## Practical Asynchronous Distributed Key Reconfiguration and Its Applications

Hanwen Feng<sup>1</sup>, Yingzi Gao<sup>2</sup>, Yuan Lu<sup>2</sup>, Qiang Tang<sup>1</sup> and Jing Xu<sup>2</sup>

<sup>1</sup>School of Computer Science, The University of Sydney <sup>2</sup>Institute of Software, Chinese Academy of Sciences

### Abstract

In this paper, we study practical constructions of asynchronous distributed key reconfiguration (ADKR), which enables an asynchronous fault-tolerant system with an existing threshold cryptosystem to efficiently generate a new threshold cryptosystem for a reconfigured set of participants. While existing asynchronous distributed threshold key generation (ADKG) protocols theoretically solve ADKR, they fail to deliver satisfactory scalability due to cubic communication overhead, even with simplifications to the reconfiguration setting.

We introduce a more efficient share-dispersal-then-agreeand-recast paradigm for constructing ADKR with preserving adaptive security. The method replaces expensive O(n) asynchronous verifiable secret sharing protocols in classic ADKG with O(n) cheaper dispersals of publicly-verifiable sharing transcripts; after consensus confirms a set of finished dispersals, it selects a small k-subset of finished dispersals for verification, reducing the total overhead to  $O(\kappa n^2)$  from  $O(n^3)$ , where  $\kappa$  is a small constant (typically ~30 or less). To further optimize concrete efficiency, we propose an interactive protocol with linear communication to generate publicly verifiable secret sharing (PVSS) transcripts, avoiding computationally expensive non-interactive PVSS. Additionally, we introduce a distributed PVSS verification mechanism, minimizing redundant computations across different parties and reducing the dominating PVSS verification cost by about one-third.

Our design also enables diverse applications: (i) given a quadratic-communication asynchronous coin-flipping protocol, it implies the first quadratic-communication ADKG; and (ii) it can be extended to realize the first quadraticcommunication asynchronous dynamic proactive secret sharing (ADPSS) protocol with adaptive security. Experimental evaluations on a global network of 256 AWS servers show up to 40% lower latency compared to state-of-the-art ADKG protocols (with simplifications to the reconfiguration setting), highlighting the practicality of our ADKR in large-scale asynchronous systems.

### 1 Introduction

Modern fault-tolerant systems heavily rely on threshold cryptosystems for achieving better performance (e.g. lower round and communication complexities [4, 22, 40, 42, 48, 52]) and enhanced security (e.g. private mempool to mitigate "miner" extractable value [15, 21, 49]). Such reliance is particularly critical in fully asynchronous fault-tolerant systems to achieve superior robustness against unpredictable communication delays, where unique threshold signatures [12] or other threshold primitives [16] are widely adopted for generating common coins to efficiently overcome the seminal Fischer-Lynch-Patterson (FLP) "impossibility" [31] to achieve agreement while retaining liveness.<sup>1</sup>

Such heavy reliance on a pre-configured threshold cryptosystem in practical asynchronous fault-tolerant systems, unfortunately, fixes the set of participating parties, thereby creating a fundamental obstacle to supporting dynamic participation and hindering their broader adoption. A naive "solution" could be running asynchronous distributed key generation (ADKG) whenever the participants are changing. However, the approach suffers from significant performance issues in a scalable network, despite recent advancements in the field [2, 3, 24, 26, 39, 46]. For instance, Das et al. reported latency of more than 40 seconds across about only 100 parties, even if using the state-of-the-art ADKG protocols [24, 26].

The above issue underscores a major gap between the current studies on practical asynchronous fault-tolerant systems—whose efficiency relies on pre-configured threshold cryptosystems across fixed participants—and the dynamic nature of real-world open blockchains.<sup>2</sup> Motivated by the key challenge, we focus on the following problem of asyn-

<sup>&</sup>lt;sup>1</sup>Without common coins, one also can construct a fully asynchronous consensus using the so-called local coins [11, 13], which, however, results in an impractical round complexity exponential in the network scale.

<sup>&</sup>lt;sup>2</sup>None of the top-30 blockchains by market capitalization (including Ripple, Chainlink, Sui, Avalanche, Stellar, Aptos, Polkadot, etc., according to CoinMarketCap) adopts fully asynchronous consensus at the time of writing (January 19, 2025). Some have even simplified DAG-based asynchronous consensus into partial synchrony by removing threshold common coins.

chronous distributed key reconfiguration (ADKR), aiming to develop more efficient mechanisms for reconfiguring threshold cryptosystems to accommodate changes in participants, even in the presence of arbitrary and unpredictable network delays caused by network asynchrony.

**Definition 1** (ADKR, **informal**). Informally, ADKR is a simplified version of ADKG, which, in the presence of an alreadyestablished threshold cryptosystem among a set  $\mathbb{M}$  of participants, distributedly generates a threshold cryptosystem among another set  $\widetilde{\mathbb{M}}$  of participants, given a fully-meshed asynchronous network consisting of all parties in  $\mathbb{M} \cup \widetilde{\mathbb{M}}$ .

Remarkably, ADKR captures the critical problem of resetting a threshold cryptosystem when two sets of participants need to hand over in an asynchronous fault-tolerant system. The key distinction between ADKR and ADKG lies in the former's potential to utilize the established threshold cryptosystem of the old participants to improve efficiency.

#### **1.1** Limits of Existing Approaches

Unsurprisingly, ADKR, as a natural formulation of threshold cryptosystem reconfiguration in the asynchronous network, was previously studied [41] and can be derived from the implications of ADKG protocols [2, 3, 24, 26, 39, 46]. Here, we briefly review these approaches and highlight their limitations.

Adapting ADKG into the coin-hybrid model still suffers from a cubic total overhead. As Figure 1 clarifies, ADKR can be immediately realized from the state-of-the-art ADKG [24, 26], with a couple of simple optimizations in the coinaided model. First, since an efficient one-round protocol for asynchronous common coins can be implemented from the old participants' threshold cryptosystem [16], we can significantly simplify the consensus part of ADKG by directly using the best so-far asynchronous consensus from these granted common coins, avoiding the usually complex protocols of generating asynchronous common coins without any threshold cryptosystem setup [10,33]. Second, the step of letting all parties perform verifiable secret sharings in ADKG is usually related to the generation of common coins in the absence of threshold cryptosystem setup [26], and once common coins are granted, the part can also be simplified, by letting 2f + 1parties (instead of all parties) to distribute their secrets.

However, such straightforward modifications of ADKG, though effective, do not scale well, as they fail to asymptotically reduce the significant cubic total overhead due to O(n) asynchronous verifiable secret sharing (AVSS)<sup>3</sup> protocols.<sup>4</sup>

In particular, Figure 5 experimentally demonstrates that while reconfiguring a high-threshold cryptosystem, the computational cost for each party still grows quadratically, exceeding 1000 seconds in a network with two thousand parties, even if assuming that the consensus component is free.



Figure 1: High-level structure of existing ADKG protocols and simple modifications of them to accommodate ADKR.

Previous sub-cubic attempt is neither concretely efficient nor adaptively secure. In addition to the previous two modifications to ADKG, Günther, Das and Kokoris-Kogias [41] also studied the exact ADKR problem and presented a construction with sub-cubic communication. They used the granted common coin to select a c-size honest-majority sub-committee from the whole network with honest super-majority and only let the sub-committee members distribute secret sharings, resulting in  $O(c\lambda n^2)$  communication cost. However, despite being asymptotically better, their design's concrete efficiency does not significantly outperform the straightforward adaptation of ADKG since their sub-committee size c is not concretely small. In particular, sampling an honest-majority subcommittee with a failure probability below  $10^{-10}$  requires a sub-committee size of about four hundred. Moreover, the design is insecure against an adaptive adversary capable of corrupting participants after observing the common coin used to select sub-committee members because the adversary can always corrupt the majority within the sub-committee, thus fully controlling the generated secret key.

Threshold cryptosystems with silent setup face performance and compatibility issues. Silent-setup threshold signature schemes [23, 34, 51] and threshold encryption schemes [35] have been proposed as alternatives to conventional threshold cryptography to avoid interactive ADKG/ADKR protocols. However, these schemes suffer from poorer performance, raising concerns for practical systems. For example, the stateof-the-art silent threshold signature scheme [23] generates signatures  $11 \times$  larger than standard BLS signatures and requires  $8 \times$  more time for verification. Similar performance limitations affect silent threshold encryption schemes. Furthermore, these threshold signature schemes are incompatible with standard signature verification algorithms, making them unsuitable as drop-in replacements in existing systems.

Moreover, as an essential application of threshold cryptography in consensus, a non-interactive unique threshold signature scheme (under ADKR/ADKG setup), such as threshold

<sup>&</sup>lt;sup>3</sup>More precisely, a strong AVSS variant asynchronous complete secret sharing (ACSS) is required in ADKG to ensure all parties receive the correct secret shares. Hereafter, we consistently refer to ACSS throughout the paper.

<sup>&</sup>lt;sup>4</sup>Note that a concurrent theoretic work [1] uses a recursive protocol structure to realize ADKG with nearly quadratic communication. But for typical recursive depth of  $\log n$ , it causes  $O(\log n)$  rounds and is not compatible with standard dLog cryptosystems since it generates group-element secret shares.

BLS [7], enables a *single-round* and highly efficient asynchronous common coin protocol [16]. Unfortunately, silent threshold signature schemes lack the uniqueness property, making it difficult to derive a common coin protocol directly<sup>5</sup>.

Given the limitation of existing techniques, we ask:

Can we design asymptotically and concretely more efficient ADKR protocols for the standard discrete-logarithm-based threshold cryptosystems (e.g., the standard BLS threshold signature), with adaptive security?

Table 1: Comparison with existing ADKR and simplified ADKG protocols. Here *n* is the total number of parties (e.g.  $|\mathbb{M}| + |\widetilde{\mathbb{M}}|$ ),  $\lambda$  is the bit length of cryptographic security parameter, and *c* and  $\kappa$  are statistic security parameters.

	Possibly adaptive?	Standard dLog?	Comm.	Round
Classic ADKGs with adaptions	$\checkmark$	$\checkmark$	$O(\lambda n^3)$	<i>O</i> (1)
GDK ADKR [41]	×	$\checkmark$	$O(c\lambda n^2)$ <sup>†</sup>	O(1)
<b>Ours</b> (§4)	√‡	$\checkmark$	$O(\kappa\lambda n^2)^{\dagger}$	<i>O</i> (1)

<sup>†</sup> Though *c* and  $\kappa$  are both statistic parameters,  $\kappa$  is significantly smaller than *c*. For a failure probability of  $10^{-10}$ , *c* is more than 400, while  $\kappa$  is typically around 30.

<sup>‡</sup> We provide an adaptive security proof for an instantiation of our design in Appendix A.

Table 2: Comparison with the asynchronous proactive secret sharing protocols having O(1) round and optimal resilience.

	Possibly adaptive?	Dynamic?	High-thld?	Comm.
CKLS [14]	$\checkmark$	×	×	$O(\lambda n^4)$
GDK [41]	×	×	$\checkmark$	$O(c\lambda n^2)$
YXXM [53]	$\checkmark$	$\checkmark$	$\checkmark$	$O(\lambda n^3)$
<b>Ours</b> (§5)	$\checkmark$	$\checkmark$	$\checkmark$	$O(\kappa \lambda n^2)$

### **1.2 Our Contribution**

We answer the above question affirmatively by proposing a novel approach to constructing a more scalable ADKR protocol for conventional discrete-logarithm (Dlog)–based threshold cryptosystems, with dedicated efforts preserving adaptive security and optimizing concrete efficiency. Specifically, our contribution can be summarized as follows.

• ADKR with asymptotically lower complexity and adaptive security. To address the cubic total overhead

caused by the O(n) ACSS protocols, we propose an efficient *share-dispersal-then-agree-and-recast* paradigm for constructing ADKR while maintaining the potential for adaptively secure implementations. The design replaces O(n) ACSS protocols with O(n) asymptotically cheaper dispersals of publicly-verifiable secret sharing (PVSS) transcripts. Once consensus confirms the completion of these dispersals,  $\kappa$  dispersals are randomly selected for reconstruction, reducing verification to these few instances. Moreover, a small  $\kappa$  (several dozen) suffices since adaptive security only requires an honest dispersal among the  $\kappa$  selected (i.e. we only require an any-trust  $\kappa$ -size sub-committee).

- Various optimizations for concrete efficiency. Inspired by the recent DXT+ ACSS [25], we introduce an interactive dealing protocol to enable each dealer to generate a publicly-verifiable sharing transcript with *linear* communication cost, to mitigate the high computational cost of non-interactive publicly-verifiable secret sharing schemes [32, 43] when instantiating the *share-dispersalthen-agree-and-recast* paradigm. We further reduce the dominating cost of verifying  $\kappa$  sharing transcripts to about 2/3, at the price of a single round all-to-all communication: for each *n*-item transcript, each item in it is pre-scheduled to 2f + 1 parties for verifying, so every party at most verifies 2f + 1 items (instead of *n*) and can "verify" the remaining *f* items by exchanging the verification results among other parties.
- Diverse applications arising from ADKR. We demonstrate various implications of our efficient ADKR design. First, our ADKR protocol can be directly adapted into a quadratic communication ADKG in the presence of common coins and PKI. When combined with the recent quadratic-communication asynchronous coin-flipping protocol from PKI [30], this completes the first quadratic-communication ADKG in the PKI setting. Additionally, we introduce the first asynchronous dynamic, proactive secret sharing (ADPSS) protocol with quadratic communication cost and adaptive security by extending our ADKR design to efficiently distribute common randomness twice across two distinct committees. These applications may be of independent interest.
- Implementation and scalable experiments. We implement our ADKR protocol and experimentally compare it with state-of-the-art ADKG protocols (including DYX+22 [26] and DYK+23 [24], with necessary simplifications for fair comparisons) in large-scale wide-area networks of up to 256 AWS servers evenly distributed across 16 cities on five continents. For n = 256, our single-threaded high-threshold ADKR implementation reduces the latency of the simplified DYK+23 protocol by 30–40%, reducing it from 103.96 seconds to approximately one minute.

<sup>&</sup>lt;sup>5</sup>Very recently, [30] proposed a silent-setup asymptotically optimal asynchronous common coin protocol using both silent threshold signature and encryption. However, its high concrete cost, such as requiring over 100 rounds in expectation, limits its suitability for frequent use in practical systems.

#### **1.3 Other Related Works**

Most DKG protocols [2, 33, 36, 46, 50] have a communication complexity of  $\Omega(\lambda n^3)$ . For a review of these classic constructions, we refer readers to [6, 27].

Recent major advances [6, 28] introduced DKG protocols with subcubic communication in the *synchronous* setting. These schemes partition the network into smaller subgroups and apply a recursive strategy, relying on at least one subgroup having an honest majority. However, this method does not naturally extend to the *asynchronous* setting, as messages from an honest-majority subgroup may be arbitrarily delayed, allowing the adversary to control the final secret key without their contributions.

While [6,28] focus on DKG without common coins, [29] and [41] explored coin-aided DKG protocols for improved performance in the synchronous and asynchronous settings, respectively. These protocols use coins to select a small committee for secret sharing, avoiding O(n) instances of verifiable secret sharing. However, this approach sacrifices strong adaptive security (i.e., tolerance of after-fact-removal attacks). [41] requires an honest-majority committee of several hundred nodes, limiting practical performance.

Recent concurrent work [1] extends the recursive method of [6, 28] to the asynchronous setting using verifiable random functions (VRFs) [9], requiring a one-time common coin after PKI setup. Compared to our coin-aided DKG, their protocol assumes a weaker setup (one-time initial coin vs. a coin oracle), but its recursive structure and internal committee sampling procedures make it unsuitable for moderatescale networks of a few hundred nodes. Moreover, theirs produces group-element secrets, while ours gives field-element secrets. [30] presented a quadratic-communication adaptive common coin protocol based on silent threshold cryptography [35], which, when combined with our coin-aided ADKG, yields quadratic-communication ADKG with a silent setup.

Finally, we note that the paradigm of coin-aided DKG resembles the idea of coin-tossing extension [5] which was studied for circumventing the impossibility of coin tossing in the *dishonest-majority* setting. Coin-aided DKG and ADKR protocols can be seen as the coin tossing extension for the honest-majority setting, for better amortized performance.

### 2 Problem Formulation and Building Blocks

We assume a fully asynchronous network consisting of two sets of participating parties,  $\mathbb{M}$  and  $\widetilde{\mathbb{M}}$ , with  $|\mathbb{M}| = n$  and  $|\widetilde{\mathbb{M}}| = \tilde{n}$ . Here,  $\mathbb{M}$  represents an "old" committee with an established threshold cryptosystem, and  $\widetilde{\mathbb{M}}$  represents a "new" committee aiming to distributedly (and efficiently) generate its own threshold cryptosystem, with the help of the old committee. This problem setting is crucial for supporting dynamic participation in many modern asynchronous fault-tolerant systems that heavily rely on threshold cryptosystems. **Modeling**. More precisely, we consider the following standard model in a computationally bounded setting, assuming a fully mesh peer-to-peer asynchronous network.

- *Public key infrastructure (PKI)*. Every party in M and M knows all other parties' public keys for verifying digital signatures and performing public key encryption through a bulletin board PKI.
- Established dLog threshold cryptosystem across the old committee. The parties in  $\mathbb{M}$  established a dLog based (n,t)-threshold cryptosystem  $\mathcal{TC}$  over a cyclic group  $\mathbb{G}$  with prime order q: (i) all parties in  $\mathbb{M}$  have common public keys  $(g^s, g^{s_1}, g^{s_2}, \ldots, g^{s_n})$ ; (ii) each  $\mathcal{P}_i \in \mathbb{M}$  holds an exclusive private key share  $s_i \in \mathbb{Z}_q$  consist with  $g^{s_i}$ ; and (iii) any t + 1 private key shares can interpolate the same *t*-degree polynomial  $\phi(x)$  with  $g^s = g^{\phi(0)}$ .  $\mathcal{TC}$  is called *high-threshold*, if the reconstruction threshold  $t = n f 1 \ge 2f$ . Similar to many quorum-based fault-tolerate systems, we might leverage a *high-threshold* cryptosystem setup for concrete efficiency.
- Asynchronous network. Each pair of honest parties in M and M can establish a secure asynchronous communication channel, where the adversary can arbitrarily delay messages but cannot tamper with them or learn any information about their content beyond their length.
- *Malicious corruptions*. Besides the power of arbitrarily delaying communication, the adversary can also corrupt up to  $f = \lfloor \frac{n-1}{3} \rfloor$  parties in  $\mathbb{M}$  and up to  $\tilde{f} = \lfloor \frac{\tilde{n}-1}{3} \rfloor$  parties in  $\widetilde{\mathbb{M}}$ . Following the standard cryptographic practice [15, 19], we consider the adversary to be probabilistic polynomial-time (PPT) bounded and can coordinate all corrupted parties to arbitrarily misbehave.
- Static vs. adaptive corruption. An adversary is static if it corrupts malicious parties before the protocol starts after the establishment of setup assumptions. In contrast, an adversary is adaptive, if it can adaptively corrupt parties during the protocol execution. We consider a strong adaptive adversary capable of performing the "after-the-fact removal" attack, i.e., it can corrupt  $\mathcal{P}_i$  and drop messages sent from  $\mathcal{P}_i$  before these messages reach their destination. This paper proposes a protocol structure for ADKR that could be instantiated with adaptive security.

**Design goals**. As briefly aforementioned, we aim at accommodating the reconfiguration of threshold cryptosystem when the current participants is rotating (e.g. some current participants are leaving and some new parties are joining), which means to solve the following problem of asynchronous distributed key reconfiguration (ADKR).

<u>SYNTAX OF ADKR</u>: An  $(n,t,\tilde{n},\tilde{t})$ -ADKR protocol for dLog cryptosystems are executed by two sets  $\mathbb{M}$  and  $\widetilde{\mathbb{M}}$  of parties, where  $|\mathbb{M}| = n$  and  $|\widetilde{\mathbb{M}}| = \tilde{n}$ . A dLog-based (n,t)-threshold cryptosystem  $\mathcal{TC}$  was established across  $\mathbb{M}$ . All parties in

 $\mathbb{M}$  and  $\widetilde{\mathbb{M}}$  inputs the public keys of  $\mathcal{TC}$ , and each party in  $\mathbb{M}$  additionally inputs its exclusive private key share belonging to  $\mathcal{TC}$ . After the protocol terminates, the honest parties in  $\widetilde{\mathbb{M}}$  set up a dLog-based  $(\tilde{n}, \tilde{t})$ -threshold cryptosystem  $\widetilde{\mathcal{TC}}$ , namely, each honest party in  $\widetilde{\mathbb{M}}$  outputs its own exclusive private key share and all public keys belonging to  $\widetilde{\mathcal{TC}}$ .

<u>SECURITY OF ADKR.</u> As ADKR is to set up threshold cryptosystem in  $\widetilde{\mathbb{M}}$  (with the help of  $\mathbb{M}$ ), it shares the same security goal as an ADKG protocol executed by  $\widetilde{\mathbb{M}}$ . Therefore, we consider the following three properties that are well-established in the literature of ADKG [24, 26, 46].

- *Termination*. Every honest party P<sub>i</sub> in M will output a vector of public keys (g<sup>z</sup>, g<sup>z1</sup>, g<sup>z2</sup>, ..., g<sup>zi</sup>) and an exclusive private key share z<sub>i</sub> ∈ Z<sub>q</sub>.
- *Key Validity.* The outputs of all honest parties in M are consistent: (i) all parties' outputs contain the same public keys (g<sup>z</sup>, g<sup>z1</sup>, g<sup>z2</sup>, ..., g<sup>zn</sup>); (ii) the private key share z<sub>i</sub> obtained by each party is consistent with the corresponding public key share of g<sup>zi</sup>; (iii) there exists a t̃-degree polynomial φ ∈ Z<sub>p</sub>[X], such that z = φ(0), and z<sub>i</sub> = φ(i) for all i ∈ [ñ].
- *Full Secrecy:* No computationally bounded adversary can (i) prevent the secret key z from being uniformly sampled over Z<sub>q</sub> or (ii) learn information about z beyond the public key g<sup>z</sup>. Formally, for any PPT adversary A which can corrupt up to f nodes, there exists a PPT simulator S which on input a uniformly sampled g<sup>z</sup> can produce a simulated view SimView, such that (SimView, g<sup>z</sup>) is computationally distinguishable with (View<sub>A</sub>, pk), where View<sub>A</sub> is the view of A is a real execution with pk as the public key.

