Practical Asynchronous Distributed Key Reconfiguration and Its Applications

Hanwen Feng[*], Yingzi Gao[2], Yuan Lu[3], Qiang Tang[1] and Jing Xu[3,4,5]

[1]School of Computer Science, The University of Sydney
[2]Shandong University
[3]Institute of Software, Chinese Academy of Sciences
[4]University of Chinese Academy of Sciences
[5]Zhongguancun Laboratory

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 an efficient *share-dispersal-then-agree-and-recast* paradigm for constructing efficient ADKR while preserving adaptive security. Our method reduces the total overhead to $O(\kappa n^2)$ from $O(n^3)$, where κ is a small constant (typically $\sim$30 or less). And our further optimizations in PVSS minimize redundant computations across different parties and reduce the dominating PVSS verification cost by about one-third.

Our techniques developed for ADKR can also be leveraged to improve the asymptotic efficiency of various other asynchronous protocols: (i) it implies the first (coin-assisted) quadratic-communication ADKG; and (ii) it can be extended to realize the first quadratic-communication asynchronous dynamic proactive secret sharing (ADPSS) with adaptive security. Experimental evaluations on a global network of 256 AWS servers show up to 40% lower latency compared to the state-of-the-art ADKG protocols that are simplified 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 to achieve better performance (e.g., lower round and communication complexities [4, 21, 40, 42, 48, 52]) and enhanced security (e.g., private mempools to mitigate "miner" extractable value [14, 20, 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 [15] are widely adopted for generating common coins to efficiently overcome the seminal Fischer-Lynch-Patterson (FLP) "impossibility" [30] to achieve agreement while retaining liveness.

Reliance on a pre-configured threshold cryptosystem in practical asynchronous fault-tolerant systems, unfortunately, fixes the set of participants, thereby creating an 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, 23, 25, 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 [23, 25].

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. [1] Motivated by the key challenge, we focus on the following problem of asynchronous 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).** ADKR *is a simplified version of* ADKG*, which, in the presence of an already-established 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*

[*]Hanwen, Yingzi, and Yuan contributed equally to this work.

[1]None of the top-30 blockchains by market capitalization adopts fully asynchronous consensus at the time of writing (Jan 19, 2025), despite recent advances on this topic (that are mostly in settings with fixed participants).

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 Related Works and Their Limitations

ADKR, as a natural formulation of threshold cryptosystem reconfiguration in the asynchronous network, was previously studied [41] and can be derived from ADKG protocols [2, 3, 23, 25, 39, 46]. Here, we briefly review these approaches and highlight their limitations.

Adapting ADKG into the coin-hybrid model still suffers from a cubic communication cost. As Figure 1 clarifies, ADKR can be immediately realized from the state-of-the-art ADKG [23, 25], with a couple of simple optimizations in the coin-hybrid model (where a common-coin oracle returns unpredictable randomness upon a sufficient number of parties query it). First, since an efficient one-round protocol for asynchronous common coins can be implemented from the old participants' threshold cryptosystem [15], we can significantly simplify the consensus part of ADKG by directly using the best so-far asynchronous consensus that rely on the granted common coins, avoiding the usually complex protocols of generating asynchronous common coins without any threshold cryptosystem setup [3, 11, 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 [25], and once common coins are granted, the part can also be simplified, by letting $2f+1$ parties (instead of all parties) 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) [2] protocols.[3] 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 assuming a free consensus component.

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

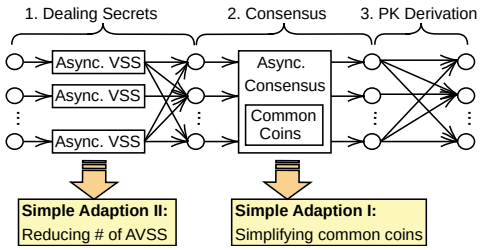


Figure 1: High-level structure of existing ADKG protocols and simple modifications of them to accommodate ADKR.

with sub-cubic communication. They used an assumed 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 outperform the straightforward adaptation of ADKG since their sub-committee size c is not concretely small. In particular, sampling an honest-majority sub-committee with a failure probability below 10^{-10} requires a sub-committee size of up to about 400. Moreover, the design is insecure against an adaptive adversary capable of corrupting participants after observing the common coin used to select sub-committee because such an 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 [22, 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 state-of-the-art silent threshold signature scheme [22] 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 (asynchronous) consensus, a non-interactive unique threshold signature scheme (assuming a setup provided by ADKR/ADKG), such as threshold BLS [8], is highly desirable to enable a *single-round* and efficient asynchronous common coin protocol [15]. Unfortunately, silent threshold signatures lack the uniqueness property, making it difficult to derive a common coin protocol directly. [4]

[2] 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 refer to ACSS throughout the paper.

[3] A very recent theoretic work [1] uses a recursive protocol structure to realize ADKG with nearly quadratic communication. But for a typical recursive depth of $\log n$, it causes $O(\log n)$ rounds and is not compatible with dLog cryptosystems since it generates group-element secret shares.

[4] Literature [29] proposed a silent-setup asymptotically optimal asynchronous common coin 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.

Given the limitations 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., standard BLS threshold signature), while being extensible to support adaptive security?*

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 optimizing concrete efficiency and preserving the capability to support adaptive security. Specifically,

- ADKR **with lower complexity**. To address the cubic total overhead caused by $O(n)$ ACSS protocols in these ADKR protocols from coin-hybrid ADKG, we propose an efficient *share-dispersal-then-agree-and-recast* paradigm for constructing ADKR. This 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, only κ dispersals are randomly selected for reconstruction, reducing verification to these few instances. And we further show that a small κ (several dozen) suffices since only one honest dispersal among the κ selected is required to achieve (adaptive) security, significantly reducing the concrete cost of prior nearly-quadratic ADKR [41] that is only statically secure.

- **Extensibility to adaptive security**. In addition to lower complexity, the proposed *share-dispersal-then-agree-and-recast* paradigm can be extended to support adaptive security with minimal additional overhead. Unlike prior nearly-quadratic ADKR [41] where an adaptive adversary could observe the sub-committee and subsequently corrupt all its members to learn their secrets corresponding to the reconstructed transcripts, our design defers the sub-committee selection until the so-far honest parties have safely erased their local secrets. Our approach defers the sub-committee selection until after all so-far honest parties have securely erased their local secrets. By instantiating the protocol with an adaptively secure PVSS scheme (such as [9] or a variant of [24]), the resulting ADKR protocol can capture adaptive security.

- **Various optimizations for concrete efficiency**. Inspired by the recent DXT+ PVSS [24], 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 [31, 43] when instantiating the *share-dispersal-then-agree-and-recast* paradigm. We further reduce the dominating cost of verifying κ 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 of new techniques arising from our** ADKR **construction**. The techniques developed for our efficient ADKR design can also be applied to construct a variety of other asynchronous protocols. 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 [29], 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 extensibility to adaptive security) by extending our ADKR design to efficiently distribute common randomness twice across two distinct committees.

- **Implementation and scalable experiments**. We implement our ADKR protocol and experimentally compare it with state-of-the-art ADKG protocols (including DYX+22 [25] and DYK+23 [23] 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.

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

	Possibly adaptive?	Standard dLog?	Comm.	Round
Classic ADKGs with adaptions	✓	✓	$O(\lambda n^3)$	$O(1)$
GDK ADKR [41]	✗	✓	$O(c\lambda n^2)$ †	$O(1)$
Ours (§4)	✓‡	✓	$O(\kappa\lambda n^2)$ †	$O(1)$

† Though c and κ are both statistic parameters, κ is significantly smaller than c. For a failure probability of 10^{-10}, the upper bound of c is about 400 [41], while κ is typically around 30.

‡ We give an adaptive security proof for an instantiation in Appendix C.

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

	Possibly adaptive?	Dynamic?	High-thld?	Comm.
CKLS [13]	✓	✗	✗	$O(\lambda n^4)$
GDK [41] †	✗	✗	✓	$O(c\lambda n^2)$
YXXM [53] †	✓	✓	✓	$O(\lambda n^3)$
Ours (§5)	✓	✓	✓	$O(\kappa\lambda n^2)$

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 distributively 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 $\mathbb{M}$ and $\widetilde{\mathbb{M}}$ 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}, such that these private key shares can interpolate a unique polynomial $\phi(x)$ with $g^s = g^{\phi(0)}$; and (iii) any subset of t (or fewer) private key shares reveal no additional information about the private key beyond the public keys, i.e., t corresponds to a privacy threshold. $\mathcal{TC}$ is called *high-threshold*, if the privacy threshold $t = n - f - 1 \geq 2f$, which is critical in many quorum-based asynchronous fault-tolerant systems (as t-threshold signatures cannot be forged without the partial signatures from $t - f$ parties [15]). Similar to these existing fault-tolerate systems, we might leverage a *high-threshold* $\mathcal{TC}$ setup for concrete efficiency.
- *Asynchronous network*. Each pair of honest parties in $\mathbb{M}$ and $\widetilde{\mathbb{M}}$ 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 $f = \lfloor \frac{n-1}{3} \rfloor$, and $\widetilde{f} = \lfloor \frac{\tilde{n}-1}{3} \rfloor$ parties in $\mathbb{M}$ and $\widetilde{\mathbb{M}}$ respectively. Following the standard cryptographic practice [14, 18], the adversary is 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 strongly adaptive adversary capable of performing the "after-the-fact removal", i.e., it can drop messages sent from a corrupted $\mathcal{P}_i$ 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 are 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).

SYNTAX OF ADKR: 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 $\widetilde{\mathcal{TC}}$.

SECURITY OF ADKR. 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 ADKG [23, 25, 46].

- *Termination*. Every honest party $\mathcal{P}_i$ in $\widetilde{\mathbb{M}}$ will output a vector of public keys $(g^z, g^{z_1}, g^{z_2}, \cdots, g^{z_{\tilde{n}}})$ and an exclusive private key share $z_i \in \mathbb{Z}_q$.
- *Key Validity*. The outputs of all honest parties in $\widetilde{\mathbb{M}}$ are consistent: (i) all parties' outputs contain the same public keys $(g^z, g^{z_1}, g^{z_2}, \cdots, g^{z_{\tilde{n}}})$; (ii) the private key share z_i obtained by each party is consistent with the corresponding public key share of g^{z_i}; (iii) there exists a unique polynomial $\phi \in \mathbb{Z}_p[X]$, such that $z = \phi(0)$, and $z_i = \phi(i)$ for all $i \in [\tilde{n}]$.
- *Full Secrecy:* No computationally bounded adversary can (i) prevent the secret key z from being uniformly sampled over $\mathbb{Z}_q$ or (ii) learn information about z beyond the public key g^z. Formally, for any PPT adversary $\mathcal{A}$ which can corrupt up to f parties, there exists a

PPT simulator $\mathcal{S}$ which on input a uniformly sampled g^z can produce a simulated view SimView, such that $(\mathsf{SimView}, g^z)$ is computationally distinguishable with $(\mathsf{View}_{\mathcal{A}}, pk)$, where $\mathsf{View}_{\mathcal{A}}$ is the view of $\mathcal{A}$ is a real execution with pk as the public key. In addition, in the context of *high-threshold* setting with a privacy threshold $\widetilde{t} = \widetilde{n} - \widetilde{f} - 1$, the adversary can access the secret key shares of at most $\widetilde{t} = \widetilde{n} - \widetilde{f} - 1$ *honest* parties in $\widetilde{\mathbb{M}}$, in addition to the shares of corrupted parties that have been available to it.

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* [8] (cf. Appendix C for details). Moreover, 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) [15]; (iii) a communication-efficient multi-valued validated asynchronous Byzantine agreement protocol (MVBA) [48]; (iv) a verifiable encryption scheme for Pedersen commitment [24]; 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 Appendix B.

Notations. Throughout the paper, let λ denote the bit length of the cryptographic security parameter, and let κ be the statistical security parameter. The notation $[i,n]$ represents the set $\{i, i+1, \ldots, n\}$, where i and n are integers with $i < n$. We may abbreviate $[1,n]$ as $[n]$. For a set $\{x_1, x_2, \ldots, x_n\}$ and a sequence $(x_1, x_2, \ldots, x_n)$, 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 $\mathsf{ProtoName[ID]}\langle\{\mathsf{ParticipantID(input)}\}\rangle$. We may omit ID or participant information when no ambiguity arises.

3 Challenges and Techniques

Challenge I: achieving near-quadratic total overhead while 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 incurs a quadratic communication. Therefore, to asymptotically reduce the cubic overhead, ADKG/ADKR can use, at most, a constant number of PVSS protocols instead of $O(n)$.

Reference [28] 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 with an honest majority but requires a significant 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: to realize near-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.

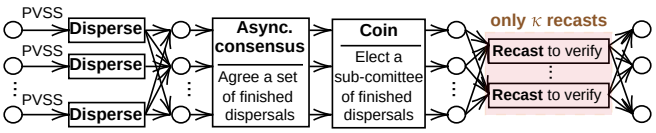


Figure 2: High-level idea of our *share-dispersal-then-agree-and-recast* paradigm (exemplified by non-interactive PVSS).

Technique I: share-dispersal-then-agree-and-recast. We introduce a *share-dispersal-then-agree-and-recast* paradigm to achieve nearly 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 κ dispersed PVSS transcripts to recover and verify, which costs only near-quadratic communication as κ 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 paradigm of *share-dispersal-then-agree-and-recast* 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 κ) that at least one of the κ 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 both concrete and asymptotic 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 [25, 31, 38, 44] or restricted applications (other than the standard dLog-based threshold cryptosystems) [9, 37, 43]. Recently, DXT+ PVSS [24] proposes an insightful approach using an interactive dealing phase with public verifiability, 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.

The above issues translate 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?

Technique II: publicly-verifiable linear-overhead dealing. We propose a simple optimization to DXT+ PVSS, thus 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, our dealing phase only sends each participant its exclusive secret share (and then has the dealer collect signatures on the Pedersen commitments to these secret shares from sufficient parties), reducing the dealer's each message size from $O(\lambda n)$ to $O(\lambda)$. Moreover, the dealer can still compute a $O(\lambda n)$-size 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 privacy 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.[5]

More concrete efficiency improvement from distributed PVSS verification. Another performance bottleneck is the verification of κ-selected PVSS transcripts. We observe that different parties perform redundant computing steps for verifying each PVSS transcript, and we therefore propose an optimized "distributed" verification approach.

