Practical Asynchronous High-threshold Distributed Key Generation and Distributed Polynomial Sampling

Sourav Das[1] Zhuolun Xiang[2] Lefteris Kokoris-Kogias[3] Ling Ren[1]
[1]*University of Illinois at Urbana-Champaign,* [2]*Aptos,* [3]*IST Austria & Mysten Labs*
souravd2@illinois.edu, xiangzhuolun@gmail.com, lefteris@mystenlabs.com, renling@illinois.edu

Abstract

Distributed Key Generation (DKG) is a technique to bootstrap threshold cryptosystems without a trusted party. DKG is an essential building block to many decentralized protocols such as randomness beacons, threshold signatures, Byzantine consensus, and multiparty computation. While significant progress has been made recently, existing asynchronous DKG constructions are inefficient when the reconstruction threshold is larger than one-third of the total nodes. In this paper, we present a simple and concretely efficient *asynchronous* DKG (ADKG) protocol among $n = 3t+1$ nodes that can tolerate up to t malicious nodes and support any reconstruction threshold $\ell \geq t$. Our protocol has an expected $O(\kappa n^3)$ communication cost, where κ is the security parameter, and only assumes the hardness of the Discrete Logarithm. The core ingredient of our ADKG protocol is an asynchronous protocol to secret share a random polynomial of degree $\ell \geq t$, which has other applications, such as asynchronous proactive secret sharing and asynchronous multiparty computation. We implement our high-threshold ADKG protocol and evaluate it using a network of up to 128 geographically distributed nodes. Our evaluation shows that our high-threshold ADKG protocol reduces the running time by 90% and bandwidth usage by 80% over the state-of-the-art.

1 Introduction

The problem of Distributed Key Generation (DKG) is to generate a public/private key pair in a distributed fashion among a set of mutually distrustful nodes so that each node holds a share of the secret key in the end. The secret-shared private keys can later be used in a threshold cryptosystem, e.g., to produce threshold signatures [8, 29], to decrypt ciphertexts of threshold encryption [23, 38], or to generate common coins [12]. To use the secret key, a threshold number of the nodes needs to reveal partial results computed using their shares. We refer to this threshold as the *reconstruction* threshold. Typically, the reconstruction threshold is set to be one higher than the number of corrupt nodes tolerated by the DKG protocol, and we refer to such constructions as *low-threshold* DKG. In contrast, in a *high-threshold* DKG construction, the reconstruction threshold can be much higher than the number of corrupt nodes.

High-threshold distributed key generation enables threshold cryptosystems with stronger secrecy as an adversary now needs to acquire extra secret shares or partial results to break security. Furthermore, many applications call for high reconstruction thresholds. For instance, many state-of-the-art Byzantine Fault Tolerant (BFT) protocols [4, 28, 33, 41, 57] rely on threshold signature with a high threshold ($2t+1$ out of $3t+1$ where t is the fault threshold) to improve the communication efficiency. Asynchronous agreement protocols rely on shared randomness to circumvent the FLP impossibility [25], and high-threshold cryptosystems give simpler and more efficient shared randomness [12, 17, 18]. Note that many of these applications assume asynchronous networks. Thus, we focus on high-threshold *asynchronous* DKG (ADKG) in this paper.

Most DKG protocols assume synchronous networks [14, 15, 26, 29, 32, 34, 44, 47, 49, 52] (see §8). ADKG has been studied only recently [3, 21, 22, 27, 39] and the state-of-the-art high-threshold ADKG protocol is very inefficient compared to its low-threshold counterpart. More specifically, the high-threshold DKG protocol of Das et al. [22] requires 500× more computation and 6× more communication than its low threshold counterpart. (We will elaborate on the reasons behind these inefficiencies in §8.)

Our results. In this paper, we design a simple and concretely efficient high-threshold asynchronous distributed key generation protocol for discrete-logarithm-based threshold cryptosystems. In an asynchronous network of $n \geq 3t+1$ nodes, where at most t nodes could be malicious, our protocol achieves an expected communication cost of $O(\kappa n^3)$. Our protocol supports any reconstruction threshold $\ell \in [t, n-t-1]$, i.e., $\ell+1$ nodes are required to use the secret key (e.g., to produce a threshold signature or decrypt a threshold encryption). At the end of our protocol, each node receives a threshold secret share of a randomly chosen secret $z \in \mathbb{Z}_q$, where $\mathbb{Z}_q$ is a

Table 1: Comparison of existing high-threshold ADKG protocols. All of these protocols can tolerate $t < n/3$ malicious nodes. We measure the computation cost in terms of number of elliptic curve group exponentiations. Abbreviations used are, Decisional Diffie-Hellman (DDH), Symmetric External Diffie-Hellman (SXDH), Decisional Composite Residuosity (DCR), and Discrete Logarithm (DL).

