftarrow \mathsf{TS.SignShare}_t(m, tsk_i)$: this algorithm generates a signature share. It takes as inputs a message m and a threshold secret key share tsk_i, and it outputs signature share $\sigma^*_{m,i}$.
- $0/1 \leftarrow \mathsf{TS.VrfyShare}_t(m, i, \sigma^*_{m,i}, tpk_i)$: this algorithm verifies a signature share. It takes as inputs a message m, an index i, a signature share $\sigma^*_{m,i}$, and a threshold public key share tpk_i. It outputs 1 iff $\sigma^*_{m,i}$ is correctly generated via $\mathsf{TS.SignShare}_t(m, tsk_i)$.
- $\sigma_m \leftarrow \mathsf{TS.Combine}_t(m, \langle i, \sigma^*_{m,i} \rangle_{i\in I})$: this algorithm generates a full signature. It takes as inputs a message m, and a set of shares $\langle i, \sigma^*_{m,i} \rangle_{i\in I}$, where $|I| > t$. The output is a full signature σ_m of message m.
- $0/1 \leftarrow \mathsf{TS.Vrfy}_t(m, \sigma_m, tpk)$: this algorithm verifies a full signature. It takes as inputs a message m, a full signature σ_m, and a threshold public key tpk. It outputs 1 iff the signature is validated by tpk.

We use Boldyreva's scheme [33] for the threshold signature, which satisfies the following properties except with negligible probability.

- *Unforgeability.* An adversary cannot forge a valid full signature of any unqueried message m.
- *Robustness.* Every honest node can generate a valid full signature as long as it receives at least $t + 1$ valid signature shares generated by TS.SignShare.

Distributed pseudorandom function (DPRF) [35], [58] allows a quorum of nodes to jointly evaluate a pseudorandom function. It includes six algorithms: DPRF.Init, DPRF.Contrib, DPRF.VrfyContrib, DPRF.Combine, DPRF.Eval, and DPRF.Vrfy.

- $\langle dsk, dpk, PC_{dsk}, VC_{dpk}, \{dsk_i, dpk_i, w^i_{dsk}, \pi^i_{dpk}\}_{i\in[n]} \rangle \leftarrow \mathsf{DPRF.Init}(t, n, 1^\lambda)$: this algorithm initializes DPRF key pairs. It takes as inputs a threshold t, a committee size n, and a security parameter λ in unary form. The output is a DPRF key pair $\langle dsk, dpk \rangle$, a polynomial commitment PC_{dsk}, a vector commitment VC_{dpk}, and n tuples $\langle dsk_i, dpk_i, w^i_{dsk}, \pi^i_{dpk} \rangle_{i\in[n]}$. Here, dsk_i is $\mathcal{P}_i$'s secret key share, dpk_i is $\mathcal{P}_i$'s public key share, w^i_{dsk} is a witness to dsk_i and π^i_{dpk} is an opening proof of dpk_i. We sometimes omit dpk and dpk_i for simplicity.
- $\langle F_i(x), \mathsf{Proof}^i_{\mathsf{dprf}} \rangle \leftarrow \mathsf{DPRF.Contrib}(x, dsk_i, dpk_i, \pi^i_{dpk})$: this algorithm generates a DPRF contribution. The input is a value x, a DPRF secret key share dsk_i, a public key share dpk_i, and an opening proof π^i_{dpk} of dpk_i. The output is a DPRF contribution $F_i(x)$ and its proof $\mathsf{Proof}^i_{\mathsf{dprf}}$.
- $0/1 \leftarrow \mathsf{DPRF.VrfyContrib}(i, x, F_i(x), \mathsf{Proof}^i_{\mathsf{dprf}})$: this algorithm verifies a DPRF contribution. It takes as inputs an index i, a value x, a contribution $F_i(x)$, and a proof $\mathsf{Proof}^i_{\mathsf{dprf}}$. It outputs 1 iff $F_i(x)$ is correctly generated via DPRF.Contrib with $(x, dsk_i, dpk_i, \pi^i_{dpk})$ as inputs.
- $y \leftarrow \mathsf{DPRF.Combine}(t, x, \{F_i(x), \mathsf{Proof}^i_{\mathsf{dprf}}\}_{i\in I})$: this algorithm generates a DPRF evaluation from at least $t + 1$ DPRF contributions. It takes as inputs a threshold t, a value x, a contribution set $\{F_i(x), \mathsf{Proof}^i_{\mathsf{dprf}}\}_{i\in I}$, where I is the index set and $|I| > t$. The output is a DPRF evaluation $y = F(x)$.
- $y \leftarrow \mathsf{DPRF.Eval}(x, dsk)$: this algorithm directly evaluates a DPRF. It takes as inputs a value x and

a DPRF secret key dsk. The output is a DPRF evaluation $y = F(x)$.

- $0/1 \leftarrow \mathsf{DPRF.Vrfy}(y, x, dpk)$: this algorithm verifies a DPRF evaluation. It takes as inputs an evaluation y, a value x, and a DPRF public key dpk. It outputs 1 iff $y = F(x)$.

We provide a specific DPRF instantiation in Algorithm 6, which is adapted from Figure 6 in the paper by Agrawal et al. [35]. A DPRF scheme satisfies the following properties except with negligible probability.

- *Consistency.* Given a fixed x, any $t+1$ valid contributions leads to the same DPRF ealuation $y = F(x)$.
- *Pseudorandomness.* A PPT adversary gains no advantage on guessing $y = F(x)$ for any unqueried x.
- *Correctness.* A PPT adversary cannot generate valid contributions that lead to an incorrect DPRF evaluation $y \neq F(x)$.

Multi-valued validated Byzantine agreement (MVBA) protocol [16], [17], [36] enables a group of nodes to submit individual proposals and reach an agreement on an output that satisfies a global predicate P_{MVBA}. The MVBA protocol satisfies the following properties except with negligible probability.

- *Termination.* If all honest nodes input proposals satisfying P_{MVBA}, then each honest node outputs a value.
- *Agreement.* If two honest nodes output in the same instance, then the output values are the same.
- *External-validity.* If an honest node outputs a value v, then v is external valid, i.e., $P_{\mathsf{MVBA}}(v) = 1$.

We utilize the MVBA protocol introduced by Lu et al. [17]. This protocol offers a time complexity of $\mathcal{O}(1)$ and a message complexity of $\mathcal{O}(n^2)$. Moreover, the communication complexity is $\mathcal{O}(n|m| + \lambda n^2)$, where $|m|$ represents the size of input.

Appendix B. DPRF Instantiation

Our DPRF instantiation is shown in Algorithm 6, which is adapted from Figure 6 in Agrawal et al's work [35].

Algorithm 5 DPRF, adapted from Fig. 6 in [35]

Let $\mathcal{H}_1 : \{0,1\}^* \to \mathbb{G}$ and $\mathcal{H}_2 : \{0,1\}^* \to \mathbb{Z}_p$ be hash functions, where $\mathbb{G}$ is a multiplicative cyclic group with g as the generator; $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ be a bilinear pairing.