The idea is to partition 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 result 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 extension to asymptotically efficient ADPSS. 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 approach from [53], which distributively 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 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 dual-committee randomness generation (DCRG) and extend our ADKR design to implement it more efficiently, with near-quadratic communication. Our improvement is achieved by modifying our ADKR 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 ADPSS with near-quadratic communication and adaptive security is therefore realized.

4 Efficient Asynchronous Distributed Key Reconfiguration Protocol

Before diving into the details of ADKR, we will first introduce an interactive PVSS scheme (an improved version of DXT+ PVSS [24]) in §4.1, which has a linear-communication asynchronous dealing phase to generate a publicly verifiable sharing transcript. Given this optimized asynchronous dealing protocol, we then present our ADKR protocol in §4.2.

[5] 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.

4.1 PVSS with Interactive Dealing

In a *non-interactive* 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 following sub-protocols and algorithms.

- $\mathsf{PVSS.Deal}\langle \mathcal{P}_d(\{pk_i\}_{i\in[n]}, s), \{\mathcal{P}_i(sk_i)\}_{i\in[n]}\rangle \to \langle \mathcal{P}_d(\mathsf{transcript}), \{\mathcal{P}_i(\mathsf{aux}_i)\}_{i\in[n]}\rangle$. The Deal stage protocol is executed among the dealer $\mathcal{P}_d$ and all receivers $\{\mathcal{P}_i\}_{i\in[n]}$. $\mathcal{P}_d$ takes the public keys $\{pk_i\}_{i\in[n]}$ of all participants and the secret s as inputs, and it outputs transcript. Each receiver $\mathcal{P}_i$ inputs its private key sk_i and receives auxiliary information aux_i.
- $\mathsf{PVSS.Verify}(\{pk_j\}_{j\in[n]}, \mathsf{transcript}) \to 0/1$. This algorithm verifies whether the sharing transcript is valid or not, using the public keys of all receivers.
- $\mathsf{PVSS.Deliver}(\mathsf{transcript}, sk_i, \mathsf{aux}_i) \to s_i$. This algorithm allows a receiver $\mathcal{P}_i$ to obtain a secret share s_i according to a valid transcript, its secret key sk_i, and the auxiliary information aux_i from Deal.

We remark that the functionality of ACSS, which ensures all honest parties 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 [24]. As briefly mentioned in §3, we give an improved version of the DXT+ PVSS protocol from [24], 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. We rely on the following components.

- A digital signature scheme with EUF-CMA security.
- A verifiable encryption (VE) scheme which satisfies IND-CPA security. A VE scheme enables a dealer to encrypt messages and produce a proof that the ciphertexts are consistent with some publicly known commitments. This allows anyone to verify that the encrypted values correspond to committed values, without learning the plaintexts. The full syntax of VE can be found in Definition 4 of [24].

  To encrypt a batch of secret shares at once, we use the batch interfaces VE.bEncProve and VE.bVerify, which are optimized to handle a batch of shares more efficiently than repeatedly invoking single-instance encryption and verification. In particular, each receiver $\mathcal{P}_j$ has a registered encryption-decryption key pair (ek_j, dk_j). We rely on the following three interfaces of the VE scheme:
  - $\mathsf{VE.bEncProve}(I, \{ek_j, a_j, v_j, b_j\}_{j\in I}) \to (\mathbf{c}, \pi_{\mathsf{VE}})$. For all $j \in I$, it encrypts each a_j under ek_j to get c_j, so it returns $\mathbf{c} = \{c_j\}_{j\in I}$. It also returns a proof π_{VE} attesting that each commitment $v_j = g^{a_j}h^{b_j}$ and encryption c_j are correspond to the same a_j.
  - $\mathsf{VE.bVerify}(I, \{ek_j, v_j\}_{j\in I}, \mathbf{c}, \pi_{\mathsf{VE}}) \to 1/0$. It verifies the above proof.
  - $\mathsf{VE.Dec}(dk_i, c) \to s$. It decrypts the ciphertext to s.
- The degree check algorithm DegCheck from [19]. It on input n group elements $\{v_j\}_{j\in[n]}$ and two generators g and h checks if there exists two t-degree polynomials ϕ and $\hat{\phi}$, such that $v_j = g^{\phi(j)}h^{\hat{\phi}}(j)$ for all $j \in [n]$.

THE PVSS PROTOCOL. 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 [24], the dealer $\mathcal{P}_d$ sends to each receiver $\mathcal{P}_j$ only the share pair $(\phi(j), \hat{\phi}(j))$, without the full commitment vector $\mathbf{v}$ (Line 3). This reduces the communication cost of the dealing stage from $O(\lambda n^2)$ to $O(\lambda n)$. In the next subsection, we show the 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 produces a publicly-verifiable sharing transcript that is ready to be dispersed.

4.2 Our ADKR Protocol

Construction details. Building upon the PVSS construction described in §4.1, we now present an efficient instantiation of our ADKR design with a focus on high threshold and static security. Nevertheless, variants supporting flexible privacy thresholds and/or adaptive security can be realized through suitable adaptations.

Algorithm 1 Our improved DXT+ PVSS scheme

$\mathsf{Deal}\langle\mathcal{P}_d(\{vk_j, ek_j\}_{j\in[n]}, s, \hat{s}),$
$\{\mathcal{P}_j(sk_j)\}_{j\in[n]}\rangle \to \langle\mathcal{P}_d(\mathsf{transcript}), (\mathsf{aux}_j)_{j\in[n]}\rangle$
// Code run by Dealer $\mathcal{P}_d$
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 $\hat{s}$.
compute $\mathbf{v} \leftarrow \{v_j = g^{\phi(j)}h^{\hat{\phi}(j)}\}_{j\in[0,n]}$
send $\textsc{Share}(\phi(j), \hat{\phi}(j))$ to $\mathcal{P}_j$, for every $j \in [n]$.
upon receiving $2f+1$ valid signatures $\{\sigma_j\}$ for distinct v_j's **do**
  Let Σ be the valid signatures set
  Let I be the indices of parties with missing valid signatures
  $\mathbf{c}, \pi_{\mathsf{VE}} \leftarrow \mathsf{VE.bEncProve}(I, \{ek_j, \phi(j), v_j, \hat{\phi}(j)\}_{j\in I})$
return $\mathsf{transcript} := (\mathbf{v}, \mathbf{c}, \pi_{\mathsf{VE}}, \Sigma, I)$
// Code run by each receiving party $\mathcal{P}_j \in \mathbb{M}$
upon receiving $\textsc{Share}(s_i, \hat{s}_i)$ from $\mathcal{P}_d$ **do**
  compute $v_i \leftarrow g^{s_i}h^{\hat{s}_i}$; $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}_i, v_i)$
  send $\textsc{Ack}(\sigma_i)$ to $\mathcal{P}_d$; **return** $\mathsf{aux}_i = (s_i, \hat{s}_i)$

$\mathsf{Verify}(\{vk_j, ek_j\}_{j\in[n]}, \mathsf{transcript}) \to 0/1$
parse transcript as: $(\mathbf{v}, \mathbf{c}, \pi_{\mathsf{VE}}, \Sigma, I)$
Check $\forall\sigma_j \in \Sigma$ is a valid signature for v_j
Check $\mathsf{DegCheck}(\mathbf{v}, t) = 1$
Check $\mathsf{VE.bVerify}(I, \{ek_j, v_j\}_{j\in I}, \mathbf{c}, \pi_{\mathsf{VE}}) = 1$
if all the checks pass **then return** 1
return 0

$\mathsf{Deliver}(\mathsf{transcript}, dk_i, \mathsf{aux}_i) \to s_i$
if $\mathsf{aux}_i = (s_i, \hat{s}_i)$ **then return** s_i | $(s_i, \hat{s}_i)$
else **return** $s_i \leftarrow \mathsf{VE.Dec}(dk_i, c_i)$ | $(s_i, \hat{s}_i) \leftarrow \mathsf{VE.Dec}(dk_i, c_i)$

Algorithm 2 ADKR, run by parties in both $\mathbb{M}$ and $\widetilde{\mathbb{M}}$

// Code run by each old committee member $\mathcal{P}_i \in \mathbb{M}$
let $store[j] \leftarrow \perp$, $lock[j] \leftarrow \perp$ for $j \in [n]$
initialize a provable dispersal instance $\mathsf{PD}[j]$ for $j \in [n]$
SHARING PHASE (old committee part; new committee part below):
randomly sample $s \leftarrow \mathbb{Z}_p$.
call $\mathsf{PVSS.Deal}[i]\langle\mathcal{P}_i(\{\widetilde{ek}_j, \widetilde{vk}_j\}_{j\in[\widetilde{n}]}, s), \{\widetilde{\mathcal{P}}_i(\widetilde{sk}_i)\}\rangle$ as the dealer
DISPERSAL PHASE (only old committee):
upon obtain $\mathsf{transcript}^{(i)}$ from $\mathsf{PVSS.Deal}[i]$ **do**
  call $\mathsf{PD}[i]\langle\mathcal{P}_i(\mathsf{transcript}^{(i)}), \mathbb{M}\rangle$
upon $\mathsf{PD}[j]$ delivers $store_j$ **do**
  $store[j] = store_j$
upon $\mathsf{PD}[j]$ delivers $lock_j$ **do**
  $lock[j] = lock_j$
  if $lock[j] \neq \perp$ **then** $T_i \leftarrow T_i \cup \{(j, lock[j])\}$
AGREE PHASE (only old committee):
  if $|T_i| = 2f+1$ **then** invoke $\mathsf{Dumbo\text{-}MVBA}\langle\{\mathcal{P}_i(T_i)\}\rangle$
RECAST PHASE (old committee part; new committee part below):
upon Dumbo-MVBA outputs T **do**
  invoke $\mathsf{Coin.Get}()\langle\mathbb{M}\rangle$ and use the coin to uniformly sample a κ-size set $T' = \{\ell_1, \ldots, \ell_\kappa\}$ from $\{\ell_z | (\ell_z, \cdot) \in T\}$
  for $\ell \in T'$ **do**
    if $store[\ell] \neq \perp$ **then** invoke $\mathsf{RC}[\ell]\langle\mathcal{P}_i(store[\ell]), \widetilde{\mathbb{M}}\rangle$

// Code run by each new committee member $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$
let $T^* \leftarrow \perp$, $\mathsf{aux}_i^{(j)} \leftarrow \perp$ for all $j \in [n]$, $\mathsf{count} = 0$, $\mathsf{PKs} \leftarrow \emptyset$
participate in all $\mathsf{PVSS.Deal}[j]$ and $\mathsf{PD}[j]$ for $j \in [n]$
SHARING PHASE (new committee part; old committee part above):
upon $\mathsf{PVSS.Deal}[j]$ returns aux **do**
  $\mathsf{aux}_i^{(j)} \leftarrow \mathsf{aux}$
RECAST PHASE (new committee part; old committee part above):
upon $\mathsf{RC}[\ell]$ returns $\mathsf{transcript}^{(\ell)}$ **do**
  $\mathsf{count} \leftarrow \mathsf{count} + 1$
  if $\mathsf{PVSS.Verify}(\{\widetilde{vk}_j, \widetilde{ek}_j\}_{j\in[\widetilde{n}]}, \mathsf{transcript}^{(\ell)}) = 1$ **then**
    $T^* \leftarrow T^* \cup \{\ell\}$
KEY DERIVATION (only new committee):
  if $\mathsf{count} = \kappa$ **then**
    $\mathsf{CompProve}(\{\mathsf{transcript}^{(\ell)}\}_{\ell\in T^*}, \widetilde{dk}_i, \{\mathsf{aux}_i^{(\ell)}\}_{\ell\in T^*}) \to (\mathsf{tpk}_i, \mathsf{proof}_i, z_i)$
    send $\textsc{Key}(\mathsf{tpk}_i, \mathsf{proof}_i)$ to all $\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}$
upon receiving $\textsc{Key}(\mathsf{tpk}_j, \mathsf{proof}_j)$ from $\widetilde{\mathcal{P}}_j$ for the first time **do**
  if $\mathsf{VrfyComp}(\{\mathsf{transcript}^{(\ell)}\}_{\ell\in T^*}, \mathsf{tpk}_j, \mathsf{proof}_j, \widetilde{ek}_j) = 1$ **then**
    $\mathsf{PKs} \leftarrow \mathsf{PKs} \cup \{\mathsf{tpk}_j\}$
  if $|\mathsf{PKs}| \geq 2\widetilde{f}+1$ **then** Interpolate tpk and any missing tpk_j
    output z_i, tpk and $\{\mathsf{tpk}_j\}_{\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}}$

$\mathsf{Predicate}(T)$ for Dumbo-MVBA
  parse T as : $\{..., (\ell, lock_\ell), ...\}$
  if $|T| = 2f+1$ and $\mathsf{ValidateLock}(\ell, lock_\ell) = 1$ for all items in T **then return** 1
  else return 0

SETUP. Recall the definition of ADKR in §2. There are two sets of participating parties: $\mathbb{M} = \{\mathcal{P}_i\}_{i\in[n]}$ and $\widetilde{\mathbb{M}} = \{\widetilde{\mathcal{P}}_i\}_{i\in[\widetilde{n}]}$. We assume all parties have generated key pairs for the digital signature scheme and the VE scheme during the PKI setup. Let (vk_i, sk_i) and (ek_i, dk_i) (resp. $(\widetilde{vk}_i, \widetilde{sk}_i)$ and $(\widetilde{ek}_i, \widetilde{dk}_i)$) denote signature key pair and encryption key pair of $\mathcal{P}_i \in \mathbb{M}$ (resp. $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$).

INGREDIENTS. Our protocol uses the following ingredients:

- The (n,t)-PVSS protocol (described in §4.1).
- The Dumbo-MVBA protocol [48].
- A variant of the APDB scheme [32, 48], which has a tuple of PD and RC protocols.
- A threshold common coin protocol [15] with an interface denoted by Coin.Get.
- A verifiable computation scheme consisting of a tuple of algorithms (CompProve, VrfyComp) for computing and verifying public key shares tpk_i from a selected subset T of PVSS transcripts. We assume access to the following interfaces:
  - $\mathsf{CompProve}(\{\mathsf{transcript}^{(\ell)}\}_{\ell\in T}, \widetilde{dk}_i, \{\mathsf{aux}_i^{(\ell)}\}_{\ell\in T}) \to (\mathsf{tpk}_i, \mathsf{proof}_i)$: A party $\mathcal{P}_i$ computes its public key share tpk_i and a corresponding proof proof_i, given dk_i and $\{\mathsf{aux}_i^{(\ell)}\}_{\ell\in T}$ as its private witness.
  - $\mathsf{VrfyComp}(\{\mathsf{transcript}^{(\ell)}\}_{\ell\in T}, \mathsf{tpk}_j, \mathsf{proof}_j, \widetilde{ek}_j) \to 0/1$: Verifies the correctness of the public key share.