Similar to many existing adaptive (ADKG) protocols such as Bingo [2], our adaptive construction aims at achieving a relaxed version of secrecy known as *oracle-aided algebraic simulatability* [7], which we recall in Def.2 in Appendix. A.2. If t = n - f - 1 and  $\tilde{t} = \tilde{n} - \tilde{f} - 1$  (where  $f = \lfloor \frac{n-1}{3} \rfloor$  and  $\tilde{f} = \lfloor \frac{\tilde{n}-1}{3} \rfloor$  represent the number of corruptions in  $\mathbb{M}$  and  $\widetilde{\mathbb{M}}$ , respectively), we say such an ADKR is *high-threshold*.

**Building blocks**. Our designs will use a few well-studied building blocks, including: (i) a threshold signature scheme (TSIG); (ii) a threshold common coin protocol (Coin) [16]; (iii) a communication-efficient multi-valued validated asynchronous Byzantine agreement protocol (MVBA) [48]; (iv) a verifiable encryption scheme for Pedersen commitment [25]; and (v) an asynchronous provable dispersal protocol, consisting of a provable dispersal (PD) subprotocol and a recovery (RC) subprotocol. Their definitions and instantiations are recalled in Sect.7.

**Notations.** Throughout the paper, let  $\lambda$  denote the bit length of the cryptographic security parameter, and let  $\kappa$  be the statistical security parameter. The notation [i,n]

represents the set  $\{i, i + 1, ..., n\}$ , where *i* and *n* are integers with i < n. We may abbreviate [1, n] as [n]. For a set  $\{x_1, x_2, ..., x_n\}$  and a sequence  $(x_1, x_2, ..., x_n)$ , we write them as  $\{x_i\}_{i \in [n]}$  and  $(x_i)_{i \in [n]}$ , respectively, for brevity. An instance of a protocol ProtoName is denoted by ProtoName[ID] $\langle \{ParticipantID(input)\} \rangle$ . We may omit ID or participant information when no ambiguity arises.

#### **3** Challenges and Techniques

Challenge I: achieving quadratic total overhead with preserving adaptive security. As illustrated in Figure 1, in a classic ADKG protocol based on O(n) PVSS protocols, (almost) every party distributes a secret across the network through an PVSS protocol. The participants then agree on a set of completed PVSS protocols, ensuring that at least one honest party's PVSS is solicited. Nevertheless, the approach causes cubic total communication cost, as PVSS has a quadratic communication lower bound. Therefore, to asymptotically reduce the cubic overhead, ADKG/ADKR can use, at most, a constant number of PVSS protocols instead of O(n).

Reference [29] suggests selecting an any-trust committee (which has at least one honest member to deal secrets in DKG) to deal secrets, which shows promising performance improvements in the synchronous network due to the significantly reduced number of VSS instances. However, its selection of any-trust sub-committee raises two major problems: (1) it, at best, can achieve weak adaptive security (i.e., cannot tolerate the after-the-fact-removal of messages); (2) in an asynchronous network, the adversary can arbitrarily delay messages, so it can prevent the only honest party from contributing its secret. To address (2), reference [41] uses a larger sub-committee size (e.g., approximately several hundred) and still fails to achieve adaptive security.

The above problem can be translated into the following challenge: if we strive to realize quadratic communication, we seemingly have to select a constant number of parties to distribute their secrets, which in turn might hurt the desired adaptive security, as a rushing adversary can wait until the sub-committee members are selected to corrupt them.



Figure 2: High-level idea of our *share-dispersal-then-agree-and-recast* paradigm (exemplified by non-interactive PVSS).

<u>Technique I: share-dispersal-then-agree-and-recast paradigm</u>. We introduce a paradigm of *share-dispersal-then-agree-and-recast* to achieve quadratic total overhead while preserving an adaptively secure protocol structure, as Figure 2 outlines. For asymptotic efficiency, this approach lets each party distribute its secret shares, through a linear communication protocol that disperses a publicly verifiable secret sharing (PVSS) transcript instead of the quadratic-communication PVSS; after the consensus component agrees on a set of n - f completed dispersals, a common coin is used to select  $\kappa$  dispersed PVSS transcripts to recover and verify, which costs only quadratic communication as  $\kappa$  is asymptotically a small constant. By combining all valid PVSS transcripts, every party can derive its exclusive secret key shares, and all public keys at the price of another round of all-to-all communication.

The *share-dispersal-then-agree-and-recast* paradigm also has the potential for adaptive security. First, for any finished dispersal confirmed by consensus, a strong adaptive adversary can no longer prevent its correct recovery or replace the PVSS transcript that was already dispersed by it. Moreover, by having all parties perform dispersals, there is an overwhelming probability (in  $\kappa$ ) that at least one of the  $\kappa$  selected dispersals originates from a sender that was not corrupted at the point when the indices of select dispersals are published. Moreover, in the erasure model, we can let non-corrupted parties erase their secrets locally before initiating their dispersals of PVSS, preventing the adversary from adaptively corrupting the selected senders to compromise the entire secrets.

Challenge II: retaining concrete efficiency for practical large-scale instantiations. An immediate instantiation of the share-dispersal-then-agree-and-recast paradigm can use non-interactive PVSS schemes, which might either have prohibitive computing cost [26, 32, 38, 44] or restricted applications (other than the standard dLog-based threshold cryptosystems) [8, 20, 37, 43]. Recently, DXT+ ACSS [25] proposes an insightful approach using an interactive dealing phase to generate a publicly verifiable sharing transcript (we call this phase as DTX+ PVSS<sup>6</sup>), offering an efficient alternative to the expensive non-interactive PVSS: the dealer first distributes secret shares along with a polynomial commitment of the shares, then it solicits signatures on the commitment from sufficient parties to produce a publicly-verifiable transcript. However, integrating the publicly verifiable dealing phase into our share-dispersal-then-agree-and-recast paradigm results in cubic total communication because of the linear size of polynomial commitment. A seeming "solution" of using compact KZG polynomial commitment [45] limits the resulting protocol to solely support pairing-friendly elliptic curves and necessitates a trusted CRS setup.

The above issues translates into another key challenge: can we instantiate the *share-dispersal-then-agree-and-recast* paradigm, using concretely more efficient components and optimizations, without hurting its asymptotic complexity?

<u>Technique II-A: publicly-verifiable linear-overhead dealing</u>. We propose a simple optimization to DXT+ PVSS, reducing the communication cost of its dealing phase from quadratic  $O(\lambda n^2)$  to linear  $O(\lambda n)$ , without extra CRS setup or hurting its generality and public verifiability. Different from DXT+ PVSS that distributes the whole polynomial commitment to all participants for verification, our dealing phase only sends each participant its exclusive secret share and the share's Pedersen commitment, reducing dealer's each message size from  $O(\lambda n)$  to  $O(\lambda)$ . Moreover, the dealer can still compute a publicly verifiable secret sharing (PVSS) transcript, by combining the signed commitments to secret shares together with verifiable encryption (VE) of other shares without signed commitments, following a method similar to that used in DXT+ PVSS.

Moreover, in the low-threshold setting when the reconstruction threshold and the maximal number of corruptions coincide, we can remove the reliance of VE, and the resulting ADKR protocol can be proven to be adaptively secure.<sup>7</sup>

<u>Technique II-B: distributedly reducing PVSS's verification cost</u>. After applying our improved PVSS to reduce the dealer's computational cost, another primary bottleneck remains to verify the  $\kappa$  selected PVSS transcripts. We observe that different parties perform many redundant computing steps for verifying each PVSS transcript, and we therefore propose an optimized "distributed" verification approach to mitigate the issue.

The idea partitions the workload of verifying each PVSS transcript into the verifications of about *n* transcript items. Each party is assigned 2f + 1 items (roughly two-thirds of the *n* items) to verify, ensuring that every item is verified by at least 2f + 1 parties. Each party then multicasts its verification results for the 2f + 1 items assigned to it. For the remaining *f* items that a party did not verify itself, it collects f + 1 consistent verification results from other parties, thus avoiding the computing steps of locally verifying these *f* items. This process ensures the correct verification of PVSS transcripts while efficiently distributing the workload across the network.

Non-trivial application to dynamic proactive secret sharing. We further extend our design methodology to improve dynamic proactive secret sharing in the asynchronous setting. Similar to ADKR, asynchronous dynamic proactive secret sharing (ADPSS) is also a handover protocol between two committees, while different from ADKR that lets a new committee set up a fresh threshold cryptosystem, ADPSS requires the private key *m* to remain the same. To meet the additional requirement of preserving the same private key, we optimize the high-level approach from [53], which distributedly samples a random value *r* and shares it twice—once among the old committee and once among the new committee. As such, the old committee can reconstruct and publish m + r, enabling the

 $<sup>^{6}</sup>$ DXT + ACSS is realized by DTX+ PVSS and a subsequent broadcast phase for broadcasting the transcript.

<sup>&</sup>lt;sup>7</sup>We conjecture that our ADKR protocol can be adaptively secure if using an adaptively secure high-threshold PVSS scheme (particularly considering our adaptive security proof in the low-threshold setting). However, we only prove static security of our efficient instantiation of high-threshold ADKR, since the VE used is not adaptively secure in the multi-user setting.

new committee to derive fresh shares of *m* by subtracting its local share of *r* from the reconstructed m + r. However, [53] implements the process with cubic communication, again, due to the invocation of O(n) PVSS protocols.

We formalize this critical process as a primitive of dualcommittee randomness generation and extend our ADKR design to implement it more efficiently, with only quadratic communication. Our improvement is achieved by modifying our ADKR protocol to include two publicly verifiable PVSS transcripts in each dispersal—one for the new committee and the other for the old committee—and introduce an equality verification to check whether the secrets of each dealer's two transcripts match. From our dual-committee randomness generation, the first quadratic-communication ADPSS with adaptive security is therefore realized.

## 4 Efficient Asynchronous Distributed Key Reconfiguration Protocol

Before diving into the details of ADKR, we first introduce an interactive PVSS scheme (an improved version of DXT+ PVSS [25]) in §4.1, which has a linear-communication asynchronous dealing phase to generate a publicly verifiable sharing transcript. From this optimized asynchronous dealing protocol, we present our ADKR protocol in §4.2.

### 4.1 PVSS with Interactive Dealing

In a *non-interactive* publicly verifiable secret sharing (PVSS) scheme [37, 38], a dealer locally runs a Deal algorithm to produce a publicly verifiable transcript. Upon observing the same valid (but potentially maliciously generated) transcript, all honest receivers can obtain correct shares of the same secret. However, the computation cost of verifying such transcripts often becomes a performance bottleneck.

To improve concrete performance, we consider an interactive variant of PVSS in this work, where transcript generation is performed via an asynchronous Deal *protocol* instead of a local algorithm.

**Syntax.** Formally, under a PKI setup where every receiver  $\mathcal{P}_i$  has generated its key pair  $(pk_i, sk_i)$  and published  $pk_i$ , an interactive PVSS scheme can be described by the next subprotocols and algorithms.

- PVSS.Deal⟨𝒫<sub>d</sub>({pk<sub>i</sub>}<sub>i∈[n]</sub>, s), {𝒫<sub>i</sub>(sk<sub>i</sub>)}⟩ → ⟨𝒫<sub>d</sub>(transcript), (au<sub>x</sub><sub>i</sub>)<sub>i∈[n]</sub>⟩. The Deal stage protocol is executed among the dealer 𝒫<sub>d</sub> and all receivers {𝒫<sub>i</sub>}. 𝒫<sub>d</sub> takes the public keys {pk<sub>i</sub>}<sub>i∈[n]</sub> of all participants and the secret s as inputs, and it outputs transcript. Each receiver 𝒫<sub>i</sub> inputs its private key sk<sub>i</sub> and receives an auxiliary information aux<sub>i</sub>.
- PVSS.Verify({pk<sub>i</sub>}<sub>i∈[n]</sub>, transcript) → 0/1. This algorithm verifies the sharing transcript transcript using the public keys of the receivers.

PVSS.Deliver(transcript, sk<sub>i</sub>, aux<sub>i</sub>) → s<sub>i</sub>. This algorithm allows a receiver P<sub>i</sub> to deliver a secret share s<sub>i</sub> according to a valid transcript, its secret key sk<sub>i</sub>, and the auxiliary information aux<sub>i</sub> from the Deal stage protocol.

We remark that the functionality of ACSS, which ensures all honest nodes receive correct shares, can be achieved by guaranteeing that all participants observe the same PVSS transcript. This can be realized using external mechanisms, such as reliable broadcast or consensus. Additionally, the public and secret keys in the above syntax can be viewed as collections of keys corresponding to various cryptographic primitives used in the protocol.

An improved instantiation of [25]. As briefly mentioned in §3, we give an improved version of the DXT+ PVSS protocol from [25], which inherits all the benefits of DXT+ PVSS (e.g., transparent setup, supporting high-threshold, and computing efficiency), and simultaneously reduces the communication cost of PVSS.Deal protocol from  $O(\lambda n^2)$  bits to  $O(\lambda n)$  bits.

INGREDIENTS. The protocol relies on the next components.

- A digital signature scheme with EUF-CMA security.
- A verifiable encryption (VE) scheme which satisfies IND-CPA security. We recall the full syntax of VE in Sect.7. In PVSS, we directly use the following three algorithms of VE, where  $(ek_j, dk_i)$  represents the encryption-decryption key pair of  $\mathcal{P}_j$ .
  - VE.bEncProve $(I, \{e_k_j\}_{j \in I}, \{a_j\}_{j \in I}, \{v_i\}_{i \in I}, \{b_j\}_{j \in I}) \rightarrow (\mathbf{c}, \pi_{VE})$ . For all  $j \in I$ , it encrypts each  $a_j$  under  $e_k_j$  to get  $c_j$ , so it returns  $\mathbf{c} = (c_j)_{j \in I}$ . It also returns a proof attesting that  $v_j = g^{a_j} h^{b_j}$  and  $c_j$  encrypts the same  $a_j$ .
  - VE.bVerify(I, {ek<sub>j</sub>}<sub>j∈I</sub>, {v<sub>j</sub>}<sub>j∈I</sub>, c, π<sub>VE</sub>) → 1 or 0. It verifies the above proof.
  - VE.Dec $(dk_i, c) \rightarrow s$ . It decrypts the ciphertext to *s*.
- The degree check algorithm DegCheck from [20]. It on input *n* group elements  $\{v_i\}_{i \in [n]}$  and two generators *g* and *h* checks if there exists two *t*-degree polynomials  $\phi$  and  $\hat{\phi}$ , such that  $v_i = g^{\phi(i)} h^{\hat{\phi}}(i)$  for all  $i \in [n]$ .

<u>THE PVSS.</u> With the above ingredients, we describe the PVSS scheme in Algorithm 1 (which includes boxed items but excludes dash-boxed items when used in our ADKR). We assume that each  $\mathcal{P}_i$  has generated the key pairs  $(ek_i, dk_i)$  and  $(vk_i, sk_i)$  w.r.t. the VE scheme and the digital signature schemes, respectively, and the public keys are known to every participant. Compared with the original scheme in [25], the dealer  $\mathcal{P}_d$  sends  $(v_j, (\phi(j), \hat{\phi}(j)))$ , instead of  $(\mathbf{v}, (\phi(j), \hat{\phi}(j)))$ , to each receiver  $\mathcal{P}_j$ , which reduces the communication cost of the dealing stage from  $O(\lambda n^2)$  to  $O(\lambda n)$ . In the next subsection, we will show our simplified scheme suffices for building a quadratic-communication ADKR, because it enables every

party in the old committee to use the linear-communication PVSS.Deal protocol to distribute a random secret across the new committee, which also produces a publicly-verifiable sharing transcript that is ready to be dispersed.

#### Algorithm 1 Our improved DXT+ PVSS scheme

 $\mathsf{Deal}\langle \mathscr{P}_d(\{vk_i, ek_i\}_{i \in [n]}, \overline{s_i}, \overline{s_i}\rangle, \{\mathscr{P}_i(sk_i)\}_{i \in [n]}\rangle \rightarrow \langle \mathscr{P}_d(\mathsf{transcript}), (\mathsf{aux}_i)_{i \in [n]}\rangle$ // Code run by Dealer  $\bar{\mathcal{P}}_d$ 1: randomly sample two *t*-degree polynomial  $\phi(\cdot)$  and  $\hat{\phi}(\cdot)$  where  $\phi(0)$  is *s* and  $\hat{\phi}(0)$  is a random secret  $\hat{s}$ . 2: compute  $\mathbf{v} \leftarrow \{v_j = g^{\phi(j)} h^{\hat{\phi}(j)}\}_{j \in [0,n]}$ 3: send SHARE $(v_i, \phi(j), \hat{\phi}(j))$  to every  $\mathcal{P}_i, j \in [n]$ . 4: **upon** receiving 2f + 1 valid signatures  $\sigma_i$  for  $v_i$  **do** Let  $\Sigma$  be the valid signatures set 5: 6: Let I be the indices of nodes with missing valid signatures  $c, \pi_{\mathsf{VE}} \leftarrow \mathsf{VE}.\mathsf{bEncProve}(I, \{ek_j\}_{j \in I}, \{\phi(j)\}_{j \in I}, \{v_i\}_{i \in I}, \{\hat{\phi}(i)\}_{i \in I})$ 7: 8: **return** transcript :=  $(v, c, \pi_{VE}, \Sigma, I)$ // Code run by each receiving party  $\mathcal{P}_i \in M$ 9: **upon** receiving SHARE $(v_i, s_i, \hat{s}_i)$  from  $\mathcal{P}_d$  **do** 10: if  $v_i = g^{s_i} h^{\hat{s}_i}$  then  $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}_i, v_i)$ 11: 12: send ACK( $\sigma_i$ ) 13: **return**  $aux_i = (s_i, \hat{s}_i)$ 14: **return**  $aux_i = \bot$  $Verify(\{vk_i, ek_i\}_{i \in [n]}, transcript) \rightarrow 0/1$ 15: parse *script* as:  $(\mathbf{v}, \mathbf{c}, \pi_{VE}, \Sigma, I)$ 16: **Check**  $\forall \sigma_i \in \Sigma$  is a valid signature for  $v_j$ 17: Check DegCheck(v,t) = 118: Check VE.bVerify $(I, \{e_k\}_{j \in I}, \{v_j\}_{j \in I}, \mathbf{c}, \pi_{VE}) = 1$ 

19: if all the checks pass then

20: return 1

21: **return** 0

Deliver(transcript,  $dk_i$ ,  $aux_i$ )  $\rightarrow s_i$ 22: if  $aux_i = (s_i, \hat{s}_i)$  then 23: return  $\boxed{s_i} \boxed{(s_i, \hat{s}_i)}$ 24: else 25: return  $s_i \leftarrow VE.Dec(dk_i, c_i) \boxed{(s_i, \hat{s}_i)} \leftarrow VE.Dec(dk_i, c_i)$ 

**Verifiable public key share computation.** In the setting of ADKR/ADKG, there are multiple instances of PVSS running in parallel, and the final secret key share is the aggregation of secret shares from a set *T* of a few PVSS instances, say  $z_i = \sum_{\ell \in T} s_i^{(\ell)}$ , where  $s_i^{(\ell)}$  is the secret share delivered in the  $\ell$ -th instance. Looking ahead, for achieving full secrecy, each node needs to compute their public key share tpk<sub>i</sub> =  $g^{z_i}$  along with a proof which demonstrating tpk<sub>i</sub> is honestly computed from a set of transcripts {transcript<sub>ℓ</sub>}<sub>ℓ∈T</sub>. Formally, we need a verifiable computation scheme {CompProve, VrfyComp} for the following computation task:

$$\mathsf{tpk}_i = g^{\sum_{\ell \in T} \mathsf{PVSS}.\mathsf{Deliver}(\mathsf{transcript}^{(\ell)}, dk_i, \mathsf{aux}_i^{(\ell)})}$$

For describing the verifiable computation scheme, we need a bit more detail about the VE scheme. In particular, for a Pedersen commitment  $v_i = g^{s_i} h^{\hat{s}_i}$ , the corresponding ciphertext  $c_i$  consists of two parts: (1)  $\bar{c}_i$ , encrypting  $s_i$ , whose detailed structure is irrelevant to the proof system; (2)  $\hat{c}_i$ , which is an ElGamal ciphertext encrypting  $h^{\hat{s}_i}$ , i.e.,  $\hat{c}_i = (g^r, ek_i^r \cdot h^{\hat{s}_i})$ .

With these facts, we develop the verifiable computation scheme in Algorithm 2 w.r.t. our improved PVSS. Note that the computation of tpk<sub>i</sub> also gives the secret key share  $z_i$  as an intermediate result. To avoid redundant calculation, we let the algorithm CompProve outputs  $z_i$  as well. In this scheme, we use the following two standard components: a NIZK pok for proving the knowledge of  $x \in \mathbb{Z}_p$  for  $g^x \in \mathbb{G}$  w.r.t. a generator g; and a NIZK DELq for proving the knowledge of x for  $(g^x, h^x)$  w.r.t. two generators g, h.

We establish the security properties of the scheme in the following Lemma 1.