	Secret key from a Field?	High Threshold?	Communication Cost (per node)	Computation Cost (per node)	Total Round Complexity	Cryptographic Assumption	Setup Assumption
Kokoris et al. [39]	✓	✓	$O(\kappa n^3)$	$O(n^3)$	$O(n)$	DDH	RO & PKI
Abraham et al. [3,21,27]	✗	✗[†]	$O(\kappa n^2)$	$O(n^2)$	$O(1)$	SXDH	RO & PKI
Das et al. [22]	✓	✓	$O(\kappa n^2)$	$O(n^3)$[‡]	$O(\log n)$	DCR & DDH	RO & PKI
This work	✓	✓	$O(\kappa n^2)$	$O(n^2)$	$O(\log n)$	DL	RO & PKI

[†] These works do not discuss whether their protocols support high-threshold or not. But we believe their protocols can be made to support high-threshold with minor modification.

[‡] Their computation cost is $O(n^3)$ elliptic curve group operations instead of elliptic curve group exponentiations.

field of size q. Our protocol can thus be used with off-the-shelf discrete-logarithm-based threshold cryptosystems [8, 23, 29]. We present an ideal-functionality-based definition of ADKG and rigorously prove the security of our ADKG protocol using the real-ideal indistinguishability paradigm.

The core ingredient of our high-threshold ADKG is a simple and concretely efficient protocol to sample a polynomial $z(\cdot) \in \mathbb{Z}_q[x]$ of degree ℓ such that each node receives an evaluation point of $z(\cdot)$. Here on, we refer to this as the *distributed polynomial sampling* protocol. We believe our distributed random polynomial sampling protocol will be of independent interest beyond high-threshold ADKG. For example, it can be used to improve the efficiency of asynchronous proactive secret sharing, robust pre-processing of asynchronous multi-party computation, etc (see §6). We also want to note that our distributed random sampling protocol can be used to sample polynomials of degrees larger than $n-t$, although this comes with additional trade-offs.

For $\ell > t$, our protocol is significantly more efficient than the best previous ADKG protocol of Das et al. [22]. Moreover, our protocol only assumes hardness of Discrete Logarithm (DL), whereas the high-threshold ADKG protocol of [22] assumes hardness of both Decisional Diffie-Hellman (DDH) and Decisional Composite Residuosity (DCR).

Along the way, we also provide a mechanism to reduce the worst-case computation cost of the state-of-the-art asynchronous multi-valued validated byzantine agreement (MVBA) protocol that does not rely on an external source for shared randomness [22]. Specifically, we reduce the per node computation cost from $O(n^3)$ elliptic curve group operation to $O(n^2)$ elliptic curve group exponentiations.

Implementation and evaluation. We implement[*] our ADKG protocol in `python` with `rust` for cryptographic operations. Our implementation supports both `curve25519` and `bls12381` elliptic curves and any reconstruction threshold $\ell \geq t$. We evaluate our protocol with up to 128 nodes in geographically distributed Amazon EC2 instances. For $\ell = 2t$, with 64 nodes and either curve, our *single-thread* implementation takes about 15 seconds and each node sends 3.5 Megabytes of data. This is less than $1/10$ of the running time and uses less than $1/5$ of the bandwidth compared to the best previous high-threshold ADKG protocol.

[*]Available at https://github.com/sourav1547/htadkg

Paper organization. The rest of the paper is organized as follows. In §2, we describe our system model, define the ADKG problem, and present an overview of our ADKG protocol. We describe preliminaries used in our protocol in §3. We present the detailed design of our high-threshold ADKG protocol and some optimizations in §4 and analyze it in §5. In §6, we discuss additional applications of our distributed polynomial sampling primitive. In §7 we provide implementation details and our evaluation results. We discuss related work in §8 and conclude in §9.