The algorithmic details of these interfaces are deferred

until after the ADKR protocol description.

Note that for presentation simplicity, we first describe our ADKR protocol assuming that parties in $\mathbb{M}$ 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, APDB, and Coin.Get within $\mathbb{M}$. Later, the variant for flexible $t \geq f$ will be discussed.

HIGH-THRESHOLD ADKR PROTOCOL. With above setup and ingredients, Algorithm 2 presents our ADKR (exemplified in the high-threshold case), which proceeds as follows.

- *Sharing phase (Lines 1-2 for $\mathcal{P}_i \in \mathbb{M}$ and Lines 16-17 for $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$):* Each party $\mathcal{P}_i \in \mathbb{M}$ uniformly samples $s \in \mathbb{Z}_p$ and deals the shares of s to $\widetilde{\mathbb{M}}$ via PVSS.Deal. Each $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ participates in all PVSS.Deal instances and receives $\mathsf{aux}_i^{(j)}$ from each instance.
- *Disperse phase (Lines 3-9 for $\mathcal{P}_i \in \mathbb{M}$):* After having $\mathsf{transcript}^{(i)}$ from PVSS.Deal, each party $\mathcal{P}_i \in \mathbb{M}$ disperses the transcript to $\mathbb{M}$ via $\mathsf{PD}[i]$. $\mathcal{P}_i$ participates in all $\mathsf{PD}[j]$ for $j \in [n]$, and stores the fragments it delivered in these instances.
- *Agree phase (Line 10 for $\mathcal{P}_i \in \mathbb{M}$):* Once $2f+1$ PD instances have completed, 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 11-14 for $\mathcal{P}_i \in \mathbb{M}$ and Lines 18-21 for $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$):* All parties in $\mathbb{M}$ jointly invoke a Coin.Get to decide on a set T' of κ indexes such that $T' \subseteq T$. Then, if $\mathcal{P}_i$ has a fragment from $\mathsf{PD}[\ell]$ for each $\ell \in T'$, it invokes $\mathsf{RC}[\ell]$ to provide the fragment to all parties in $\widetilde{\mathbb{M}}$. Each $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ can thus obtain $\mathsf{transcript}^{(\ell)}$ via $\mathsf{RC}[\ell]$ for every $\ell \in T'$.
- *Key derivation (Lines 22-29 for $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$):* After receiving κ sharing transcripts from the RC protocols, each $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ calculates its secret key share $z_i = \sum_{\ell \in T^*} \mathsf{PVSS.Deliver}(\mathsf{transcript}^{(\ell)}, \widetilde{dk}_i, \mathsf{aux}_i^{(\ell)})$, and runs CompProve to generate the public key share $\mathsf{tpk}_i = g^{z_i}$ along with proof_i attesting that tpk_i is honestly generated. Then, $\widetilde{\mathcal{P}}_i$ multicasts $(\mathsf{tpk}_i, \mathsf{proof}_i)$ to all parties in $\widetilde{\mathbb{M}}$. Finally, after receiving $2\widetilde{f}+1$ valid public key shares (along with their proofs), each $\widetilde{\mathcal{P}}_i$ can obtain the group public key tpk by interpolating these tpk_j's in the exponent.

VERIFIABLE COMPUTATION FOR PUBLIC KEY SHARE. To achieve full secrecy, we now instantiate the verifiable computation scheme $(\mathsf{CompProve}, \mathsf{VrfyComp})$ for computing $\mathsf{tpk}_i = g^{z_i}$ with a proof attesting that tpk_i is correctly computed from a set of PVSS transcripts $\{\mathsf{transcript}^{(\ell)}\}_{\ell \in T}$.

We first need a bit more detail about the underlying 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., $\hat{c}_i = (g^r, \widetilde{ek}_i^{\,r} \cdot h^{\hat{s}_i})$.

With these facts, we develop the verifiable computation scheme in Algorithm 3 w.r.t. our improved PVSS. Note that the computation of tpk_i also gives the secret key share z_i as an intermediate result. To avoid redundant calculation, we let the algorithm CompProve output z_i as well. In this scheme, we employ the following non-interactive zero-knowledge proof-of-knowledge (zk-pok) for dLog and DH tuple:

- $\mathsf{pok}_{\mathsf{dLog}}.\mathsf{prove}(x; g, g^x)$ for proving the knowledge of $x \in \mathbb{Z}_p$ for $g^x \in \mathbb{G}$ w.r.t. a generator g,
- $\mathsf{pok}_{\mathsf{DH}}.\mathsf{prove}(x; g, g^x, h, h^x)$ for proving the knowledge of x for (g^x, h^x) w.r.t. two generators g, h.

We establish the next security properties of the scheme.

Lemma 1. *Conditioned on that the input $\{\mathsf{transcript}^{(\ell)}\}_{\ell \in T}$ are valid, $\{\mathsf{CompProve}, \mathsf{VrfyComp}\}$ in Algorithm 3 satisfies the following properties:*

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

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

For *zero-knowledge*, we can construct a simulator, trivially invoking the black-box simulators for zk-pok of dLog and zk-pok of DH tuple, to simulate a valid proof $(\mathsf{tpk}_i, \mathsf{proof}_i)$ without accessing the witness.

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

Algorithm 3 Verifiable Public Key Share Computation

$\mathsf{CompProve}(\{\mathsf{transcript}^{(\ell)}\}_{\ell\in T}, \widetilde{dk}_i, \{\mathsf{aux}_i^{(\ell)}\}_{\ell\in T})$

Parse $\mathsf{transcript}^{(\ell)}$: $(\mathbf{v}^{(\ell)}, \mathbf{c}^{(\ell)}, \pi_{\mathsf{VE}}^{(\ell)}, \Sigma^{(\ell)}, I^{(\ell)})$
for $\ell \in T$ **do**
  if $\mathsf{aux}_i^{(\ell)} = \perp$ **then** Decrypt $s_i^{(\ell)}, h^{\hat{s}_i^{(\ell)}}$ from $c_i^{(\ell)} = (\bar{c}_i^{(\ell)}, \hat{c}_i^{(\ell)})$
  else Parse $\mathsf{aux}_i^{(\ell)} = (s_i^{(\ell)}, \hat{s}_i^{(\ell)})$
$z_i \leftarrow \sum_{\ell\in T} s_i^{(\ell)}$, and $\forall j \in [\widetilde{n}]$, $v_j \leftarrow \prod_{\ell\in T} v_j^{(\ell)}$
let $S_i = \{\ell | \ell \in T \text{ and } i \in I^{(\ell)}\}$, $S_i' = T \setminus S_i$
$\hat{H}_i \leftarrow \prod_{\ell\in S_i} h^{\hat{s}_i^{(\ell)}}$, $\hat{z}_i' \leftarrow \sum_{\ell\in S_i'} \hat{s}_i^{(\ell)}$

// $\hat{c}_i^{\ell}$ is an ElGamal ciphertext with $\hat{c}_i[0] = g^{r_i}$ and $\hat{c}_i^{(\ell)}[1] = \widetilde{ek}_i^{r_i} \cdot h^{\hat{s}_i}$

$\mathsf{X}_i \leftarrow \prod_{\ell\in S_i} \hat{c}_i^{(\ell)}[0]$, $\mathsf{Y}_i \leftarrow (\prod_{\ell\in S} \hat{c}_i^{(\ell)}[1])/\hat{H}_i$
$\pi_i^*[0] = \mathsf{pok}_{\mathsf{dLog}}.\mathsf{prove}(z_i; g, g^{z_i})$
$\pi_i^*[1] = \mathsf{pok}_{\mathsf{dLog}}.\mathsf{prove}(\hat{z}_i'; h, h^{\hat{z}_i'})$
$\pi_i^*[2] = \mathsf{pok}_{\mathsf{DH}}.\mathsf{prove}(\widetilde{dk}_i; g, \widetilde{ek}_i, \mathsf{X}_i, \mathsf{Y}_i))$
$\mathsf{proof}_i = (h^{\hat{z}_i'}, \mathsf{Y}_i, (\pi_i^*[0], \pi_i^*[1], \pi_i^*[2]))$
return ($\mathsf{tpk}_i = g^{z_i}$, proof_i, z_i)

$\mathsf{VrfyComp}(\{\mathsf{transcript}^{(\ell)}\}_{\ell\in T}, \mathsf{tpk}_j, \mathsf{proof}_j, \widetilde{ek}_j)$

Compute X_j and v_j from $\{\mathsf{transcript}^{(\ell)}\}_{\ell\in T}$ as in $\mathsf{CompProve}$
Parse $\mathsf{proof}_j = (h^{\hat{z}_j'}, \mathsf{Y}_j, (\pi_j^*[0], \pi_j^*[1], \pi_j^*[2]))$
Compute $\hat{H}_j = (\prod_{\ell\in S} \hat{c}_j^{(\ell)}[1])/\mathsf{Y}_j$
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**
  return 1
return 0

Variants. While Algorithm 2 focuses on a high-threshold ADKR with static security (also practical as later demonstrated in §6), it can be easily adapted to the following variants, which may be of independent (theoretic) interest:

- ADKR **with a flexible privacy 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 [22, 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).

- ADKR **with adaptive security.** First, it can be instantiated with adaptive security and a flexible threshold $t \in [f, 2f]$ for group-element secret, using the PVSS scheme from [9]. Moreover, when $t = f$, we can provide an adaptive instantiation for field-element secret and dLog cryptosystem, as detailed in Appendix C.

- ADKG **in the coin-hybrid model.** When $\mathbb{M} = \widetilde{\mathbb{M}}$, Algorithm 2 describes an ADKG assisted by a common-coin oracle, given that the need for existing (n,t)-dLog threshold setup is replaced with a silent-setup threshold signature scheme. [6]

[6] With recent advancements in silent-setup common coin protocols [29], this yields a quadratic ADKG under the PKI setup.

Concrete optimization by distributed PVSS verification. In the above protocol, for each transcript, every new committee member independently verifies all $\widetilde{n} - \widetilde{f}$ signatures and calls bVerify to verify $\widetilde{f}$ encrypted shares corresponding to the set I. It is clearly redundant and costly, as all parties perform κn same group-exponentiation operations to verify κ PVSS transcripts, making a major performance bottleneck.

Nonetheless, we can further reduce the verification cost as follows: instead of verifying all $\widetilde{n}$ items in the transcript, we let each party verify $2\widetilde{f}+1$ consecutive items. For example, $\mathcal{P}_i$ verifies the signature or encrypted share of the items indexed by $\{j \mod \widetilde{n}\}_{j\in[i,i+2\widetilde{f}]}$ in the transcript. So if successfully verifying these $\widetilde{n} - \widetilde{f}$ items assigned to it, each new committee member multicasts a bit of 1 (represents all these $\widetilde{n} - \widetilde{f}$ items valid); otherwise, it multicasts a bit of 0 (indicating the sharing transcript is invalid). Then, for each item in the rest $\widetilde{f}$ items that are not assigned to locally verify, the party waits for messages from other parties (that are actually assigned to verify the item) such that at least $\widetilde{f}+1$ of these messages carry the same bit b: If the bit b is 1, it is secure to mark the item as valid; and if the bit b is 0, it is secure to conclude that the sharing transcript is invalid. Finally, when all items are verified as valid (either locally or distributedly), the sharing transcript is also verified.

This approach reduces the computation cost of verifying the sharing transcript by about $1/3$, at a concretely and asymptotically small price: *one* round of all-to-all communication in which each node just multicasts a *single* bit.

4.3 Analysis

Number of sharing transcripts selected to recover. To keep communication and computation costs within $O(n^2)$, we select a κ-size sub-committee 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 ensure the pseudo-randomness of the generated secret key, it suffices to have *one* recovered transcript in the sub-committee to 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 in the output set of MVBA. Thus, the probability that a κ-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 $f+1$, κ is the number of "draw" (committee size), and 0

represents the bad event of no "success". In Fig. 3, we plot the required committee size as a function of the total number of parties $n = 3f+1$, for given probabilities p.

As the total number of parties increases, a constant-sized committee can ensure the presence of at least one honest party except for the given probabilities $1-p$.

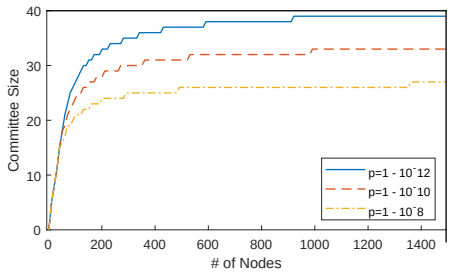


Figure 3: Sub-committee size κ to ensure some selected dispersal from a non-corrupted sender with probability p.

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

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 dealer and initiates one PD protocol as sender while participating in other PD instances as non-sender. In this phase, each party performs $O(n+\widetilde{n})$ group-exponentiation operations. During the consensus phase, the number of group exponentiation for each party is $O(n+\widetilde{n})$. For parties in $\widetilde{\mathbb{M}}$, each party verifies κ sharing transcripts, each containing $O(\widetilde{n})$ items, resulting in $O(\kappa\widetilde{n})$ group-exponentiation operations. The number of group-exponentiation operations in the key derivation phase is $O(\widetilde{n}\log\widetilde{n})$ (assuming DFT in the exponent). Therefore, the number of group exponentiation operations per party is $\tilde{O}(\kappa\widetilde{n}+\widetilde{n}\log\widetilde{n}+n)$.

Security analysis. We establish the following theorem regarding the security of ADKR in Algorithm 2.