function $\mathsf{DPRF.Init}(t, n, 1^\lambda)$ ▷ initialize DPRF keys
  $dsk \leftarrow_\$ \mathbb{Z}_p$
  $dpk \leftarrow g^{dsk}$
  $\{PC_{dsk}, \langle dsk_i, w^i_{dsk}\rangle_{i\in[n]}\} \leftarrow \mathsf{VSS.Share}(t, n, dsk)$
  $\boldsymbol{v}_{dpk} \leftarrow \{dpk_1, \cdots, dpk_n\}$, where $dpk_i \leftarrow g^{dsk_i}$ for $i \in [n]$
  $VC_{dpk} \leftarrow \mathsf{VCom.Commit}(\boldsymbol{v}_{dpk})$
  for $i \in [n]$ **do**
    $\langle dpk_i, \pi^i_{dpk}\rangle \leftarrow \mathsf{VCom.Open}(\boldsymbol{v}_{dpk}, i)$
  return $\langle dsk, dpk, PC_{dsk}, VC_{dpk}, \{dsk_i, dpk_i, w^i_{dsk}, \pi^i_{dpk}\}_{i\in[n]}\rangle$
▷ verify a DPRF key share
function $\mathsf{DPRF.VrfyKey}(dsk_i, dpk_i, PC_{dsk}, VC_{dpk}, w^i_{dsk}, \pi^i_{dpk})$
  if $dpk_i = g^{dsk_i} \wedge \mathsf{VSS.Verify}(PC_{dsk}, i, dsk_i, w^i_{dsk}) = 1$ **then**
    if $\mathsf{VCom.Vrfy}(VC_{dpk}, i, dpk_i, \pi^i_{dpk}) = 1$ **then**
      return 1
  return 0
▷ generate a DPRF contribution
function $\mathsf{DPRF.Contrib}(x, dsk_i, dpk_i, \pi^i_{dpk})$
  $w \leftarrow \mathcal{H}_1(x)$
  $W_i \leftarrow w^{dsk_i}$
  $r_i \leftarrow_\$ \mathbb{Z}_p$
  $c_i \leftarrow \mathcal{H}_2(W_i, w, dpk_i, g, w^{r_i})$
  $u_i \leftarrow r_i - c_i \times dsk_i$
  denote W_i as $F_i(x)$
  $\mathsf{Proof}^i_{\mathsf{dprf}} \leftarrow \langle w, c_i, u_i, dpk_i, \pi^i_{dpk}\rangle$
  return $\langle F_i(x), \mathsf{Proof}^i_{\mathsf{dprf}}\rangle$
function $\mathsf{DPRF.VrfyContrib}(i, x, W_i, \mathsf{Proof}^i_{\mathsf{dprf}})$
  parse $\mathsf{Proof}^i_{\mathsf{dprf}}$ as $\langle w, c_i, u_i, dpk_i, \pi^i_{dpk}\rangle$
  if $\mathcal{H}_1(x) \neq w$ **then**
    return 0
  wait until receiving VC_{dpk}
  if $\mathsf{VCom.Vrfy}(VC_{dpk}, i, dpk_i, \pi^i_{dpk}) = 1$ **then**
    if $c_i = \mathcal{H}_2(W_i, w, dpk_i, g, w^{u_i} W_i^{c_i})$ **then**
      return 1
  return 0
▷ combine $t+1$ contributions
function $\mathsf{DPRF.Combine}(t, x, \{F_i(x), \mathsf{Proof}^i_{\mathsf{dprf}}\}_{i\in I})$
  if $|I| > t$ **then**
    if $\mathsf{DPRF.VrfyContrib}(i, x, F_i(x), \mathsf{Proof}^i_{\mathsf{dprf}}) = 1, \forall i \in I$ **then**
      interpolate $F(x)$ from $\{F_i(x)\}_{i\in I}$
      return $F(x)$
function $\mathsf{DPRF.Eval}(x, dsk)$ ▷ output a DPRF evaluation
  return $F(x) \leftarrow \mathcal{H}_1(x)^{dsk}$
function $\mathsf{DPRF.Vrfy}(y, x, dpk)$ ▷ verify a DPRF evaluation
  if $e(y, g) = e(\mathcal{H}_1(x), dpk)$ **then**
    return 1
  return 0

Appendix C. Secret Sharing and Reconstruction in DPSS

C.1. Secret Sharing in DPSS

In our DPSS.Handoff protocol (Section 7), we require that each old node initially holds a commitment vector $\boldsymbol{v} = \{g^{s_i}\}_{i\in[n]}$, where s_i is $\mathcal{P}_i$'s share of s. Hence, in epoch $e = 1$, we need a sharing protocol DPSS.Share that distributes valid shares and the required commitment vector among all honest nodes. We observe that the ACSS protocol by Das et al. [1] meets our requirements, so we adapt their protocol in Algorithm 7.

The dealer $\mathcal{P}_d$ first invokes VSS.Share to generate n secret shares $\{s_i\}_{i\in[n]}$. Then, for each $i \in [n]$, it invokes $\mathsf{EncAndProve}(pk_i, g, s_i)$ and output $\langle v_i, c_i, \pi_i\rangle$, where $v_i = g^{s_i}$ is the commitment to s_i, c_i is the encryption of s_i using $\mathcal{P}_i$'s encryption public key pk_i, and π_i is a zero-knowledge proof that proves v_i and c_i corresponds to the same s_i. Finally, $\mathcal{P}_d$ uses a reliable broadcast instance RBC to broadcast $(\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi})$ to the committee $\mathcal{U}^1$, where $\boldsymbol{v} = \{v_i\}_{i\in[n]}, \boldsymbol{c} = \{c_i\}_{i\in[n]}, \boldsymbol{\pi} = \{\pi_i\}_{i\in[n]}$. When a node

Algorithm 6 DPRF, adapted from Fig. 6 in [35]

Let $\mathcal{H}_1 : \{0,1\}^* \to \mathbb{G}$ and $\mathcal{H}_2 : \{0,1\}^* \to \mathbb{Z}_p$ be hash functions, where $\mathbb{G}$ is a multiplicative cyclic group with g as the generator; $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ be a bilinear pairing.

function DPRF.Init($t, n, 1^\lambda$) ▷ initialize DPRF keys
  $dsk \leftarrow_\$ \mathbb{Z}_p$
  $dpk \leftarrow g^{dsk}$
  $\{PC_{dsk}, \langle dsk_i, w^i_{dsk}\rangle_{i\in[n]}\} \leftarrow$ VSS.Share(t, n, dsk)
  $\boldsymbol{v}_{dpk} \leftarrow \{dpk_1, \cdots, dpk_n\}$, where $dpk_i \leftarrow g^{dsk_i}$ for $i \in [n]$
  $VC_{dpk} \leftarrow$ VCom.Commit$(\boldsymbol{v}_{dpk})$
  for $i \in [n]$ **do**
    $\langle dpk_i, \pi^i_{dpk}\rangle \leftarrow$ VCom.Open$(\boldsymbol{v}_{dpk}, i)$
  return $\langle dsk, dpk, PC_{dsk}, VC_{dpk}, \{dsk_i, dpk_i, w^i_{dsk}, \pi^i_{dpk}\}_{i\in[n]}\rangle$
▷ verify a DPRF key share
function DPRF.VrfyKey($dsk_i, dpk_i, PC_{dsk}, VC_{dpk}, w^i_{dsk}, \pi^i_{dpk}$)
  if $dpk_i = g^{dsk_i} \wedge$ VSS.Verify$(PC_{dsk}, i, dsk_i, w^i_{dsk}) = 1$ **then**
    if VCom.Vrfy$(VC_{dpk}, i, dpk_i, \pi^i_{dpk}) = 1$ **then**
      return 1
  return 0
▷ generate a DPRF contribution
function DPRF.Contrib($x, dsk_i, dpk_i, \pi^i_{dpk}$)
  $w \leftarrow \mathcal{H}_1(x)$
  $W_i \leftarrow w^{dsk_i}$
  $r_i \leftarrow_\$ \mathbb{Z}_p$
  $c_i \leftarrow \mathcal{H}_2(W_i, w, dpk_i, g, w^{r_i})$
  $u_i \leftarrow r_i - c_i \times dsk_i$
  denote W_i as $F_i(x)$
  $\mathsf{Proof}^i_{\mathsf{dprf}} \leftarrow \langle w, c_i, u_i, dpk_i, \pi^i_{dpk}\rangle$
  return $\langle F_i(x), \mathsf{Proof}^i_{\mathsf{dprf}}\rangle$
function DPRF.VrfyContrib($i, x, W_i, \mathsf{Proof}^i_{\mathsf{dprf}}$)
  parse $\mathsf{Proof}^i_{\mathsf{dprf}}$ as $\langle w, c_i, u_i, dpk_i, \pi^i_{dpk}\rangle$
  if $\mathcal{H}_1(x) \neq w$ **then**
    return 0
  wait until receiving VC_{dpk}
  if VCom.Vrfy$(VC_{dpk}, i, dpk_i, \pi^i_{dpk}) = 1$ **then**
    if $c_i = \mathcal{H}_2(W_i, w, dpk_i, g, w^{u_i} W_i^{c_i})$ **then**
      return 1
  return 0
▷ combine $t+1$ contributions
function DPRF.Combine($t, x, \{F_i(x), \mathsf{Proof}^i_{\mathsf{dprf}}\}_{i\in I}$)
  if $|I| > t$ **then**
    if DPRF.VrfyContrib$(i, x, F_i(x), \mathsf{Proof}^i_{\mathsf{dprf}}) = 1, \forall i \in I$ **then**
      interpolate $F(x)$ from $\{F_i(x)\}_{i\in I}$
      return $F(x)$
function DPRF.Eval(x, dsk) ▷ output a DPRF evaluation
  return $F(x) \leftarrow \mathcal{H}_1(x)^{dsk}$
function DPRF.Vrfy(y, x, dpk) ▷ verify a DPRF evaluation
  if $e(y, g) = e(\mathcal{H}_1(x), dpk)$ **then**
    return 1
  return 0