2 System Model and Overview

2.1 Notations and System Model

We use κ to denote the security parameter; for example, when we use a collision-resistant hash function, κ denotes the size of the hash function's output. We use $|S|$ to denote the size of a set S. Let $\mathbb{Z}_q$ be a finite field of order q. For any integer a, we use $[a]$ to denote the ordered set $\{1, 2, \ldots, a\}$. Also, for two integers a and b where $a < b$, we use $[a, b]$ to denote the ordered set $\{a, a+1, \ldots, b\}$.

For any element $x \in \mathbb{Z}_q$, we use $[\![x]\!]$ to denote the $(n, t+1)$ secret sharing of x, i.e., x is secret shared using a polynomial of degree t. Also, for any node i, we use $[\![x]\!]_i$ to denote the share held by node i. For any vector $\boldsymbol{x}$, we use $[\![\boldsymbol{x}]\!]$ to denote element-wise secret sharing of the vector $[\![\boldsymbol{x}]\!]$. Similarly, we use $[\![\boldsymbol{x}]\!]_i$ to denote the share of $\boldsymbol{x}$ held by node i.

Threat model and network assumption. We consider a network of n nodes where every pair of nodes are connected via a pairwise private and authenticated channel. We consider the presence of a malicious *static* adversary $\mathcal{A}$ that can corrupt up to t nodes out of at least $3t+1$ nodes in the network. Once the ADKG protocol terminates, $\mathcal{A}$ also sees $\ell - t$ additional shares of the secret key. We want to emphasize that, for the security

Functionality $\mathcal{F}_{\mathsf{ADKG}}$

Parameters: Maximum number of malicious nodes t, the total number of nodes $n \geq 3t+1$, and the reconstruction threshold $\ell \in [t, n-t-1]$. Let $\mathbb{G}$ be an elliptic curve group of order q with a random generator g and scalar field $\mathbb{Z}_q$.

1. Wait for $\mathcal{C}_1$ and $\mathcal{C}_2$, the set of nodes $\mathcal{A}$ will corrupt and the set of additional nodes whose shares $\mathcal{A}$ will learn. Check that $|\mathcal{C}_1| \leq t$ and $|\mathcal{C}_1 \cup \mathcal{C}_2| \leq \ell$.
2. Sample a uniformly random element $z \in \mathbb{Z}_q$. Generate $(n, \ell+1)$ Shamir secret shares of z, denoted as $[\![z]\!]$.
3. Compute $g^z, g^{[\![z]\!]}$ and send $(g, g^z, [\![z]\!]_i, g^{[\![z]\!]})$ to party i.

Figure 1: Asynchronous Distributed Key Generation functionality

guarantees of our protocol to hold, it is important that for the latter set of nodes, $\mathcal{A}$ only learns their final ADKG shares and not their internal states. We consider a static adversary in the sense that both the set of nodes $\mathcal{A}$ will corrupt and the set of additional nodes whose share $\mathcal{A}$ will learn are fixed a priori before the start of the protocol. We assume the network is asynchronous, i.e., $\mathcal{A}$ can arbitrarily delay any message but must eventually deliver all messages sent between honest nodes.

Our threat model is inspired by the typical use cases of high-threshold cryptographic schemes. For example, BFT protocols using threshold signatures often require that the signature remains unforgeable for any adversary that corrupts t nodes and also learns partial signatures from t additional nodes. More generally, cryptographic schemes that use high-threshold ADKG allow $\mathcal{A}$ to corrupt up to t nodes during the execution of the ADKG protocol; and after the ADKG protocol finishes, these schemes seek to prevent $\mathcal{A}$ from computing any function of the ADKG secret key z even if $\mathcal{A}$ is given some information that is computed from $\ell - t$ additional shares that belong to honest nodes. Our threat model captures this by allowing $\mathcal{A}$ to acquire these $\ell - t$ additional shares.

We also assume a public key infrastructure (PKI). This is because we use Asynchronous Complete Secret Sharing (ACSS) as a building block, and state-of-the-art ACSS protocols use PKI [21]. ACSS can also be instantiated without PKI at a higher cost [5], so the PKI assumption of our protocol can be removed at the cost of efficiency if the application calls for it.