**Lemma 1.** Conditioned on that the input transcripts  $(\text{transcript}^{(\ell)})_{\ell \in T}$  are valid, {CompProve, VrfyComp} in Algorithm 2 satisfies the following properties:

- Completeness: Honestly generated (tpk<sub>i</sub>, proof<sub>i</sub>) can always pass the verification.
- Zero-knowledge: An honestly generated  $(tpk_i, proof_i)$ does not leak anything beyond  $(tpk_i, h^{z'_i})$ . I.e., there is a PPT simulator which, on input  $(tpk_i, h^{z'_i})$  and the transcripts, generates simproof<sub>i</sub>, s.t.  $(tpk_i, proof_i)$  and  $(tpk_i, simproof_i)$  are computationally indistinguishable.
- Soundness: Any PPT adversary cannot provide a valid proof for  $tpk'_i \neq g\sum_{\ell \in T} PVSS.Deliver(transcript^{(\ell)}, dk_i, aux_i^{(\ell)})$ .

*Proof.* For *completeness*, it follows the completeness of the proof-of-knowledge for discrete logarithm, that of the proof-of-knowledge for DLEq, and the structure of Pedersen commitment and the ElGammal-style ciphertext.

For *zero-knowledge*, we can construct a simulator, which first computes the public element  $ek_i^r \leftarrow \frac{(\prod_{\ell \in S} \hat{c}_i^{(\ell)}[1])}{v_i/(h^{s_i^\ell} \cdot tpk)}$  and then trivially invokes the black-box simulators for zk-pok of discrete logarithm and DLEq, to simulate a valid proof  $(tpk_i, proof_i)$  without accessing the witness.

For *soundness*, if there is an adversary  $\mathcal{A}$  feasibly breaks it, that means with some non-negligible probability  $\delta$ ,  $\mathcal{A}$  can compute (tpk'\_i, proof'\_i) passing the verification of VrfyComp but tpk'\_i  $\neq g^{\sum_{\ell \in T} \text{PVSS.Deliver}(\text{transcript}^{(\ell)}, dk_i, \text{aux}_i^{(\ell)})}$  in polynomial time. We therefore can construct another adversary  $\mathcal{B}$ , interacting with  $\mathcal{A}$ , and uses black-box extractor of the proofof-knowledge for dLog and DH-tuple, to extract the witness satisfying the relationships. Knowing two set of satisfying witnesses,  $\mathcal{B}$  therefore can feasibly break the discrete logarithm problem.

#### 4.2 Our ADKR Protocol

**Setup and Ingredients.** Recall the definition of ADKR in §2. There are two sets of participating parties:  $\mathbb{M} = \{\mathcal{P}_i\}_{i \in [n]}$  and

#### Algorithm 2 Verifiable Public Key Share Computation

CompProve((transcript<sup>( $\ell$ )</sup>)<sub> $\ell \in T$ </sub>,  $dk_i$ , (aux<sup>( $\ell$ )</sup><sub> $\ell \in T$ </sub>))) Parse each transcript<sup>( $\ell$ )</sup> as:  $(\Sigma^{(\ell)}, I^{(\ell)}, \mathbf{v}^{(\ell)}, (\bar{c}_i^{(\ell)}, \hat{c}_i^{(\ell)}), \pi_{VF}^{(\ell)})$ 1: for  $\ell \in T$  do 2: if  $aux_i^{(\ell)} = \bot$  then 3: Decrypt  $s_i^{(\ell)}$  and  $h^{\hat{s}_i^{(\ell)}}$  from  $\bar{c}_i^{(\ell)}$  and  $\hat{c}_i^{(\ell)}$ 4: 5: else else Parse  $\mathsf{aux}_i^{(\ell)} = (s_i^{(\ell)}, \hat{s}_i^{(\ell)})$   $z_i \leftarrow \sum_{\ell \in T} s_i^{(\ell)}, \text{ and } \forall j \in [n], v_j \leftarrow \prod_{\ell \in T} v_j^{(\ell)}$ let  $S_i = \{\ell | \ell \in T \text{ and } i \in I^{(\ell)}\}, S_i' = T \setminus S_i$ 6: 7: 8:  $\hat{H}_i \leftarrow \prod_{\ell \in S_i} h^{\hat{s}_i^{(\ell)}}, \hat{z}_i' \leftarrow \sum_{\ell \in S_i'} \hat{s}_i^{(\ell)}$ 9: //  $\hat{c}_i^\ell$  is an ElGamal ciphertext with  $\hat{c}_i[0] = g^{r_i}$  and  $\hat{c}_i^{(\ell)}[1] = ek_i^{r_i} \cdot h^{\hat{s}_i}$  $\begin{array}{l} g_i^r \leftarrow \prod_{\ell \in S_i} \hat{c}_i^{(\ell)}[0], \mathsf{ek}_i^r \leftarrow (\prod_{\ell \in S} \hat{c}_i^{(\ell)}[1]) / \hat{H}_i \\ \pi_i^*[0] = \mathsf{pok}.\mathsf{prove}(z_i, g, g^{z_i}) \end{array}$ 10. 11:  $\pi_i^*[1] = \mathsf{pok.prove}(\hat{z}_i', h, h^{\hat{z}_i'})$ 12:  $\pi_i^*[2] = \mathsf{DLEq.prove}(g, \mathsf{ek}_i, g_i^r, \mathsf{ek}_i^r))$ 13: 14: **return** (tpk<sub>*i*</sub> =  $g^{z_i}$ , proof<sub>*i*</sub> = ( $h^{\hat{z}'_i}$ ,  $ek^r_i$ , ( $\pi^*_i[0]$ ,  $\pi^*_i[1]$ ,  $\pi^*_i[2]$ ),  $z_i$ ))  $VrfyComp((transcript^{(\ell)})_{\ell \in T}, tpk_i, proof_i)$ Compute  $g_i^r$  and  $v_j$  from  $(transcript^{(\ell)})_{\ell \in T}$  as in CompProve 15: Parse proof<sub>*j*</sub> =  $(h^{\hat{z}'_j}, ek'_i, (\pi^*_i[0], \pi^*_i[1], \pi^*_i[2]))$ 16: Compute  $\hat{H}_j = (\prod_{\ell \in S} \hat{c}_i^{(\ell)}[1]) / ek_i^r$ 17: if  $\pi_i^*[0]$ ,  $\pi_i^*[1]$ , and  $\pi_i^*[2]$  are valid, and  $v_j = \hat{H}_j \cdot h^{\hat{z}'_j} \cdot \mathsf{tpk}_j$  then 18: 19: return 1 20: return 0

 $\widetilde{\mathbb{M}} = \{\widetilde{\mathcal{P}}_i\}_{i \in [\widetilde{n}]}$ . We assume all parties have generated key pairs for the digital signature scheme and the VE scheme during the PKI setup. Let  $(ek_i, dk_i)$  and  $(vk_i, sk_i)$  (resp.  $(\widetilde{ek}_i, \widetilde{vk}_i)$  and  $(\widetilde{vk}_i, \widetilde{sk}_i)$ ) denote encryption key pair and signature key pair of  $\mathcal{P}_i \in \mathbb{M}$  (resp.  $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ ).

Our protocol uses the following primitives as ingredients: (1) the (n,t)-PVSS protocol and its associated verifiable computation {CompProve, VrfyComp} (described in §4.1), (2) the Dumbo-MVBA protocol [48], (3) the PD protocol (cf. Appendix 7), and (4) the threshold common coin Coin.Get (recalled in Appendix 7). Note that parties in M have established a dLog-based (n,t)-threshold cryptosystem with t = n - f - 1 and  $f = \lfloor \frac{n-1}{3} \rfloor$ , providing sufficient setup to execute Dumbo-MVBA, PD, and Coin.Get within M.

**The protocol.** With above setup and ingredients, we present our ADKR protocol in Algorithm 3, and describe its execution flow in the following.

- Sharing phase (Lines 1-2 for P<sub>i</sub> ∈ M and Lines 18-20 for P̃<sub>i</sub> ∈ M̃): Each node P<sub>i</sub> ∈ M uniformly samples s ∈ Z<sub>p</sub> and deals the shares of s to M̃ via PVSS.Deal. Each P̃<sub>i</sub> ∈ M̃ participates in all PVSS.Deal instances and receives aux<sup>(j)</sup><sub>i</sub> from each instance.
- Disperse phase (Lines 3-10 for P<sub>i</sub> ∈ M): After having transcript<sup>(i)</sup> from PVSS.Deal, each node P<sub>i</sub> ∈ M disperses the transcript to M via PD[i]. P<sub>i</sub> participates in all PD[j] for j ∈ [n], and stores the fragments it delivered in these instances.

#### Algorithm 3 ADKR, run by parties in both $\mathbb{M}$ and $\mathbb{M}$

// Code run by each old committee member  $\mathcal{P}_i \in \mathbb{M}$ let *store*[*j*]  $\leftarrow \bot$ , *lock*[*j*]  $\leftarrow \bot$  for *j*  $\in$  [*n*] **initialize** a provable dispersal instance PD[j] for  $j \in [n]$ SHARING PHASE (old committee part): 1: randomly sample  $s \leftarrow \mathbb{Z}_p$ . 2: call PVSS.Deal[i] $\langle \mathcal{P}_i(\{\widetilde{ek}_i, \widetilde{vk}_i\}_{i \in [\widetilde{n}]}, s), \{\widetilde{\mathcal{P}}_i(\widetilde{sk}_i)\}\rangle$  as the dealer DISPERSAL PHASE: **upon** obtain transcript<sup>(i)</sup> from PVSS.Deal[i] **do** 3: call  $\mathsf{PD}[i] \langle \mathcal{P}_i(\mathsf{transcript}^{(i)}), \mathbb{M} \rangle$ 5: **upon** PD[i] delivers *store* i **do** 6:  $store[j] = store_i$ 7. **upon** PD[j] delivers  $lock_j$  **do** 8:  $lock[j] = lock_i$ if  $lock[j] \neq \bot$  then 9: 10:  $T_i \leftarrow T_i \cup \{(j, lock[j])\}$ AGREE PHASE: 11: **if**  $|T_i| = 2f + 1$  **then** invoke Dumbo-MVBA $\langle \{ \mathcal{P}_i(T_i) \} \rangle$ 12: RECAST PHASE (old committee part) 13: upon Dumbo-MVBA outputs T do call Coin.Get() $\langle \mathbb{M} \rangle$  to get  $T' = \{\ell_1, \dots, \ell_{\kappa}\}$ , s.t.  $(\ell_z, \cdot) \in T$  for  $z \in [\kappa]$  $14 \cdot$ for  $\ell \in T'$  do 15: if  $store[\ell] \neq \bot$  then 16: invoke  $\mathsf{RC}[\ell] \langle \mathcal{P}_i(store[\ell]), \widetilde{\mathbb{M}} \rangle$ 17: // Code run by each new committee member  $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ let  $T^* \leftarrow \bot$ ,  $\operatorname{aux}_i^{(j)} \leftarrow \bot$  for all  $j \in [n]$ ,  $\operatorname{count} = 0$ ,  $\mathsf{PKs} \leftarrow \emptyset$ 18: participate in all PVSS.Deal[j] and PD[j] for  $j \in [n]$ SHARING PHASE (new committee part): 19: upon PVSS.Deal[j] returns aux do  $\mathsf{aux}_i^{(j)} \leftarrow \mathsf{aux}$ 20: RECAST PHASE (new committee part): 21: upon  $RC[\ell]$  returns transcript<sup>( $\ell$ )</sup> do 22:  $\texttt{count} \gets \texttt{count} + 1$ if PVSS.  $Verify(\{\tilde{vk}_i, \tilde{ek}_i\}_{i \in [\tilde{n}]}, transcript^{(\ell)}) = 1$  then 23: 24:  $T^* \leftarrow T^* \cup \{\ell\}$ **KEY DERIVATION:** 25: if  $count = \kappa$  then CompProve((transcript<sup>(l)</sup>)<sub> $l \in T^*$ </sub>,  $\widetilde{dk}_i$ , (aux<sup>(l)</sup>)<sub> $l \in T^*$ </sub>) 26:  $(\mathsf{tpk}_i, \mathsf{proof}_i, z_i)$ send  $\text{KEY}(\text{tpk}_i, \text{proof}_i)$  to all  $\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}$ 27: 28: **upon** receiving  $\text{KEY}(\text{tpk}_i, \text{proof}_i)$  from  $\widetilde{\mathcal{P}}_i$  for the first time **do** if VrfyComp( $(transcript^{(\ell)})_{\ell \in T^*}, tpk_i, proof_i) = 1$  then 29.  $\mathsf{PKs} \leftarrow \mathsf{PKs} \cup \{\mathsf{tpk}_i\}$ 30: if  $|\mathsf{PKs}| \ge 2\tilde{f} + 1$  then Interpolate tpk and any missing tpk i 31: 32. **output**  $z_i$ , tpk and  $\{tpk_j\}_{\widetilde{P}_i \in \widetilde{\mathbb{M}}}$  $\mathsf{Predicate}(T)$  for  $\mathsf{Dumbo-MVBA}$ 33: parse *T* as : {...,  $(\ell, lock_{\ell}), ...$ }

34: if |T| = 2f + 1 and ValidateLock $(\langle ID, \ell \rangle, lock_{\ell}) = 1$  for all items in *T* then

- 35: return 1
- 36: else return 0

- Agree phase (Lines 11-22 for  $\mathcal{P}_i \in \mathbb{M}$ ): After 2f + 1PD instances finish, each  $\mathcal{P}_i \in \mathbb{M}$  collects a set  $T_i$  of the indexes and proofs of those finished PD instances and provides it as input to Dumbo-MVBA. Then, all parties in  $\mathbb{M}$  agree on T, which is a set of 2f + 1 finished PD instances.
- Recast phase (Lines 13-17 for P<sub>i</sub> ∈ M and Lines 21-24 for P̃<sub>i</sub> ∈ M̃): All parties in M jointly invoke a common coin to decide on a set of κ indexes T', such that all indexes in T' are included in T. Then, if P<sub>i</sub> has a fragment from PD[ℓ] for each ℓ ∈ T', it invokes RC[ℓ] to provide the fragment to all parties in M̃. Each P̃<sub>i</sub> ∈ M̃ can thus obtain transcript<sup>(ℓ)</sup> via RC[ℓ] for every ℓ ∈ T'.
- *Key derivation (Lines 25-33 for <i>P*<sub>i</sub> ∈ *M*): After receiving κ sharing transcripts from the RC protocols, each *P*<sub>i</sub> ∈ *M* calculates its secret key share *z<sub>i</sub>* = Σ<sub>ℓ∈T\*</sub> PVSS.Deliver(transcript<sup>(ℓ)</sup>, *dk*<sub>i</sub>, aux<sub>i</sub><sup>(ℓ)</sup>), and runs CompProve to generate the public key share tpk<sub>i</sub> = g<sup>z<sub>i</sub></sup> along with proof<sub>i</sub> attesting that tpk<sub>i</sub> is honestly generated. Then, *P*<sub>i</sub> multicasts (tpk<sub>i</sub>, proof<sub>i</sub>) to all parties in *M*. Finally, after receiving 2*f* + 1 valid public key shares (along with their proofs), each *P*<sub>i</sub> can obtain the group public key tpk by interpolating these tpk<sub>j</sub>'s in the exponent.

**Two natural variants.** While Algorithm 3 describes an ADKR protocol with a high threshold t = 2f, it can be easily adapted to support the following functionalities, which may be of independent interest:

- ADKR with a flexible threshold  $t \in [f, 2f]$ . Our PVSS already supports a flexible threshold, and we use t = 2f primarily because the underlying Dumbo-MVBA and PD require threshold signatures with t = 2f. However, by leveraging silent-setup threshold signatures [23, 34, 51], Dumbo-MVBA and PD can also be implemented, which enables a flexible (n,t)-dLog threshold setup in  $\mathbb{M}$  (as long as such setup still provides common coins).
- ADKG in the coin-aided model. When  $\mathbb{M} = \widetilde{\mathbb{M}}$ , Algorithm 3 describes an ADKG where the need for an existing (n,t)-dLog threshold setup can be eliminated by using a silent-setup threshold signature scheme combined with a coin oracle. This results in a quadratic communication ADKG in the coin-aided model. <sup>8</sup>

A concrete optimization: distributed verification of sharing transcript. In the above protocol, for each transcript, every new committee member needs to independently verify all  $\tilde{n} - \tilde{f}$  signatures and call bVerify to verify the  $\tilde{f}$  encrypted shares corresponding in the set *I*. It is clearly redundant and

costly, as all parties perform  $\kappa n$  same group-exponentiation operations, making a major performance bottleneck.

Nonetheless, we can further reduce the verification cost as follows: instead of verifying all  $\tilde{n}$  items in the transcript, we let each party verify  $2\tilde{f} + 1$  consecutive items. For example,  $\mathcal{P}_i$  verifies the signature or encrypted share of the items indexed by  $\{j \mod \tilde{n}\}_{j \in [i,i+2\tilde{f}]}$  in the transcript. So after verifying these  $\tilde{n} - \tilde{f}$  items assigned to it, each new committee member multicasts  $\tilde{n} - \tilde{f}$  bits within the new committee, where each bit encodes one item's verification result (where 1 represents "accept" and 0 represents "reject"); Then, for other  $\tilde{f}$  items that are not locally verified, the party waits for  $\tilde{f} + 1$  messages from other parties, such that these messages carry the same bit encoding the item's verification result, and it can securely use the bit to indicate whether the item is valid or not.

This approach reduces the computation cost of verifying sharing transcripts by about one-third, at a concretely small price: (i) it costs one round of all-to-all communication within the new committee members, and (ii) each new committee member multicasts a  $(\tilde{n} - \tilde{f})$ -bit message, though causing asymptotically cubic collective communication, it is still concretely small as  $(\tilde{n} - \tilde{f})\tilde{n}^2$  bits approximate the size of  $\tilde{n}^2$  group elements for typical system scales (e.g. about 500).<sup>9</sup>

#### 4.3 Analysis

Number of recovered sharing transcripts. To keep communication and computation costs within  $O(n^2)$ , we select a  $\kappa$ size subset from the output set of MVBA in ADKR. The sharing transcripts dispersed by parties in this sub-committee are recovered, verified, and then used to derive the threshold keys. To guarantee the pseudo-randomness of the generated secret key, at least one recovered transcript in the sub-committee shall be dispersed by a non-corrupted party.

We use the hypergeometric distribution to determine the appropriate sub-committee size, as the sub-committee is randomly sampled from the MVBA output using a threshold common coin. Since the adversary can fully control the order of message delivery in an asynchronous network, the indices of all f corrupted parties could be included by the output set of MVBA. Thus, the probability that a  $\kappa$ -size sub-committee contains at least one non-corrupted party is given by:

$$p = 1 - \mathcal{H}(0, 2f + 1, f + 1, \kappa) \tag{1}$$

where  $\mathcal{H}$  is the hypergeometric distribution with a population size of 2f + 1, the number of "success" objects (honest parties) is f + 1, and  $\kappa$  is the committee size. In Figure 3, we plot the required committee size as a function of the total number of parties n = 3f + 1, for some given probabilities p.

As the total number of parties increases, it becomes evident that a constant-sized committee can ensure the presence of at least one honest party except for the given probabilities p.

<sup>&</sup>lt;sup>8</sup>With recent advancements in silent-setup common coin protocols [30], this yields a quadratic ADKG under the PKI and CRS setup.

<sup>&</sup>lt;sup>9</sup>Without the distributed verification optimization for concrete efficiency,



Figure 3: Sub-committee size  $\kappa$  to ensure some selected dispersal from a non-corrupted sender with probability *p*.

**Communication complexity.** We first call *n* instances of the PD protocol to disperse the  $O(\lambda \tilde{n})$ -size scripts, which incurs a communication cost of  $O(\lambda n \cdot \tilde{n} + \lambda n^2)$  bits. Next, the Dumbo-MVBA requires  $O(\lambda n \cdot \tilde{n} + \lambda n^2)$  communication, where each party provides an input of size  $O(\lambda \tilde{n})$ . Subsequently, each  $\mathcal{P}_i \in \mathbb{M}$  multicasts  $O(\lambda)$ -bit *store* messages across all  $\kappa$  RC instances to  $\mathcal{P}_i \in \tilde{M}$  resulting in a communication cost of  $O(\kappa\lambda n \cdot \tilde{n})$ . Therefore, the total communication cost of the ADKR for parties in  $\mathbb{M}$  is  $O(\kappa\lambda n \cdot \tilde{n} + \lambda n^2 + \lambda n \cdot \tilde{n})$ . For parties in  $\tilde{\mathbb{M}}$ , each party multicasts  $O(\lambda)$ -size KEY message, leading to a communication cost of  $O(\lambda \tilde{n}^2)$ . Thus, the overall communication complexity of our ADKR is  $O(\kappa\lambda n \cdot \tilde{n} + \lambda n^2 + \lambda \tilde{n}^2)$ .

**Computation complexity.** Here we count the number of group exponentiation operations, which corresponds to the dominating cryptographic cost. For parties in  $\mathbb{M}$ , everyone invokes an PVSS.Deal instance as a dealer and initiates one PD protocol as sender while participating in other PD instances as non-sender. In this phase, each party performs  $O(n + \tilde{n})$  group-exponentiation operations. During the consensus phase, the number of group exponentiation for each party is  $O(n + \tilde{n})$ . For parties in  $\mathbb{M}$ , each party verifies  $\kappa$  sharing transcripts, each containing  $O(\tilde{n})$  items, resulting in  $O(\kappa \tilde{n})$  group-exponentiation operations. The number of group-exponentiation parts is  $O(\tilde{n}\log \tilde{n})$  (assuming DFT in the exponent). Therefore, the number of group exponentiation operations per party is  $\tilde{O}(\kappa \tilde{n} + \tilde{n}\log \tilde{n} + n)$ .

**Security analysis.** We establish the following theorem regarding the security of our ADKR.

**Theorem 1.** The algorithms shown in Algorithm 3 realize ADKR in a fully asynchronous network model with up to  $\lfloor \frac{n-1}{3} \rfloor$  corruptions in  $\mathbb{M}$  and  $\lfloor \frac{\tilde{n}-1}{3} \rfloor$  corruptions in  $\widetilde{\mathbb{M}}$ , conditioned on the hardness of Discrete Log problem and that the underlying primitives are all secure.