$\mathcal{P}_i \in \mathcal{U}^1$ outputs $(\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi})$ from RBC, it verifies that (1) the elements in $\boldsymbol{v}$ can be interpolated by any subset $\boldsymbol{v}' \subset \boldsymbol{v}$, $|\boldsymbol{v}'| = t+1$, which implies $\boldsymbol{v}$ contains the commitments to a degree-t sharing polynomial; (2) the commitment-encryption pair $(\boldsymbol{v}, \boldsymbol{c})$ can be verified by $\boldsymbol{\pi}$, which ensures that all commitments are bound with the encrypted shares. If the verification pass, each node $\mathcal{P}_i$ decrypts its own share s_i from $c_i \in \boldsymbol{c}$, signs $\boldsymbol{v}$, and multicasts the signature. Finally, when $\mathcal{P}_i$ receives $t+1$ valid and consistent signatures on $\boldsymbol{v}$, it outputs $\langle s_i, \boldsymbol{v}\rangle$. Note that the $t+1$ signatures are also locally stored by $\mathcal{P}_i$.

Algorithm 7 DPSS.Share, adapted from Alg. 2 in [1]

Let $\mathcal{P}_d$ be the dealer; $\mathcal{U}^1$ be the initial committee with n nodes, t of which are corrupted; $\{sk_i, pk_i\}_{i\in[n]}$ be the encryption key pairs of all nodes, where each $\mathcal{P}_i$ holds sk_i and $\{pk_i\}_{i\in[n]}$.

As a dealer $\mathcal{P}_d$:
upon receiving s as input **do**
  generate $\{s_i\}_{i\in[n]}$ by VSS.Share(t, n, s)
  for $i \in [n]$ **do**
    $\langle v_i, c_i, \pi_i\rangle \leftarrow$ EncAndProve(pk_i, g, s_i)
  $\boldsymbol{v} \leftarrow \{v_i\}_{i\in[n]}$ ▷ $v_i = g^{s_i}$
  $\boldsymbol{c} \leftarrow \{c_i\}_{i\in[n]}$ ▷ $c_i =$ Encrypt(pk_i, s_i)
  $\boldsymbol{\pi} \leftarrow \{\pi_i\}_{i\in[n]}$ ▷ π_i proves v_i and c_i corresponds to the same s_i
  input $(\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi})$ to RBC ▷ broadcast to $\mathcal{U}^1$
As a node $\mathcal{P}_i \in \mathcal{U}^1$:
upon outputting $(\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi})$ from RBC **do**
  if all elements in $\boldsymbol{v}$ can be interpolated by any subset $\boldsymbol{v}' \subset \boldsymbol{v}$, where $|\boldsymbol{v}'| = t+1$ **then**
    if the verification of $\boldsymbol{\pi}$ w.r.t. $(\boldsymbol{v}, \boldsymbol{c})$ returns true **then**
      $s_i \leftarrow$ Decrypt(sk_i, c_i) ▷ c_i is the i-th element in $\boldsymbol{c}$
      sign $\boldsymbol{v}$ and multicast the signature
      wait for $t+1$ valid and consistent signatures on $\boldsymbol{v}$
        output $\langle s_i, \boldsymbol{v}\rangle$

Algorithm 8 DPSS.Recon[ID]

Let $\boldsymbol{v}_{\mathsf{new}} = \{g^{s'_1}, ..., g^{s'_{n'}}\}$ be the commitments to the current shares

As a new node $\mathcal{P}'_i \in \mathcal{U}'$: ▷ send shares
upon receiving RECON from client $\mathcal{L}$ **do**
  $VC_{\mathsf{new}} \leftarrow$ VCom.Commit$(\boldsymbol{v}_{\mathsf{new}})$
  $\langle g^{s'_i}, \pi^i_{\mathsf{new}}\rangle \leftarrow$ VCom.Open$(\boldsymbol{v}_{\mathsf{new}}, i)$
  send $\langle$RECON, ID, $VC_{\mathsf{new}}, s'_i, \pi^i_{\mathsf{new}}\rangle$ to client $\mathcal{L}$
As a client $\mathcal{L}$: ▷ receive shares
upon receiving $\langle$RECON, ID, $VC_{\mathsf{new}}, s'_i, \pi^i_{\mathsf{new}}\rangle$ from $\mathcal{P}_i$ **do**
  if VCom.Vrfy$(VC_{\mathsf{new}}, i, g^{s'_i}, \pi^i_{\mathsf{new}}) = 1$ **then**
    record $\langle$ID, $VC_{\mathsf{new}}, i, s'_i\rangle$
    if recorded $t'+1$ valid tuples with same ID and VC_{new} **then**
      interpolate s from $\{(i, s'_i)\}_{i\in I}$, I is the corresponding index set
      return s

Given a RBC protocol [59] with $\mathcal{O}(n|m|+\lambda n^2)$ communication complexity, where $|m|$ is the input size, the overall communication complexity of DPSS.Share is $\mathcal{O}(\lambda n^2)$.

C.2. Secret Reconstruction in DPSS

The reconstruction protocol DPSS.Recon (Algorithm 8) allows a client $\mathcal{L}$ to reconstruct the secret from the new committee $\mathcal{U}'$. The client $\mathcal{L}$ has no information about the commitments to the current shares, i.e., $\boldsymbol{v}_{\mathsf{new}}$.

As shown in Algorithm 8, upon invocation, every node within the current epoch (denoted by $\mathcal{P}'_i$) sends its share s'_i to the client $\mathcal{L}$, along with a vector commitment VC_{new} and an opening proof π^i_{new} of $g^{s'_i}$. The client $\mathcal{L}$ waits for $t'+1$ valid messages with the same VC_{new}, and then it interpolates the secret s from $t'+1$ valid shares $\{s'_i\}_{i\in I}$, where I is the index set such that $|I| = t'+1$.

We remark that DPSS.Recon can also be implemented by an online error correcting (OEC) algorithm, where the client has to wait at least $2t'+1$ shares before reconstruction.

Appendix D. Deferred Proofs for wpACSS

Lemma 1 (Termination of wpACSS)**.** *If the dealer $\mathcal{P}_d$ is honest, then all honest nodes output valid shares in* wpACSS.Share*, and $\mathcal{P}_d$ outputs a valid proof of sharing.*

Proof. If the dealer $\mathcal{P}_d$ is honest, then it will generate a valid value-proof tuple $\langle V_i, \Pi_i \rangle$ for each node $\mathcal{P}_i \in \mathcal{U}$ (lines 1–15, Algorithm 1, same below). Hence, every honest node $\mathcal{P}_i$ receives such a tuple containing its deserved valid share. Before $\mathcal{P}_i$ outputting this tuple, it has to receive a valid READY message which contains a valid full signature $\sigma_{\mathsf{r},d}$ (line 39). This condition is satisfied since the dealer $\mathcal{P}_d$ will receive at least $n-t$ ECHO messages as the responses to the SHARE messages, each containing a valid signature share (lines 29–38). These shares are sufficient to formulate a valid signature $\sigma_{\mathsf{r},d}$ in READY message (lines 16–22). Therefore, if the honest dealer $\mathcal{P}_d$ invokes wpACSS.Share, all honest nodes will output valid shares in line 43.

On the other side, before the honest nodes outputting, they will each send a FINISH message to $\mathcal{P}_d$ (line 42). Each FINISH message contains a valid signature share, so $\mathcal{P}_d$ is guaranteed to formulate a full signature $\sigma_{\mathsf{f},d}$ (lines 23–28) and output a valid proof of sharing $\langle m_d, \sigma_{\mathsf{f},d} \rangle$, where m_d is generated by itself during wpACSS.Share (lines 1–17). □

Lemma 2 (Provability of wpACSS)**.** *If the dealer $\mathcal{P}_d$ outputs a valid proof of sharing from* wpACSS.Share*, then at least $t+1$ honest nodes output valid shares in* wpACSS.Share*, and these honest nodes are able to help others to recover shares via* wpACSS.BatchedRecover.