Theorem 1. *The protocol shown in Algorithm 2 realizes* ADKR *in a fully asynchronous network model with $n/3$ static corruptions in $\mathbb{M}$ and $\widetilde{n}/3$ static corruptions in $\widetilde{\mathbb{M}}$, conditioned on the hardness of the Discrete Log (dLog) problem and that the underlying primitives are all secure.*

Below, we will first prove two intermediate results in Lemma 2 and 3. Based on them, we prove termination, key validity, and full secrecy in Lemma 4, 5, and 6, 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 κ 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 $(\widetilde{n},\widetilde{t})$-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 *(Algorithm 2), then each honest $\widetilde{\mathcal{P}}_i\in\widetilde{\mathbb{M}}$ will compute an aggregate commitment v_j for all $\widetilde{\mathcal{P}}_j\in\widetilde{\mathbb{M}}$ (line 5 in Alg. 3).*

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 a 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 their *store* values. These parties will send their *store* values to parties in $\widetilde{\mathbb{M}}$ by invoking RC$[\ell]$ 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 eliminating invalid PVSS transcripts (via PVSS.Verify), every honest party in $\widetilde{\mathbb{M}}$ generate an aggregate commitment v_j for all $\widetilde{\mathcal{P}}_j\in\widetilde{\mathbb{M}}$ during the execution of CompProve. □

Lemma 3. *For any two honest parties $\widetilde{\mathcal{P}}_i$ and $\widetilde{\mathcal{P}}_j$ that generate the set of aggregate commitments $\mathbf{v}=\{v_j\}_{\widetilde{\mathcal{P}}_j\in\widetilde{\mathbb{M}}}$ and $\mathbf{v}'=\{v'_j\}_{\widetilde{\mathcal{P}}_j\in\widetilde{\mathbb{M}}}$ respectively in the* ADKR *protocol (line 5 in Algorithm 3), then $\mathbf{v}=\mathbf{v}'$.*

Proof. 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 transcript$^{(\ell)}$ 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 $\mathsf{RC}[\ell]$, and no honest party will input to any $\mathsf{RC}[\ell]$ that $\ell \notin T'$ or input to a $\mathsf{RC}[\ell]$ 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 an RC instance only if it receives at least $f+1$ valid fragments for the same vc, all honest parties in $\widetilde{\mathbb{M}}$ can only output from $\mathsf{RC}[\ell]$ that $\ell \in T'$.

So we prove the lemma by contradiction. Assuming that there exist two honest parties in $\widetilde{\mathbb{M}}$ that generate aggregate commitment set $\mathbf{v} \neq \mathbf{v}'$. Since any honest party's commitment set is aggregated from the transcripts outputs of RC instances, $\widetilde{\mathcal{P}}_i$ and $\widetilde{\mathcal{P}}_j$ must receive different outputs $\mathsf{transcript}^{(\ell)} \neq \mathsf{transcript}'^{(\ell)}$ from at least one $\mathsf{RC}[\ell]$ that $\ell \in T'$. Recall that all valid *store* values in one RC instance must be consistent with 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 $\mathbf{v} = \{v_j\}_{\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}}$ will generate the same one. □

Lemma 4 (Termination). *If all honest parties in $\mathbb{M}$ and $\widetilde{\mathbb{M}}$ invoke* ADKR *(Algorithm 2), then every honest party $\widetilde{\mathcal{P}}_i$ in $\widetilde{\mathbb{M}}$ will output the public keys $(g^z, g^{z_1}, g^{z_2}, \cdots, g^{z_{\widetilde{n}}})$ and a private key share z_i.*

Proof. From Lemmas 2, 3, every honest party in $\widetilde{\mathbb{M}}$ will generate the same aggregate commitment set $\{v_j\}_{\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}}$. According to the protocol description, each honest $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ will output its private key z_i, public key share $\mathsf{tpk}_i = g^{z_i}$ and proof_i from CompProve. Then honest $\widetilde{\mathcal{P}}_i$ will send a KEY message containing tpk_i with proof_i. Each proof_i generated by honest parties in $\widetilde{\mathbb{M}}$ can be verified by any honest party in $\widetilde{\mathbb{M}}$ according to the *completeness* in Lemma 1. So each honest party in $\widetilde{\mathbb{M}}$ can collect at least $2\widetilde{f}+1$ valid tpk_j corresponding to the aggregated commitment set. 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 $\mathsf{tpk} = g^z$ and all missing public key shares. □

Lemma 5 (Key Validity). *The outputs of Algorithm 2 provide the same public keys $(g^z, g^{z_1}, g^{z_2}, \cdots, g^{z_{\widetilde{n}}})$. Any private key share z_i carried by the outputs is consistent with the public key of g^{z_i}. In addition, any $2\widetilde{f}+1$-subset of private key shares consistent with public keys can recover the same polynomial and therefore the same private key z.*

Proof. From Lemma 3, any honest party in $\widetilde{\mathbb{M}}$ generates the same set of aggregate commitments $\mathbf{v} = \{v_j\}_{\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}}$. From the description of the ADKR (Algorithm 2) and CompProve (Algorithm 3), the commitment set $\mathbf{v}$ is aggregated from the same set of verified $\mathsf{transcript}^{(\ell)}$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 $\mathsf{transcript}^{(\ell)}$. So the private key share z_j for each $\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}$ is on the polynomial of degree on more than $2\widetilde{f}$. Therefore, any $2\widetilde{f}+1$-subset of private key shares will recover the same polynomial and thus the same z.

From the description of the protocol, each $\mathsf{transcript}^{(\ell)}$ verified by PVSS.Verify satisfies that $\{v_j^{(\ell)}\}_{\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}}$ is a Pedersen commitment to the sharing polynomial of degree no more than $2\widetilde{f}$. Note that the private key shares correspond to a unique polynomial $\phi(\cdot)$ (and $\hat{\phi}(\cdot)$) of degree no more than $2\widetilde{f}$. Then $\{v_j\}_{\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}}$ is a Pedersen commitment to $\phi(\cdot)$ (and $\hat{\phi}(\cdot)$) since the Pedersen commitment is additively holomorphic.

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

Each honest party in $\widetilde{\mathbb{M}}$ receives a KEY message from $\widetilde{\mathcal{P}}_j$ containing $\mathsf{tpk}_j = g^{z_j}$ and a proof_j which demonstrates that $\widetilde{\mathcal{P}}_j$ knows: z_j, the sum of all $\hat{z}_j^{(\ell)}$ for $\sigma_j^{(\ell)} \in \Sigma^{(\ell)}$, and the product of all $h^{\hat{z}_j^{(\ell)}}$ for $c_j^{(\ell)} \in \mathbf{c}^{(\ell)}$.

With 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 $2\widetilde{f}+1$ shares g^{z_j}. □

Lemma 6 (Full Secrecy). *No computationally bounded static 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 in the* ADKR *of Algorithm 2.*

Proof. We prove the full secrecy of our ADKR protocol by constructing a PPT simulator $\mathcal{S}$ that, given as input a uniformly random group element $y \in \mathbb{G}$, can simulate an execution of ADKR where y serves as the group public key. We show that any PPT adversary $\mathcal{A}$, statically corrupting up to $f = \lfloor \frac{n-1}{3} \rfloor$ parties in $\mathbb{M}$ and $\widetilde{f} = \lfloor \frac{\widetilde{n}-1}{3} \rfloor$ parties in $\widetilde{\mathbb{M}}$, cannot distinguish the simulated execution from a real execution that outputs y as the group public key tpk. In addition to the internal states of corrupted parties, $\mathcal{A}$ is also granted access to secret key shares of a static set of $\widetilde{n} - 2\widetilde{f} - 1$ honest parties in $\widetilde{\mathbb{M}}$ (since we consider high-threshold privacy, so $\widetilde{t} - \widetilde{f} = \widetilde{n} - 2\widetilde{f} - 1$).

We describe the simulator $\mathcal{S}$ 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 parties in $\mathbb{M}$ and $\widetilde{\mathbb{M}}$ are represented as G and $\widetilde{G}$. We also denote the set of honest parties whose secret shares are leaked to the adversary as $\widetilde{L} \subset \widetilde{G}$. As $\mathcal{A}$ is static, we assume w.l.o.g. that $\widetilde{B} = \{\widetilde{\mathcal{P}}_1, ..., \widetilde{\mathcal{P}}_{\widetilde{f}}\}$, $\widetilde{G} = \{\widetilde{\mathcal{P}}_{\widetilde{f}+1}, ..., \widetilde{\mathcal{P}}_{\widetilde{n}}\}$ (with B and G defined similarly), and $\widetilde{L} = \{\widetilde{\mathcal{P}}_{\widetilde{f}+1}, \ldots, \widetilde{\mathcal{P}}_{2\widetilde{f}}\}$

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

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

Simulating PVSS.Deal*:* For $i \in [1, 2\widetilde{f}]$, $\mathcal{S}$ uniformly samples $s_i^{(i^*)}$ and $\hat{s}_i^{(i^*)}$, and computes $v_i^{(i^*)} = g^{s_i^{(i^*)}} \cdot h^{\hat{s}_i^{(i^*)}}$. $\mathcal{S}$ uniformly samples $v_0^{(i^*)}$. For all $i \in [2\widetilde{f}+1, \widetilde{n}]$, interpolate $v_i^{(i^*)} = \prod_{j \in [0,2\widetilde{f}]} (v_j^{(i^*)})^{\lambda_j(i)}$, where $\lambda_j(x) = \prod_{k \in [0,2\widetilde{f}], k \neq j} \frac{j-x}{j-k}$.

Then, $\mathcal{S}$ sends $(v_i^{(i^*)}, s_i^{(i^*)}, \hat{s}_i^{(i^*)})$ to each $\widetilde{\mathcal{P}}_i \in \widetilde{B} \cup \widetilde{L}$, on behalf of $\mathcal{P}_{i^*}$. $\mathcal{S}$ simply sends $(v_i^{(i^*)}, \perp)$ to honest parties in $\widetilde{G} \setminus \widetilde{L}$. On behalf of $\widetilde{\mathcal{P}}_i \in \widetilde{G}$, $\mathcal{S}$ signs each $v_i^{(i^*)}$, and sends the signature to $\mathcal{P}_{i^*}$ on behalf of honest $\widetilde{\mathcal{P}}_i$.

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

Regarding other honest parties, $\mathcal{S}$ honestly executes the protocol on their behalf in PVSS.Deal instances, except for their responses to the messages from $\mathcal{P}_{i^*}$.

Simulating the Dispersal Phase: $\mathcal{S}$ acts honestly on behalf of each honest party by executing the protocol until Dumbo-MVBA outputs.

Simulating the Agree Phase: When Dumbo-MVBA outputs T (Line-11, Algorithm 2), if $(i^*, \cdot)$ is not included in T, $\mathcal{S}$ shall **abort** the current simulated execution, sample a new $i^* \leftarrow G$, rewind $\mathcal{A}$ to the beginning of the simulation, and **restart** a new simulated execution. If $(i^*, \cdot)$ is included in T, $\mathcal{S}$ continues to simulate the subsequent phases.

Simulating the Recast Phase: For Coin.Get, recall that the coin value is $\mathsf{RO}(\mathsf{ThldSig})$. $\mathcal{S}$ computes the unique threshold signature Thld on the default message of this execution (since $\mathcal{S}$ knows the secret keys of all honest parties), and *programs* RO such that $i^* \in T' = \mathsf{RO}(\mathsf{ThldSig})$. $\mathcal{S}$ honestly executes the protocol on behalf of honest parties for the subsequent operations of the Recast Phase.

Simulating the Key Derivation: First, $\mathcal{S}$ computes the public key shares of honest parties as follows:

- For $j \in [1, 2\widetilde{f}]$, computes $\mathsf{tpk}_j^{(i^*)} = g^{s_j^{(i^*)}}$.
- For $j \in [\widetilde{f}+1, \widetilde{n}]$, $\bar{z}_j = \sum_{\ell \in T^* \setminus \{i^*\}} s_j^{(\ell)}$, $\bar{\mathsf{tpk}}_j = g^{\bar{z}_j}$.
- Interpolate $\bar{\mathsf{tpk}}$ from $\{\bar{\mathsf{tpk}}_j\}$, $j \in [\widetilde{f}+1, n]$.
- Compute $\mathsf{tpk}_0^{(i^*)} = y / \bar{\mathsf{tpk}}$.
- Interpolate $\mathsf{tpk}_j^{(i^*)}$ for $j \in [2\widetilde{f}+1, \widetilde{n}]$ from $\{\mathsf{tpk}_j^{(i^*)}\}_{j \in [0, 2\widetilde{f}]}$
- For $\widetilde{\mathcal{P}}_j \in \widetilde{G}$, compute $\mathsf{tpk}_j = \mathsf{tpk}_j^{(i^*)} \cdot \bar{\mathsf{tpk}}_j$.

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

- Define $S_i = \{\ell | \ell \in T^* \text{ and } i \in I^{(\ell)}\}$, and $S_i' = T^* \setminus S_i$. Recall that T^* is the set of identities selected by the common coin and with valid transcripts (see Line-21, Algorithm 2); I^ℓ is the set of identities included in $\mathsf{transcript}^{(\ell)}$, which suggests that the encrypted shares of parties in I^ℓ are in the transcript (see Lines 6-7, Algorithm 1).
- If $i^* \in S_i$, compute $\hat{z}_i' = \sum_{\ell \in S_i'} \hat{s}_i^{(\ell)}$, where $\hat{s}_i^{(\ell)}$ was sent by $\mathcal{P}_\ell$ during the Sharing Phase.
- If $i^* \in S_i'$, compute $\hat{z}_i'' = \sum_{\ell \in S_i', \ell \neq i^*} \hat{s}_i^{(\ell)}$, $h^{\hat{s}_i^{(i^*)}} = \frac{v_i^{(i^*)}}{\mathsf{tpk}_j^{(i^*)}}$, and $h^{\hat{z}_i'} = h^{\hat{z}_i''} \cdot h^{\hat{s}_i^{(i^*)}}$.
- Invoke the CompProve simulator with $(\mathsf{tpk}_i, h^{\hat{z}_i'})$ as input and obtain proof_j.

For each honest party $\widetilde{\mathcal{P}}_j \in \widetilde{L}$, the computation proof proof_j and the threshold public key share tpk_j can be computed honestly. With $(\mathsf{tpk}_i, \mathsf{proof}_i)$ for all honest parties in $\widetilde{G}$, $\mathcal{S}$ honestly executes the rest of the protocol. And the secret key shares of parties in $\widetilde{L}$ can be honestly computed and provided to the adversary in the end of the execution.