In the following, we first sketch the security intuition, and then prove the termination (in Lemma 4), key-validity (in Lemma 5 and 6), and full secrecy(in Lemma 7), respectively.

**Security intuition.** The *termination* of the protocol directly follows from the *termination* of MVBA, Coin, and APDB.

For Key Validity, the agreement of MVBA and the consistency of Coin ensure that the same set of scripts is selected, while the recast-ability of APDB guarantees the reconstruction of each script is consistent among all parties.

For *Full Secrecy*, as long as at least one of the  $\kappa$  scripts is provided by an honest party, it ensures the *pseudo-randomness* of the secret key *z*. Since all secret shares are encrypted and we use an  $(\tilde{n}, \tilde{t})$ -Shamir secret sharing scheme, the adversary cannot learn any information about *z* beyond the public keys.

**Lemma 2.** If all honest parties in  $\mathbb{M}$  and  $\widetilde{\mathbb{M}}$  invoke ADKR, then every honest party  $\mathcal{P}_i \in \widetilde{\mathbb{M}}$  will generate an aggregate commitment  $\{v_j\}_{\mathcal{P}_i \in \widetilde{\mathbb{M}}}$ .

*Proof.* From the *termination* of APDB, if all honest parties in  $\mathbb{M}$  invoke ADKR and pass transcripts into the corresponding PD instances, then all honest parties in  $\mathbb{M}$  can output *lock* from at least 2f + 1 PD instances. Consequently, each honest party  $\mathcal{P}_i \in \mathbb{M}$  will input valid set  $T_i$  to MVBA, so all of them can output the same set T and call Coin.Get() according to the *termination* and *agreement* of MVBA. Then from *termination* and *consistency* of Coin, all honest parties can get the same subset T' from T. The *external-validity* of MVBA requires that the output satisfies the predicate Q, so for any index in T', there exists a valid *lock* value which implies that at least f + 1 honest parties have its *store* values. These parties will send their *store* values to parties in  $\widetilde{M}$  by invoking RC[ID,  $\ell$ ] instances for  $\ell \in T'$ .

Then all honest parties in  $\widetilde{\mathbb{M}}$  can receive at least f + 1 valid fragments and output in each RC instance corresponding to the index in T'. Thus, after calling PVSS.Verify to eliminate incorrect scripts, all honest parties in  $\widetilde{\mathbb{M}}$  will generate an aggregate commitment  $\{v_j\}_{\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}}$  when CompProve is invoked.

**Lemma 3.** For any two honest parties  $\widetilde{\mathcal{P}}_i$  and  $\widetilde{\mathcal{P}}_j$  that generate aggregate commitment  $\mathbf{v}$  and  $\mathbf{v}'$  respectively in the ADKR protocol,  $\mathbf{v} = \mathbf{v}'$ .

From the *agreement* of MVBA, any honest party in  $\mathbb{M}$  who outputs from MVBA will output the same set T and then select the same subset T' by calling the Coin.Get. According to the *external-validity* of MVBA, for any script<sup>( $\ell$ )</sup> that  $\ell \in T'$ , there are at least f + 1 honest parties who have stored the *store* values of it. Thus, for any  $\ell \in T'$ , at least f + 1 honest parties will input *store* values for the same vector commitment vc to RC[ID, $\ell$ ], and no honest party will input to any RC[ID, $\ell$ ] that  $\ell \notin T'$  or input to a RC[ID, $\ell$ ] that  $\ell \in T'$  with a *store* value that the vc is inconsistent with the *lock*[k] output in MVBA. Since a party in  $\widetilde{\mathbb{M}}$  will output from a RC instance only if it receives at least f + 1 valid fragments for the same

our design has asymptotically quadratic communication overhead.

vc, all honest parties in  $\widetilde{\mathbb{M}}$  can only output from  $\mathsf{RC}[\mathsf{ID}, \ell]$  that  $\ell \in T'$ .

So we prove the lemma by a contradiction. Assuming that there exist two honest parties in  $\widetilde{\mathbb{M}}$  that generate aggregate commitment  $\mathbf{v} \neq \mathbf{v}'$ . Since any honest party's  $\mathbf{v}$  is aggregated from the transcripts outputs of RC instances,  $\widetilde{\mathcal{P}}_i$  and  $\widetilde{\mathcal{P}}_j$  must receive different outputs transcript<sup>(\ell)</sup>  $\neq$  transcript'<sup>(\ell)</sup> from at least one RC[ID,  $\ell$ ] that  $\ell \in T'$ . Recall that all valid *store* values in one RC instance must have the same vc, this will immediately break the *correctness* of erasure code or the *collision resistance* of the hash function. Hence, any honest party in  $\widetilde{\mathbb{M}}$  that generates an aggregate commitment  $\mathbf{v} = \{v_j\}_{\mathcal{P}_j \in \widetilde{\mathbb{M}}}$ will generate the same one.

**Lemma 4 (Termination).** If all honest parties in  $\mathbb{M}$  and  $\mathbb{\tilde{M}}$  invoke ADKR, then every honest party  $\tilde{\mathcal{P}}_i$  in  $\mathbb{\tilde{M}}$  will output the public keys  $(g^z, g^{z_1}, g^{z_2}, \cdots, g^{z_{\tilde{n}}})$  and a private key share  $z_i$ .

*Proof.* From Lemma 2 and 3, every honest party in  $\widetilde{\mathbb{M}}$  will generate the same aggregate commitment  $\{v_j\}_{\mathscr{P}_j \in \widetilde{\mathbb{M}}}$ . Then each honest  $\widetilde{\mathscr{P}}_i \in \widetilde{\mathbb{M}}$  will calculate its private key  $z_i$ , public key share tpk<sub>i</sub> and a proof<sub>i</sub> from CompProve. Then honest  $\widetilde{\mathscr{P}}_i$  will send a KEY message containing tpk<sub>i</sub> =  $g^{z_i}$ . The proofs generated by any honest parties in  $\widetilde{\mathbb{M}}$  can be verified by any honest parties in  $\widetilde{\mathbb{M}}$  can be verified by any honest party in  $\widetilde{\mathbb{M}}$  according to Lemma 3 and the *completeness* of the proofs. So each honest party in  $\widetilde{\mathbb{M}}$  can collect at least  $2\widetilde{f} + 1$  valid public key shares tpk<sub>j</sub> corresponding to the aggregated commitment. Given that the degree of all sharing polynomials generated in the PVSS instances in  $T^*$  is no more than  $2\widetilde{f} + 1$ , they can interpolate the public key tpk =  $g^z$  and all missing public key shares.

**Lemma 5** (Key Validity 3). Any  $2\tilde{f} + 1$ -subset of private key shares provided by the outputs can recover the same master private key *z*.

*Proof.* From Lemma 3, any honest party in  $\mathbb{M}$  generates the same aggregate commitment  $\mathbf{v} = \{v_j\}_{\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}}$ . From the description of the ADKR and CompProve, the commitment  $\mathbf{v}$  is aggregated from the same set of verified transcript<sup>( $\ell$ )</sup> s that  $\ell \in T^*$ , where the degree of each sharing polynomial is no more than  $2\widetilde{f}$ , and each share  $z_j^{(\ell)}$  is bound to the commitment value  $v_j^{(\ell)}$  in transcript<sup>( $\ell$ )</sup>. So the private key share  $z_j$  for each  $\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}$  is on the polynomial of degree  $\leq 2\widetilde{f}$ . Therefore, any  $2\widetilde{f} + 1$ -subset of private key shares recover the same z.

**Lemma 6 (Key Validity 1&2).** The outputs provide the same public keys  $(g^z, g^{z_1}, g^{z_2}, \dots, g^{z_{\tilde{n}}})$ . Any private key share  $z_i$  carried by the outputs is consistent with the public key of  $g^{z_i}$ .

*Proof.* From the description of the protocol, each transcript<sup>( $\ell$ )</sup> verified by PVSS.Verify satisfies that  $\{v_j^{(\ell)}\}_{j\in \widetilde{\mathbb{M}}}$  is a Pedersen commitment to the sharing polynomial with degree  $\leq 2\widetilde{f}$ . From Lemma 5, the private key shares

corresponds to a unique polynomial  $\phi(\cdot)$  (and  $\hat{\phi}(\cdot)$ ) of degree  $\leq 2\tilde{f}$ . Then  $\{v_j\}_{j\in\tilde{\mathbb{M}}}$  is a Pedersen commitment to  $\phi(\cdot)(\text{and }\hat{\phi}(\cdot))$  since the Pedersen commitment is additively holomorphic.

Now we prove that the honest party can output all the public key shares.

Each honest party in  $\widetilde{\mathbb{M}}$  receives a KEY message from  $\widetilde{\mathcal{P}}_j$  containing tpk<sub>j</sub> =  $g^{z_j}$  and a proof<sub>j</sub> Each proof<sub>j</sub> contains 3 proofs:

- $\pi_i^*[0]$  proves that  $\mathcal{P}_j$  knows  $z_j$ .
- $\pi_j^*[1]$  proves that  $\widetilde{\mathcal{P}}_j$  knows the sum of all  $\hat{z}_j^{(\ell)}$  for  $\sigma_j^{(\ell)} \in \Sigma^{(\ell)}$ .
- $\pi_j^*[2]$  proves that  $\widetilde{\mathscr{P}}_j$  knows the product of all  $h^{\hat{z}_j^{(\ell)}}$  for  $c_i^{(\ell)} \in c^{(\ell)}$ .

Through the above two proofs  $\pi_j^*[1]$  and  $\pi_j^*[2]$ ,  $h^{\hat{z}_j} = \hat{H}_j \cdot h^{\hat{z}'_j}$  can be calculated. Then by the binding property of Pedersen commitment, we know that  $z_j = \phi(j)$  and  $h^{\hat{z}_j} = h^{\phi(j)}$ . Therefore, the public key  $g^{\phi(0)}$  and all missing public key shares can be interpolated from any  $2\tilde{f} + 1$  shares  $g^{z_j}$ .

**Lemma 7 (Full Secrecy).** No computationally bounded adversary can (i) bias the uniform sampling of master secret key z over  $\mathbb{Z}_q$  or (ii) learn information about z beyond the released public keys.

*Proof.* We prove the full secrecy of our ADKR by building a PPT simulator S, which on inputs a uniformly distributed group element  $y \in \mathbb{G}$  can simulate an execution of ADKR where y is group public key, such that any PPT adversary  $\mathcal{A}$  that corrupts up to  $\lfloor \frac{n-1}{3} \rfloor$  nodes in  $\mathbb{M}$  and  $\lfloor \frac{\tilde{n}-1}{3} \rfloor$  nodes in  $\widetilde{\mathbb{M}}$  cannot distinguish the simulated execution and a real execution which returns y as the group public key tpk.

We describe the simulator S as follows.

**Notations:** We denote the set of parties controlled by the adversary  $\mathcal{A}$  in  $\mathbb{M}$  and  $\widetilde{\mathbb{M}}$  as B and  $\widetilde{B}$ , respectively. The sets of honest party in  $\mathbb{M}$  and  $\widetilde{\mathbb{M}}$  are represented as G and  $\widetilde{G}$ . Without loss of generality, let  $B = \{\mathcal{P}_1, ..., \mathcal{P}_f\}, G = \{\mathcal{P}_{f+1}, ..., \mathcal{P}_n\}$  (with  $\widetilde{B}$  and  $\widetilde{G}$  defined similarly).

**Input:** a public key 
$$y \in \mathbb{G}$$
.

**Simulation:** Firstly, sample  $i^* \leftarrow [n]$  which represents a party in  $\mathbb{M}$ . S generates encryption key pairs  $(ek_i, dk_i)$  and signature key pairs for all parties in  $\widetilde{\mathbb{M}}$ . After that, S simulates an execution as follows.

<u>Simulating PVSS.Deal</u>: S simulates PVSS.Deal $[i^*]$  by the following strategy: For  $i \in [1,t]$ , S uniformly samples  $s_i^{(i^*)}$  and  $\hat{s}_i^{(i^*)}$ , and computes  $v_i^{(i^*)} = g^{s_i^{(i^*)}} \cdot h^{\hat{s}_i^{(i^*)}}$ . S uniformly samples  $v_0^{(i^*)}$ . For all  $i \in [2t + 1, n]$ , interpolate  $v_i^{(i^*)} = \prod_{j \in [0,t]} (v_j^{(i^*)})^{\lambda_j(i)}$ , and  $\lambda_j(x) = \prod_{k \in [0,t], k \neq j} \frac{j-x}{j-k}$ .

Then, S sends  $(v_i^{(i^*)}, s_i^{(i^*)}, \hat{s}_i^{(i^*)})$  to each  $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ , on the behalf of  $\mathcal{P}_{i^*} S$  simply sends  $(v_i^{(i^*)}, \bot)$  to honest nodes. On the behalf of  $\widetilde{\mathcal{P}}_i \in \widetilde{G}$ , S signs each  $v_i^{(i^*)}$ , and sends the signature to  $\mathcal{P}_{i^*}$  on the behalf of honest  $\widetilde{\mathcal{P}}_i$ .

Finally, S can collects valid signatures for  $\{v_i^{(i^*)}\}_{i\in S}$ , where  $S \in [\tilde{n}]$  with  $|S| = 2\tilde{f} + 1$ . For every  $j \notin S$ , if  $j \in B'$ , S honestly uses VE to encrypt  $(s_i^{(i^*)}, \hat{s}_i^{(i^*)})$ ; Otherwise, S encrypts a random value under  $\tilde{ek}_j$  and simulates a proof. By doing so, S can obtain transcript $(i^*)$ .

Regarding other honest nodes, S honestly executes the protocol on the behalf of them in PVSS.Deal instances, except their responses to the messages from  $\mathcal{P}_{i^*}$  (which we have specified above).

Simulating the Dispersal Phase: For all operations until  $\overline{\text{Dumbo-MVBA}}$  outputs, S acts on the behalf of each honest nodes by executing the protocol.

Simulating the Agree Phase: When Dumbo-MVBA outputs  $\overline{T'}$ , if  $(i^*, \cdot)$  is not included in T', S shall **abort** the current simulated execution, sample a new  $i^* \leftarrow G$ , rewind  $\mathcal{A}$  to the begining of the simulation, and **restart** a new simulated execution. If  $(i^*, \cdot)$  is included in T', S continues to simulate the subsequent phases.

Simulating the Recast Phase: For Coin.Get, recall that the coin value is RO(ThldSig). S computes the unique threshold signature Thld on the default message of this execution (since S knows the secret keys of all honest nodes), and programs RO such that  $i^* \in \text{RO}(\text{ThldSig})$ . S honestly executes the protocol on the behalf of honest nodes for the subsequent operations of the Recast Phase.

*Simulating the Key Derivation:* First, *S* computes the public key shares of honest nodes as follows:

- For  $j \in [1, 2\tilde{f}]$ , computes  $\mathsf{tpk}_{j}^{(i^*)} = g^{a_j}$ .
- For  $j \in [\tilde{f}+1, \tilde{n}], \bar{z}_j = \sum_{\ell \in T^* \setminus \{i^*\}} s_j^{(\ell)}, \mathsf{tpk}_j = g^{\bar{z}_j}.$
- Interpolate tpk from  $\{tpk_i\}, j \in [2\tilde{f}+1, n]$ .
- Compute  $tpk_0^{(i^*)} = y/tpk$ .
- Interpolate  $tpk_j^{(i^*)}$  for  $j \in [2\tilde{f} + 1, \tilde{n}]$  from  $\{tpk_j^{(i^*)}\}$  for  $j \in [0, 2\tilde{f}]$
- For  $\widetilde{\mathcal{P}}_j \in \widetilde{G}$ , compute  $\mathsf{tpk}_j = \mathsf{tpk}_j^{(i^*)} \cdot \bar{\mathsf{tpk}}_j$ .

Then, S simulates the computation proof  $\text{proof}_i$  for each  $i \in [2\tilde{f} + 1, \tilde{n}]$  as follows.

- Define  $S_i = \{\ell | \ell \in T \text{ and } i \in I^{(\ell)}\}$ , and  $S'_i = T^* \setminus S_i$
- If  $i^* \in S_i$ , compute  $\hat{z}'_i = \sum_{\ell \in S'_i} \hat{s}^{(\ell)}_i$ , where  $\hat{s}^{(\ell)}_i$  was sent by  $\mathcal{P}_{\ell}$  during the Sharing Phase.

- If  $i^* \in S'_i$ , compute  $\hat{z}''_i = \sum_{\ell \in S'_i, \ell \neq i^*} \hat{s}_i^{(\ell)}$ ,  $h^{\hat{s}_i^{(*)}} = \frac{v_i^{(i^*)}}{\mathsf{tpk}_j^{(i^*)}}$ , and  $h^{\hat{z}'_i} = h^{\hat{z}''_i} \cdot h^{\hat{s}_i^{(i^*)}}$ .
- Invoke the CompProve simulator with (tpk<sub>i</sub>, h<sup><sup>2</sup>/<sub>i</sub></sup>) as input and obtain proof<sub>j</sub>.

With  $(tpk_i, proof_i)$  for all honest nodes in  $\widetilde{G}$ , S honestly executes the rest of the protocol on the behalf of all honest nodes.

**Public key in the simualted execution.** Now we demonstrate that the final public key tpk in the simulated execution is the input of S, namely, y. First, all tpk<sub>j</sub> for parties in  $\tilde{G}$  are computed as tpk<sub>j</sub><sup>(i<sup>\*</sup>)</sup>  $\cdot t\bar{p}k_j$ . Since that  $y = tpk_0^{(i^*)} \cdot t\bar{p}k$  and both  $tpk_0^{(i^*)}$  and  $t\bar{p}k$  are interpolated from  $2\tilde{f} + 1$  shares. y can be interpolated from  $2\tilde{f} + 1$  tpk<sub>j</sub><sup>(i<sup>\*</sup>)</sup>. For any  $\tilde{P}_j \in \tilde{B}$ , it knows  $z_j$  for an opening of tpk<sub>j</sub>. From the *knowledge soundnesss* of NIZK and the binding property of Pedersen commitment. All tpk<sub>j</sub>' with a valid proof<sub>j</sub> are on the *t*-degree polynomial defined by  $\{tpk_j\}$  that  $\tilde{P}_j \in \tilde{G}$ . Thus the output of S is exactly the input public key y.

**Indistinguishability between the simulated execution and a real execution.** Then, we show that a PPT adversary cannot distinguish the simulated execution and a real execution, by considering the following hybrids.

**Hybrid 0:** It is identical to a real execution, where the simulator honestly executes the protocol on the behalf of all honest nodes.

**Hybrid 1:** It is almost identical to **Hybrid 0**, except that if Coin.Get returns  $T^*$  which does not contain an honest node  $i' \in G$ , the simulator aborts the current execution, rewidns the adversary  $\mathcal{A}$  to the beginnig, and restarts the simulation.

**Hybrid 2:** It is almost identical to **Hybrid 1**, except that the simulator randomly selects  $i^* \leftarrow G$  in the beginning.

**Hybrid 3:** It is almost identical to **Hybrid 2**, except that during PVSS.Deal[ $i^*$ |old] and PVSS.Deal[ $i^*$ |new], when  $\mathcal{P}_{i^*}$  needs to encrypt shares for corrupted parties, the simulator encrypts random values and generates a simulated proof.

**Hybrid 4:** It is almost identical to **Hybrid 3**, except that if Dumbo-MVBA outputs T' and  $(i^*, \cdot) \notin T'$ , the simulator aborts the current execution aborts, rewinds the adversary  $\mathcal{A}$  to the begining, and restarts the simulated execution with a freshly sampled  $i^* \leftarrow G$ .

**Hybrid 5:** It is almost identical to **Hybrid 4**, except that after Dumbo-MVBA outputs T', the simulator programs RO at ThldSig such that  $i^* \in RO(ThldSig)$ .

**Hybrid 6:** It is almost identical to **Hybrid 5**, except that during the key derivation phase, the simulator invokes the CompProve simulator to generate proof<sub>i</sub> on the behalf of all  $\tilde{\mathcal{P}}_i \in \tilde{G}$ .

Hybrid 7: It is the simulated execution.

First, from  $\mathcal{A}$ 's perspective, the difference between Hybrid 0 and Hybrid 1 is that in Hybrid 0, the output of Coin.Get is a uniformly sampled subset of  $\kappa$  indexes in T', while in Hybrid 1, the output must include at least one honest  $i' \in G$ . Since the probability that  $\kappa$  randomly sampled indexes do not include any  $i' \in G$  is negligible, the statistical distance between the output distributions of Coin.Get in Hybrid 0 and Hybrid 1 is negligible. Thus,  $\mathcal{A}$  cannot distinguish between the two hybrids with a non-negligible advantage. Additionally, as the probability that the simulator needs to rewind  $\mathcal{A}$  and restart the simulation is negligible, the simulator's expected running time is polynomial in the security parameter.

Next, Hybrids 1 and 2 are identical from  $\mathcal{A}$ 's perspective.

Then, by the IND-CPA security of the underlying VE,  $\mathcal{A}$  cannot distinguish between Hybrids 2 and 3 except with negligible probability.

Furthermore, since the choice of  $i^*$  is independent of  $\mathcal{A}$ 's view, Hybrids 3 and 4 are indistinguishable from  $\mathcal{A}$ 's perspective. As T' contains at least f + 1 honest nodes and  $i^*$  is uniformly chosen from G, the probability that  $(i^*, \cdot) \notin T'$  is at most  $\frac{f}{2f+1}$ . Thus, the simulator is expected to rewind  $\mathcal{A}$  approximately twice, keeping the simulator's running time polynomial in the security parameter.