Proof. In our model, the threshold key pairs are initialized by a trusted setup. Hence, the adversary cannot generate a proof of sharing by itself. If the dealer $\mathcal{P}_d$ outputs a valid proof of sharing from wpACSS.Share, then $\mathcal{P}_d$ has received at least $n-t$ FINISH messages from distinct nodes (lines 23–28), among which $n-2t \geq t+1$ nodes are honest. These honest nodes will output value-proof tuples in line 43, each tuple contains a secret share.

We now prove the shares output by these honest nodes are valid. Without loss of generality, we denote these honest nodes as $\{\mathcal{P}_i\}_{i\in I}$, where I is the index set. The verifications in lines 32–34 ensure that each tuple $\langle V_i, \Pi_i \rangle$ is correctly formulated, i.e., the values in V_i is valid w.r.t. the commitments in Π_i, where $i \in I$. Besides, according to the verification in line 40, each node $\mathcal{P}_i$ ensures that the other nodes in $\{\mathcal{P}_i\}_{i\in I}$ receive the same m_d, which includes the commitments to the sharing and recovery polynomials (line 36). Hence, the nodes $\{\mathcal{P}_i\}_{i\in I}$ will output valid tuples that are bound with the same set of commitments. Namely, at least $t+1$ honest nodes obtain valid shares from wpACSS.Share.

Next, we prove these honest nodes are able to help others to recover shares in wpACSS.BatchedRecover. For each caller $\mathcal{P}_j$, the honest nodes $\{\mathcal{P}_i\}_{i\in I}$ will generate valid responses (lines 4–10, Algorithm 2), using the tuples $\langle V_i, \Pi_i \rangle_{i\in I}$ output from wpACSS.Share. Hence, the conditions in lines 12–14 of Algorithm 2 will be satisfied, so that $\mathcal{P}_j$ can recover its share by going through lines 15–19. □

Lemma 3 (Correctness of wpACSS)**.** *If the dealer $\mathcal{P}_d$ inputs a secret s in* wpACSS.Share *and outputs a valid proof of sharing, then there exists a fixed secret $\hat{s}$, such that any $t+1$ shares from honest nodes can reconstruct the secret $\hat{s}$. If the dealer is honest, then $s = \hat{s}$.*

Proof. By Lemma 2, we derive that if the dealer $\mathcal{P}_d$ outputs a valid proof of sharing, then every honest node can obtain output a valid share from either wpACSS.Share or wpACSS.BatchedRecover. We use $\mathcal{U}_{\mathsf{sh}}$ and $\mathcal{U}_{\mathsf{re}}$ to include the honest nodes that have outputs in wpACSS.Share and wpACSS.BatchedRecover, respectively.

We first prove that all nodes in $\mathcal{U}_{\mathsf{sh}}$ output consistent shares. An honest node $\mathcal{P}_i \in \mathcal{U}_{\mathsf{sh}}$ only outputs the value-proof tuple $\langle V_i, \Pi_i \rangle$ when it receives a valid signature $\sigma_{\mathsf{r},d}$ (line 39, Algorithm 1). Given there are at most t corrupted nodes, if there are two distinct message-signature pairs $\langle m_d, \sigma_{\mathsf{r},d} \rangle$ and $\langle m'_d, \sigma'_{\mathsf{r},d} \rangle$, such that $m_d \neq m'_d$ and both signatures are valid, then at least one honest node have signed to both m_d and m'_d, leading to a contradiction. Hence, there is at most one valid message-signature pair $\langle m_d, \sigma_{\mathsf{r},d} \rangle$, which implies that the value-proof tuples output by all honest nodes in $\mathcal{U}_{\mathsf{sh}}$ are bound with the same m_d. As m_d contains the polynomial commitment PC_s to the sharing polynomial, the shares output by the honest nodes in $\mathcal{U}_{\mathsf{sh}}$ are bound with the same PC_s.

Next, we prove the shares output by the nodes in $\mathcal{U}_{\mathsf{re}}$ are also bound with PC_s. According to the conditions in lines 12–14, Algorithm 2, an honest node $\mathcal{P}_j \in \mathcal{U}_{\mathsf{re}}$ it has to wait for $t+1$ valid HELP responses with the same commitments $(VC^d_{dpk}, PC^d_{\mathsf{mask}})$ before recovering its share. As there are at most t corrupted nodes, at least one response comes from honest nodes, which ensures the commitments are correctly computed. The vector commitment VC^d_{dpk} ensures that the DPRF evaluation in line 18 is correct. Furthermore, the polynomial commitment PC^d_{mask} guarantees that the masked share is correctly computed from the secret share and DPRF evaluation. Hence, the recovered share is consistent with the shares held by $\mathcal{U}_{\mathsf{sh}}$, the shares of both $\mathcal{U}_{\mathsf{sh}}$ and $\mathcal{U}_{\mathsf{re}}$ are bounded by PC_s.

Finally, we prove any $t+1$ shares from honest nodes lead to a fixed $\hat{s}$. We have proved that all shares output by honest nodes in $\mathcal{U}_{\mathsf{sh}}$ and $\mathcal{U}_{\mathsf{re}}$ are bound with the same polynomial commitment PC_s. Hence, there exists a sharing polynomial $\hat{f}(x)$, such that $\mathsf{PCom.Commit}(\hat{f}(x)) = PC_s$, $\hat{f}(0) = \hat{s}$, and the degree of $\hat{f}(x)$ is t. Therefore, any $t+1$ shares from honest nodes can reconstruct the secret $\hat{s}$. Moreover, if the dealer is honest, then $\hat{f}(x) = f(x)$, $s = \hat{s}$. □

Lemma 4 (Secrecy of wpACSS)**.** *If the dealer $\mathcal{P}_d$ is honest and it shares a uniformly random secret $s \in \mathbb{Z}_q$ via* wpACSS*, then a static PPT adversary learns no extra information about the secret beyond g^s.*

Simulator Sim_1 for $\mathsf{wpACSS.Share}[\langle \mathsf{ID}, d\rangle]$

1: $pp_1 \leftarrow \mathsf{PCom.Setup}(t, 1^\lambda)$
2: $pp_2 \leftarrow \mathsf{VCom.Setup}(n, 1^\lambda)$
3: $\{tpk, \{tpk_i\}_{i\in[n]}, \{tsk_i\}_{i\in[n]}\} \leftarrow \mathsf{TS.KeyGen}(t, n, 1^\lambda)$
4: Sim_1 reserves the trapdoor τ during the setup of PCom
5: Sim_1 samples t uniformly random shares $\{s_i\}_{\mathcal{P}_i\in\mathcal{U}_{\mathsf{crp}}}$
6: Denote $f(x)$ as the sharing polynomial, such that $f(0) = s$ for an unknown s and $f(i) = s_i$ for each $\mathcal{P}_i \in \mathcal{U}_{\mathsf{crp}}$
7: Sim_1 interpolates $PC_s = g^{f(\tau)}$ using g^s and $\{g^{s_i}\}_{\mathcal{P}_i\in\mathcal{U}_{\mathsf{crp}}}$
8: Sim_1 computes $w_s^i = g^{\frac{f(\tau)-s_i}{\tau-i}} = (\frac{PC_s}{g^{s_i}})^{(\tau-i)^{-1}}$ for each $\mathcal{P}_i \in \mathcal{U}_{\mathsf{crp}}$
9: $PC_z \leftarrow PC_s/g^s$
10: $w_z^0 \leftarrow (PC_z)^{\tau^{-1}}$
11: $\langle dsk_i, \{\phi_k(i)\}_{k\in[\ell]}, \mathsf{Proof}^i_{\mathsf{rec}}\rangle_{i\in[n]} \leftarrow \mathsf{GenRecPoly}[\langle \mathsf{ID}, d\rangle](t, n)$
12: Sim_1 constructs the vector $\boldsymbol{v}_s$ as follows:
(1) Set $\boldsymbol{v}_s[i]$ as g^{s_i} for each $\mathcal{P}_i \in \mathcal{U}_{\mathsf{crp}}$
(2) Interpolate $\boldsymbol{v}_s[j]$ using g^s and $\{g^{s_i}\}_{\mathcal{P}_i\in\mathcal{U}_{\mathsf{crp}}}$ for each $\mathcal{P}_j \in \mathcal{U}\backslash\mathcal{U}_{\mathsf{crp}}$
13: $VC_s \leftarrow \mathsf{VCom.Commit}(\boldsymbol{v}_s)$
14: $\langle g^{s_i}, \pi_s^i\rangle \leftarrow \mathsf{VCom.Open}(\boldsymbol{v}_s, i)$ for $i \in [n]$
15: For each corrupted node $\mathcal{P}_i \in \mathcal{U}_{\mathsf{crp}}$:
(1) $V_i \leftarrow \langle s_i, dsk_i, \{\phi_k(i)\}_{k\in[\ell]}\rangle$
(2) $\Pi_i \leftarrow \langle g^s, PC_s, w_s^i, PC_z, w_z^0, VC_s, \pi_s^i, \mathsf{Proof}^i_{\mathsf{rec}}\rangle$
(3) Sim_1 sends $\langle \textsc{Share}, \mathsf{ID}, V_i, \Pi_i\rangle$ to $\mathcal{P}_i$
16: For each honest node $\mathcal{P}_i \in \mathcal{U}\backslash\mathcal{U}_{\mathsf{crp}}$:
(1) $V_i \leftarrow \langle 0, dsk_i, \{\phi_k(i)\}_{k\in[\ell]}\rangle$
(2) $\Pi_i \leftarrow \langle g^s, PC_s, \perp, PC_z, w_z^0, VC_s, \pi_s^i, \mathsf{Proof}^i_{\mathsf{rec}}\rangle$
(3) Sim_1 sends $\langle \textsc{Share}, \mathsf{ID}, V_i, \Pi_i\rangle$ to $\mathcal{P}_i$
17: Sim_1 follows lines 16–43 of Algorithm 1 to interact with $\mathcal{A}$
18: Sim_1 outputs a proof of sharing as the honest dealer $\mathcal{P}_d$, and outputs $\langle V_i, \Pi_i\rangle$ as each honest node $\mathcal{P}_i$