Public key in the simulated execution. Now we demonstrate that the final public key tpk in the simulated execution is the input of $\mathcal{S}$, namely, y. First, all tpk_j for parties in $\widetilde{G}$ are computed as $\mathsf{tpk}_j^{(i^*)} \cdot \bar{\mathsf{tpk}}_j$. Since $y = \mathsf{tpk}_0^{(i^*)} \cdot \bar{\mathsf{tpk}}$ and both $\mathsf{tpk}_0^{(i^*)}$ and $\bar{\mathsf{tpk}}$ are interpolated from $2\widetilde{f}+1$ shares, y can be interpolated from $2\widetilde{f}+1$ $\mathsf{tpk}_j^{(i^*)}$. For any $\widetilde{\mathcal{P}}_j \in \widetilde{B}$, it knows z_j for an opening of tpk_j. From the *knowledge soundness* of NIZK and the binding property of Pedersen commitment, all tpk_j' with a valid proof_j are on the $2\widetilde{f}$-degree polynomial defined by $\{\mathsf{tpk}_j\}$ that $\widetilde{\mathcal{P}}_j \in \widetilde{G}$. Thus, the output of $\mathcal{S}$ 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 the real execution, by considering the following hybrids.

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

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

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

Hybrid 3: It is identical to **Hybrid 2**, except that during PVSS.Deal$[i^*|\mathsf{old}]$ and PVSS.Deal$[i^*|\mathsf{new}]$, 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 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 beginning, and restarts the simulated execution with a freshly sampled $i^* \leftarrow G$.

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

Hybrid 6: It is identical to **Hybrid 5**, except that during the key derivation phase, $\mathcal{S}$ invokes the CompProve simulator to generate proof_i on behalf of all $\widetilde{\mathcal{P}}_i \in \widetilde{G}$.

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 κ indexes in T', while in Hybrid 1, the output must include at least one honest $i' \in G$. Since the probability that κ 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 parties 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 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

We now present an ADPSS scheme that achieves near-quadratic communication, extending our ADKR protocol.

Definition 2 (ADPSS)**.** *In an $(n,t,\tilde{n},\tilde{t})$-ADPSS protocol between two sets $\mathbb{M}$ and $\widetilde{\mathbb{M}}$ of parties, with $|\mathbb{M}| = n$ 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 $\mathcal{P}_i \in \mathbb{M}$ 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 $\mathcal{P}_i \in \widetilde{\mathbb{M}}$ that terminates will output a secret share m'_i of secret m, such that parties in $\widetilde{\mathbb{M}}$ can eventually reconstruct a secret $m' = m$.*
- Secrecy: *No PPT adversary can learn any information of the secret m. Notably, this holds even when the adversary can additionally access the secret key shares of up to $t - f - 1$ honest parties in $\mathbb{M}$ and $\tilde{t} - \tilde{f} - 1$ honest parties in $\widetilde{\mathbb{M}}$, after all honest parties in $\widetilde{\mathbb{M}}$ have received the output shares.*

The above ADPSS definition aligns with that in [53], which focuses on a one-shot resharing to handle the handover between old and new committees. We also note that although [53] and other ADPSS studies such as [14, 41] target a mobile adversary that can periodically rotate the set of corrupted nodes during a long-term execution, the formal modeling of such mobile corruption over a long-term execution in a fully asynchronous setting (which lacks a global physical clock and requires a sequence of repeatedly occurring resharings rather than a single resharing) remains largely open. Moreover, Alexandru et al. [6] established an impossibility result for certain ADPSS protocols and pointed out that all existing ADPSS protocols may all rely on implicit extra assumptions (e.g., synchronous windows or proactively secure channels) in long-term executions.

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.

SYNTAX OF DCRG. 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 values $s, \hat{s}$, s.t. 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. Concretely, at the end of the

execution, parties in $\mathbb{M}$ agree on a vector of Pedersen commitments $(g^s h^{\hat{s}}, g^{s_1} h^{\hat{s}_1}, \ldots, g^{s_n} h^{\hat{s}_n})$, 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 parties are (n,t) Shamir secret sharing of s and $\hat{s}$, respectively. At the same time, parties in $\widetilde{\mathbb{M}}$ outputs $(g^s h^{\hat{s}}, g^{s'_1} h^{\hat{s}'_1}, \ldots, g^{s_{\widetilde{n}}} h^{\hat{s}_{\widetilde{n}}})$, and each $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ receives a private output $(s'_i, \hat{s}'_i)$, s.t. $\{s'_i\}$, $\{\hat{s}'_i\}$ of honest parties are (n,t) Shamir secret sharing of s, $\hat{s}$, respectively.

ADPSS PROTOCOL BASED ON DCRG. 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 $\{\mathsf{cm}_j\}_{\mathcal{P}_j \in \mathbb{M}}$, where $\mathsf{cm}_j = g^{m_j} h^{\hat{m}_j}$. 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 $\{\widetilde{\mathsf{cm}}_j\}_{\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}}$.

At a high level, there are three steps to realize this goal. (1) Parties in $\mathbb{M}$ and $\widetilde{\mathbb{M}}$ jointly invoke a DCRG instance to generate two sets of secret shares for the random numbers r and $\hat{r}$. (2) Parties in $\mathbb{M}$ uses shares of r and $\hat{r}$ to mask the shares of m and $\hat{m}$. Then, they reconstruct the masked values $m+r$ and $\hat{m}+\hat{r}$, and multicast to parties in $\widetilde{\mathbb{M}}$. Since all parties have the same view of all corresponding Pedersen commitments, all parties can obtain the same reconstructed values. (3) After receiving the masked values, each $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ can obtain its share $m'_i = (m+r) - r'_i$ and $\hat{m}'_i = (\hat{m}+\hat{r}) - \hat{r}'_i$, and the corresponding commitments can be updated homomorphically. The ADPSS protocol is presented in Algorithm 4.

Algorithm 4 ADPSS run by parties in both $\mathbb{M}$ and $\widetilde{\mathbb{M}}$

// Code run by each $\mathcal{P}_i \in \mathbb{M}$
initialize $S \leftarrow \emptyset$
$\mathcal{P}_i$ takes m_i, $\hat{m}_i$ and $\{\mathsf{cm}_j\}_{j \in \mathbb{M}}$ as input
1: invoke DCRG as a party in $\mathbb{M}$
2: **wait** for DCRG output $r_i, \hat{r}_i, \{v_j\}_{\mathcal{P}_j \in \mathbb{M}}$
3: **multicast** REC$(m_i + r_i, \hat{m}_i + \hat{r}_i)$ to all parties in $\mathbb{M}$
4: **upon** receiving REC$(\mathsf{sh}_j, \hat{\mathsf{sh}}_j)$ from $\mathcal{P}_j \in \mathbb{M}$ **do**
5: **if** $\mathsf{cm}_j \cdot v_j = g^{\mathsf{sh}_j} \cdot h^{\hat{\mathsf{sh}}_j}$ **then** $S \leftarrow S \cup \{(j, (\mathsf{sh}_j, \hat{\mathsf{sh}}_j))\}$
6: **if** $|S| \geq 2f+1$ **then**
7: Interpolate $m+r$ and $\hat{m}+\hat{r}$ from S
8: **multicast** TRANS$(m+r, \hat{m}+\hat{r})$ to all parties in $\widetilde{\mathbb{M}}$

// Code run by each $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$
9: invoke DCRG as a party in $\widetilde{\mathbb{M}}$
10: **wait** for DCRG output $r'_i, \hat{r}'_i, \{v'_j\}_{\mathcal{P}_j \in \widetilde{\mathbb{M}}}$
11: **upon** receiving $f+1$ TRANS messages with the same $(m+r)$ and $(\hat{m}+\hat{r})$ from distinct parties in $\mathbb{M}$ **do**
12: $m'_i \leftarrow (m+r) - r'_i$, $\hat{m}'_i \leftarrow (\hat{m}+\hat{r}) - \hat{r}'_i$
13: **for** $\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}$ **do** $\mathsf{cm}'_j \leftarrow (g^{m+r} h^{\hat{m}+\hat{r}}) / v'_j$
14: **output** m'_i, $\hat{m}'_i$ and $\{\mathsf{cm}'_j\}_{\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}}$

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.

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.

SETUP AND INGREDIENTS. 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 $\hat{s}$.

This 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 [24], the VE scheme from [38] suffices for this purpose. This variant of PVSS is also described in Algorithm 1 (including dash-boxed items, excluding boxed items).

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

- Each $\mathcal{P}_i \in \mathbb{M}$ samples two secrets s and $\hat{s}$, and it invokes PVSS.Deal$[i|\mathsf{old}]$ (with threshold t) and PVSS.Deal$[i|\mathsf{new}]$ (with threshold $\widetilde{t}$) to share them with $\mathbb{M}$ and $\widetilde{\mathbb{M}}$, respectively. All parties in both $\mathbb{M}$ and $\widetilde{\mathbb{M}}$ can obtain the corresponding auxiliary information.
- After obtaining $\mathsf{transcript}^{(i|\mathsf{old})}$ and $\mathsf{transcript}^{(i|\mathsf{new})}$ from PVSS.Deal$[i|\mathsf{old}]$ and PVSS.Deal$[i|\mathsf{new}]$, respectively, $\mathcal{P}_i \in \mathbb{M}$ disperses them (as a single message) to $\mathbb{M}$ via PD$[i]$. Every $\mathcal{P}_j \in \mathbb{M}$ participates in these PD instances and stores the associated 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 κ indexes T', such that all indexes in T' have been included in T. Then, parties in $\mathbb{M}$ recast both $\mathsf{transcript}^{(\ell|\mathsf{old})}$ and $\mathsf{transcript}^{(\ell|\mathsf{new})}$ for all $\ell \in T$ to all parities in $\mathbb{M} \cup \widetilde{\mathbb{M}}$.
- After receiving $\mathsf{transcript}^{(\ell|\mathsf{old})}$ and $\mathsf{transcript}^{(\ell|\mathsf{new})}$ from RC$[\ell]$, each party checks whether they are valid and if they contain the same $v_0^{(\ell)}$ (which ensures the two transcripts sharing the same pair of $s^{(\ell)}$ and $\hat{s}^{(\ell)}$). Let $T^* = \{\ell\}$ such that $\mathsf{transcript}^{(\ell|\mathsf{old})}$ and $\mathsf{transcript}^{(\ell|\mathsf{new})}$ passed the check. Each $\mathcal{P}_i \in \mathbb{M}$ (resp. $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$) processes $\{\mathsf{transcript}^{(\ell|\mathsf{old})}\}_{\ell \in T^*}$ (resp. $\{\mathsf{transcript}^{(\ell|\mathsf{new})}\}_{\ell \in T^*}$) 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

Algorithm 5 DCRG, run by parties in both $\mathbb{M}$ and $\widetilde{\mathbb{M}}$

// Code run by each old committee member $\mathcal{P}_i \in \mathbb{M}$

let $store[j] \leftarrow \perp$, $\mathsf{aux}_i^{(j)}$ $lock[j] \leftarrow \perp$ for $j \in [n]$. $\mathsf{count} \leftarrow 0$

initialize a provable dispersal instance $\mathsf{PD}[j]$ and $\mathsf{PVSS.Deal}[j|\mathsf{old}]$ for $j \in [n]$

1: randomly sample $s, \hat{s}$.
2: call $\mathsf{PVSS.Deal}[i|\mathsf{old}]\langle \mathcal{P}_i(\{ek_i, vk_i\}_{i\in[n]}, s, \hat{s}), \{\mathcal{P}_i(sk_i)\}\rangle$
3: call $\mathsf{PVSS.Deal}[i|\mathsf{new}]\langle \mathcal{P}_i(\{\widetilde{ek}_i, \widetilde{vk}_i\}_{i\in[\widetilde{n}]}, s, \hat{s}), \{\widetilde{\mathcal{P}}_i(\widetilde{sk}_i)\}\rangle$
4: **upon** $\mathsf{PVSS.Deal}[j|\mathsf{old}]$ returns aux **do**
5: $\quad \mathsf{aux}_i^{(j)} \leftarrow \mathsf{aux}$
6: **upon** obtain $\mathsf{transcript}^{(i|\mathsf{old})}$ from $\mathsf{PVSS.Deal}[i|\mathsf{old}]$ and $\mathsf{transcript}^{(i|\mathsf{new})}$ from $\mathsf{PVSS.Deal}[i|\mathsf{new}]$ **do**
7: $\quad$ call $\mathsf{PD}[i]\langle \mathcal{P}_i(\mathsf{transcript}^{(i|\mathsf{old})}|\mathsf{transcript}^{(i|\mathsf{new})}), \mathbb{M}\rangle$

Then, Execute Lines 5-14 in Algorithm 2 while replacing Line 14 there with Line 8 below:

8: $\quad$ **if** $store[\ell] \neq \perp$ **then** invoke $\mathsf{RC}[\ell]\langle \mathcal{P}_i(store[\ell]), \mathbb{M}\cup\widetilde{\mathbb{M}}\rangle$ ▷ Recast to all parties
9: **upon** $\mathsf{RC}[\ell]$ returns $\mathsf{transcript}^{(\ell|\mathsf{old})}|\mathsf{transcript}^{(\ell|\mathsf{new})}$ **do**
10: $\quad \mathsf{count} \leftarrow \mathsf{count} + 1$
11: $\quad$ parse $\mathsf{transcript}^{(\ell|\mathsf{old})} = (\{v_j^{(\ell|\mathsf{old})}\}_{j\in[n]}, \cdot)$ and $\mathsf{transcript}^{(\ell|\mathsf{new})} = (\{v_j^{(\ell|\mathsf{new})}\}_{j\in[\widetilde{n}]}, \cdot)$
12: $\quad$ **if** both $\mathsf{transcript}^{(\ell|\mathsf{old})}$ and $\mathsf{transcript}^{(\ell|\mathsf{new})}$ are valid, and $v_0^{(\ell|\mathsf{old})} = v_0^{(\ell|\mathsf{new})}$ **then**
13: $\quad\quad T^* \leftarrow T^* \cup \{\ell\}$
14: $\quad\quad (s_i^{(\ell)}, \hat{s}_i^{(\ell)}) \leftarrow \mathsf{PVSS.Deliver}(\mathsf{transcript}^{(\ell|\mathsf{old})}, dk_i, \mathsf{aux}_i^{(j)})$
15: $\quad$ **if** $\mathsf{count} = \kappa$ **then**
16: $\quad\quad$ **return** $(\{\prod_{\ell\in T^*} v_j^{(\ell|\mathsf{old})}\}_{j\in[n]}, \sum_{\ell\in T^*} s_i^{(\ell)}, \sum_{\ell\in T^*} \hat{s}_i^{(\ell)})$

// Code run by each new committee member $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$

let $T^* \leftarrow \perp$, $\mathsf{aux}_i^{(j)} \leftarrow \perp$ for all $j \in [n]$, $\mathsf{count} = 0$, $\mathsf{PKs} \leftarrow \emptyset$

17: participate in all $\mathsf{PVSS.Deal}[j|\mathsf{new}]$ and $\mathsf{PD}[j]$ for $j \in [n]$
18: **upon** $\mathsf{PVSS.Deal}[j|\mathsf{new}]$ returns aux **do**
19: $\quad \mathsf{aux}_i^{(j)} \leftarrow \mathsf{aux}$

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

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 [7].