Next, the unforgeability of the unique threshold signature ensures that  $\mathcal{A}$  cannot obtain ThldSig before honest participants invoke Coin.Get(). Consequently, the probability that  $\mathcal{A}$  queries RO with ThldSig before RO is programmed is negligible. Hence,  $\mathcal{A}$  cannot distinguish between Hybrids 4 and 5 except with negligible probability.

Then, ensured by the zero-knowledge property of CompProve, the adversary cannot distinguish Hybrid 5 and Hybrid 6 except with a negligible probability.

Finally, Hybrids 6 and 7 are indistinguishable from  $\mathcal{A}$ 's perspective. Therefore,  $\mathcal{A}$  cannot distinguish between the real execution and the simulated execution except with negligible probability.

## 5 Application to Asynchronous Dynamic Proactive Secret Sharing

In this section, we present a quadratic-communication asynchronous dynamic proactive secret sharing (ADPSS) scheme, based on an extension to our ADKR protocol.

**Definition.** In an  $(n, t, \tilde{n}, \tilde{t})$ -ADPSS protocol executed among two sets  $\mathbb{M}$  and  $\widetilde{\mathbb{M}}$  of parties, where  $|\mathbb{M}| = n$  and  $|\widetilde{\mathbb{M}}| = \tilde{n}$ . Each  $\mathcal{P}_i \in \mathbb{M}$  hold a share  $m_i$  of a secret m. After the protocol terminates, each honest  $\mathcal{P}_i \in \widetilde{\mathbb{M}}$  will output a refreshed share  $m'_i$  of the secret m. Such an ADPSS satisfies the following properties with all but negligible probability:

• *Termination:*. If all honest parties invoke ADPSS, then all honest parties will eventually terminate the protocol.

- *Correctness:* Any honest 𝒫<sub>i</sub> ∈ M̃ that terminates will output a *t*-threshold share m'<sub>i</sub> of the secret m.
- *Secrecy:* No PPT adversary can learn any information of the secret *m*.

# 5.1 A generic construction of ADPSS using DCRG [53]

Our construction closely follows a (semi)-generic construction introduced in [53], which, essentially, builds an ADPSS from a component which we term by the following *Dual-Committee Randomness Generation* (DCRG) scheme.

<u>SYNTAX OF DCRG.</u> In an  $(n,t,\tilde{n},\tilde{t})$ -DCRG, there are two committees:  $\mathbb{M} = \{\mathcal{P}_i\}_{i \in [n]}$  and  $\widetilde{\mathbb{M}} = \{\widetilde{\mathcal{P}}_i\}_{i \in [\tilde{n}]}$ . DCRG aims to sample two uniform random values *s* and  $\hat{s}$ , such that parties in  $\mathbb{M}$  have (n,t) Shamir secret sharings of  $(s,\hat{s})$ , while parties in  $\widetilde{\mathbb{M}}$  have  $(\tilde{n},\tilde{t})$  Shamir secret sharings of them. More specifically, at the end of the execution, parties in  $\mathbb{M}$  agree on a vector of Pedersen commitments  $(g^s h^{\hat{s}}, g^{s_1} h^{\hat{s}_1}, \dots, g^{s_n} h^{\hat{s}_n})$ , and each  $\mathcal{P}_i \in \mathbb{M}$  receives a private output  $(s_i, \hat{s}_i)$ , such that  $\{s_i\}$  and  $\{\hat{s}_i\}$  of honest nodes are (n,t) Shamir secret sharing of *s* and  $\hat{s}$ , respectively. At the same time, parties in  $\widetilde{\mathbb{M}}$ outputs  $(g^s h^{\hat{s}}, g^{s'_1} h^{s'_1}, \dots, g^{s_{\overline{n}}} h^{\hat{s}_{\overline{n}}})$ , and each  $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$  receives a private output  $(s'_i, \hat{s}'_i)$ , such that  $\{s'_i\}$  and  $\{s'_i\}$  of honest nodes are (n,t) Shamir secret sharing of *s* and  $\hat{s}$ , respectively.

<u>ADPSS PROTOCOL BASED ON DCRG.</u> Now we describe how to realize ADPSS by using DCRG. In the ADPSS protocol, each party  $\mathcal{P}_i \in \mathbb{M}$  holds shares  $m_i$  and  $\hat{m}_i$  of the secrets m and a hiding randomness  $\hat{m}$ , respectively, along with a set of Pedersen commitments  $\{\mathrm{cm}_j\}_{\mathcal{P}_j \in \mathbb{M}}$ , where  $\mathrm{cm}_j = g^{m_j} h^{\hat{m}_j}$ . The goal is to enable the parties in  $\widetilde{\mathbb{M}}$  to have  $(\widetilde{n}, \widetilde{t})$  Shamir secret shares of the same m and  $\hat{m}$ , as long with an updated set of commitments  $\{\widetilde{\mathrm{cm}}_j\}_{\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}}$ .

The ADPSS protocol is presented in Algorithm 4. At a high level, it leverages the DCRG protocol to generate two sets of distinct shares for the same pair of random numbers r and  $\hat{r}$  across two configurations. The old and new configurations use them to transmit a pair of masked secrets m + r and  $\hat{m} + \hat{r}$ . The new configuration derives new shares of m by computing  $m + r - r'_i$ . Specifically:

- *Initialization:* Each party P<sub>i</sub> ∈ M holds shares m<sub>i</sub> and m̂<sub>i</sub> of the secrets m and a random number m̂, respectively, along with a set of Pedersen commitments {cm<sub>j</sub>}<sub>P<sub>j</sub>∈M</sub>, where where cm<sub>j</sub> = g<sup>m<sub>j</sub></sup>h<sup>m̂<sub>j</sub></sup>.
- Invocation of DCRG (Lines 1-2, Lines 10-11): parties in both M and M̃ jointly invoke an instance of the DCRG protocol to generate two sets of secret shares for the random numbers r and r̂. Specifically, P<sub>i</sub> ∈ M obtains shares r<sub>i</sub>, r̂<sub>i</sub> along with a set of Pedersen commitments {g<sup>r<sub>j</sub></sup>h<sup>r̂<sub>j</sub></sup>}<sub>P<sub>i</sub>∈M</sub>. Correspondingly, a node P'<sub>i</sub> ∈ M̃

receives shares  $r'_i$ ,  $\hat{r'}_i$  and a set of Pederson commitments  $\{g^{r'_j}h^{\hat{r'}_j}\}_{\mathcal{P}_i \in \widetilde{\mathbb{M}}}$ .

- *Reconstruction of the Masked Secrets (Lines 3-8):* Each *P<sub>i</sub>* ∈ M computes the masked shares (*m<sub>i</sub>*+*r<sub>i</sub>*, *m̂<sub>i</sub>*+*r̂<sub>i</sub>*) and multicasts them to all parties in M via a REC message to reconstruct *m*+*r* and *m̂*+*r̂*. When *P<sub>i</sub>* ∈ M receives 2*f*+1 REC(sh<sub>j</sub>, sh<sub>j</sub>) satisfying the condition *v<sub>j</sub>*cm<sub>j</sub> == g<sup>sh<sub>j</sub></sup>h<sup>sh<sub>j</sub></sup>, it uses Lagrange interpolation to reconstruct *m*+*r* and *m̂*+*r̂*.
- Share Transmission (Lines 9, 12): parties in M then transmit the masked values m + r and m̂ + r̂ to the parties in M̃ with TRANS messages. Due to the randomness of r and r̂, the TRANS messages will not leak any information about m and m̂. parties in M̃ ensure the consistency of the masked secrets by receiving at least f + 1 TRANS messages with the same values.
- Generation of New Shares (Lines 13-16): Each 𝒫'<sub>i</sub> ∈ M̃ computes their respective shares of m and m̂ by calculating (m + r) − r'<sub>i</sub> and (m̂ + r̂) − r̂'<sub>i</sub>. It also outputs a set of commitment for the new shares, defined as cm'<sub>i</sub> = g<sup>m+r</sup>h<sup>m̂+r̂</sup>/c'<sub>i</sub>.

Al	gorithm 4 ADPSS run by parties in both IVI and IVI
	// Code run by each $\mathcal{P}_i \in \mathbb{M}$
	initialize $S \leftarrow \emptyset$
	$\mathcal{P}_i$ takes $m_i$ , $\hat{m}_i$ and $\{cm_j\}_{j\in\mathbb{M}}$ as input
1:	invoke DCRG as a party in $\mathbb{M}$
2:	<b>wait</b> for DCRG output $r_i, \hat{r}_i, \{v_j\}_{\mathcal{P}_j \in \mathbb{M}}$
3:	<b>multicast</b> $\operatorname{REC}(m_i + r_i, \hat{m}_i + \hat{r}_i)$ to all parties in $\mathbb{M}$
4:	<b>upon</b> receiving $\operatorname{Rec}(\operatorname{sh}_j, \widehat{\operatorname{sh}}_j)$ from $\mathscr{P}_j \in \mathbb{M}$ <b>do</b>
5:	<b>if</b> $\operatorname{cm}_j \cdot v_j = g^{\operatorname{sh}_j} \cdot h^{\widehat{\operatorname{sh}}_j}$ <b>then</b>
6:	$S \leftarrow S \cup \{(j, (sh_j, \hat{sh}_j))\}$
7:	if $ S  \ge 2f + 1$ then
8:	Interpolate $m + r$ and $\hat{m} + \hat{r}$ from S
9:	<b>multicast</b> TRANS $(m+r, \hat{m}+\hat{r})$ to all parties in $\widetilde{\mathbb{M}}$
	// Code run by each $\mathcal{P} \in \widetilde{\mathbb{M}}$
10.	invoke DCRG as a party in $\widetilde{\mathbb{M}}$
11:	wait for DCRG output $r'_i$ , $\hat{r'}_i$ , $\{v'_i\}_{i=1}^{\infty}$
12.	<b>upon</b> receiving $f + 1$ TRANS messages with the same $(m + r)$ and
12.	$(\hat{m} + \hat{r})$ from distinct parties in $\mathbb{M}$ do
13.	$m' \leftarrow (m+r) - r', \ \hat{m}' \leftarrow (\hat{m}+\hat{r}) - \hat{r}'.$
14.	for $\mathcal{O}_{\mathcal{O}} \subset \widetilde{\mathbb{M}}$ do
15.	$\operatorname{cm}' \leftarrow (a^{m+r}h^{\hat{m}+\hat{r}})/v'$
1.5.	$cm_i \times (s n)/v_i$

16: **output**  $m'_i$ ,  $\hat{m}'_i$  and  $\{\mathsf{cm}'_j\}_{\mathcal{P}_i \in \widetilde{\mathbb{M}}}$ 

• 41

Note that except the DCRG part, the communication cost of the remaining part is merely  $O(\lambda n^2)$ . However, [53] realizes DCRG by using *n* instances of a variant of PVSS and thus incurs  $O(\lambda n^3)$  communication cost. So we strive to reduce the overhead of DCRG, from cubic to quadratic, towards realizing quadratic-commutation ADPSS.

#### 5.2 Our quadratic-communication DCRG

Here we describe how to realize a DCRG protocol with  $O(\lambda n^2)$  communication cost, by extending our ADKR design. A quadratic-communication ADPSS is immediately achieved by replacing the component in Algorithm 4.

<u>SETUP AND INGREDIENTS.</u> Our DCRG assumes the same setup and ingredients as our ADKR, except that the underlying PVSS needs to be slightly modified.

In particular, since in DCRG, honest parties should output shares of both *s* and  $\hat{s}$ , such that  $g^s h^{\hat{s}}$  is the public Pedersen commitment, the PVSS scheme should support the dealer to specify both *s* and  $\hat{s}$  and enable each receiver to deliver the shares of both *s* and  $\hat{s}$ .

The above functionality can be supported by using a VE scheme to encrypt both  $s_i$  and  $\hat{s}_i$  for a receiver  $\mathcal{P}_i$ , so that  $\mathcal{P}_i$  can obtain both shares by decrypting the ciphertexts. As discussed in [25], the VE scheme from [38] suffices for this purpose. This variant of PVSS is also described in Algorithm 1 (including dash-boxed items but excluding boxed items).

Algorithm 5	DCRG, run by part	ies in t	both $\mathbb M$ and $\widetilde{\mathbb M}$
// Code run	by each old committee	e membe	er $\mathcal{P}_i \in \mathbb{M}$
<b>let</b> <i>store</i> [ <i>j</i> ] $f$ <b>initialize</b> a $j$ $j \in [n]$	$\leftarrow \bot$ , aux <sup>(j)</sup> $lock[j] \leftarrow \bot$ provable dispersal instance	for $j \in$ the PD[ $j$ ]	[ <i>n</i> ]. count $\leftarrow 0$ and PVSS.Deal[ <i>j</i>  old] for
1: randomly sa	ample $s, \hat{s}$ .		
2: call PVSS.	$Deal[i old] \langle \mathscr{P}_i(\{ek_i, vk_i\}_i$	$\in [n], s, \hat{s})$	$\{\mathcal{P}_i(sk_i)\}\$ as the dealer
3: call PVSS. 4: upon PVS	Deal[ <i>i</i>  new] $\langle \mathcal{P}_i(\{ek_i, vk_i\}   S.Deal[j old] returns aut$	$k_{i\in [\widetilde{n}]}, s, \hat{s}$ × <b>do</b>	$(\mathcal{P}_{i}(sk_{i}))$ as the dealer
5: $\operatorname{aux}_{i}^{(j)} \leftarrow$	aux		
6: upon ob	tain transcript <sup>(i old)</sup>	from	PVSS.Deal[i old] and
transcript <sup>(i</sup>	<sup>(new)</sup> from PVSS.Deal[ <i>i</i>	new] do	
7: call PD[i	$]\langle \mathcal{P}_i(transcript^{(i old)} tra)]$	nscript <sup>(i</sup>	$ new), \mathbb{M}\rangle$
Then, Execu	ate Lines 5-17 in Algorit	hm 3 wh	ile Line 17 is changed to
8: invoke R	$C[\ell] \langle \mathcal{P}_i(store[\ell]), \mathbb{M} \cup \widetilde{\mathbb{M}}$	$\rangle$	▷ Recast to all partie
9: <b>upon</b> RC[ℓ]	returns transcript <sup>(<math>\ell</math> old)</sup>	transcri	$pt^{(\ell new)} \mathbf{do}$
$10: \text{ count} \leftarrow$	count + 1		
1: parse tra	$\operatorname{nscript}^{(\ell \operatorname{old})} = (\{v_j^{(\ell \operatorname{old})}\})$	$\}_{j\in[n]},\cdot),$	and transcript <sup><math>(\ell new)</math></sup>
2: <b>if</b> both tr	anscript $(\ell old)$ and trans	cript <sup>(<i>l</i> ne</sup>	$(e^{i})$ are valid, and $v_0^{(\ell \mid old)}$
$v_{\rm o}^{(\ell {\rm new})}$ then	1		0
$3: T^* \leftarrow$	$T^* \cup \{\ell\}$		
4: $(s_i^{(\ell)}, \hat{s}_i)$	${\ell \choose i} \leftarrow PVSS.Deliver(tr)$	anscript	$d^{(\ell old)}, dk_i, aux_i^{(j)})$
5: <b>if</b> count 6: <b>return</b>	$= \kappa \text{ then } \\ \mathfrak{l} \left( \left\{ \prod_{\ell \in T^*} v_j^{(\ell old)} \right\}_{j \in [n]}, \Sigma \right.$	$_{\ell\in T^*} s_i^{(\ell)},$	$\sum_{\ell\in T^*} \hat{s}_i^{(\ell)})$
// Code run let $T^* \leftarrow \bot$ 17: participate i 18: upon PVS	by each new committee , $aux_i^{(j)} \leftarrow \bot$ for all $j \in$ in all PVSS.Deal[ <i>j</i>  new] S.Deal[ <i>j</i>  new] returns a	e memb [n], coun   and PD ux <b>do</b>	er $\widetilde{\mathscr{P}}_i \in \widetilde{\mathbb{M}}$ et = 0, PKs $\leftarrow \emptyset$ [ $j$ ] for $j \in [n]$

19:  $\operatorname{aux}_i^{(j)} \leftarrow \operatorname{aux}_i$ 

Then, follow the same logic of Lines 9-16 above to obtain shares and commitments from  $\{\texttt{transcript}^{(\ell|new)}\}_{\ell\in T^*}$ 

<u>THE PROTOCOL.</u> With above setup and ingredients, we present our DCRG scheme in Algorithm 5, and describe its execution flow in the following.

- Each 𝒫<sub>i</sub> ∈ M samples two secrets *s* and *ŝ*, and it invokes PVSS.Deal[*i*|old] (with threshold *t*) and PVSS.Deal[*i*|new] (with threshold *t̃*) to share them with M and M̃, respectively. All parties in both M and M̃ can obtain the corresponding auxiliary information.
- After obtaining transcript<sup>(i|old)</sup> and transcript<sup>(i|new)</sup> from PVSS.Deal[i|old] and PVSS.Deal[i|new], respectively, *P<sub>i</sub>* ∈ M disperses them (as a single message) to M via PD[i]. Every *P<sub>j</sub>* ∈ M participates these PD instances and stores the associate information.
- All parties in  $\mathbb{M}$  invoke Dumbo-MVBA to agree on T, which is a set of 2f + 1 finished PD instances. Then, they call Coin.Get to select a set of  $\kappa$  indexes T', such that all indexes in T' have been included in T. Then, parties in  $\mathbb{M}$  recast both transcript<sup>( $\ell$ |old)</sup> and transcript<sup>( $\ell$ |new)</sup> for all  $\ell \in T$  to all parities in  $\mathbb{M} \cup \widetilde{\mathbb{M}}$ .
- After receiving transcript<sup>(ℓ|old)</sup> and transcript<sup>(ℓ|new)</sup> from RC[ℓ], each party checks whether they are valid and if they contain the same v<sub>0</sub><sup>(ℓ)</sup> (which ensures the two transcripts sharing the same pair of s<sup>(ℓ)</sup> and ŝ<sup>(ℓ)</sup>). Let T\* = {ℓ} such that transcript<sup>(ℓ|old)</sup> and transcript<sup>(ℓ|new)</sup> passed the check. Each 𝒫<sub>i</sub> ∈ M (resp. 𝒫<sub>i</sub> ∈ M) processes {transcript<sup>(ℓ|old)</sup>}<sub>ℓ∈T\*</sub> (resp. {transcript<sup>(ℓ|new)</sup>}<sub>ℓ∈T\*</sub>) to obtain their outputs, as specified in Lines 12-16.

We remark that the committee  $\mathbb{M}$  is assumed to have a setup for an (n,t) threshold cryptosystem, which enables threshold common coin and subsequently enables Dumbo-MVBA. So the new committee  $\mathbb{M}$  should also set up its (n,t) threshold cryptosystem after DCRG, to support a future execution with a newer committee. Therefore, besides the operations in Algorithm 5, all parties shall run an ADKR in parallel, to accomplish the setup <sup>10</sup>.

#### 5.3 Analysis

We establish the following results about our ADPSS protocol.

**Theorem 2.** Using DCRG in Algorithm 5 to instantiate the generic ADPSS in Algorithm 4 gives a secure ADPSS with  $O(\lambda \kappa (n + \tilde{n})^2)$  communication complexity and O(1) rounds.

**Communication Complexity.** In the DCRG protocol, parties in  $\mathbb{M}$  invoke ADKR where the input size of each PD is  $O(\lambda(n+\tilde{n}))$  instead of  $O(\lambda\tilde{n})$ , resulting in a communication complexity of  $O(\kappa\lambda n^2 + \kappa\lambda n \cdot \tilde{n})$ . Multicasting COMMIT messages cost  $O(\kappa \lambda n \cdot \tilde{n})$ . For parties in  $\widetilde{\mathbb{M}}$ , the DCRG costs  $O(\lambda \tilde{n}^2 + \kappa \lambda n \cdot \tilde{n})$  bits.

The subsequent *n*-to-*n*(or *n*-to- $\tilde{n}$ ) multicast messages each have a size of  $O(\lambda)$  for parties in  $\mathbb{M}$ , contributing an additional  $O(\lambda n \cdot \tilde{n} + \lambda n^2)$  in communication costs. Therefore, the total communication complexity of ADPSS is  $O(\kappa \lambda n^2 + \kappa \lambda n \cdot \tilde{n} + \lambda \tilde{n}^2)$ .

**Computation Complexity.** Each party in  $\mathbb{M}$  performs  $O(\kappa n + \tilde{n})$  exponentiations when invoking *DCRG*. Then verifying shares in the REC messages and interpolating m + r (and  $\hat{m} + \hat{r}$ ) incurs a computation cost of O(n) per party. For each party in  $\widetilde{\mathbb{M}}$ , it costs  $O(\kappa \tilde{n})$  in the DCRG protocol, followed by  $O(\tilde{n})$  exponentiations to compute  $\operatorname{cm}'_j$  for all  $\mathcal{P}_j \in \widetilde{\mathbb{M}}$  Therefore, the overall computation complexity of ADPSS is  $O(\kappa n + \kappa \tilde{n})$  per party.

**Security Analysis.** Hereunder, we prove the properties of termination, correctness, and secrecy, one-by-one.

**Lemma 8.** The ADPSS protocol (instantiating Algorithm 4 with DCRG from Algorithm 5) satisfies the termination property.

*Proof.* First, following the proof about the termination of ADKR in Lemma 4, we can show that every honest party in  $\mathbb{M} \cup \widetilde{\mathbb{M}}$  can terminate in DCRG.

Then, after receiving output  $(r_i, \hat{r}_i, \{v_j\}_{j \in [n]})$ , every honest  $\mathcal{P}_i \in \mathbb{M}$  can multicast  $(m_i + r_i, \hat{m}_i + \hat{r}_i)$ , which satisfies  $\operatorname{cm}_i \cdot v_i = g^{m_i + r_i} h^{\hat{m}_i + \hat{r}_i}$  (condition in Line 5 of Algorithm 4). Therefore, all honest nodes in  $\mathbb{M}$  can obtain *S* such that  $|S| \ge 2f + 1$  (condition in Line 7), so they can recover  $(m + r, \hat{m} + \hat{r})$  (Line 8) and multicast it to all parties in  $\widetilde{\mathbb{M}}$ ; then, they can terminate.