Figure 6: Simulator Sim_1 for $\mathsf{wpACSS.Share}[\langle \mathsf{ID}, d\rangle]$

Proof. We prove that for a uniformly random secret s, given g^s, there exists a PPT simulator Sim_1 that can simulate the view of any static PPT adversary $\mathcal{A}$. We denote the set of corrupted nodes as $\mathcal{U}_{\mathsf{crp}}$, such that $\mathcal{U}_{\mathsf{crp}} \subset \mathcal{U}$ and $|\mathcal{U}_{\mathsf{crp}}| \leq t$.

We first construct a simulator for $\mathsf{wpACSS.Share}[\langle \mathsf{ID}, d\rangle]$ as detailed in Figure 6. The major steps are as follows.

(1) Sim_1 runs the setups for the underlying components, including PCom, VCom, and TS. Notably, it reserves the trapdoor τ for the PCom instantiation.

(2) Sim_1 samples t uniformly random shares $\{s_i\}_{\mathcal{P}_i\in\mathcal{U}_{\mathsf{crp}}}$. For ease of expression, we denote $f(x)$ as the sharing polynomial, such that $f(0) = s$ for an unknown s and $f(i) = s_i$ for each $\mathcal{P}_i \in \mathcal{U}_{\mathsf{crp}}$. Using the trapdoor τ and the $t+1$ exponents $\{g^s, \{g^{s_i}\}_{\mathcal{P}_i\in\mathcal{U}_{\mathsf{crp}}}\}$, Sim_1 interpolates the polynomial commitment $PC_s = g^{f(\tau)}$ at position $x = \tau$. It also computes the witnesses $\{w_s^i\}_{\mathcal{P}_i\in\mathcal{U}_{\mathsf{crp}}}$ and the commitment-witness pair $\langle PC_z, w_z^0\rangle$ of the zero polynomial.

(3) Sim_1 invokes the GenRecPoly function to initiate the DPRF key pairs and the recovery polynomials. In addition, Sim_1 generates the vector $\boldsymbol{v}_s$ using interpolation and computes its commitment VC_s and the opening proofs $\{\pi_s^i\}_{i\in[n]}$.

(4) With the above elements, Sim_1 can compose a valid SHARE message for each corrupted node $\mathcal{P}_i \in \mathcal{U}_{\mathsf{crp}}$, and it then interacts with the corrupted nodes following lines 16–43 of Algorithm 1 on behalf of the simulated dealer $\mathcal{P}_d$. Notably, Sim_1 uses the threshold signing keys of honest nodes to generate the READY messages sent to the corrupted nodes. In the end, Sim_1 outputs a proof of sharing as the honest dealer $\mathcal{P}_d$, and outputs $\langle V_i, \Pi_i\rangle$ as each honest node $\mathcal{P}_i$.

Next, we prove that the view of $\mathcal{A}$ in its interaction with Sim_1 is computationally indistinguishable from its view in the real-world protocol. In the adversary's view, it receives t SHARE and READY messages from $\mathcal{P}_d$. Each SHARE message contains a value-proof tuple $\langle V_i, \Pi_i\rangle$, where $\mathcal{P}_i \in \mathcal{U}_{\mathsf{crp}}$. For ease of comparison, we denote the simulated tuple as $\langle \hat{V}_i, \hat{\Pi}_i\rangle$. The adversary's task is to distinguish the following two tuples.

$$V_i = \langle s_i, dsk_i, \{\phi_k(i)\}_{k\in[\ell]}\rangle,$$
$$\Pi_i = \langle g^s, PC_s, w_s^i, PC_z, w_z^0, VC_s, \pi_s^i, \mathsf{Proof}^i_{\mathsf{rec}}\rangle$$

and

$$\hat{V}_i = \langle \hat{s}_i, dsk_i, \{\phi_k(i)\}_{k\in[\ell]}\rangle,$$
$$\hat{\Pi}_i = \langle g^s, PC_{\hat{s}}, w_{\hat{s}}^i, PC_{\hat{z}}, w_{\hat{z}}^0, VC_{\hat{s}}, \pi_{\hat{s}}^i, \mathsf{Proof}^i_{\mathsf{rec}}\rangle$$

Note that the elements $\langle dsk_i, \{\phi_k(i)\}_{k\in[\ell]}, \mathsf{Proof}^i_{\mathsf{rec}}\rangle$ are honestly generated by the GenRecPoly function in both worlds, so they are computationally indistinguishable. Since both $s_i \in V_i$ and $\hat{s}_i \in \hat{V}_i$ are uniformly random, and the simulated elements $\langle PC_{\hat{s}}, w_{\hat{s}}^i, PC_{\hat{z}}, w_{\hat{z}}^0, VC_{\hat{s}}, \pi_{\hat{s}}^i\rangle \in \hat{\Pi}_i$ are ensured to pass all validations in lines 32–34, Algorithm 1, the adversary cannot distinguish these elements in the two tuples. Therefore, the two tuples are computationally indistinguishable. Then, in each READY message, there is a combination of commitments and an honestly generated signature, where the commitments are extracted from Π_i. Hence, the indistinguishability of the value-proof tuples also implies that the READY messages are also indistinguishable by $\mathcal{A}$.

In the wpACSS.BatchedRecover sub-protocol, the adversary $\mathcal{A}$ may call for help on behalf of corrupted nodes. As claimed in VSSR [18], the responses from honest nodes reveal no information about the secret. Hence, we omit the simulation and the indistinguishability discussion for wpACSS.BatchedRecover for simplicity.

Altogether, the view of the adversary $\mathcal{A}$ in the interaction with the simulator is computationally indistinguishable from the view in real-world execution. Hence, we conclude that $\mathcal{A}$ learns no extra information about the uniformly random secret s beyond g^s. □

Combining Lemma 1–4, we get Theorem 5 as below.

Theorem 5. *The tuple of protocols described in Algorithm 1 and Algorithm 2 satisfies the termination, provability, correctness, and secrecy properties of wpACSS except with negligible probability.*

Appendix E. Deferred Proofs for DPSS.Handoff

Lemma 6 (Termination of handoff)**.** *If all honest nodes in committees $\mathcal{U}$ and $\mathcal{U}'$ invoke* DPSS.Handoff*, then all honest old nodes halt, and all honest new nodes have outputs.*

Proof. We first prove that all honest old nodes halt in the handoff protocol. Following our adversary model, there are

three types of old nodes: (1) corrupted nodes, (2) honest in epoch $e-1$ but corrupted in epoch e, and (3) the others. The term "honest old node" refers to the third type. In current epoch e, there are at least $t+1$ honest old nodes, and these nodes have old shares and a common commitment vector $\boldsymbol{v}_{\mathsf{old}}$. Hence, these honest nodes can compose the COM messages and invoke wpACSS.Share with their old shares as inputs (lines 5–6, Algorithm 3). By the termination of wpACSS (Lemma 1), every honest node $\mathcal{P}_i$ that invokes wpACSS.Share is guaranteed to receive a proof of sharing as output, which is used to formulate the PROOF message. Hence, all honest old nodes will multicast the necessary messages and halt.