2.2 Definition of ADKG

As mentioned in §1, in this paper, we focus on ADKG for discrete logarithm-based cryptosystems such as ElGamal encryption [24] and BLS signatures [8, 9].

A distributed key generation protocol for a discrete logarithm cryptosystem amounts to secret sharing a uniformly random value $z \in \mathbb{Z}_q$ and making public the value $y = g^z$, where g is a random generator of a group $\mathbb{G}$ of order q. With n nodes, at the end of the protocol, each node outputs a $(n, \ell+1)$-threshold Shamir share [51] of the secret z, where $\ell+1$ valid shares are needed to use z. More precisely, let $z(\cdot) \in \mathbb{Z}_q[x]$ be a random polynomial of degree ℓ such that $z(0) = z$. At the end of the DKG protocol, the i^{th} node outputs its share of the secret key $z(i) = [\![z]\!]_i$, and every node outputs the public key $y = g^z$. Additionally, applications of DKG such as threshold signatures and threshold encryption, require that in addition to y, threshold public keys of all nodes, i.e., $g^{z(i)}$ for all $i \in [n]$, are also publicly known.

For an ADKG protocol, the polynomial degree ℓ used for sharing the secret key is called the *reconstruction* threshold. We say an ADKG protocol is low-threshold if $\ell = t$, and is high-threshold if $\ell > t$.

We formalize the above using the ideal functionality $\mathcal{F}_{\mathsf{ADKG}}$ defined in Figure 1. Intuitively, for any given reconstruction threshold $\ell \geq t$, $\mathcal{F}_{\mathsf{ADKG}}$ samples a uniform random polynomial $z(x)$ of degree ℓ over the field $\mathbb{Z}_q$. Let $z(0)$ be the ADKG secret key. $\mathcal{F}_{\mathsf{ADKG}}$ then outputs one share to each node, i.e., outputs $z(i)$ to node i. $\mathcal{F}_{\mathsf{ADKG}}$ additionally outputs the ADKG public key g^z, and the threshold public keys $g^{[\![z]\!]}$ to all nodes.

An ADKG protocol that realizes the functionality $\mathcal{F}_{\mathsf{ADKG}}$ is called (t, ℓ)-secure if the following *Security* property holds in the presence of an adversary $\mathcal{A}$ that corrupts a set C_1 of up to t nodes and observes up to $\ell - t$ additional shares from a set of C_2 nodes.

Security. For every probabilistic polynomial-time (PPT) adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that given g^z, $g^{[\![z]\!]}$, and $[\![z]\!]_i$ for each $i \in \mathcal{C}_1 \cup \mathcal{C}_2$ by the ideal functionality $\mathcal{F}_{\mathsf{ADKG}}$, $\mathcal{S}$ produces a view such that the joint-distribution of $\mathcal{A}$'s view and honest parties' outputs in the ideal world is indistinguishable from that of the real world.

Remark. Note that our security definition captures all the properties of the DKG definitions presented in [22, 29].

2.3 Overview of our Protocol

Overview of existing ADKG protocols. Existing DKG protocols have the following typical structure: Each node runs a concurrent instance of verifiable secret sharing (VSS) to share a randomly chosen secret with every other node. For any reconstruction threshold ℓ, nodes use degree ℓ polynomial to share their secret. Once nodes agree on $t+1$ finished secret-sharing instances via an agreement protocol, they locally aggregate the corresponding shares to compute the share of the final secret key z. Briefly, the intuition is that the aggregated secret key contains the contribution of at least one honest node and thus remains hidden from the adversary.