Regarding each party  $\widehat{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ , it can receive outputs from DCRG and  $(m + r, \hat{m} + \hat{r})$  from at least f + 1 parties in  $\mathbb{M}$  (since all honest parties in  $\mathbb{M}$  send the same pair). Therefore, each  $\widetilde{\mathcal{P}}_i \widetilde{\mathbb{M}}$  can execute the code after Line 12 and then terminate.

**Lemma 9.** The ADPSS protocol (instantiating Algorithm 4 with DCRG from Algorithm 5) satisfies the correctness property.

*Proof.* First, following the proof about the key validity of ADKR in Lemma 5 and Lemma 6, we can show that all parties in  $\mathbb{M}$  can obtain a vector of Pedersen commitment  $\{v_j\}_{j\in[n]}$ , such that there exits two *t*-degree polynomials  $\phi$  and  $\hat{\phi}$ , and  $v_j^{\text{old}} = g^{\phi(j)}h^{\hat{\phi}(j)}$ . And every honest  $\mathcal{P}_i \in \mathbb{M}$  obtains  $(\phi(i), \hat{\phi}(i))$ . Similarly, we can show all parties in  $\widetilde{\mathbb{M}}$  can obtain a vector of Pedersen commitment  $\{v_j\}_{j\in[n]}$ , such that there exits two *t*-degree polynomials  $\phi'$  and  $\hat{\phi}'$ , and  $v_j^{\text{old}} = g^{\phi(j)}h^{\hat{\phi}(j)}$ . And every honest  $\{v_j\}_{j\in[n]}$ , such that there exits two *t*-degree polynomials  $\phi'$  and  $\hat{\phi}'$ , and  $v_j^{\text{old}} = g^{\phi(j)}h^{\hat{\phi}(j)}$ . And every honest  $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$  obtains  $(\phi'(i), \hat{\phi}'(i))$ . Moreover, since we require  $v_0^{(\ell|\text{old})} = v_0^{(\ell|\text{new})}$  for all  $\ell \in T^*$ ,  $v_0 = \prod_{\ell \in T^*} v_0^{(\ell|\text{old})} = v_0'$ . So  $g^{\phi(0)}h^{\hat{\phi}(0)} = g^{\phi'(0)}h^{\hat{\phi}'(0)}$ . Due to the binding property of

<sup>&</sup>lt;sup>10</sup>It is not hard to see the ADKR and DCRG can be batched: the network can use the same PD, Dumbo-MVBA, and Coin to disperse, agree, and select PVSS transcripts for DCRG and ADKR.

Pedersen commitment (implied by the dLog assumption), we have  $\phi(0) = \phi'(0)$ , and  $\hat{\phi}(0) = \hat{\phi}'(0)$ . I.e., the parties in  $\mathbb{M}$  and the parties in  $\widetilde{\mathbb{M}}$  have secret shares of the same pair  $(r, \hat{r})$ .

Next, since every  $(m_i + r_i, \hat{m}_i + \hat{r}_i)$  needs to satisfy  $\operatorname{cm}_j \cdot v_j = g^{m_i + r_i} h^{\hat{m}_i + \hat{r}_i}$  to be included in the set *S*, it is easy to see every  $\mathcal{P}_i \in \mathbb{M}$  can reconstruct the same  $(m + r, \hat{m} + \hat{r})$ . Then, the updated secret share  $m'_i = (m + r) - r'_i$  is F(i) for a  $\tilde{t}$ degree polynomial  $F(x) = (m + r) - \phi'(x)$ . So,  $F(0) = m + r - \phi'(0) = m$ , which means  $\{m'_i\}_{i \in [\tilde{n}]}$  are  $(\tilde{n}, \tilde{t})$  Shamir shares of *m*. Following a similar argument, we can show  $\{\hat{m}'_i\}_{i \in [\tilde{n}]}$ are  $(\tilde{n}, \tilde{t})$  Shamir shares of  $\hat{m}$ .

# **Lemma 10.** The ADPSS protocol (instantiating Algorithm 4 with DCRG from Algorithm 5) satisfies the secrecy property.

*Proof.* We prove the secrecy of our ADPSS by building a PPT simulator S, which on inputs a Pedersen commitment cm can simulate an execution of ADPSS where *C* is the commitment to the secret cm, such that any PPT adversary  $\mathcal{A}$  that corrupts up to  $\lfloor \frac{n-1}{3} \rfloor$  nodes in  $\mathbb{M}$  and  $\lfloor \frac{\tilde{n}-1}{3} \rfloor$  nodes in  $\widetilde{\mathbb{M}}$  cannot distinguish the simulated execution and a real execution where the commitment to the secret is cm.

We describe the simulator S as follows.

**Notations:** We denote the set of parties controlled by the adversary  $\mathcal{A}$  in  $\mathbb{M}$  and  $\widetilde{\mathbb{M}}$  as B and  $\widetilde{B}$ , respectively. The sets of honest party in  $\mathbb{M}$  and  $\widetilde{\mathbb{M}}$  are represented as G and  $\widetilde{G}$ . Without loss of generality, let  $B = \{\mathcal{P}_1, ..., \mathcal{P}_f\}, G = \{\mathcal{P}_{f+1}, ..., \mathcal{P}_n\}$  (with  $\widetilde{B}$  and  $\widetilde{G}$  defined similarly).

**Input:** a Pedersen commitment cm, which is a group element in  $\mathbb{G}$ .

**Simulation:** Firstly, sample  $i^* \leftarrow [n]$  which represents a party in  $\mathbb{M}$ . S generates encryption key pairs  $(ek_i, dk_i)$  and the signature key pairs for all parties in  $\widetilde{\mathbb{M}}$ . After that, S simulates an execution as follows.

Initial configuration in  $\mathbb{M}$ : For all  $i \in [t]$ , S uniformly samples  $m_i$  and  $\hat{m}_i$ , and computes  $\operatorname{cm}_i = g^{m_i} \cdot h^{\hat{m}_i}$ . For all  $i \in [t+1,n]$ , interpolate  $\operatorname{cm}_i = \prod_{j \in [0,t]} \operatorname{cm}_j^{\lambda_j(i)}$ , where  $\operatorname{cm}_0 = \operatorname{cm}$  and  $\lambda_j(x) = \prod_{k \in [0,t], k \neq j} \frac{j-x}{j-k}$ .

Simulating DCRG: For  $\mathcal{P}_{i^*}$ ,  $\mathcal{S}$  simulates its actions by the following strategy:

• For simulating PVSS.Deal[ $i^*$ ], S samples *t*-degree polynomials  $\varphi$  and  $\hat{\varphi}$ , and  $\tilde{t}$ -degree polynomials  $\varphi'$  and  $\hat{\varphi}'$ , such that  $\varphi(0) = \varphi'(0)$  and  $\hat{\varphi}(0) = \hat{\varphi}'(0)$ . Then, S computes  $v_0^{(i^*)} = \frac{g^{\varphi(0)} \cdot h^{\varphi(0)}}{cm_0}$ ,

$$v_i^{(i^*)} = \frac{g^{\varphi(i)} \cdot h^{\hat{\varphi}(i)}}{\mathsf{cm}_i}, \forall i \in [n],$$
(2)

and  $v_i^{(i^*)'} = \frac{g^{\varphi'(i)} \cdot h^{\varphi'(i)}}{cm_i}$  for all  $i \in [\tilde{n}]$ . Then, S sends  $(v_i^{(i^*)}, (m_i + \varphi(i), \hat{m}_i + \hat{\varphi}(i)))$  to each  $\mathcal{P}_i \in \mathbb{M}$ , and  $(v_i^{(i^*)'}, (m_i + \varphi'(i), \hat{m}_i + \hat{\varphi}'(i)))$  to each  $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ , on the behalf of  $\mathcal{P}_{i^*}$  S simply sends  $(v_i^{(i^*)}, \bot)$  or  $(v_i^{(i^*)'}, \bot)$  to honest nodes. On the behalf of each honest  $\mathcal{P}_i \in G$  and  $\widetilde{\mathcal{P}}_i \in \widetilde{G}$ , S signs each  $v_i^{(i^*)}$  and  $v_i^{(i^*)'}$ , and sends the signature to  $\mathcal{P}_{i^*}$  on the behalf of honest  $\mathcal{P}_i$  and  $\widetilde{\mathcal{P}}_i$ .

Finally, *S* can collects valid signatures for  $\{v_i^{(i^*)}\}_{i\in S}$  and  $\{v_i^{(i^*)'}\}_{i\in S'}$ , where  $S \in [n]$  with |S| = 2f + 1, and  $S' \in [\tilde{n}]$  with  $|S'| = 2\tilde{f} + 1$ . For every  $j \notin S$  (resp  $j \notin S'$ ), if  $j \in B$  (resp.  $j \in B'$ ), *S* honestly uses VE to encrypt  $(m_j + \varphi(j), \hat{m}_j + \hat{\varphi}(j))$  (resp.  $(m'_j + \varphi'(j), \hat{m}'_j + \hat{\varphi}'(j))$ ); Otherwise, *S* encrypts a random value under  $ek_j$  (resp.  $\tilde{ek}_j$ ) and simulates a proof. By doing so, *S* can obtain transcript  $(i^*|\text{old})$  and transcript  $(i^*|\text{new})$ .

Regarding other honest nodes, S honestly executes the protocol on the behalf of them in PVSS.Deal instances, except their responses to the messages from  $\mathcal{P}_{i^*}$  (which we have specified above).

- For all operations until Dumbo-MVBA outputs, S acts on the behalf of each honest nodes by executing the protocol. When Dumbo-MVBA outputs T', if (i\*, ·) is not included in T', S shall **abort** the current simulated execution, sample a new i\* ← G, rewind A to the begining of the simulation, and **restart** a new simulated execution. If (i\*, ·) is included in T', S continues to execute the protocol on the behalf of all honest nodes, until Coin.Get.
- For Coin.Get, recall that the coin value is RO(ThldSig). S computes the unique threshold signature Thld on the default message of this execution (since S knows the secret keys of all honest nodes), and *programs* RO such that  $i^* \in RO(ThldSig)$ .
- For operations after GetCoin, S acts on the behalf of each honest nodes by executing the protocol. In particular, on the behalf of each honest P<sub>i</sub> ∈ G and P̃<sub>i</sub> ∈ G, S obtains (r<sub>i</sub><sup>(ℓ)</sup>, r̂<sub>i</sub><sup>(ℓ)</sup>) and (r<sub>i</sub><sup>(ℓ)'</sup>, r̂<sub>i</sub><sup>(ℓ)'</sup>) for all ℓ ∈ T<sup>\*</sup>, ℓ ≠ i<sup>\*</sup>; It skips the deliver operation for ℓ = i<sup>\*</sup>.

Simulating other operations in ADPSS: On the behalf of all honest nodes, S executes the rest of the protocol honestly, except for Line 3 of Algorithm 4. Since S does not have the secret share  $(r_i, \hat{r}_i)$  of  $\mathcal{P}_i \in G$ , it multicasts

$$(\mathsf{sh}_{i} = \varphi(i) + \sum_{\ell \in T^{*}, \ell \neq i^{*}} r_{i}^{(\ell)}, \hat{\mathsf{sh}}_{i} = \hat{\varphi}(i) + \sum_{\ell \in T^{*}, \ell \neq i^{*}} \hat{r}_{i}^{(\ell)}).$$
(3)

Recall Eq.2 where S computes  $v_i^{(i^*)} = \frac{g^{\phi(i)} \cdot h^{\hat{\phi}(i)}}{\mathsf{cm}_i}$ . Therefore,  $(\mathsf{sh}_i, \hat{\mathsf{sh}}_i)$  can pass the check, since

$$\operatorname{cm}_{i} \cdot v_{i} = \operatorname{cm}_{i} \cdot v_{i}^{(i^{*})} \cdot \prod_{\ell \in T^{*}, \ell \neq i^{*}} v_{i}^{(\ell)}$$

$$= g^{\varphi(i) + \sum_{\ell \in T^{*}, \ell \neq i^{*}} r_{i}^{(\ell)}} \cdot h^{\hat{\varphi}(i) + \sum_{\ell \in T^{*}, \ell \neq i^{*}} \hat{r}_{i}^{(\ell)}} = g^{\operatorname{sh}_{i}} h^{\hat{sh}_{i}}.$$

$$(4)$$

Now, we turn to prove that an adversary  $\mathcal{A}$  cannot distinguish a real execution and a simulated execution provided by  $\mathcal{S}$ . To argue this fact, we consider the following hybrids.

**Hybrid 0:** It is identical to a real execution, where the simulator honestly executes the protocol on the behalf of all honest nodes.

**Hybrid 1:** It is almost identical to **Hybrid 0**, except that when during DCRG, if Coin.Get returns  $T^*$  which does not contain an honest node  $i' \in G$ , the simulator aborts the current execution, rewidns the adversary  $\mathcal{A}$  to the begining, and restarts the simulation.

**Hybrid 2:** It is almost identical to **Hybrid 1**, except that the simulator randomly selects  $i^* \leftarrow G$  in the beginning.

**Hybrid 3:** It is almost identical to **Hybrid 2**, except that during PVSS.Deal[ $i^*$ |old] and PVSS.Deal[ $i^*$ |new], when  $\mathcal{P}_{i^*}$  needs to encrypt shares for corrupted parties, the simulator encrypts random values and generates a simulated proof.

**Hybrid 4:** It is almost identical to **Hybrid 3**, except that when during DCRG, if Dumbo-MVBA outputs T' and  $(i^*, \cdot) \notin T'$ , the simulator aborts the current execution aborts, rewinds the adversary  $\mathcal{A}$  to the begining, and restarts the simulated execution with a freshly sampled  $i^* \leftarrow G$ .

**Hybrid 5:** It is almost identical to **Hybrid 4**, except that during DCRG, after Dumbo-MVBA outputs T', the simulator programs RO at ThldSig such that  $i^* \in \text{RO}(\text{ThldSig})$ .

Hybrid 6: It is the simulated execution.

First, from  $\mathcal{A}$ 's perspective, the difference between Hybrid 0 and Hybrid 1 is that in Hybrid 0, the output of Coin.Get is a uniformly sampled subset of  $\kappa$  indexes in T', while in Hybrid 1, the output must include at least one honest  $i' \in G$ . Since the probability that  $\kappa$  randomly sampled indexes do not include any  $i' \in G$  is negligible, the statistical distance between the output distributions of Coin.Get in Hybrid 0 and Hybrid 1 is negligible. Thus,  $\mathcal{A}$  cannot distinguish between the two hybrids with a non-negligible advantage. Additionally, as the probability that the simulator needs to rewind  $\mathcal{A}$  and restart the simulation is negligible, the simulator's expected running time is polynomial in the security parameter.

Next, Hybrids 1 and 2 are identical from  $\mathcal{A}$ 's perspective.

Then, by the IND-CPA security of the underlying VE,  $\mathcal{A}$  cannot distinguish between Hybrids 2 and 3 except with negligible probability.

Furthermore, since the choice of  $i^*$  is independent of  $\mathcal{A}$ 's view, Hybrids 3 and 4 are indistinguishable from  $\mathcal{A}$ 's perspective. As T' contains at least f + 1 honest nodes and  $i^*$  is uniformly chosen from G, the probability that  $(i^*, \cdot) \notin T'$  is at most  $\frac{f}{2f+1}$ . Thus, the simulator is expected to rewind  $\mathcal{A}$  approximately twice, keeping the simulator's running time polynomial in the security parameter.

Next, the unforgeability of the unique threshold signature ensures that  $\mathcal{A}$  cannot obtain ThldSig before honest participants invoke Coin.Get(). Consequently, the probability that

 $\mathcal{A}$  queries RO with ThldSig before RO is programmed is negligible. Hence,  $\mathcal{A}$  cannot distinguish between Hybrids 4 and 5 except with negligible probability.

Finally, Hybrids 5 and 6 are indistinguishable from  $\mathcal{A}$ 's perspective. Therefore,  $\mathcal{A}$  cannot distinguish between the real execution and the simulated execution except with negligible probability.

#### 6 Implementation and Evaluations

This section will demonstrate the concrete efficiency and scalability of our ADKR protocol via large-scale experiments.

### 6.1 Instantiations and Test Environment

**Implementation**. We implemented our high-threshold ADKR protocol, along with two ADKRs naturally adapted from the high-threshold ADKG protocols in [26] and [24] through using the existing threshold common coins generated by the old committee members. In addition, we implemented two versions of the protocol adapted from [26]. Note that the ACSS scheme in [26] is based a VE scheme. In the first version (denoted as "adapted DYX+22-VE"), we replaced the original VE scheme with the VE for Pedersen commitments proposed in [25]. In the second version (denoted as "adapted DYX+22-ACSS"), we replaced the entire ACSS scheme in [26] with DXT+ ACSS (with our improved PVSS generation in Algorithm 1). Both versions are implemented and evaluated to show each portion of the improvement.

All implementations are written in Python 3.8, using the identical cryptographic library and security parameters across. Concurrency is managed using the gevent library. We implemented all protocols over BLS12-381 curve, so they can support pairing-based threshold cryptosystems. We used the BLS12-381 implementation from Zcash and its Python wrapper as described in [26]. we assume that the parties in the old configuration have established a high-threshold cryptosystem capable of performing non-interactive threshold BLS signatures, which can be set up during the last ADKR. Furthermore, in the implementation of VE for Pedersen commitment, each node signs the commitment value using ECDSA algorithm.

**Choices of other parameters**. While the system scales from n = 127 to n = 256 nodes, we consider two choices of subcommittee size  $\kappa$ , ensuring the selected sub-committee contains a dispersal from an honest sender with probabilities  $p_1 = 1 - 10^{-8}$  and  $p_2 = 1 - 10^{-10}$ , respectively. Intuitively, if the system reconfigures its threshold key every 10 minutes, it is expected to fail approximately once every 1902 years under  $p_1$  and even longer for  $p_2$ .

We show an optimization scheme for the good case where most nodes are honest and the network latency is small: When each party executes the dealing protocol to generate a PVSS transcript, it waits for a short time  $\delta$  after distributing the



Figure 4: Performance in comparison with the state-of-the-art asynchronous protocols (in the WAN setting).

shares, instead of proceeding immediately after receiving 2f + 1 signatures. Here  $\delta$  is set to 1 sec for n = 127 and 196, and 2 sec for n = 256. We also adopted another good-case optimization from [24], which attempts to wait for KEY messages from all parties during the key derivation phase. This optimization can be applied to all of the protocols. We organized an additional set of experiments in Fig. 4d to compare all protocols under this optimization.

Setup of test environment. Our experimental evaluations include both wide-area network (WAN) and local evaluations. All evaluations were conducted on Amazon's EC2 c5.xlarge instances, equipped with 4 virtual CPUs and 8GB of RAM. In the WAN setting, the number n of nodes ranges from 127 to 256. The nodes were evenly distributed across 16 geographical regions on 5 continents. The local evaluation was conducted on a single c5.xlarge instance, where we measured all computation time for a single node during one execution.

#### 6.2 Evaluations in Large-scale WAN

**Running time**. We presented the evaluation of the running time of the ADKR protocols. For DYX+22, we did not measure the running time for node sizes greater than 127, as it was clearly much slower than the other two protocols.

As shown in Figure 4a, our ADKR protocol demonstrated a clear performance advantage than the other two protocols. For n = 127, the running time of our ADKR protocol is reduced by more than 90% compared to the adapted DYX+22-VE and about 75% compared to the adapted DYX+22-ACSS. For larger node sizes (n = 196 and 256), when the failure probability is  $10^{-8}$ , the running time of our protocol is only 71.5% and 57.7% of that of adapted DXK+23, respectively. Similarly, When the failure probability is  $10^{-10}$ , it is 78.8% and 66.2% of that of adapted DXK+23, respectively.

**Bandwidth usage**. We evaluated the bandwidth usage of the protocols by measuring the number of bytes sent by each node during a single execution. As shown in Figure 4b, for n = 127, our protocol (with  $p_1 = 1 - 10^{-8}$ ) consumes approximately 24.7% and 48.6% of the bandwidth used by the adapted DYX+22-VE and adapted DYX+22-ACSS, respectively. Similarly, with  $p_2 = 1 - 10^{-10}$ , our protocol uses about

26.2% and 51.5% of their bandwidth.

Our ADKR protocol maintains a linear growth in bandwidth usage, while the adapted DXK+23 protocol exhibits a quadratic growth trend.<sup>11</sup> As the node scales increase from 127 to 256, the bandwidth usage of our protocol with  $p_1 = 1 - 10^{-8}$  is approximately 71.1% to 39.8% of that of the adapted DXK+22. With  $p_2 = 1 - 10^{-10}$ , it is 61.7% to 41.7% of that of the adapted DXK+22.

**Running time under optimistic conditions**. We also measured the running time of the protocols with optimizations for some good cases, such as the network latency being small and almost all honest parties being honest.

First, during the process of generating sharing transcript, we introduced a waiting period of length  $\delta$  in both our protocol and the adapted DYX+22-ACSS protocol to collect additional signatures. This optimization significantly improved performance, as shown in Figure 4c. Particularly, compared to the adapted DXK+23 protocol, our protocol achieves a running time reduction of 40% to 54% with  $p_1 = 1 - 10^{-8}$ , for n = 127-256 nodes. Similarly, with  $p_2 = 1 - 10^{-10}$ , the running time is reduced by 36.4% to 50.8%, for n = 127-256.

In another good case where the interpolations of f public key shares are skipped, our performance advantage became even more pronounced. As shown in Figure 4d, the running time of our protocol is only about 18.2% and 18.8% of that of the adapted DYX+22-ACSS, respectively, with  $p_1$  and  $p_2$ . When the number of nodes increases from 127 to 256, our protocol is 41.4% to 68% faster than adapted DXK+23 with  $p_1$ , and 43.2% to 65.3% faster with  $p_2$ .