5.2 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* $O(\lambda\kappa(n+\widetilde{n})^2)$ *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 $O(\lambda(n+\widetilde{n}))$ instead of $O(\lambda\widetilde{n})$, resulting in a communication complexity of $O(\kappa\lambda n^2 + \kappa\lambda n\cdot\widetilde{n})$. Multicasting COMMIT messages cost $O(\kappa\lambda n\cdot\widetilde{n})$. For parties in $\widetilde{\mathbb{M}}$, the DCRG costs $O(\lambda\widetilde{n}^2 + \kappa\lambda n\cdot\widetilde{n})$ bits.

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

Computation Complexity. Each party in $\mathbb{M}$ performs $O(\kappa n + \widetilde{n})$ 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 $O(\kappa\widetilde{n})$ in the DCRG protocol, followed by $O(\widetilde{n})$ exponentiations to compute cm'_j for all $\mathcal{P}_j \in \widetilde{\mathbb{M}}$ Therefore, the overall computation complexity of ADPSS is $O(\kappa n + \kappa\widetilde{n})$ per party.

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

Lemma 7. *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 $\mathsf{cm}_i \cdot v_i = g^{m_i+r_i} h^{\hat{m}_i+\hat{r}_i}$ (condition in Line 5 of Algorithm 4). Therefore, all honest nodes in $\mathbb{M}$ can obtain S such that $|S| \geq 2f+1$ (condition in Line 6), so they can recover $(m+r, \hat{m}+\hat{r})$ (Line 7) and multicast it to all parties in $\widetilde{\mathbb{M}}$; then, they can terminate.

Regarding each party $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$, 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 11 and then terminate. □

Lemma 8. *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, 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 ϕ and $\hat{\phi}$, and $v_j^{\mathsf{old}} = g^{\phi(j)} h^{\hat{\phi}(j)}$. 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 ϕ' and $\hat{\phi}'$, and $v_j^{\mathsf{old}} = g^{\phi(j)} h^{\hat{\phi}(j)}$. And every honest $\widetilde{\mathcal{P}}_i \in \widetilde{\mathbb{M}}$ obtains $(\phi'(i), \hat{\phi}'(i))$. Moreover, since we require $v_0^{(\ell|\mathsf{old})} = v_0^{(\ell|\mathsf{new})}$ for all $\ell \in T^*$, $v_0 = \prod_{\ell\in T^*} v_0^{(\ell|\mathsf{old})} = v'_0$.

[7] 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.

So $g^{\phi(0)}h^{\hat{\phi}(0)} = g^{\phi'(0)}h^{\hat{\phi}'(0)}$. Due to the binding property of 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 $\mathsf{cm}_j \cdot v_j = g^{m_i+r_i}h^{\hat{m}_i+\hat{r}_i}$ 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 $\widetilde{t}$-degree polynomial $F(x) = (m+r) - \phi'(x)$. So, $F(0) = m + r - \phi'(0) = m$, which means $\{m'_i\}_{i\in[\widetilde{n}]}$ are $(\widetilde{n}, \widetilde{t})$ Shamir shares of m. Following a similar argument, we can show $\{\hat{m}'_i\}_{i\in[\widetilde{n}]}$ are $(\widetilde{n}, \widetilde{t})$ Shamir shares of $\hat{m}$. □

Lemma 9. *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 $\mathcal{S}$, 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 $\lfloor\frac{\widetilde{n}-1}{3}\rfloor$ nodes in $\widetilde{\mathbb{M}}$ cannot distinguish the simulated execution and a real execution where the commitment to the secret is cm. In addition to the internal states of corrupted parties, $\mathcal{A}$ is also granted access to secret key shares of a static set of $n-2f-1$ honest parties in $\mathbb{M}$ and $\widetilde{n}-2\widetilde{f}-1$ honest parties in $\widetilde{\mathbb{M}}$.

We describe the simulator $\mathcal{S}$ 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 $\widetilde{G}$. We also denote the set of honest parties whose secret shares are leaked to the adversary as $L \subset G$ and $\widetilde{L} \subset \widetilde{G}$. As $\mathcal{A}$ is static, we assume w.l.o.g. that 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 $\mathbb{M}$. $\mathcal{S}$ generates encryption key pairs (ek_i, dk_i) and the signature key pairs for all parties in $\widetilde{\mathbb{M}}$. After that, $\mathcal{S}$ simulates an execution as follows.

Initial configuration in $\mathbb{M}$*:* For all $i \in [t]$, $\mathcal{S}$ uniformly samples m_i and $\hat{m}_i$, and computes $\mathsf{cm}_i = g^{m_i} \cdot h^{\hat{m}_i}$. For all $i \in [t+1, n]$, interpolate $\mathsf{cm}_i = \prod_{j\in[0,t]} \mathsf{cm}_j^{\lambda_j(i)}$, where $\mathsf{cm}_0 = \mathsf{cm}$ 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$[i^*]$, $\mathcal{S}$ samples t-degree polynomials φ and $\hat{\varphi}$, and $\widetilde{t}$-degree polynomials φ' and $\hat{\varphi}'$, such that $\varphi(0) = \varphi'(0)$ and $\hat{\varphi}(0) = \hat{\varphi}'(0)$. Then, $\mathcal{S}$ computes $v_0^{(i^*)} = \frac{g^{\varphi(0)} \cdot h^{\hat{\varphi}(0)}}{\mathsf{cm}_0}$,

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

and $v_i^{(i^*)'} = \frac{g^{\varphi'(i)} \cdot h^{\hat{\varphi}'(i)}}{\mathsf{cm}_i}$ for all $i \in [\widetilde{n}]$.

Then, $\mathcal{S}$ sends $(v_i^{(i^*)}, (m_i + \varphi(i), \hat{m}_i + \hat{\varphi}(i)))$ to each $\mathcal{P}_i \in \mathbb{B} \cup L$, and $(v_i^{(i^*)'}, (m_i + \varphi'(i), \hat{m}_i + \hat{\varphi}'(i)))$ to each $\widetilde{\mathcal{P}}_i \in \widetilde{B} \cup \widetilde{L}$, on the behalf of $\mathcal{P}_{i^*}$

$\mathcal{S}$ simply sends $(v_i^{(i^*)}, \perp)$ or $(v_i^{(i^*)'}, \perp)$ to honest nodes. On the behalf of each honest $\mathcal{P}_i \in G \setminus L$ and $\widetilde{\mathcal{P}}_i \in \widetilde{G} \setminus \widetilde{L}$, $\mathcal{S}$ 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, $\mathcal{S}$ 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 $S' \in [\widetilde{n}]$ with $|S'| = 2\widetilde{f}+1$. For every $j \notin S$ (resp $j \notin S'$), if $j \in B \cup L$ (resp. $j \in \widetilde{B} \cup \widetilde{L}$), $\mathcal{S}$ 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, $\mathcal{S}$ encrypts a random value under ek_j (resp. $\widetilde{ek}_j$) and simulates a proof. By doing so, $\mathcal{S}$ can obtain $\mathsf{transcript}(i^*|\mathsf{old})$ and $\mathsf{transcript}(i^*|\mathsf{new})$.

Regarding other honest nodes, $\mathcal{S}$ 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, $\mathcal{S}$ acts on the behalf of each honest nodes by executing the protocol. When Dumbo-MVBA outputs T', if $(i^*, \cdot)$ is not included in T', $\mathcal{S}$ shall **abort** the current simulated execution, sample a new $i^* \leftarrow G$, rewind $\mathcal{A}$ to the beginning of the simulation, and **restart** a new simulated execution. If $(i^*, \cdot)$ is included in T', $\mathcal{S}$ 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). $\mathcal{S}$ computes the unique threshold signature Thld on the default message of this execution (since $\mathcal{S}$ knows the secret keys of all honest nodes), and *programs* RO such that $i^* \in$ RO(ThldSig).

- For operations after GetCoin, $\mathcal{S}$ acts on the behalf of each honest nodes by executing the protocol. In particular, on the behalf of each honest $\mathcal{P}_i \in G$ and $\widetilde{\mathcal{P}}_i \in \widetilde{G}$, $\mathcal{S}$ obtains $(r_i^{(\ell)}, \hat{r}_i^{(\ell)})$ and $(r_i^{(\ell)'}, \hat{r}_i^{(\ell)'})$ for all $\ell \in T^*, \ell \neq i^*$; It skips the deliver operation for $\ell = i^*$.

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

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

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

$$\begin{aligned} \mathsf{cm}_i \cdot v_i &= \mathsf{cm}_i \cdot v_i^{(i^*)} \cdot \prod_{\ell \in T^*, \ell \neq i^*} v_i^{(\ell)} \\ &= g^{\varphi(i) + \sum_{\ell \in T^*, \ell \neq i^*} r_i^{(\ell)}} \cdot h^{\hat{\varphi}(i) + \sum_{\ell \in T^*, \ell \neq i^*} \hat{r}_i^{(\ell)}} = g^{\mathsf{sh}_i} h^{\hat{sh}_i}. \end{aligned} \tag{4}$$

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, rewinds the adversary $\mathcal{A}$ to the beginning, 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$[i^*|\mathsf{old}]$ and PVSS.Deal$[i^*|\mathsf{new}]$, 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 beginning, 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 $i^* \in \mathsf{RO}(\mathsf{ThldSig})$.

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 κ indexes in T', while in Hybrid 1, the output must include at least one honest $i' \in G$. Since the probability that κ 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

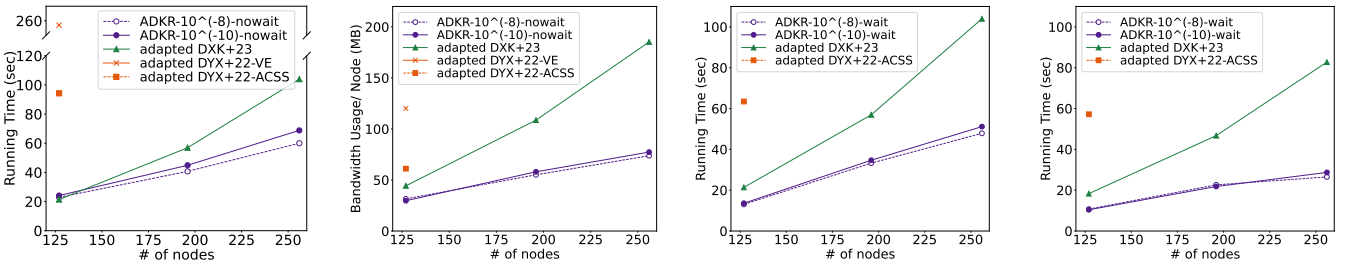
This section demonstrates the concrete efficiency and scalability of our ADKR through large-scale experiments.

6.1 Instantiations and Test Environment

Implementation. We implemented the high-threshold version of our ADKR protocol, along with two ADKRs adapted from the high-threshold ADKG protocols in [25] and [23] with simplifications to use existing threshold common coins generated by the old committee. In addition, we implemented two versions of the protocol adapted from [25]. Note that the ACSS scheme in [25] is based on a VE scheme. In the first version (denoted as "adapted DYX+22-VE"), we replaced the original VE with the VE for Pedersen commitments proposed in [24]. In the second version (denoted as "adapted DYX+22-ACSS"), we replaced the entire ACSS in [25] with DXT+ ACSS (with our improvement in Algorithm 1). Both versions are implemented and evaluated to show the benefits of each improvement individually.

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 [54] and its Python wrapper as described in [25]. We assume that the parties in the old committee have established a high-threshold cryptosystem capable of performing non-interactive threshold BLS signatures. 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 256, we consider two choices of sub-committee size κ, 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



(a) Running time.

(b) Bandwidth usage.

(c) Running time, if wait for δ time to solicit signatures while dealing.

(d) Running time in a good case if interpolations of PK shares skipped.

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

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.

Table 3: Sub-committee size for varying n under different failure probabilities (e.g., 10^{-8} and 10^{-10}).

	# of nodes (n)			Failure
	127	196	256	probability
Any-trust committee size (κ)	22	23	24	10^{-8}
	26	28	29	10^{-10}
Honest-majority committee size (c)	79	107	125	10^{-8}
	85	117	139	10^{-10}

Table 3 lists the corresponding sub-committee sizes under p_1 and p_2 for $n = 127$, 196, and 256. We note that although GDK ADKR [41] also employs sub-committee election to reduce communication overhead, its approach requires significantly larger sub-committees to ensure honest-majority guarantees at the same probabilities. In the network scales we consider, the sub-committee size c required by GDK ADKR becomes comparable to $2f+1$. As a result, GDK ADKR offers little performance advantage at these scales (compared to the trivial simplification of selecting a $2f+1$-sized subset to run ACSS), and we therefore omit it from our performance comparison. [8]

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 δ after distributing the shares, instead of proceeding immediately after receiving $2f+1$ signatures. Here δ is set to 1 sec for $n = 127$ and 196, and 2 sec for $n = 256$. We also adopted another good-case optimization from [23], 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.

[8] We note that [13,53] primarily focus on APSS rather than ADKR. These approaches are not only asymptotically more expensive (cf. Table 2) but also concretely more costly than a simplified cubic ADKG, due to the additional overhead of maintaining the same secret key. Therefore, we exclude them from the baseline comparisons for ADKR.

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 8 GB 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 over 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.