Then, we prove that all honest new nodes have outputs in the handoff protocol. Since at least $t+1$ honest old nodes have the same commitment vector $\boldsymbol{v}_{\mathsf{old}}$, each honest new node $\mathcal{P}'_i$ is guaranteed to receive $t+1$ valid COM messages with the same VC_{old} and interpolate the commitments in line 10. Besides, the honest new nodes will receive at least $t+1$ valid PROOF messages, so all honest new nodes input valid proposals to MVBA (lines 11–16). By the termination of MVBA, each honest new node obtains the same proposal $\overline{W}$ and index set I (lines 17–18). For each $j \in I$, there exists a valid proof of sharing $\langle m\bar{n}_j, \bar{\sigma}_j \rangle \in \overline{W}$. Hence, if $\mathcal{P}'_i$ invokes wpACSS.BatchedRecover (line 23), it is guaranteed to recover the missing shares by Lemma 2. Then, $\mathcal{P}'_i$ is ensured to obtain all sub-shares $\{s_{j,i}\}_{j\in I}$ and interpolate its new share s'_i (line 24). Afterward, each $\mathcal{P}'_i$ calls GetNewCom (Algorithm 4) to generate the new commitment vector $\boldsymbol{v}_{\mathsf{new}}$.

In GetNewCom, if all new nodes multicast valid NEW-COM messages, then each $\mathcal{P}'_i$ interpolates the correct $g^{s'_0}$ and returns $\boldsymbol{v}_{\mathsf{new}}$ (lines 1–4, Algorithm 4), leading to the output of DPSS.Handoff. If any ERROR message appears, each new node waits for $t'+1$ sets of $\{g^{s_{k,j}}\}_{k\in I}$ for distinct j (lines 12–14, Algorithm 4). When an honest node $\mathcal{P}'_i$ receives a valid AUX message from $\mathcal{P}'_j$, there might be two cases: (1) $I_j = I$ and (2) $I_j \neq I$. In the first case, the j-th commitment $g^{s'_j}$ can be directly interpolated from the information in the AUX message (lines 19–20, Algorithm 4). As for the second case, since there exists a valid proof of sharing for each $k \in I \backslash I_j$, at least $t'+1$ honest new nodes have output valid shares, and they will contribute $g^{s_{k,*}}$ via the AUX messages. Upon collecting $t'+1$ such valid AUX messages that contain $g^{s_{k,*}}$, $\mathcal{P}'_i$ interpolates the missing $g^{s_{k,j}}$ and enter lines 19–20 (Algorithm 4) to compute $g^{s'_j}$. In both cases, $\mathcal{P}'_i$ obtains the j-th new commitment. Hence, $\mathcal{P}'_i$ can collect at least $t'+1$ new commitments and compute the new vector $\boldsymbol{v}_{\mathsf{new}}$. □

Lemma 7 (Correctness of handoff)**.** *The secret value s stays invariant across the executions of* DPSS.Handoff.

Proof. The adversary has the ability to eliminate the commitment vector $\boldsymbol{v}_{\mathsf{old}}$ or forge a fake vector in the corrupted nodes. However, considering that the adversary can corrupt at most t nodes, they cannot generate $t+1$ valid signatures. Hence, at the beginning of each DPSS.Handoff, every honest old node checks whether it has $t+1$ valid signatures on the commitment vector $\boldsymbol{v}_{\mathsf{old}}$, ensuring the consistency and correctness of $\boldsymbol{v}_{\mathsf{old}}$. Then, following the procedures of DPSS.Handoff, the new nodes obtain a common index set I (line 18, Algorithm 3), where $|I| = t+1$. For each $j \in I$, the old node $\mathcal{P}_j$ reshares its share s_j using a degree-t' sharing polynomial $f_j(x)$, such that $f_j(0) = s_j$. The commitment vector $\boldsymbol{v}_{\mathsf{old}}$ ensures that the input to each wpACSS.Share instance is the correct old share. By the external validity of MVBA, the proofs of sharing from $\{\mathcal{P}_j\}_{j\in I}$ are all valid. Hence, by the provability of wpACSS (Lemma 2), each new node $\mathcal{P}'_i$ outputs valid shares $\{s_{j,i}\}_{j\in I}$, where $s_{j,i} = f_j(i)$. The $t+1$ polynomials $\{f_j(x)\}_{j\in I}$ form a degree-$\langle t', t\rangle$ bivariate polynomial $B(x,y)$, such that $B(x,j) = f_j(x)$. Each new node $\mathcal{P}'_i$ interpolates its new share $s'_i = B(i,0)$ from $\{s_{j,i}\}_{j\in I} = \{B(i,j)\}_{j\in I}$. Besides, the old share held by $\mathcal{P}_i$ can be represented by $s_i = B(0,i)$. Hence, the secret value $s = B(0,0)$ can be reconstructed from $t+1$ old shares or $t'+1$ new shares, so the secret value s remains invariant. □

Simulator Sim_2 for DPSS.Handoff[ID]

1: Sim_2 obtains $\{s_i, \boldsymbol{v}_{\mathsf{old}}\}_{\mathcal{P}_i\in\mathcal{U}_{\mathsf{crp}}}$ of the prior epoch from the simulated execution of DPSS.Handoff, where $\boldsymbol{v}_{\mathsf{old}} = \{g^{s_1}, g^{s_2}, ..., g^{s_n}\}$
2: Sim_2 initialize the setups for PCom, VCom, TS, and PKI, and it reserves the trapdoor τ of PCom
3: Sim_2 simulates each honest old node $\mathcal{P}_i$ following lines 1–7, Algorithm 3. Specially, Sim_2 runs Sim_1 for each wpACSS.Share[$\langle$ID, $i\rangle$] instance, with $g^{s_i} \in \boldsymbol{v}_{\mathsf{old}}$ as input
4: Sim_2 simulates honest new nodes following lines 8–17, Algorithm 3, and obtains the common index set I
5: Sim_2 runs the wpACSS.BatchedRecover instances (possibly invoked by $\mathcal{A}$) following lines 18–23, Algorithm 3
6: Sim_2 extracts the new shares $\{s'_i\}_{\mathcal{P}'_i\in\mathcal{U}'_{\mathsf{crp}}}$ of corrupted nodes as follows:
(1) For each honest dealer $\mathcal{P}_j$ satisfying $j \in I$, Sim_2 already knows the simulated subshare $s_{j,i}$ held by each corrupted new node $\mathcal{P}'_i$
(2) For each corrupted dealer $\mathcal{P}_j$ satisfying $j \in I$, since there exists a proof of share, at least $t'+1$ simulated honest new nodes have received valid subshares. Sim_2 uses these subshares to reconstruct the sharing polynomial $f_j(x)$ and then evaluates $s_{j,i} = f_j(i)$ for $\mathcal{P}'_i \in \mathcal{U}'_{\mathsf{crp}}$
(3) The new share s'_i for $\mathcal{P}'_i$ is interpolated from $\{(j, s_{j,i})\}_{j\in I}$
7: Sim_2 constructs the new commitment vector $\boldsymbol{v}_{\mathsf{new}}$ as follows:
(1) Set $\boldsymbol{v}_{\mathsf{new}}[i]$ as $g^{s'_i}$ for each $\mathcal{P}_i \in \mathcal{U}_{\mathsf{crp}}$
(2) Interpolate $\boldsymbol{v}_{\mathsf{new}}[j]$ using g^s and $\{g^{s'_i}\}_{\mathcal{P}_i\in\mathcal{U}_{\mathsf{crp}}}$ for each $\mathcal{P}_j \in \mathcal{U}\backslash\mathcal{U}_{\mathsf{crp}}$
8: Sim_2 runs lines 25–28, Algorithm 3, using $\boldsymbol{v}_{\mathsf{new}}$

Figure 7: Simulator Sim_2 for DPSS.Handoff[ID]

Lemma 8 (Secrecy of handoff)**.** *Given a uniformly random secret s, a static PPT adversary does not gain any additional knowledge about the secret from* DPSS.Handoff *beyond g^s.*

Proof. We prove that for a uniformly random secret s, given g^s, there exists a PPT simulator Sim that can simulate the view of any static PPT adversary $\mathcal{A}$. We denote the sets of corrupted nodes in the old and new committees as $\mathcal{U}_{\mathsf{crp}}$ and $\mathcal{U}'_{\mathsf{crp}}$, respectively, such that $\mathcal{U}_{\mathsf{crp}} \subset \mathcal{U}$, $\mathcal{U}'_{\mathsf{crp}} \subset \mathcal{U}'$, $|\mathcal{U}_{\mathsf{crp}}| \leq t$, and $|\mathcal{U}'_{\mathsf{crp}}| \leq t'$.