#### 6.3 Evaluations of Local Computing Cost

As shown in Figure 5, we also compared the local CPU time for all evaluated protocols, in order to estimate the computing cost under even larger network scales. This result aligns with our computational complexity analysis, demonstrating that our protocol exhibits linear growth, in contrast to the quadratic

<sup>&</sup>lt;sup>11</sup>Careful readers may notice a discrepancy between the bandwidth usage of the adapted DXK+23 protocol in our measurements and those reported in [24]. This difference arises because [24] implemented RBC with a fast path in a "very good case", while we implemented RBC from [17] (which has no good-case path but causes less communication in the bad case).

behavior observed in protocols adapted from classic ADKG. Particularly, if the failure probability is  $10^{-8}$ , the computing time of our design is 410.88 seconds for n = 2041, achieving a reduction of 68.2% compared to that of DXK+23.



Figure 5: Computation time (without PK interpolations).

## 7 Preliminary

**Non-interactive Threshold Signature** (TSIG). Given a (n, t) threshold signature scheme among a set  $\mathbb{M} := \{\mathcal{P}_i\}_{i \in [n]}$  of n parties, each party  $\mathcal{P}_i \in \mathbb{M}$  has a private function denoted by SignShare $(sk_i, \cdot)$  to produce its partial signature, and there are also three public functions VerifyShare, Combine and Verify, which can respectively verify the partial signature, combine partial signatures into a full signature, and validate the full signature. Note that t represents the reconstruction threshold (i.e. at least t + 1 valid partial signatures are required to compute a valid full signature). Throughout the paper, we require a so-called *high-threshold* threshold scheme, requiring t = n - f - 1 where f is the maximal number of malicious parties. More formally, a non-interactive (n, t)-threshold signature scheme TSIG consists of hereunder algorithms/protocols:

- TSIG.Setup $(\lambda, t, n) \rightarrow (pk, sk)$ . Given t, n and the cryptographic security parameter  $\lambda$ , the algorithm/protocol generates the public keys pk and a vector of secret key shares sk =  $(s_1, \dots, s_n)$ , where pk is published and  $s_i$  is exclusively obtained by  $\mathcal{P}_i$ .
- TSIG.SignShare(s<sub>i</sub>, m) → ρ<sub>i</sub>. On input a message m and a secret key share s<sub>i</sub>, this algorithm outputs a partial signature share ρ<sub>i</sub> for the message m.
- TSIG.VerifyShare(m, (i, ρ<sub>i</sub>)) → 0/1. Given a message m, a partial signature ρ<sub>i</sub> and an index i (along with the implicit input of public keys pk), this algorithm outputs 1 (accept) or 0 (reject).
- TSIG.Combine(m, {(i, ρ<sub>i</sub>)}<sub>i∈S</sub>) → σ/⊥. Given the implicit input of public keys pk, a message m, and a set of indexed partial signatures {(i, ρ<sub>i</sub>)}<sub>i∈S</sub> with |S| > t and S ⊂ M, this algorithm outputs a full signature σ for message m (or a special symbol ⊥).
- TSIG.Verify(m, σ) → 0/1. Given a message m and a combined full signature σ (along with the implicit input

of public keys pk), this algorithms outputs 1 (accept) or 0 (reject).

We require a (n,t)-TSIG scheme to satisfy *correctness*, *robustness* and *unforgeability*:

- *Correctness*. The partial signatures and full signatures that are correctly computed can be verified.
- *Robustness*. Combining any *t* + 1 valid partial signatures (even if some of them are computed by the adversary) produces a valid full signature, with all but negligible probability.
- Unforgeability. The unforgeability can be defined by a threshold and adaptive version of Existential UnForgeability under Chosen Message Attack game [47]. Intuitively, the unforgeability ensures that no P.P.T. adversary  $\mathcal{A}$  that corrupts f parties can produce a valid full signature except with negligible probability in  $\lambda$ , unless  $\mathcal{A}$  queries sufficient partial signatures from at least t f + 1 honest parties.

*Instantiations*. According to the granted setup, we choose the best-performing instantiation of high-threshold signature:

- *DKG setup (or a trusted dealer).* In such scenarios, any high-threshold signature scheme supported by the DKG setup can be adopted. For instance, if the DKG setup is for dLog over a pairing-friendly elliptic curve, the threshold BLS signature scheme can be used. Noticeably, this is the case of high-threshold ADKR, where the old committee participants indeed share an already-established high-threshold cryptosystem.
- *PKI setup only*. In the case, we can adopt a threshold signature scheme with silent setup. This could be the case in a general ADKR problem, where the old committee members do not have a granted DKG setup for high-threshold signature, as their established threshold cryptosystem has a "low" reconstruction threshold t = f < n f 1.

**Threshold common coin** (Coin). Given a (n,t)-threshold common coin scheme among a set  $\mathbb{M} := \{\mathcal{P}_i\}_{i \in [n]}$  of *n* parties, each party  $\mathcal{P}_i \in \mathbb{M}$  is provided with an interface Coin.Get $(\cdot)$ to invoke the protocol with taking  $x \in \{0,1\}^*$  as input, and if *t* honest parties in  $\mathbb{M}$  invoke the protocol with the same input *x*, all honest parties will return the same random value  $r \in \{0,1\}^{\lambda}$  for *x*. We require a (n,t)-threshold Coin scheme to satisfy *termination*, *consistency* and *pseudorandomness*, with all but negligible probability:

• *Termination*. If *t* + 1 honest parties invoke Coin.Get(*x*) using the same input *x*, then any honest party invoking Coin.Get(*x*) will obtain some output *r* from Coin.Get(*x*), despite the influence of the adversary.

- *Consistency*. For any two honest parties, if they obtain *r* and *r'* from Coin.Get(*x*) respectively, then *r* = *r'*.
- *Pseudorandomness*. For any PPT adversary  $\mathcal{A}$  who can corrupt up to f parties, let view denote  $\mathcal{A}$ 's view (i.e. all internal states of corrupted parties and all protocol messages that have been generated) before t f honest parties invoke Coin.Get(x), then  $|\Pr[r' = r : r \leftarrow \text{Coin.Get}(x) \text{ and } r' \leftarrow \mathcal{A}(\text{view})] \frac{1}{2\lambda}| \le \varepsilon(\lambda)$ .

*Instantiation.* In the setting of ADKR, it is straightforward to let the old committee participants leverage their established threshold cryptosystem to implement Coin. For instance, if the old committee's threshold cryptosystem is based on dLog over a pairing-friendly elliptic curve, Coin can be realized from threshold BLS signature in the random oracle model.

Multi-valued validated asynchronous Byzantine agreement (MVBA). In an MVBA protocol executed among a set  $\mathbb{M}$  of *n* parties, the honest participants reach an agreement on an output *v* satisfying a predefined global predicate *Q* s.t. Q(v) = 1. Particularly, an MVBA satisfies the next properties of *termination*, *agreement* and *external-validity*, except with negligible probability:

- *Termination*. If all honest parties in M input some values satisfying Q, then all honest parties in M will output.
- Agreement. If any two honest parties in  $\mathbb{M}$  output *v* and *v*', respectively, then v = v';
- *External-Validity:* If an honest party in  $\mathbb{M}$  outputs a value v, then v is valid w.r.t. Q, i.e., Q(v) = 1.

Instantiation. For efficiency consideration, we instantiate MVBA from Dumbo-MVBA [48] in the paper, since given a high-threshold signature scheme established across  $\mathbb{M}$ , the construction can attain a communication cost of  $O(Ln + \lambda n^2)$  where *L* represents the bit length of the input value, which is necessary to make our ADKR design preserve a collective quadratic communication overhead.

**Verifiable encryption for Pedersen commitment.** We include a full syntax of VE as follows.

- VE.Setup(1<sup>λ</sup>, Cm) → pp<sub>VE</sub>. Input security parameter λ and a commitment scheme Cm, the algorithm outputs public parameter pp<sub>VE</sub>.
- VE.KeyGen $(pp_{VE}) \rightarrow (\{ek\}, \{dk\})$ . Input public parameters, the algorithm outputs the encryption keys ek and decryption keys dk of the encryption scheme.
- VE.EncAndProve $(pp_{VE}, ek, s, v, w) \rightarrow (c, \pi_{VE})$ .Input message *s*, commitment *v* and witness *w*, where  $v, w \leftarrow$ Cm. COMMIT(*s*). The algorithm outputs the encryption *c* of the tuple  $(s, \pi = \text{Cm.Open}(v, s, w))$  and a correct encryption NIZK proof  $\pi_{VE}$ .

- VE.Dec(dk,c) → s,π. Input the ciphertext c and a decryption key dk, the algorithm outputs a decryption of c using dk.
- VE.Verify $(pp_{VE}, ek, v, c, \pi_{VE}) \rightarrow 0/1$ . Input public parameters  $pp_{VE}$ , encryption key ek, commitment v, ciphertext c and its proof  $\pi_{VE}$ . If  $\pi_{VE}$  is a valid proof that  $\alpha, \pi$  exist, and  $\alpha, \pi = \text{VE.Dec}(dk, c)$  and Cm.Verify $(v, \alpha, \pi) = 1$ , the algorithm outputs 1.
- VE.bEncProve(*pp*<sub>VE</sub>, *I*, {*ek*}, *s*, *v*, *w*) → (*c*, π<sub>VE</sub>). Input a set *I*, a set of encryption keys *ek*, a vector *s* of messages, their commitments *v*, corresponding witness *w*, the algorithm outputs encryptions *c* for each *s<sub>i</sub>* ∈ *s*, along with a NIZK proof π<sub>VE</sub> that satisfy VE.bVerify.
- VE.bVerify(*pp*<sub>VE</sub>, *I*, {*ek*}, *v*, *c*, π<sub>VE</sub>) → 0/1. Input parameters *pp*<sub>VE</sub>, a set of encryption keys *ek*, commitments *v*, encryptions *c* and a proof π<sub>VE</sub> the algorithm outputs 1 if π<sub>VE</sub> is a valid proof that, for each *i* ∈ *I* there exists (α<sub>i</sub>, π<sub>i</sub>) such that α<sub>i</sub>, π<sub>i</sub> = VE.Dec(*dk*<sub>i</sub>, *c*<sub>i</sub>) and PC.Verify(*v*, α<sub>i</sub>, π<sub>i</sub>) = 1.

A VE scheme satisfies the standard IND-CPA security. In addition, in the random oracle model, it allows a simulator (which can program random oracles) to simulate a proof  $\pi_{VE}$  without knowing the plaintext and encryption randomness.

Asynchronous provable dispersal. An asynchronous provable dispersal broadcast (APDB) protocol (adapted from [48]) allows a designated sender to disperse a message to a set  $\mathbb{M}$ of parties, such that the dispersed message can be recovered by the message's designated receiving parties.

Syntactically, APDB consists of two phases—provable dispersal (PD) and recovery (RC):

- The PD phase is executed by a designated sender  $\mathcal{P}_s$  and a set of parties denoted by  $\mathbb{M}$ . Here  $\mathcal{P}_s$  inputs a messages m, and aims to split the message into  $|\mathbb{M}|$  encoded fragments and disperses each fragment to the corresponding party in  $\mathbb{M}$ . From the PD phase, each party in  $\mathbb{M}$  obtains two outputs *store* and *lock*.
- The RC phase is executed by the parties in M and a set of receiving parties denoted by M'. The honest parties in M take *store* and *lock* as their input (if they have obtained *store* and *lock* from the PD phase), and aim to help M' to recover some common message.

An APDB protocol with identifier ID satisfies *termination* and *recast-ability*, except with negligible probability:

• *Termination*. If the sender  $\mathcal{P}_s$  is honest, then all honest parties in  $\mathbb{M}$  will output *store* and valid *lock* from PD in a constant number of asynchronous rounds, where the valid *lock* satisfies ValidateLock(ID, *lock*) = 1;

Algorithm 6 PD protocol, with identifier ID and sender  $\mathcal{P}_s$ /\* Protocol for the sender  $\mathcal{P}_s$  \*/ 1: upon receiving the input value m do  $\{m_j\}_{j \in n} \leftarrow \operatorname{Enc}(m), vc \leftarrow \operatorname{VCom}(\{m_i\})$ 2: 3: for each  $j \in [\mathbb{M}]$  do  $\pi_i \leftarrow \mathsf{Open}(\mathsf{vc}, m_i, j)$ 4: 5: let *store* :=  $\langle vc, m_i, j, \pi_i \rangle$ 6: send STORE(ID, store) to  $\mathcal{P}_i$ 7: wait until |S| = 2f + 18:  $\sigma_{\mathsf{PD}} \leftarrow \mathsf{TSIG}.\mathsf{Combine}(\langle \mathsf{STORED}, \mathsf{ID}, \mathsf{vc} \rangle, S)$ let  $\mathit{lock} := \langle vc, \sigma_{PD} \rangle$  and multicast DISPERSAL(ID,  $\mathit{lock}$ ) to all 9.  $\mathcal{P}_{i} \in \mathbb{M}$ 10: **upon** receiving STORED(ID,  $\rho_i$ ) from  $\mathcal{P}_i$  for the first time **do** if TSIG.VerifyShare( $(STORED, ID, vc), (j, \rho_j)$ ) = 1 then 11:  $S \leftarrow S \cup (j, \rho_j)$ 12: /\* Protocol for each party  $\mathcal{P}_i \in \mathbb{M}$  \*/ 13: upon receiving STORE(ID, store) from sender  $\mathcal{P}_s$  for the first time do  $14 \cdot$ **if** ValidateStore(i, store) = 1 **then** 15: **deliver** store and parse store it as  $\langle vc, m_i, i, \pi_i \rangle$  $\rho_i \leftarrow \mathsf{TSIG}.\mathsf{SignShare}(\mathit{sk}_i, \langle \mathsf{STORED}, \mathsf{ID}, \mathsf{vc} \rangle)$ 16: send STORED(ID,  $\rho_i$ ) to  $\mathcal{P}_s$ 17: 18: **upon** receiving DISPERSAL(ID, *lock*) from  $\mathcal{P}_s$  for the first time **do** if ValidateLock(ID, lock) = 1 then 19: 20deliver lock ValidateStore(*i*', *store*): 21: parse *store* as:  $\langle vc, m_i, i, \pi_i \rangle$ **return** VerifyOpen(vc,  $m_i, i, \pi_i$ )  $\land i = i'$ 22.

ValidateLock(ID, *lock*): 23: parse *lock* as:  $\langle vc, \sigma_{PD} \rangle$ 

```
24: return TSIG.VerifyThld(\langleSTORED, ID, vc\rangle, \sigma_{PD})
```

• *Recast-ability*. If all honest parties in  $\mathbb{M}$  enter the RC phase and at least one of them has received a valid *lock* satisfying ValidateLock(ID, *lock*) = 1, then: (i) all honest parties in  $\mathbb{M}'$  output some common value m'; (ii) if the sender  $\mathcal{P}_s$  dispersed *m* during the PD phase and was not corrupted before f + 1 honest parties in  $\mathbb{M}$  output *store*, then m' = m.

Instantiation. Throughout the paper, we adapt the APDB protocol from [48] to realize APDB as illustrated in in Algorithm 6 and Algorithm 7. The construction relies on a high-threshold signature scheme across the parties  $\mathbb{M}$  to preserve low communication overhead. Noticeably, if we consider the high-threshold ADKR problem, the old committee  $\mathbb{M}$  members already have high-threshold signature for granted; even if we consider the ADKR problem where  $\mathbb{M}$  has only "low-threshold" setup, the APDB construction still can be instantiated from high-threshold signature with silent setup.

## 8 Conclusion

We present an efficient ADKR protocol, reducing communication complexity from  $O(n^3)$  to  $O(\kappa n^2)$  with preserving adaptive security (where  $\kappa \approx 30$ ). Using a *share-dispersalthen-agree-and-recast* paradigm and optimizations like (i) linear-communication interactive PVSS generation and (ii)

## Algorithm 7 RC protocol, with identifier ID for each party in $\mathbb{M}$ and $\mathbb{M}'$ .

1:	/* Protocol for each $\mathcal{P}_i \in \mathbb{M}$ */ <b>upon</b> receiving input <i>store</i> <b>do</b>
2:	<b>multicast</b> $RCSTORE(ID, store)$ to all parties in $\mathbb{M}'$
	/* Protocol for each party $\mathcal{P}_i \in \mathbb{M}' */$
3:	<b>upon</b> receiving RCSTORE(ID, <i>store</i> ) from sender $\mathcal{P}_i \in \mathbb{M}$ <b>do</b>
4:	if ValidateStore $(i, store) = 1$ then
5:	parse store as: $\langle vc, m_i, j, \pi_i \rangle$
6:	$C[vc] \leftarrow (C[vc] \cup (j, m_j))$
7:	if $ C[vc]  = f + 1$ then
8:	$v \leftarrow Dec(C[vc])$
9:	<b>if</b> $VCom(Enc(v)) = vc$ <b>then</b>
0:	return v
11:	else return $\perp$

distributed PVSS verification, it achieves up to 40% latency reduction in 256-node experiments. Additionally, the design supports broader applications, including the first quadraticcommunication ADPSS protocol with adaptive security.

## References

- Ittai Abraham, Renas Bacho, Julian Loss, and Gilad Stern. Nearly quadratic asynchronous distributed key generation. Cryptology ePrint Archive, Paper 2025/006, 2025.
- [2] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, and Gilad Stern. Bingo: Adaptivity and asynchrony in verifiable secret sharing and distributed key generation. In *Annual International Cryptology Conference*, pages 39–70. Springer, 2023.
- [3] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 363–373, 2021.
- [4] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync hotstuff: Simple and practical synchronous state machine replication. In 2020 IEEE Symposium on Security and Privacy (SP), pages 106– 118, 2020.
- [5] Damiano Abram, Jack Doerner, Yuval Ishai, and Varun Narayanan. Constant-round simulation-secure coin tossing extension with guaranteed output. In *EUROCRYPT* (5), volume 14655 of *Lecture Notes in Computer Science*, pages 122–154. Springer, 2024.
- [6] Renas Bacho, Christoph Lenzen, Julian Loss, Simon Ochsenreither, and Dimitrios Papachristoudis. Grandline: Adaptively secure DKG and randomness beacon with (almost) quadratic communication complexity. In *CCS*. ACM, 2024.

- [7] Renas Bacho and Julian Loss. On the adaptive security of the threshold BLS signature scheme. In CCS, pages 193–207. ACM, 2022.
- [8] Renas Bacho and Julian Loss. Adaptively secure (aggregatable) pvss and application to distributed randomness beacons. In CCS. ACM, 2023.
- [9] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In CCS, pages 913–930. ACM, 2018.
- [10] Akhil Bandarupalli, Adithya Bhat, Saurabh Bagchi, Aniket Kate, and Michael K Reiter. Random beacons in monte carlo: Efficient asynchronous random beacon without threshold cryptography. In *Proceedings of the* 2024 on ACM SIGSAC Conference on Computer and Communications Security, pages 2621–2635, 2024.
- [11] Michael Ben-Or. Another advantage of free choice (extended abstract) completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, 1983.
- [12] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffiehellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46, 2002.
- [13] Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In Proceedings of the second annual ACM symposium on Principles of distributed computing, pages 12–26, 1983.
- [14] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th* ACM Conference on Computer and Communications Security, pages 88–97, 2002.
- [15] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541, 2001.
- [16] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. volume 3, pages 219–246, 2005.
- [17] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *Proc. SRDS 2005*, pages 191–201, 2005.

- [18] Ran Canetti, Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Adaptive security for threshold cryptosystems. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 98–115. Springer, 1999.
- [19] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 42–51, 1993.
- [20] Ignacio Cascudo and Bernardo David. Scrape: Scalable randomness attested by public entities. In *International Conference on Applied Cryptography and Network Security*, pages 537–556. Springer, 2017.
- [21] Arka Rai Choudhuri, Sanjam Garg, Julien Piet, and Guru-Vamsi Policharla. Mempool privacy via batched threshold encryption: Attacks and defenses. In 33rd USENIX Security Symposium (USENIX Security 24), 2024.
- [22] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In Proceedings of the Seventeenth European Conference on Computer Systems, pages 34–50, 2022.
- [23] Sourav Das, Philippe Camacho, Zhuolun Xiang, Javier Nieto, Benedikt Bünz, and Ling Ren. Threshold signatures from inner product argument: Succinct, weighted, and multi-threshold. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–370, 2023.
- [24] Sourav Das, Zhuolun Xiang, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling. In USENIX Security 2023, 2023.
- [25] Sourav Das, Zhuolun Xiang, Alin Tomescu, Alexander Spiegelman, Benny Pinkas, and Ling Ren. Verifiable secret sharing simplified. In *IEEE S&P 2025*.
- [26] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *IEEE S&P* 2022, 2022.
- [27] Hanwen Feng, Zhenliang Lu, and Qiang Tang. Dragon: Decentralization at the cost of representation after arbitrary grouping and its applications to sub-cubic DKG and interactive consistency. Cryptology ePrint Archive, Paper 2024/168, 2024.
- [28] Hanwen Feng, Zhenliang Lu, and Qiang Tang. Dragon: Decentralization at the cost of representation after arbitrary grouping and its applications to sub-cubic DKG

and interactive consistency. In *PODC*, pages 469–479. ACM, 2024.