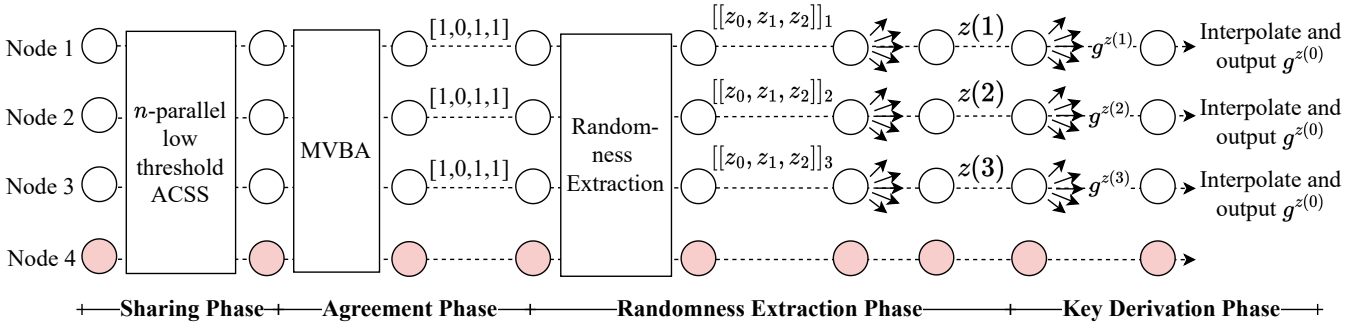

Figure 2: Overview of our protocol in a network of 4 nodes where node 4 is malicious. During the sharing phase, each node secret shares two random secrets using a low-threshold ACSS protocol. During the agreement phase, nodes run the Multi-valued Validated Byzantine Agreement (MVBA) protocol to agree on a subset of size $n-t$ valid ACSS instances. During the randomness extraction phase, nodes use the randomness extractor to extract secret shares of $\ell+1$ random coefficients of the polynomial $z(x) = z_0 + z_1 x + z_2 x^2 + \cdots + z_\ell x^\ell$. Nodes then interact with each other to assist node i in computing $z(i)$. During the key derivation phase, nodes interact to compute the ADKG public key $g^{z(0)}$ and threshold public keys of every other node.

In asynchrony, instead of running VSS, nodes run asynchronous complete secret sharing (ACSS) to share their random secrets (§3.1). The main source of inefficiency of high-threshold ADKG comes from high-threshold ACSS. In particular, the high-threshold ACSS designed and used in [22] is two to three orders of magnitude more expensive computationally than its low-threshold counterpart. To put things into perspective, with $n = 128$ nodes and `curve25519` as the underlying elliptic curve, running n parallel high-threshold ACSS takes 504 seconds, whereas running n parallel low-threshold ACSS takes only 0.19 seconds. Similarly, in their high-threshold ADKG, each node incurs about $6\times$ higher communication cost than in the low-threshold counterpart.

Our Approach. Our protocol deviates from the common wisdom that high-threshold ADKG needs to use high-threshold ACSS to *secret share* high-degree polynomials. Instead, we design a high-threshold ADKG by sampling a random degree ℓ polynomial in a distributed manner, such that each node obtains one evaluation point on the random polynomial. This way, our approach only uses low-threshold ACSS (i.e., to share degree-t polynomial), which is orders of magnitude faster than the best-known high-threshold ACSS schemes. In addition to high-threshold ADKG, this random polynomial sampling protocol can also improve the efficiency of asynchronous proactive secret sharing, robust pre-processing of asynchronous multi-party computation, and possibly other problems (see §6).

We now provide a simplified overview of our construction for the specific case of $\ell = 2t$. Besides low-threshold ACSS, our construction also uses a *Multi-valued Validated Byzantine Agreement* (MVBA) [11] subroutine, where each node inputs a value and agrees on a set of at least $n-t$ values.

Each node samples two randomly chosen secrets and shares them using a low-threshold ACSS scheme, i.e., an ACSS scheme where $t+1$ valid secret shares are sufficient to reconstruct the secret. Nodes then run a MVBA protocol to agree on a subset of nodes, denoted T, whose ACSS terminated at all nodes. The MVBA protocol guarantees that T includes at least $n-t$ nodes.

Once the MVBA protocol terminates, each node locally holds shares of two secret shared vectors of size at least $n-t$ each. Each vector contains $(n, t+1)$ threshold secret shares of up to possibly t biased secrets from the malicious nodes and at least $n-2t$ uniformly random secrets from the honest nodes. The idea is that nodes use these shares of the possibly biased secret shared vectors to generate $(n, t+1)$ threshold secret shares of $\ell+1$ random coefficients of a polynomial.

In order to produce threshold secret shares of the $\ell+1$ uniformly random coefficients, our protocol uses a randomness extractor, which outputs uniformly random values from a mixed set of random and biased values. One approach for randomness extraction is to multiply the above vector of randomness with a hyperinvertible matrix [6], which intuitively ensures each output value has contributions from at least one new uniformly random input and therefore is also uniformly random.