We design the simulator Sim_2 for our DPSS.Handoff as detailed in Figure 7. The specific steps are as follows.

(1) Sim_2 first obtains $\{s_i, \boldsymbol{v}_{\mathsf{old}}\}_{\mathcal{P}_i\in\mathcal{U}_{\mathsf{crp}}}$ of the prior epoch from the simulated execution of DPSS.Handoff. Then, Sim_2 initializes the underlying components, including PCom,

VCom, TS, and the PKI for signing keys. Besides, Sim_2 reserves the trapdoor τ for PCom.

(2) Sim_2 simulates each honest old node $\mathcal{P}_i$, following lines 1–7, Algorithm 3. Specially, for each invocation of wpACSS.Share$[\langle \mathsf{ID}, i\rangle]$, Sim_2 runs Sim_1 with $g^{s_i} \in \boldsymbol{v}_{\mathsf{old}}$ as input. Sim_2 also simulates the honest new nodes following lines 8–23, running the MVBA[ID] instance and any wpACSS.BatchedRecover instances invoked by either honest nodes or the adversary. The common index set obtained in line 17 is denoted as I.

(3) Sim_2 extracts the new shares $\{s'_i\}_{\mathcal{P}'_i \in \mathcal{U}'_{\mathsf{crp}}}$ of corrupted new nodes as follows. For each honest dealer $\mathcal{P}_j$ identified by I, Sim_2 already knows the simulated subshare $s_{j,i}$ held by each corrupted node $\mathcal{P}'_i$. For each corrupted dealer $\mathcal{P}_j$ identified by I, due to the external validity of MVBA, there exists a valid proof of sharing for $\mathcal{P}_j$, so at least $t'+1$ honest new nodes have received a valid subshare from $\mathcal{P}_j$. Sim_2 then uses these subshares to reconstruct the sharing polynomial $f_j(x)$ and then evaluates $s_{j,i} = f_j(i)$ for each $\mathcal{P}'_i \in \mathcal{U}'_{\mathsf{crp}}$. Once Sim_2 recovers all subshares sent by nodes $\{\mathcal{P}_j\}_{j\in I}$, it interpolates the new share s'_i for each $\mathcal{P}'_i \in \mathcal{U}'_{\mathsf{crp}}$ from $\{(j, s_{j,i})\}_{j\in I}$.

(4) Sim_2 constructs the new commitment vector $\boldsymbol{v}_{\mathsf{new}}$ using g^s and $\{g^{s'_i}\}_{\mathcal{P}_i \in \mathcal{U}_{\mathsf{crp}}}$, where s'_i is obtained in the prior step. Finally, Sim_2 runs lines 25–28, Algorithm 3, using $\boldsymbol{v}_{\mathsf{new}}$, which includes the GetNewCom[ID] instance and the interactions to obtain $t'+1$ signatures on $\boldsymbol{v}_{\mathsf{new}}$.

Next, we prove that the view of $\mathcal{A}$ in its interaction with Sim_2 is computationally indistinguishable from its view in the real-world protocol.

During the execution of DPSS.Handoff, $\mathcal{A}$ receives $n-t$ COM messages (line 5) , $n-t$ PROOF messages (line 7), and it interacts with the honest nodes in n wpACSS.Share instances (line 6), one MVBA instance (line 16), several optional wpACSS.BatchedRecover instances (line 23), and one GetNewCom instance (line 25). Moreover, $\mathcal{A}$ interacts with honest nodes through lines 26–28.

Firstly, since the old shares in both worlds are uniformly random, the simulated vector $\boldsymbol{v}_{\mathsf{old}}$ has the same distribution as that in the real world. Therefore, the COM messages, which are honestly generated using the vector $\boldsymbol{v}_{\mathsf{old}}$ in both worlds, are indistinguishable by the adversary. Next, by the secrecy of wpACSS (Lemma 4), the wpACSS.Share and wpACSS.BatchedRecover instances are indistinguishable. As the PROOF messages each contains an output of wpACSS.Share instance, they are also indistinguishable by $\mathcal{A}$. Besides, as the MVBA instance is honestly executed, the views of $\mathcal{A}$ within MVBA in the two worlds are identical. Finally, the messages in the GetNewCom instance are derived from the simulated elements in the wpACSS.Share instances, so the adversary cannot distinguish whether the GetNewCom instance is executed in the real world or the ideal world.

Altogether, the view of $\mathcal{A}$ in the interaction with the simulator is computationally indistinguishable from the view in real-world execution. Hence, $\mathcal{A}$ does not gain any extra knowledge about the secret from DPSS.Handoff beyond g^s. □

DPSS (cf. Section 7)	**bDPSS (cf. Section 8.1)**
1: Each $\mathcal{P}_i$ reshares its old share s_i to the new committee. $\mathcal{P}_i$ multicasts the proofs of sharing to the new committee.	1: Each $\mathcal{P}_i$ shares $B/(n-2t)$ random values to both old and new committees, using different polynomials. $\mathcal{P}_i$ multicasts the proofs of sharing to the old committee.
2: The new committee runs MVBA to identify $t+1$ old nodes who correctly reshare their old shares.	2: The old committee runs MVBA to identify $n-t$ old nodes who correctly share all random values. The old committee disseminates the index set I to the new committee.
	3: Each $\mathcal{P}_i$ (resp., $\mathcal{P}'_i$) extracts B shares, denoted as $\{r_{k,i}\}_{k\in[B]}$ (resp., $\{r'_{k,i}\}_{k\in[B]}$) based on the index set I.
	4: The old committee reconstructs $\{s_k + r_k\}_{k\in[B]}$ and sends them to the new committee.
3: Each $\mathcal{P}'_i$ computes its new share s'_i according to the MVBA output.	5: Each $\mathcal{P}'_i$ waits for $t+1$ consistent reconstructed values and computes $\{s'_{k,i} = s_k + r_k - r'_{k,i}\}_{k\in[B]}$ as its new shares.
4: The new nodes interact to generate the commitment vector $\boldsymbol{v}_{\mathsf{new}}$, which contains commitments to all new shares.	

Figure 8: Comparison of DPSS and bDPSS.

Combining Lemma 6–8, we obtain Theorem 9 stated below.