Moreover, across node scales from 127 to 256, our protocol consistently consumes less bandwidth than the adapted DXK+22. [9] The bandwidth advantage becomes increasingly

[9] Careful readers may notice a discrepancy between the bandwidth usage of the adapted DXK+23 protocol in our measurements and [23]. This difference arises because [23] implemented RBC with a fast path in a "very good case", while we implemented RBC from [16] (which has no good-case path but causes less communication in the bad case).

evident as the scale grows: the usage is 71.1% to 39.8% of that of the adapted DXK+22 with p_1; and ranges from 61.7% to 41.7% with p_2.

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

During the interactive dealing phase of generating PVSS transcript, we introduce a waiting period of length δ (e.g., 1–2 sec as aforementioned) in both our protocol and the adapted DYX+22-ACSS protocol to collect additional signatures. This significantly improves good-case 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 $\widetilde{f}$ public key shares are skipped (in all protocols), our performance advantage is 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 measured the local CPU time to evaluate the computation overhead of each protocol at larger scales. The result aligns with our complexity analysis: our protocol shows a slower growth trend in computation cost as the network scale increases. 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.

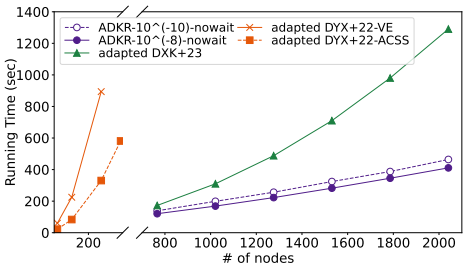


Figure 5: Computation time (without PK interpolations).

7 Conclusion and Future Work

We presented an efficient protocol for *asynchronous distributed key reconfiguration* (ADKR), which reduces the communication complexity from $O(n^3)$ to $O(\kappa n^2)$ with $\kappa \approx 30$ while preserving adaptive security, if the underlying publicly verifiable secret sharing (PVSS) scheme is adaptively secure. Our protocol follows a *share-dispersal–then–agree–and–recast* paradigm and introduces two key optimizations: (i) linear-communication interactive PVSS generation, and (ii) distributed PVSS verification. In a 256-node deployment, our implementation achieves up to 40% reduction in latency, compared with ADKGs simplified to the reconfiguration setting. Furthermore, the underlying techniques extend beyond ADKR, offering improvements for other distributed primitives, for instance, enabling an *asynchronous dynamic proactive secret sharing* (ADPSS) with near-quadratic communication complexity per resharing.

A natural direction for future work is to further optimize communication and computation, particularly by eliminating the κ factor in the asymptotic cost. Another open question is whether adaptive security can be achieved without relying on memory-erasure assumptions. In addition, the new paradigm of leveraging existing threshold cryptosystems to bootstrap new ones presents promising opportunities for extension to broader settings, which we believe is an exciting direction for future exploration.

Acknowledgements

The work of Yingzi and Yuan is supported in part by the National Key R&D Project of China (2022YFB2701600) and in part by the CAS Project for Young Scientists in Basic Research (YSBR-035). The work of Hanwen and Qiang is supported in part by ARC Discovery Project (DP250101739), and research awards from the Ethereum Foundation, Protocol Labs, and the Stellar Development Foundation. The work of Jing Xu is supported in part by the Natural Science Foundation of China (62172396 and 62572460).

References

[1] 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 *CRYPTO*, pages 39–70, 2023.

[3] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *PODC*, 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 *IEEE S&P*, 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] Andreea B Alexandru, Erica Blum, Jonathan Katz, and Julian Loss. State machine replication under changing network conditions. In *Asiacrypt*, pages 681–710, 2022.

[7] 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*, 2024.

[8] Renas Bacho and Julian Loss. On the adaptive security of the threshold BLS signature scheme. In *CCS*, pages 193–207. ACM, 2022.

[9] Renas Bacho and Julian Loss. Adaptively secure (aggregatable) pvss and application to distributed randomness beacons. In *CCS*. ACM, 2023.

[10] 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.

[11] 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 *CCS*, 2024.

[12] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *PKC*, pages 31–46, 2002.

[13] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *CCS*, pages 88–97. ACM, 2002.

[14] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, pages 524–541, 2001.

[15] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3), 2005.

[16] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *Proc. SRDS 2005*, pages 191–201, 2005.

[17] Ran Canetti, Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Adaptive security for threshold cryptosystems. In *CRYPTO*, 1999.

[18] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, pages 42–51, 1993.

[19] Ignacio Cascudo and Bernardo David. Scrape: Scalable randomness attested by public entities. In *ACNS*, pages 537–556. Springer, 2017.

[20] Arka Rai Choudhuri, Sanjam Garg, Julien Piet, and Guru-Vamsi Policharla. Mempool privacy via batched threshold encryption: Attacks and defenses. In *USENIX Security*, 2024.

[21] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *EuroSys*, pages 34–50, 2022.

[22] 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 *CCS*, pages 356–370. ACM, 2023.

[23] 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.

[24] Sourav Das, Zhuolun Xiang, Alin Tomescu, Alexander Spiegelman, Benny Pinkas, and Ling Ren. Verifiable secret sharing simplified. In *IEEE S&P 2025*.

[25] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *IEEE S&P 2022*, pages 2518–2534.

[26] 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.

[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. In *PODC*, pages 469–479. ACM, 2024.

[28] Hanwen Feng, Tiancheng Mai, and Qiang Tang. Scalable and adaptively secure any-trust distributed key generation and all-hands checkpointing. In *CCS*, pages 2636–2650. ACM, 2024.

[29] Hanwen Feng and Qiang Tang. Asymptotically optimal adaptive asynchronous common coin and DKG with silent setup. In *CRYPTO*, pages 642–675, 2025.

[30] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2), 1985.

[31] Pierre-Alain Fouque and Jacques Stern. One round threshold discrete-log key generation without private channels. In *PKC*, pages 300–316, 2001.

[32] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-NG: Fast asynchronous bft consensus with throughput-oblivious latency. In *CCS*, pages 1187–1201. ACM, 2022.

[33] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. In *IEEE 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 *IEEE S&P 2024*, pages 3034–3052.

[35] Sanjam Garg, Dimitris Kolonelos, Guru-Vamsi Policharla, and Mingyuan Wang. Threshold encryption with silent setup. In *CRYPTO*, pages 352–386, 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 *EUROCRYPT*, 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 *EUROCRYPT*, pages 370–400, 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 *DSN*, 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*, pages 147–176, 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 *CCS*, pages 4286–4300. ACM, 2024.

[45] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, 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 *CCS*, 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 *PODC*, 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 *PODC*, pages 129–138, 2020.

[49] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *CCS*, pages 31–42. ACM, 2016.

[50] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, pages 129–140. Springer, 1991.

[51] Tian Qiu and Qiang Tang. Predicate aggregate signatures and applications. In *Asiacrypt*, 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 *PODC*, 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 *USENIX Security*, pages 5413–5430, 2023.

[54] zkcrypto. Pairing: Pairing-friendly elliptic curve library. https://github.com/zkcrypto/pairing, 2025. Accessed: 2025-10-10.

A Additional 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 [7, 26].

Recent major advances [7, 27] 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 [7, 27] focus on DKG without a common-coin setup, [28] and [41] explored coin-hybrid 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). Moreover, as previously discussed, [41] also requires an honest-majority committee of several hundred nodes, limiting its practical performance.

A recent concurrent work [1] extends the recursive method of [7, 27] to the asynchronous setting using verifiable random functions (VRFs) [10], 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 moderate-scale networks of a few hundred nodes. Moreover, theirs produces group-element secrets, while ours gives field-element secrets. Moreover, [29] presented a quadratic-communication adaptive common coin protocol based on silent threshold cryptography [35], which, when combined with our coin-hybrid ADKG (as a variant of our ADKR instantiation), yields a quadratic-communication ADKG with a silent setup.

Finally, we note that the paradigm of coin-hybrid DKG resembles the idea of coin-tossing extension [5] which was studied for circumventing the impossibility of coin tossing in the *dishonest-majority* setting. ADKR (particularly its variant of coin-hybrid ADKG) can be viewed as an extension of the underlying asynchronous coin-tossing protocol, offering better amortized performance (i.e., even if the underlying coin-tossing protocol is very expensive, it can extend these coin-tossing protocols to generate multiple common coins with reduced amortized complexity).

B 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 $\mathsf{SignShare}(sk_i, \cdot)$ 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:

- $\mathsf{TSIG.Setup}(\lambda, t, n) \to (\mathsf{pk}, \mathsf{sk})$. Given t, n and the cryptographic security parameter λ, the algorithm/protocol generates the public keys pk and a vector of secret key shares $\mathsf{sk} = (s_1, \cdots, s_n)$, where pk is published and s_i is exclusively obtained by $\mathcal{P}_i$.
- $\mathsf{TSIG.SignShare}(s_i, m) \to \rho_i$. On input a message m and a secret key share s_i, this algorithm outputs a partial signature share ρ_i for the message m.
- $\mathsf{TSIG.VerifyShare}(m, (i, \rho_i)) \to 0/1$. Given a message m, a partial signature ρ_i and an index i (along with the implicit input of public keys pk), this algorithm outputs 1 (accept) or 0 (reject).
- $\mathsf{TSIG.Combine}(m, \{(i, \rho_i)\}_{i\in S}) \to \sigma/\perp$. Given the implicit input of public keys pk, a message m, and a set of indexed partial signatures $\{(i, \rho_i)\}_{i\in S}$ with $|S| > t$ and $S \subset \mathbb{M}$, this algorithm outputs a full signature σ for message m (or a special symbol $\perp$).
- $\mathsf{TSIG.Verify}(m, \sigma) \to 0/1$. 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 λ, 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 $\mathsf{Coin.Get}(\cdot)$ 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 $\mathsf{Coin.Get}(x)$ using the same input x, then any honest party invoking $\mathsf{Coin.Get}(x)$ will obtain some output r from $\mathsf{Coin.Get}(x)$, despite the influence of the adversary.
- *Consistency*. For any two honest parties, if they obtain r and r' from $\mathsf{Coin.Get}(x)$ 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 $\mathsf{Coin.Get}(x)$, then $|\Pr[r' = r : r \leftarrow \mathsf{Coin.Get}(x) \text{ and } r' \leftarrow \mathcal{A}(\mathsf{view})] - \frac{1}{2^\lambda}| \leq \varepsilon(\lambda)$.

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 $\mathbb{M}$ input some values satisfying Q, then all honest parties in $\mathbb{M}$ 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.

- $\mathsf{VE.Setup}(1^\lambda, \mathsf{Cm}) \to pp_{\mathsf{VE}}$. Input security parameter λ and a commitment scheme Cm, the algorithm outputs public parameter pp_{VE}.
- $\mathsf{VE.KeyGen}(pp_{\mathsf{VE}}) \to (\{ek\}, \{dk\})$. Input public parameters, the algorithm outputs the encryption keys ek and decryption keys dk of the encryption scheme.
- $\mathsf{VE.EncAndProve}(pp_{\mathsf{VE}}, ek, s, v, w) \to (c, \pi_{\mathsf{VE}})$.Input message s, commitment v and witness w, where $v, w \leftarrow \mathsf{Cm.}\,\textsc{Commit}(s)$. The algorithm outputs the encryption c of the tuple $(s, \pi = \mathsf{Cm.Open}(v, s, w))$ and a correct encryption NIZK proof π_{VE}.
- $\mathsf{VE.Dec}(dk, c) \to s, \pi$. Input the ciphertext c and a decryption key dk, the algorithm outputs a decryption of c using dk.
- $\mathsf{VE.Verify}(pp_{\mathsf{VE}}, ek, v, c, \pi_{\mathsf{VE}}) \to 0/1$. Input public parameters pp_{VE}, encryption key ek, commitment v, ciphertext c and its proof π_{VE}. If π_{VE} is a valid proof that α, π exist, and $\alpha, \pi = \mathsf{VE.Dec}(dk, c)$ and $\mathsf{Cm.Verify}(v, \alpha, \pi) = 1$, the algorithm outputs 1.
- $\mathsf{VE.bEncProve}(pp_{\mathsf{VE}}, I, \{ek\}, s, v, w) \to (\mathbf{c}, \pi_{\mathsf{VE}})$. 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 $s_i \in s$, along with a NIZK proof π_{VE} that satisfy VE.bVerify.
- $\mathsf{VE.bVerify}(pp_{\mathsf{VE}}, I, \{ek\}, v, \mathbf{c}, \pi_{\mathsf{VE}}) \to 0/1$. Input parameters pp_{VE}, a set of encryption keys ek, commitments v, encryptions c and a proof π_{VE} the algorithm outputs 1 if π_{VE} is a valid proof that, for each $i \in I$ there exists (α_i, π_i) such that $\alpha_i, \pi_i = \mathsf{VE.Dec}(dk_i, c_i)$ and $\mathsf{PC.Verify}(v, \alpha_i, \pi_i) = 1$.

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 π_{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.

Algorithm 6 PD protocol, with identifier ID and sender $\mathcal{P}_s$

/* Protocol for the sender $\mathcal{P}_s$ */
upon receiving the input value m **do**
  $\{m_j\}_{j\in n} \leftarrow \mathsf{Enc}(m)$, $\mathsf{vc} \leftarrow \mathsf{VCom}(\{m_j\})$
  for each $j \in [\mathbb{M}]$ **do**
    $\pi_j \leftarrow \mathsf{Open}(\mathsf{vc}, m_j, j)$
    let $store := \langle \mathsf{vc}, m_j, j, \pi_j \rangle$
    send $\textsc{Store}(\mathsf{ID}, store)$ to $\mathcal{P}_j$
  wait until $|S| = 2f+1$
    $\sigma_{\mathsf{PD}} \leftarrow \mathsf{TSIG.Combine}(\langle \textsc{Stored}, \mathsf{ID}, \mathsf{vc}\rangle, S)$
    let $lock := \langle \mathsf{vc}, \sigma_{\mathsf{PD}} \rangle$ and **multicast** $\textsc{Dispersal}(\mathsf{ID}, lock)$ to all $\mathcal{P}_i \in \mathbb{M}$
upon receiving $\textsc{Stored}(\mathsf{ID}, \rho_j)$ from $\mathcal{P}_j$ for the first time **do**
  if $\mathsf{TSIG.VerifyShare}(\langle \textsc{Stored}, \mathsf{ID}, \mathsf{vc}\rangle, (j, \rho_j)) = 1$ **then**
    $S \leftarrow S \cup (j, \rho_j)$

/* Protocol for each party $\mathcal{P}_i \in \mathbb{M}$ */
upon receiving $\textsc{Store}(\mathsf{ID}, store)$ from sender $\mathcal{P}_s$ for the first time **do**
  if $\mathsf{ValidateStore}(i, store) = 1$ **then**
    deliver *store* and parse *store* it as $\langle \mathsf{vc}, m_i, i, \pi_i \rangle$
    $\rho_i \leftarrow \mathsf{TSIG.SignShare}(sk_i, \langle \textsc{Stored}, \mathsf{ID}, \mathsf{vc}\rangle)$
    send $\textsc{Stored}(\mathsf{ID}, \rho_i)$ to $\mathcal{P}_s$
upon receiving $\textsc{Dispersal}(\mathsf{ID}, lock)$ from $\mathcal{P}_s$ for the first time **do**
  if $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$ **then**
    deliver *lock*

$\mathsf{ValidateStore}(i', store)$:
  parse *store* as: $\langle \mathsf{vc}, m_i, i, \pi_i \rangle$
  return $\mathsf{VerifyOpen}(\mathsf{vc}, m_i, i, \pi_i) \wedge i = i'$
$\mathsf{ValidateLock}(\mathsf{ID}, lock)$:
  parse *lock* as: $\langle \mathsf{vc}, \sigma_{\mathsf{PD}} \rangle$
  return $\mathsf{TSIG.VerifyThld}(\langle \textsc{Stored}, \mathsf{ID}, \mathsf{vc}\rangle, \sigma_{\mathsf{PD}})$

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 $\mathbb{M}$ and a set of receiving parties denoted by $\mathbb{M}'$. The honest parties in $\mathbb{M}$ take *store* and *lock* as their input (if they have obtained *store* and *lock* from the PD phase), and aim to help $\mathbb{M}'$ 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 $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$;
- *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 $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$, 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.