More specifically, each node locally computes the threshold secret shares of the $\ell+1$ random coefficients $z_0, z_1, z_2, \ldots, z_\ell$ by locally applying the randomness extractor to the ACSS outputs included in the MVBA output T. Here, we crucially use the linearity of the randomness extractor. Then, consider the polynomial $z(x)$ defined as:

$$z(x) = z_0 + z_1 x + z_2 x^2 + \cdots + z_\ell x^\ell$$

Our protocol uses $z_0 = z(0)$ as the ADKG secret key and $z(k)$ as the ADKG secret key share of node k. However, so far, each node k, only has $[\![z_0, z_1, z_2, \ldots, z_\ell]\!]_k$. Thus, the next step of our protocol is to assist node k in computing $z(k)$. In particular, each node i, uses $[\![z_0, z_1, z_2, \ldots, z_\ell]\!]_i$ to locally compute secret share of $z(k)$ for each $k \in [n]$, i.e., $[\![z(k)]\!]_i$, and sends it to node k via private channel. Node k, upon receiving sufficiently many shares of $z(k)$, recovers $z(k)$ using error

Table 2: Notations used in the paper

Notation	Description
n	Total number of nodes
t	Maximum number of malicious nodes
ℓ	Reconstruction threshold of ADKG
$\mathbb{Z}_q$	Field of order q where q is prime
$\mathbb{G}$	Group of order q with hard Discrete Logarithm
g, h	Random and independent generators of $\mathbb{G}$
z, g^z	ADKG secret and public key
$z(i)$	ADKG secret share of i^{th} node
$g^{z(i)}$	ADKG threshold public key of i^{th} node
κ	Security parameter
pk_i, sk_i	Public and private keys of i^{th} node.
a_i, b_i	Secrets chosen by i^{th} node during sharing phase
$a_i(\cdot), b_i(\cdot)$	Polynomial chosen by i^{th} node to share a_i, b_i
$\boldsymbol{v}_i, \boldsymbol{u}_i$	Pedersen commitment of $a_i(\cdot)$ and $b_i(\cdot)$
$\mathsf{pok}(\cdot)$	NIZK proof for Proof of Knowledge
T	MVBA protocol output

correcting code.

Challenges. Proving the security of our high-threshold ADKG protocol is more challenging than one might expect. The primary source of difficulty is due to the fact that, in our protocol an adversary corrupting any node i, in addition to the evaluation point $z(i)$, also learns secret shares of the coefficients of $z(\cdot)$, i.e., $[\![z_0, z_1, z_2, \ldots, z_\ell]\!]_i$, and all the secret shares of $z(i)$, i.e., $[\![z(i)]\!]_k$ for each $k \in [n]$. This introduces further challenges in ensuring that nodes output the correct ADKG public key $g^{z(0)}$ and the threshold public key $g^{z(k)}$ of each node $k \in [n]$. Addressing these challenges while maintaining the efficiency and simplicity of the overall protocol is quite challenging. We will discuss these in more detail in §5 when we have the appropriate context.

3 Preliminaries

In this section, we describe the preliminaries used in our protocol. We summarize the notations in Table 2.

3.1 Asynchronous Complete Secret Sharing

An ACSS protocol consists of two phases: Sharing and Reconstruction. During the sharing phase, a dealer L shares a secret $s \in \mathbb{Z}_q$ using Sh. During the reconstruction phase, nodes use Rec to recover the secret. We say that $(\mathsf{Sh}, \mathsf{Rec})$ is a t-resilient ACSS protocol if the following properties hold with probability $1 - \mathsf{negl}(\kappa)$ against any non-uniform probabilistic polynomial time (PPT) adversary that corrupts up to t nodes:

- **Correctness.** If L is honest, then Sh will result in every honest node i eventually outputting $[\![s]\!]_i$. Once Sh is complete, if all honest nodes start Rec, they will output s as long as at most t nodes are malicious.
- **Secrecy.** If L is honest, then for any non-uniform PPT adversary $\mathcal{A}$ controlling up to t nodes, there exists a PPT simulator $\mathcal{S}$ such that the output of $\mathcal{S}$ and $\mathcal{A}$'s view in the real-world protocol are indistinguishable.
- **Agreement.** If any honest nodes outputs in Sh, then there exists a secret $\tilde{s} \in \mathbb{Z}_q$ such that each honest node i eventually outputs $[\![\tilde{s}]\!]_i$ and $\tilde{s}$ is guaranteed to be correctly reconstructed in Rec. Moreover, if L is honest, $\tilde{s} = s$.