Theorem 9. *The protocol described in Algorithm 3 satisfies the termination, correctness, and secrecy properties of the* DPSS.Handoff *protocol except with negligible probability.*

Appendix F. Details of our Batched DPSS

Algorithm 9 shows details of bDPSS.Handoff described in Section 8.1, where B is the batch size and M is a hyperinvertible matrix (referred to as H matrix). The primary differences between bDPSS and DPSS is highlighted in Figure 8. The detailed procedure is as follows:

- Each old node $\mathcal{P}_i$ locally samples $B/(n-2t)$ random values, denoted as $\{r^i_{\mathsf{local},k}\}_{k\in[B/(n-2t)]}$. For each $k \in [B/(n-2t)]$, $\mathcal{P}_i$ shares the random value $r^i_{\mathsf{local},k}$ to $\mathcal{U}$ (resp., $\mathcal{U}'$) with a threshold t (resp., t'). At the end of sharing, $\mathcal{P}_i$ outputs a set of proofs of sharing $\langle m_{k,i}, m'_{k,i}, \sigma_{\mathsf{f},k,i}, \sigma'_{\mathsf{f},k,i}\rangle_{k\in[B/(n-2t)]}$, where $m_{k,i}$ and $m'_{k,i}$ have a common prefix $\mathsf{ID}||i||g^{r^i_{\mathsf{local},k}}$. This proof is multicasted to the old committee.
- The old committee collects $n-t$ proofs of sharing and then runs MVBA to identify $n-t$ old nodes who correctly shares all $B/(n-2t)$ random values. The predicate of MVBA requires that all signatures in the proofs of sharing are valid, and every pair of $\langle m_{k,d}, m'_{k,d}\rangle$ has the same prefix, ensuring that the dealer (old node) $\mathcal{P}_d$ shares the same $r^d_{\mathsf{local},k}$ to the old and new committees. Then,

Algorithm 9 bDPSS.Handoff[ID] between $\mathcal{U}$ and $\mathcal{U}'$, with B as the batch size and M as a hyperinvertible matrix

Let $\mathcal{U}$ be the old committee with n nodes, t of which are corrupted;
Let $\mathcal{U}'$ be the new committee with n' nodes, t' of which are corrupted;
Let the predicate of MVBA be: $P_{\mathsf{MVBA}}(W) \equiv (W$ contains $n-t$ sets of $\{m_{k,i}, m'_{k,i}, \sigma_{k,i}, \sigma'_{k,i}\}_{k\in[B/(n-2t)]}$ for distinct indexes i, such that every pair of $m_{k,i}$ and $m'_{k,i}$ have the same prefix $\mathsf{ID}||i||g^{r^i_{\mathsf{local},k}}$, and $\langle \sigma_{k,i}, \sigma'_{k,i}\rangle$ are valid w.r.t. $\langle m_{k,i}, m'_{k,i}\rangle)$

As an old node $\mathcal{P}_i \in \mathcal{U}$:
Init: $W_i \leftarrow \emptyset$; $I_i \leftarrow \emptyset$; $S^i_{\mathsf{help}} \leftarrow \emptyset$; start $\mathsf{wpACSS.Share}[\langle \mathsf{ID}, d, k, 1\rangle]$ for all $d \in [n]$ and $k \in [B/(n-2t)]$ to receive shares
▷ share $B/(n-2t)$ random values

1: **upon** receiving batch size B as input **do**
2: **for** $k \in [B/(n-2t)]$ **do**
3: randomly sample $r^i_{\mathsf{local},k}$
4: $\langle m_{k,i}, \sigma_{k,i}\rangle \leftarrow \mathsf{wpACSS.Share}[\langle \mathsf{ID}, i, k, 1\rangle](\mathcal{U}, t, n, r^i_{\mathsf{local},k})$
5: $\langle m'_{k,i}, \sigma'_{k,i}\rangle \leftarrow \mathsf{wpACSS.Share}[\langle \mathsf{ID}, i, k, 2\rangle](\mathcal{U}', t', n', r^i_{\mathsf{local},k})$
6: multicast $\langle \textsc{Proof}, \mathsf{ID}, \{m_{k,i}, m'_{k,i}, \sigma_{k,i}, \sigma'_{k,i}\}_{k\in[B/(n-2t)]}\rangle$ to $\mathcal{U}$

7: **upon** receiving $\langle \textsc{Proof}, \mathsf{ID}, \{m_{k,j}, m'_{k,j}, \sigma_{k,j}, \sigma'_{k,j}\}_{k\in[B/(n-2t)]}\rangle$ from $\mathcal{P}_j$ **do**
8: **if** for each $k \in [B/(n-2t)]$, $m_{k,j}$ and $m'_{k,j}$ have the same prefix $\mathsf{ID}||j||g^{r^j_{\mathsf{local},k}} \wedge \langle \sigma_{k,j}, \sigma'_{k,j}\rangle$ are valid w.r.t. $\langle m_{k,j}, m'_{k,j}\rangle$ **then**
9: $I_i \leftarrow I_i \cup \{j\}$
10: $W_i \leftarrow W_i \cup \{m_{k,j}, m'_{k,j}, \sigma_{k,j}, \sigma'_{k,j}\}_{k\in[B/(n-2t)]}$
11: **if** $|I_i| = n - t$ **then**
12: input W_i into $\mathsf{MVBA}[\mathsf{ID}]$

13: **upon** receiving $\overline{W}$ from $\mathsf{MVBA}[\mathsf{ID}]$ **do**
14: let I be the index set corresponds to $\overline{W}$, such that $|I| = n - t$
15: **if** not yet receive any of $\{r^j_{\mathsf{local},k,i}\}_{k\in[B/(n-2t)],j\in I}$ **then**
16: recover the missing shares by $\mathsf{wpACSS.BatchedRecover}$
17: **for** $k \in [B/(n-2t)]$ **do**
18: $[r_{k,1,i}, r_{k,2,i}, \cdots, r_{k,n-2t,i}] \leftarrow M \cdot [\{r^j_{\mathsf{local},k,i}\}_{j\in I}]$
19: reorder $[r_{k,1,i}, r_{k,2,i}, \cdots, r_{k,n-2t,i}]_{k\in[B/(n-2t)]}$ as $\{r_{k,i}\}_{k\in[B]}$
20: **for** $k \in [B]$ **do**
21: $s_{\mathsf{mask},k,i} \leftarrow s_{k,i} + r_{k,i}$ ▷ used in batch reconstruction
22: batch-reconstruct $\{s_k + r_k\}_{k\in[B]}$ ▷ cf. ReconsPubl in [37]
23: multicast $\langle \textsc{Mask}, \{s_k + r_k\}_{k\in[B]}\rangle$ to $\mathcal{U}'$
24: $\{m_1, \cdots, m_n\} \leftarrow \mathsf{RSEnc}(t, n, I)$ ▷ encode I using RS code
25: multicast $\langle \textsc{Ecc}, \mathsf{ID}, m_i\rangle$ to $\mathcal{U}'$
26: delete all private information and halt ▷ halt

As a new node $\mathcal{P}'_i \in \mathcal{U}'$:
Init: start $\mathsf{wpACSS.Share}[\langle \mathsf{ID}, d, k, 2\rangle]$ for all $d \in [n]$ and $k \in [B/(n-2t)]$ to receive shares

27: **for** $0 \le r \le t$ **do**
28: **upon** receiving $2t + 1 + r$ Ecc messages from distinct $\mathcal{P}_i$ **do**
29: run Online-Error-Correcting (OEC) algorithm until obtain I
30: **upon** receiving I **do**
31: follow line 15–19 to compute $\{r'_{k,i}\}_{k\in[B]}$
32: **wait** for $t + 1$ Mask messages with the same $\{s_k + r_k\}_{k\in[B]}$
33: **for** $k \in [B]$ **do**
34: $s'_{k,i} \leftarrow s_k + r_k - r'_{k,i}$
35: **output** $\{s'_{k,i}\}_{k\in[B]}$

the old committee aggregates the indexes of the $n-t$ selected nodes into I, and disseminates it to the new committee. Note that directly multicasting I would incur $\mathcal{O}(\lambda n^3)$ communication complexity, so we use error-correcting codes to keep the optimistic communication complexity within $\mathcal{O}(\lambda n^2)$.

- For each $k \in [B/(n-2t)]$, the old node $\mathcal{P}_i$ multiplies the H matrix with the $n-t$ selected shares. The multiplication for a specific k yields $n-2t$ random shares. Hence, each $\mathcal{P}_i$ obtains B random shares $\{r_{k,i}\}_{k\in[B]}$, where $r_{k,i}$ is the share of a uniformly random value r_k. Each new node $\mathcal{P}'_i$ receives the index set I from the old committee and then extracts B random shares $\{r'_{k,i}\}_{k\in[B]}$ in the same way.
- Each old node $\mathcal{P}_i$ masks their old shares by computing $\{s_{k,i}+r_{k,i}\}_{k\in[B]}$. Then, $\mathcal{P}_i$ reconstructs $\{s_k + r_k\}_{k\in[B]}$ using the batched reconstruction algorithm in [37]. The reconstructed values are sent to the new committee.
- Each new node $\mathcal{P}'_i$ waits for $t+1$ messages with the same reconstructed values and computes the new share $s'_{k,i} = s_k + r_k - r'_{k,i}$ for every $k \in [B]$. The new nodes are no longer required to generate the commitment vector $\boldsymbol{v}_{\mathsf{new}}$ because the inputs for the provable sharing instances are random values rather than the old shares.