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

/* Protocol for each $\mathcal{P}_i \in \mathbb{M}$ */
upon receiving input *store* **do**
  multicast $\textsc{RcStore}(\mathsf{ID}, store)$ to all parties in $\mathbb{M}'$

/* Protocol for each party $\mathcal{P}_i \in \mathbb{M}'$ */
upon receiving $\textsc{RcStore}(\mathsf{ID}, store)$ from sender $\mathcal{P}_j \in \mathbb{M}$ **do**
  if $\mathsf{ValidateStore}(i, store) = 1$ **then**
    parse *store* as: $\langle \mathsf{vc}, m_j, j, \pi_j \rangle$
    $C[\mathsf{vc}] \leftarrow (C[\mathsf{vc}] \cup (j, m_j))$
    if $|C[\mathsf{vc}]| = f+1$ **then**
      $v \leftarrow \mathsf{Dec}(C[\mathsf{vc}])$
      if $\mathsf{VCom}(\mathsf{Enc}(v)) = \mathsf{vc}$ **then**
        return v
      else return $\perp$

C Adpatively Secure ADKR

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 [8] 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 $\mathsf{DL}_g(\cdot)$.

Formally, the termination property and key validity defined in Sect.2 naturally extend to the adaptive setting. We recall the secrecy definition from [8] below, originally defined for DKG but applicable to DKR with minimal modifications.

Definition 3 (Oracle-aided Algebraic Simulatability)**.** *A protocol Π satisfies k-oracle-aided algebraic simulatability if for every PPT adversary $\mathcal{A}$ adaptively corrupts at most f parties in $\mathbb{M}$ and $\widetilde{f}$ parties in $\widetilde{\mathbb{M}}$, there exists a PPT simulator $\mathcal{S}^{\mathcal{A}}$ 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 Π for $\mathcal{A}$. In particular,*

- ***Algebraic oracle queries:*** *When $\mathcal{S}^{\mathcal{A}}$ 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 $\mathsf{view}_{\mathcal{A},y,\Pi}$ the view of $\mathcal{A}$ in an execution of Π outputting $pk = y$. $\mathcal{S}^{\mathcal{A}}$ can output $\mathsf{simview}_{\mathcal{A}}$ and pk, such that $(\mathsf{view}_{\mathcal{A},y,\Pi}, y)$ is computationally indistinguishable with $(\mathsf{simview}_{\mathcal{A}}, pk)$.*

- ***Invertible simulatability martrix:*** *Let g_i denote the i-th query by $\mathcal{S}$ to $\mathsf{DL}_g(\cdot)$. 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*

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

C.1 Choices of Adaptively Secure PVSS

The protocol in §4 only achieves 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 [9] 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 elliptic-curve 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. [7] 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 [7]. As summarized in [7], 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 κ 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 [7] 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 [24]), 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 [24] for better efficiency for the "low" threshold case. For completeness, we include a description of the PVSS protocol in Algorithm 8.

C.2 Instantiation and Analysis (with $t = f$)

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. The adaptive instantiation largely follows §4, but relies on a non-interactive high-threshold signature scheme without DKG setup, which can be instantiated with any adaptively secure scheme with a silent setup, such as [22, 34, 51]. The PVSS is instantiated using the variant based on [24], as described above. Additionally, since Feldman commitments are employed, each party can locally derive tpk and $\{\mathsf{tpk}_j\}_{j\in[\tilde{n}]}$ during the key derivation phase, as detailed in Algorithm 9.

The underlying threshold common coin must be adaptive. For the ADKR setting, it can be instantiated with an adaptively secure unique threshold signature scheme [8, 15]. 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 [29].

To achieve adaptive security, we assume *memory erasures* [17], requiring each node $\mathcal{P}_i$ in the old committee to *erase*

Algorithm 8 A Variant of PVSS in [24]

$\mathsf{Deal}\langle \mathcal{P}_d(\{vk_j\}_{j\in[n]}, s), \{\mathcal{P}_j(sk_j)\}_{j\in[n]}\rangle \rightarrow$
$\langle \mathcal{P}_d(\mathsf{transcript}), (\mathsf{aux}_j)_{j\in[n]}\rangle$
// Code run by Dealer $\mathcal{P}_d$
1: randomly sample a t-degree polynomial $\phi(\cdot)$ where $\phi(0)$ is s.
2: compute $\mathbf{v} \leftarrow \{v_j = g^{\phi(j)}\}_{j\in[0,n]}$
3: **for** $\mathcal{P}_j \in \mathbb{M}$ **do**
4: **send** $\textsc{Share}(\phi(j))$
5: **upon** receiving $2f+1$ valid signatures $\{\sigma_j\}$ for distinct v_j's **do**
6: **Let** Σ be the valid signatures set
7: **Let** I be the indices of nodes with missing valid signatures
8: **return** $\mathsf{transcript} \leftarrow (\mathbf{v}, \Sigma, \{\phi(i)\}_{i\in I})$
// Code run by each $\mathcal{P}_i \in M$
9: **upon** receiving $\textsc{Share}(s_i)$ from $\mathcal{P}_d$ **do**
10: $\sigma_i \leftarrow \mathsf{Sign}(\mathsf{sk}_i, v_i = g^{s_i})$
11: **send** $\textsc{Ack}(\sigma_i)$
12: **return** $\mathsf{aux}_i = s_i$

$\mathsf{Verify}(\{vk_j\}_{j\in[n]}, \mathsf{transcript}) \rightarrow 0/1$
13: parse $\mathsf{transcript}$ as: $(\mathbf{v}, \Sigma, I)$
14: **Check** $\forall \sigma_j \in \Sigma$ is a valid signature for v_j
15: **Check** $\mathsf{DegCheck}(\mathbf{v}, t) = 1$
16: **if** all the checks pass **then return** 1
17: **return** 0

$\mathsf{Deliver}(\mathsf{transcript}, \mathsf{aux}_i) \rightarrow s_i$
18: **if** $\mathsf{aux}_i = s_i$ **then return** s_i
19: **else return** s_i from $\mathsf{transcript}$

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

Each $\mathcal{P}_i \in \widetilde{\mathbb{M}}$ has obtained $\{\mathsf{transcript}^{(\ell)}\}_{\ell\in T^*}$, and $\mathsf{transcript}^{(\ell)}$ 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: $\mathsf{tpk}_j \leftarrow \prod_{\ell\in T^*} v_j^{(\ell)}$ for all $\mathcal{P}_j \in \widetilde{\mathbb{M}}$
3: $\mathsf{tpk} = \prod_{j\in[t]} \mathsf{tpk}_j^{\lambda_j}$, where λ_j is the Lagrange coefficient.
4: **return** $(\mathsf{tpk}, (\mathsf{tpk}_i)_{i\in[n]}, z_i)$

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, and Lemma 5. In the following, we prove our protocol satisfies the oracle-aided algebraic simulatability.

Lemma 10. *When the underlying* PVSS *is the signature-based* 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 $\mathcal{S}$ in the following. $\mathcal{S}$ takes as inputs $2\widetilde{f}+1$ random group elements and can make up to $2\widetilde{f}$ queries to DLog_g.

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

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

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

$\mathcal{S}$ runs the protocol with $\mathcal{A}$ by acting on 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 behalf of all nodes in $\mathcal{H} \cap \mathbb{M}$ except $\mathcal{P}_{i^*}$, it honestly executes the protocol.
- On behalf of $\mathcal{P}_{i^*}$, 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 $\mathsf{DLog}_g(\cdot)$ with v_j, and obtains a_j such that $v_j = g^{a_j}$. It sends $\textsc{Share}(a_j)$ to all $\mathcal{P}_j \in \mathcal{C} \cap \widetilde{\mathbb{M}}$, and sends $(v_j, \perp)$ to all honest nodes in $\widetilde{\mathbb{M}}$.

  Then, after receiving $(2\widetilde{f}+1)$ valid signatures from distinct nodes in $\widetilde{M}$ (the signature set is denoted as Σ), it queries requires the DLog oracle $\mathsf{DLog}_g(\cdot)$ with v_j for all $\mathcal{P}_j \in I$ (if v_j has not been queried before), where $\{\mathcal{P}_j\}_{j\in I}$ is the set of f nodes in $\widetilde{\mathbb{M}}$ which have not returned the signature. It obtains $\{a_j\}_{j\in I}$ accordingly. Finally, it prepares the script as $(v, \Sigma, (a_j)_{j\in I})$. After that, it executes the protocol honestly on behalf of $\mathcal{P}_{i^*}$.
- On 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 $\mathcal{P}_{i^*} \in \mathbb{M}$ without verifying the openings.

Simulating the Dispersal Phase: $\mathcal{S}$ acts honestly on behalf of each honest party by executing the protocol until Dumbo-MVBA outputs.

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

Simulating the Recast Phase: For Coin.Get, recall that the coin value is $\mathsf{RO}(\mathsf{ThldSig})$. $\mathcal{S}$ computes the unique threshold signature Thld on the default message of this execution (since $\mathcal{S}$ knows the secret keys of all honest parties), and *programs* RO such that $i^* \in T' = \mathsf{RO}(\mathsf{ThldSig})$. $\mathcal{S}$ honestly executes the protocol on behalf of honest parties for the subsequent operations of the Recast Phase.

Simulating verification and key derivation: On 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 invoking PD), then $\mathcal{S}$ **aborts** the current simulated execution, samples a

new $i^* \leftarrow [n]$, and **rewinds** $\mathcal{A}$ to the beginning of the protocol. If $\mathcal{P}_j \in \widetilde{\mathbb{M}}$ is corrupted, it queries the DLog oracle $\mathsf{DLog}_g(\cdot)$ with v_j (if v_j has not been queried before) and obtains $a_j \in \mathbb{Z}_p$. It then returns all internal states of $\mathcal{P}_j$ 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.3.

Running time of $\mathcal{S}$. First, we show that the expected running time of $\mathcal{S}$ is polynomial in the security parameters and n. Note that $\mathcal{S}$ 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 is trivial to check that each simulated execution takes polynomial time. Thus, it is sufficient to show that $\mathcal{S}$ only needs to rewind $\mathcal{A}$ for polynomial many times in expectation.

Note that $\mathcal{S}$ rewinds $\mathcal{A}$ when the randomly sampled $\mathcal{P}_{i^*}$ is either corrupted before $\mathcal{P}_{i^*}$ invokes PD or not included in T. 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 $\mathsf{Coin.Get}()$ is at least $\frac{n-f}{n}$. Being included in T means that i^* is included in the output of Dumbo-MVBA, which, under the condition that $\mathcal{P}_i$ is not corrupted before the output of $\mathsf{Coin.Get}()$, happens with the probability of at least $\frac{f+1}{2f+1}$. Therefore, the probability that a simulated execution does not abort is at least $\frac{(n-f)(f+1)}{n^2}$, and the expected time that $\mathcal{S}$ rewinds $\mathcal{A}$ is not greater than $\frac{n^2}{(n-f)(f+1)}$, 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: $\mathcal{S}$ needs to query the DLog oracle $\mathsf{DLog}_g(\cdot)$ for all v_j, such that $\widetilde{\mathcal{P}}_j \in \widetilde{\mathbb{M}}$ is corrupted or does not respond its signature at the time of generating transcript. There are at most $\widetilde{f}$ corrupted nodes and at most $\widetilde{f}$ nodes having not returned their signatures. Thus, $\mathcal{S}$ queries the oracle for at most $2\widetilde{f}$ times.

For each query v_j, $\mathcal{S}$ can provide its algebraic representation over $(g, \zeta_0, \ldots, \zeta_{2\widetilde{f}})$ as $(0, 1, j, j^2, \ldots, j^{2\widetilde{f}})$. Note that the final public key tpk is computed as

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

Without loss of generality, we assume $\mathcal{S}$ has queried the oracle for $2\widetilde{f}$ times with $\{v_j\}_{j\in S}$, for $S = j_1, \ldots, j_{2\widetilde{f}}$

We know that $\prod_{j\in[2\widetilde{f}+1]}(v_j^{(i^*)})^{\lambda_j} = \zeta_0$. For $\ell \neq i^*$, the simulator $\mathcal{S}$ can reconstruct the secret $s^{(\ell)}$ such that $\prod_{j\in[2\widetilde{f}+1]}(v_j^{(\ell)})^{\lambda_j} = g^{s^{(\ell)}}$. Therefore, $\mathcal{S}$ can provide the algebraic representation of tpk over $(g, \zeta_0, \ldots, \zeta_{2\widetilde{f}})$ as $(\sum_{\ell\in T^*, \ell\neq i^*} s^{(\ell)}, 1, 0, \ldots, 0)$.

Therefore, the simulatability matrix is as follows:

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

It is easy to verify the above matrix is invertible. □