We also require the ACSS scheme to satisfy the following *Homomorphic-Partial-Commitment* property.

- **Homomorphic-Partial-Commitment:** If some honest node terminates Sh for a secret s, then every honest node outputs commitments of $[\![s]\!]_i$ for each $i \in [n]$. Furthermore, these commitments are additively homomorphic across different ACSS instances.

We observe that if an ACSS protocol outputs a Pedersen commitment of the underlying polynomial, then it guarantees Homomorphic-Partial-Commitment. We describe Pedersen commitment and Pedersen polynomial commitment [46] next.

Pedersen commitment. Let $g, h \in \mathbb{G}$, be two uniformly random and independent generators of an elliptic curve group $\mathbb{G}$. Given g, h, a Pedersen commitment c to a message m, is $c = g^m \cdot h^r$. The opening proof of a Pedersen commitment is the tuple (m, r). Upon receiving opening to a commitment c, the verifier checks its correctness by checking that $c = g^m h^r$. Pedersen commitment is information-theoretically hiding and, assuming the hardness of discrete logarithm, computationally binding.

Pedersen polynomial commitment. To commit to a degree-d polynomial $a(x)$, the committer samples a random degree d polynomial $\hat{a}(x)$, where:

$$a(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d$$
$$\hat{a}(x) = \hat{a}_0 + \hat{a}_1 x + \hat{a}_2 x^2 + \cdots + \hat{a}_d x^d$$

Here, the coefficients a_k and $\hat{a}_k$ for $k \in [0, d]$ are elements of $\mathbb{Z}_q$ and $\hat{a}_k$ are uniformly random. Then the commitment to $a(x)$ is the vector $\boldsymbol{v}$ computed as:

$$\boldsymbol{v} = \left[g^{a_0} h^{\hat{a}_0}, g^{a_1} h^{\hat{a}_1}, g^{a_2} h^{\hat{a}_2}, \ldots, g^{a_d} h^{\hat{a}_d}\right] \tag{1}$$

Note that given the Pedersen commitment of a polynomial $a(\cdot)$ with randomness $\hat{a}(\cdot)$, we can compute $g^{a(i)} h^{\hat{a}(i)}$, the Pedersen commitment of $a(i)$, by evaluating in the exponent. The polynomial commitment is additively homomorphic and information theoretically hiding. Moreover, assuming the hardness of discrete logarithm, the polynomial commitment is computationally binding. The size of the commitment is linear in the degree of the polynomial.

Given a commitment $\boldsymbol{v}$ and share $\alpha_i, \hat{\alpha}_i$, a node checks whether $\alpha_i = a(i)$ and $\hat{\alpha}_i = \hat{a}(i)$ by checking whether

$$g^{\alpha_i} h^{\hat{\alpha}_i} = \prod_{k=1}^{d} (\boldsymbol{v}[k])^{i^k} \tag{2}$$

Our paper uses the low-threshold ACSS scheme from Das et al. [21], which improves upon the low-threshold ACSS of Yurek et al. [58]. For completeness, we describe the Sharing phase of the ACSS scheme of [21] in Appendix B. We want to note that it is possible to further improve the concrete computation cost of the low-threshold ACSS phase using recent techniques from [60].

3.2 Multi-valued Validated Byzantine Agreement

Multi-valued validated Byzantine agreement (MVBA) [11] is an agreement protocol that guarantees a set of nodes, each with an input value, to agree on the same value satisfying a predefined external predicate $P(v) : \{0,1\}^{|v|} \rightarrow \{0,1\}$ globally known to all the nodes. A MVBA protocol with predicate $P(\cdot)$ provides the following guarantees except for negligible probability.

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

Remark 1. Due to the FLP [25] impossibility result, a deterministic agreement is impossible under asynchrony and thus requires randomness (commonly via a threshold-signature based common-coin). To break this circularity, we use the recent ideas from [22], whose Sharing, Key Proposal, and Agreement phases can be viewed as a MVBA protocol. Looking ahead, the MVBA construction of [22], after our improvements in §4.6, has communication cost of $O(\kappa n^3)$, expected latency of $O(\log n)$ rounds, and each node incurs a computation cost of $O(n^2)$ group exponentiations.