- [29] Hanwen Feng, Tiancheng Mai, and Qiang Tang. Scalable and adaptively secure any-trust distributed key generation and all-hands checkpointing. In *Proceedings* of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, pages 2636–2650, 2024.
- [30] Hanwen Feng and Qiang Tang. Asymptotically optimal adaptive asynchronous common coin and DKG with silent setup. Cryptology ePrint Archive, Paper 2024/2098, 2024.
- [31] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374– 382, 1985.
- [32] Pierre-Alain Fouque and Jacques Stern. One round threshold discrete-log key generation without private channels. In *Public Key Cryptography: 4th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2001 Cheju Island, Korea, February 13–15, 2001 Proceedings 4*, pages 300–316, 2001.
- [33] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. In 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS), pages 246–257, 2022.
- [34] Sanjam Garg, Abhishek Jain, Pratyay Mukherjee, Rohit Sinha, Mingyuan Wang, and Yinuo Zhang. hints: Threshold signatures with silent setup. In 2024 IEEE Symposium on Security and Privacy (SP), pages 3034– 3052. IEEE, 2024.
- [35] Sanjam Garg, Dimitris Kolonelos, Guru-Vamsi Policharla, and Mingyuan Wang. Threshold encryption with silent setup. In *Annual International Cryptology Conference*, pages 352–386. Springer, 2024.
- [36] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. J. Cryptol., 20(1):51– 83, 2007.
- [37] Craig Gentry, Shai Halevi, and Vadim Lyubashevsky. Practical non-interactive publicly verifiable secret sharing with thousands of parties. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 458–487, 2022.
- [38] Jens Groth. Non-interactive distributed key generation and key resharing. *IACR Cryptol. ePrint Arch.*, 2021:339, 2021.

- [39] Jens Groth and Victor Shoup. Fast batched asynchronous distributed key generation. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 370–400. Springer, 2024.
- [40] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In 2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN), pages 568–580, 2019.
- [41] Christoph Günther, Sourav Das, and Lefteris Kokoris-Kogias. Practical asynchronous proactive secret sharing and key refresh. *IACR Cryptol. ePrint Arch.*, page 1586, 2022.
- [42] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous bft closer to practice. *Cryptology ePrint Archive*, 2022.
- [43] Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. In *EUROCRYPT (1)*, volume 12696 of *Lecture Notes in Computer Science*, pages 147–176. Springer, 2021.
- [44] Aniket Kate, Easwar Vivek Mangipudi, Pratyay Mukherjee, Hamza Saleem, and Sri Aravinda Krishnan Thyagarajan. Non-interactive vss using class groups and application to dkg. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4286–4300, 2024.
- [45] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16, pages 177–194, 2010.
- [46] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *Proceedings of the 2020* ACM SIGSAC Conference on Computer and Communications Security, pages 1751–1767, 2020.
- [47] Benoît Libert, Marc Joye, and Moti Yung. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 303–312, 2014.

- [48] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Proceedings of the 39th symposium on principles of distributed computing*, pages 129–138, 2020.
- [49] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pages 31–42, 2016.
- [50] Torben P. Pedersen. Non-interactive and informationtheoretic secure verifiable secret sharing. In *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.
- [51] Tian Qiu and Qiang Tang. Predicate aggregate signatures and applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 279–312. Springer, 2023.
- [52] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, pages 347–356, 2019.
- [53] Thomas Yurek, Zhuolun Xiang, Yu Xia, and Andrew Miller. Long live the honey badger: Robust asynchronous {DPSS} and its applications. In 32nd USENIX Security Symposium (USENIX Security 23), pages 5413– 5430, 2023.

## A Adpatively Secure ADKR with t = f

In this section, we demonstrate that when instantiating with adaptively secure components our framework yields an adaptively secure ADKR protocol.

**Definition: Oracle-Aided Algebraic Simulatability.** The adaptive security achieved by our protocol is termed oracle-aided algebraic simulatability, introduced by Bacho and Loss [7] to capture the adaptive security of many practical DKG protocols. This notion is at least sufficient for instantiating the key generation of the threshold BLS signature protocol. At a high level, it ensures that all information accessible to the adversary during protocol execution can be simulated by an algebraic algorithm with access to a discrete logarithm oracle  $DL_g(\cdot)$ .

Formally, the termination property and key validity defined in Sect.2 naturally extend to the adaptive setting. We recall the secrecy definition from [7] below, originally defined for DKG but applicable to DKR with minimal modifications.

**Definition 2** (Oracle-aided Algebraic Simulatability). A protocol  $\Pi$  satisfies k-oracle-aided algebraic simulatability if for every PPT adversary  $\mathcal{A}$  adaptively corrupts at most f parties in  $\mathbb{M}$  and f parties in  $\widetilde{\mathbb{M}}$ , there exists a PPT simulator  $S^{\mathcal{A}}$  which on input  $\zeta = (g^{z_1}, \ldots, g^{z_k}) \in \mathbb{G}^k$ , can query the DLog oracle  $\mathsf{DL}_g(\cdot)$  for at most k - 1 times, and simulate an execution of  $\Pi_{\mathsf{DKG}}$  for  $\mathcal{A}$ . In particular,

- Algebraic oracle queries: When  $S^{\mathcal{A}}$  queries  $\mathsf{DL}_g(\cdot)$  with a group element g', it must provides the algebraic expression of g' in the term of  $(g, g^{z_1}, \ldots, g^{z_k})$ , i.e., a vector  $(\hat{a}, a_1, \ldots, a_k) \in \mathbb{F}^{k+1}$  such that  $g' = g^{\hat{a}} \prod_{j \in [k]} (g^{z_j})^{a_j}$ . The oracle will return  $a \in \mathbb{F}$  s.t.  $g' = g^a$ .
- Indistinguishable simulation: Denote by view<sub> $\mathcal{A},y,\Pi$ </sub> the view of  $\mathcal{A}$  in an execution of  $\Pi$  outputting pk = y.  $\mathcal{S}^{\mathcal{A}}$  can output simview<sub> $\mathcal{A}$ </sub> and pk, such that (view<sub> $\mathcal{A},y,\Pi$ </sub>, y) is computationally indistinguishable with (simview<sub> $\mathcal{A}$ </sub>, pk).
- Invertible simulatability martrix: Let  $g_i$  denote the *i*th query by S to  $DL_g(\cdot)$ . Let  $(\hat{a}_i, a_{i,1}, \ldots, a_{i,k})$  be the corresponding algebraic coefficients of  $g_i$ , i.e.,  $g_i =$  $g^{\hat{a}_i}\prod_{j=1}^k (g^{z_j})^{a_{i,j}}$  and set  $(\hat{a}, a_{0,1}, \ldots, a_{0,k})$  as the algebraic coefficients of pk. Then, the following matrix over  $\mathbb{F}$  is invertible

$$L := egin{pmatrix} a_{0,1} & a_{0,2} \cdots & a_{0,k} \ a_{1,1} & a_{1,2} \cdots & a_{1,k} \ dots & dots & dots \ a_{k-1,1} & a_{k-1,2} \cdots & a_{k-1,k} \end{pmatrix}.$$

#### A.1 Choices of Adaptively Secure PVSS

The protocol in Section 4 only achieves the static security because the underlying PVSS scheme is static. On the other hand, while there are a few candidates for adaptive and verifiable ACSS, they come with certain limitations. We discuss the choices in the following.

Adaptive PVSS with Group-Element Secrets. Bacho and Loss [8] recently proved that a variant of the PVSS scheme from [43] is secure against adaptive adversaries. However, this PVSS scheme only deals with secrets in a pairing-friendly ECC group, making it incompatible with major Dlog cryptosystems that require secret keys in a scalar field like  $\mathbb{Z}_p$ .

On the other hand, Bacho et al. [6] presented a threshold signature scheme with group-element secret keys and utilized the PVSS scheme to construct a distributed key generation (DKG) protocol for their threshold signature scheme.

In principle, instantiating our ADKR protocol with this adaptive PVSS scheme could yield an ADKR protocol for the threshold signature scheme in [6]. As summarized in [6], the threshold signature scheme is secure as long as the public and secret key shares are derived from an aggregated PVSS transcript, which "is just an aggregation of several initially sampled PVSS transcripts with contributions from at least one honest party." Our ADKR/ADKG protocol satisfies this condition: the network recasts  $\kappa$  randomly sampled PVSS transcripts, ensuring at least one is contributed by an honest party except with negligible probability.

A formal security analysis of Bacho et al.'s threshold signature scheme [6] under our ADKR setup is left for future work.

**Signature-based** PVSS with a privacy threshold of *f*. Alternatively, if the application does not need a high threshold of t = 2f, we can employ an adaptive variant of the PVSS in Algorithm 1 (also a variant of [25]), which supports a privacy threshold of *f*.

Compared with Algorithm 1, we make the following changes. First, since the secrecy goal has been relaxed to oracle-aided algebraic simulatability, we can use more efficient Feldman commitment, *i.e.*,  $g^s$  as a commitment to *s*. Second, as the privacy threshold is just *f*, we do not need to use a VE to encrypt the shares for parties who have not returned their shares. Instead, we can just publish those shares. This strategy is also used in [25] for better efficiency for the "low" threshold case. For completeness, we include a description of the PVSS protocol in Algorithm 8.

#### Algorithm 8 A Variant of PVSS in [25]

 $\mathsf{Deal}\langle \mathscr{P}_d(\mathsf{ek},\mathsf{pk},s), \mathbb{M}(\mathsf{sk}_i) \rangle \rightarrow \langle \mathscr{P}_d(\mathsf{transcript}), \mathbb{M}(s_i) \rangle$ // Code run by Dealer  $\mathcal{P}_d$ 1: randomly sample a *t*-degree polynomial  $\phi(\cdot)$  where  $\phi(0)$  is *s*. 2: compute  $\mathsf{v} \leftarrow \{g^{\phi(j)}\}_{\mathscr{P}_i \in \mathbb{M}}$ 3: for  $\mathcal{P}_i \in \mathbb{M}$  do send SHARE $(v_i, \phi(j))$ 4: 5: **upon** receiving 2f + 1 valid signatures  $\sigma_i$  for  $v_i$  **do** Let  $\Sigma$  be the valid signatures set 6: Let I be the indices of nodes with missing valid signatures 7. 8: **return** transcript  $\leftarrow (v, \Sigma, \{\phi(i)\}_{i \in I})$ // Code run by each  $\mathcal{P}_i \in M$ 9: **upon** receiving SHARE $(v_i^{(j)}, s_i^{(j)})$  from  $\mathcal{P}_i$  **do** if  $v_i^{(j)} = g^{s_i^{(j)}}$  then 10:  $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}_i, v_i^{(j)})$ 11: send ACK( $\sigma_i$ ) 12: return  $s_i^{(j)}$ 13: return  $\perp$ 14:  $Verify(pk, transcript) \rightarrow 0/1$ 15: parse *script* as:  $(v, \Sigma, \{(s_z^{(j)})\}_{z \in I})$ 16: Check  $\forall \sigma_j \in \Sigma$  is a valid signature for  $v_j$ 17: Check DegCheck(v,t) = 118: Check  $v_z^{(j)} = g^{s_z^{(j)}}$  for  $z \in I$ 19: if all the checks pass then 20: return 1 21: return 0

#### A.2 Instantiation and Analysis

In the following, we formally analyze the adaptive security of our ADKR protocol when instantiating the PVSS with the signature-based PVSS protocol described above. **Instantiation.** For clarity, we specify the instantiations of our adaptive ADKR/ADKG protocol. As outlined in Sect. 4, our protocol is built upon an ACSS protocol (with verifiable transcripts), Dumbo-MVBA [48], APDB, and a threshold common coin. Both Dumbo-MVBA and APDB rely on a non-interactive high-threshold signature, which can be instantiated with any adaptively secure threshold signature scheme with a silent setup, such as [23, 34, 51].

The PVSS protocol is instantiated using the signaturebased construction described above. Note that the key derivation phase of our ADKR/ADKG protocol is tailored to the underlying PVSS protocol. With the signature-based PVSS, the key derivation phase is detailed in Algorithm 9.

## **Algorithm 9** KEY DERIVATION PHASE for the PVSS scheme in Algorithm 8

Each  $P_i \in \widetilde{\mathbb{M}}$  has obtained a set of verified ACSS transcripts  $\{\operatorname{script}^{(\ell)}\}_{\ell \in T^*}$ , and  $\operatorname{script}^{(\ell)}$  can be parsed as  $(\mathbf{v}^{(\ell)}, \Sigma, (s_j^{(\ell)})_{j \in I})$ 1:  $z_i \leftarrow \sum_{\ell \in T^*} s_i^{(\ell)}$ 2:  $\operatorname{tpk}_j \leftarrow \prod_{\ell \in T^*} v_j^{(\ell)}$  for all  $\mathcal{P}_j \in \widetilde{\mathbb{M}}$ 3:  $\operatorname{tpk} = \prod_{j \in [t]} \operatorname{tpk}_j^{\lambda_j}$ , where  $\lambda_j$  is the Lagrange coefficient. 4: return  $(\operatorname{tpk}, (\operatorname{tpk}_i)_{i \in [n]}, z_i)$ 

The underlying threshold common coin must be adaptive. For the ADKR setting, it can be instantiated with an adaptively secure unique threshold signature scheme [7, 16]. In the ADKG setting, it can be instantiated with an adaptively secure quadratic-communication common coin protocol that does not require a DKG setup, such as [30].

To achieve adaptive security, we assume *memory erasures* [18], requiring each node  $P_i$  in the old committee to *erase* its secrets as a PVSS dealer (e.g., the secret polynomial  $\phi^{(i)}$ ) *before* invoking the dispersal protocol PD.

Adaptive Security Analysis. The termination property and key validity against adaptive adversaries easily follow the analysis presented in Lemma 4, Lemma 5, and Lemma 6. In the following, we prove our protocol satisfies the oracle-aided algebraic simulatability.

**Lemma 11.** When the underlying PVSS is the signaturebased PVSS in Algorithm 8, and the other underlying components are adaptively secure, our ADKR protocol satisfies the oracle-aided algebraic simulatability.

*Proof.* For any PPT adversary  $\mathcal{A}$ , we can build a simulator  $\mathcal{S}^{\mathcal{A}}$  in the following.  $\mathcal{S}^{\mathcal{A}}$  takes as inputs 2f + 1 random group elements and can make up to 2f queries to  $\mathsf{DLog}_{q}$ .

**Notations:** Let  $\mathcal{H}$  be the set of so-far-honest nodes, and let C be the set of all corrupted nodes.

**Input:** 2f + 1 random group elements  $\zeta_0, \ldots, \zeta_{2f}$ .

**Simulation:** First, uniformly sample  $i^* \leftarrow [n]$ , which represents a node in the old committee

 $S^{\mathcal{A}}$  runs the protocol with  $\mathcal{A}$  by acting on the behalf of all so-far-honest nodes with the following strategy:

Simulating the Setup: It honestly generates all public parameters and key pairs for all nodes in  $\mathcal{H}$ 

#### Simulating the sharing/dealing phase:

- On the behalf of all nodes in  $\mathcal{H} \cap \mathbb{M}$  except  $P_{i^*}$ , it honestly executes the protocol.
- On the behalf of  $P_{i^*}$ , it computes  $v_j = \prod_{z \in [0,2f]} \zeta_z^{j^z}$  for all  $\mathcal{P}_j \in \widetilde{\mathbb{M}}$ . Then, for all  $\mathcal{P}_j \in \mathcal{C}$ , it queries the DLog oracle  $\mathsf{DLog}_{\mathfrak{g}}(\cdot)$  with  $v_i$ , and obtains  $a_i$  such that  $A_i = g^{a_i}$ . It sends SHARE $(v_i, a_i)$  to all  $P_i \in C \cap \mathbb{M}$ , and sends  $(v_i, \bot)$ to all honest nodes in  $\widetilde{\mathbb{M}}$ .

Then, after receiving (2f+1) valid signatures from distinct nodes in M (the signature set is denoted as  $\Sigma$ ), it queries requires the DLog oracle  $DLog_{g}(\cdot)$  with  $v_{j}$  for all  $\mathcal{P}_i \in I$  (if  $v_i$  has not been queried before), where  $\{P_i\}_{i \in I}$ is the set of f nodes in  $\mathbb{M}$  which have not returned the signature. It obtains  $\{a_i\}_{i \in I}$  accordingly. Finally, it prepares the script as  $(v, \Sigma, (a_j)_{j \in I})$ . After that, it executes the protocol honestly on the behalf of  $P_{i^*}$ .

• On the behalf of all nodes in  $\mathcal{H} \cap \widetilde{\mathbb{M}}$ , it honestly executes the protocol except that it directly returns the digital signature for  $v_{i^*}$  to  $P_{i^*} \in \mathbb{M}$  without verifying the openings.

Simulating the consensus phase: On the behalf of all honest nodes, it executes the protocol honestly. However, if Coin.Get() returns T' such that  $i^* \notin T'$ , it **aborts** the current simulated execution, samples a new  $i^* \leftarrow [n]$ , rewinds  $\mathcal{A}$ to the beginning of the protocol, and restarts the simulation. Simulating verification and key derivation: On the behalf of all honest nodes, it executes the protocol honestly.

*Handle Corruption*: If  $\mathcal{P}_{i^*}$  is corrupted before  $\mathcal{P}_{i^*}$  is supposed to erase its secret polynomial (say, at the time of inovking PD), then  $\mathcal{S}^{\mathcal{A}}$  **aborts** the current simulated execution, samples a new  $i^* \leftarrow [n]$ , and **rewinds**  $\mathcal{A}$  to the beginning of the protocol. If  $\mathcal{P}_i \in \widetilde{\mathbb{M}}$  is corrupted, it queries the DLog oracle  $\mathsf{DLog}_{\mathfrak{g}}(\cdot)$ with  $v_i$  (if  $v_i$  has not been queried before) and obtains  $a_i \in \mathbb{Z}_p$ . It then returns all internal states of  $\mathcal{P}_i$  to the adversary, while the secret share  $s_j^{(i^*)}$  from  $\mathcal{P}_{i^*} \in \mathbb{M}$  is set to be  $a_j$ . In other cases, it returns the internal states of the node to the adversary. Termination: Upon an honest node terminates, it returns all information available to  $\mathcal{A}$  in the last simulated execution and the public key outputted by honest nodes.

Now, we analyze the simulator satisfies the requirements outlined in Def.2.

Running time of  $S^{\mathcal{A}}$ . First, we show that the expected running time of  $S^{\mathcal{A}}$  is polynomial in the security parameters and *n*. Note that  $\mathcal{S}^{\mathcal{A}}$  may rewind the adversary  $\mathcal{A}$  under certain conditions and restart a new simulated execution until a successful

execution where an honest node can terminate. It it trivial to check that each simulated execution takes polynomial time. Thus, it is sufficient to show that  $S^{\mathcal{A}}$  only needs to rewind  $\mathcal{A}$ for polynomial many times in expectation.

Note that  $S^{\mathcal{A}}$  rewinds  $\mathcal{A}$  when the randomly sampled  $\mathcal{P}_{i^*}$  is either corrupted before  $\mathcal{P}_{i^*}$  invokes PD or not included in  $L^*$ . Since the choice of  $i^*$  is independent of the view of  $\mathcal{A}$  before rewinding, and  $\mathcal{A}$  corrupts up to f nodes, the probability that  $\mathcal{P}_{i^*}$  is *not* corrupted before the output of Coin.Get() is at least  $\frac{n-f}{n}$ . Being included in L\* means that i\* is included in the output of Dumbo-MVBA and then selected by Coin.Get(), which, under the condition that  $\mathcal{P}_i$  is not corrupted before the output of Coin.Get(), happens with the probability of at least  $\frac{\kappa}{n}$ . Therefore, the probability that a simulated execution does not abort is at least  $\frac{\kappa(n-f)}{n^2}$ , and the expected time that  $S^{\mathcal{A}}$ rewinds  $\mathcal{A}$  is not greater than  $\frac{n^2}{\kappa(n-f)}$ , which is polynomial in *n* and the security parameters.

Indistinguishable simulation. It is easy to see that the view of an adversary in the last simulated execution which does not abort is identically distributed with its view in a real execution. Algebraic oracle queries and invertible simulatability matrix:  $\mathcal{S}^{\bar{\mathcal{A}}}$  needs to query the DLog oracle  $\mathsf{DLog}_{\rho}(\cdot)$  for all  $A_{j}$ , such that  $\mathcal{P}_i \in \widetilde{\mathbb{M}}$  is corrupted or does not respond its signature at the time of generating transcript script. There are at most f

corrupted nodes and at most f nodes having not returned their signatures. Thus,  $S^{\mathcal{A}}$  queries the oracle for at most 2f times. For each query  $A_i$ ,  $S^{\mathcal{A}}$  can provide its algebraic representa-

tion over  $(g, \zeta_0, \ldots, \zeta_{2f})$  as  $(0, 1, j, j^2, \ldots, j^{2f})$ . Note that the final public key tpk is computed as

$$\prod_{j \in [2f+1]} \mathsf{tpk}_j^{\lambda_j} = \prod_{j \in [2f+1]} (\prod_{\ell \in T^*} v_j^{(\ell)})^{\lambda_j} = \prod_{\ell \in T^*} \prod_{j \in [2f+1]} (v_j^{(\ell)})^{\lambda_j}.$$

Without loss of generality, we assume  $S^{\mathcal{A}}$  has queried the oracle for 2*f* times with  $\{A_j\}_{j \in S}$ , for  $S = j_1, \dots, j_{2f}$ 

We know that  $\prod_{j \in [2f+1]} (v_i^{(i^*)})^{\lambda_j} = \zeta_0$ . For  $\ell \neq i^*$ , the simulator  $S^{\mathcal{A}}$  can reconstruct the secret  $s^{(\ell)}$  such that  $\prod_{j \in [2f+1]} (v_j^{(\ell)})^{\lambda_j} = g^{s^{(\ell)}}$ . Therefore,  $S^{\mathcal{A}}$  can provide the algebraic representation of tpk over  $(g, \zeta_0, \dots, \zeta_t)$  as  $(\sum_{\ell \in L^*, \ell \neq i^*} s^{(\ell)}, 1, 0, \dots, 0).$ Therefore, the simulatability matrix is as follows:

$$L := \begin{pmatrix} 1 & 0 \cdots & 0 \\ 1 & j_1 \cdots & j_1^{2f} \\ \vdots & \vdots & \vdots \\ 1 & j_{2f} \cdots & j_{2f}^{2f} \end{pmatrix}.$$

It is easy to verify the above matrix is invertible.

Ì