Remark 2. Our protocol also uses an MVBA with slightly strong validity requirement [59], where the predicate $P(v,e)$ additionally can have some variable e depending on the execution state of the node as the input. Indeed, the MVBA of Das et al. [22] satisfies this property. We will explain more details in §4.2.

3.3 Randomness Extraction using a Hyperinvertible Matrix

We use the randomness extraction technique based on hyperinvertible matrices [6], which are matrices whose every square sub-matrix is invertible. Using a hyperinvertible matrix, each node can perform a series of local linear operations on the shares of m input secrets and extracts shares of $m-t$ uniform random secrets.

Let $[\![x_1, x_2, \ldots, x_m]\!]_i$ be shares of m secrets $x_1, x_2, \ldots, x_m$ held by node i. We use the following Vandermonde matrix. Then, nodes compute their shares of $m-t$ output secrets $y_1, y_2, \ldots, y_{m-t}$ as follows:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{m-t} \end{bmatrix} = \begin{bmatrix} 1 & \omega_1 & \ldots & \omega_1^{m-1} \\ 1 & \omega_2 & \ldots & \omega_2^{m-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{m-t} & \ldots & \omega_{m-t}^{m-1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{m-1} \\ x_m \end{bmatrix}$$

If at least $m-t$ input secrets are independent and uniformly random, and the matrix is hyper-invertible, then the $m-t$ output secrets $y_1, y_2, \ldots, y_{m-t}$ are guaranteed to be independent and uniformly random. Looking ahead, this suits our purpose because at most t input secrets in our protocol come from corrupt nodes, and the remaining $m-t$ will be independent and uniformly random.

4 Design

We summarize our ADKG protocol in Algorithm 1 and describe it in this section.

The public parameters for our ADKG protocol are a pair of randomly and independently chosen generators (g,h) of a group $\mathbb{G}$ of prime order q, in addition to any public parameters of the MVBA protocol. Our protocol consists of four phases: *Sharing, Agreement, Randomness Extraction, and Key Derivation* phase.

4.1 Sharing Phase

During the sharing phase, each node i samples two uniformly random secrets $a_i, b_i \in \mathbb{Z}_q$ and secret-shares them with all other nodes using ACSS schemes (lines 2-3 in Algorithm 1). Looking ahead in our protocol, each invocation of the randomness extractor will produce $t+1$ random values (cf. §4.3). Since we need $\ell+1$ random values and ℓ can be up to $2t$, we need each node to share two uniform random values.

Nodes use the low-threshold ACSS from [21, §5.3] to share the secrets. Let $a_i(\cdot), b_i(\cdot) \in \mathbb{Z}_q[x]$ as shown below be the two polynomials of degree t used during the ACSS scheme.

$$a_i(x) = a_i + a_{i,1}x + a_{i,2}x^2 + \ldots + a_{i,t}x^t$$
$$b_i(x) = b_i + b_{i,1}x + b_{i,2}x^2 + \ldots + b_{i,t}x^t$$

where $a_{i,k}, b_{i,k} \in \mathbb{Z}_q$ are chosen at random. Let $\hat{a}_i$ and $\hat{b}_i$ be the randomness used in the Pedersen commitment of a_i and b_i, respectively.

The Agreement property of the ACSS scheme guarantees that, once the sharing phase of i^{th} ACSS instance terminates, each honest node outputs one evaluation points on $a_i(\cdot)$ and $b_i(\cdot)$. Also, each node j will also output $[\![\hat{a}_i]\!]_j$, and $[\![\hat{b}_i]\!]_j$. Each node additionally outputs the Pedersen commitments u_i and v_i of a_i and b_i, respectively, where:

$$u_i = g^{a_i} h^{\hat{a}_i}; \quad v_i = g^{b_i} h^{\hat{b}_i} \tag{3}$$

Algorithm 1 High-threshold ADKG for node i

INPUT: $\ell, g, h, sk_i, \{pk_j\}$ for each $j \in [n]$
OUTPUT: $z(i), g^z, \{g^{z(j)}\}$ for each $j \in [n]$

SHARING PHASE:

$S := \{\}, K := \{\}, R := \{\}, H := \{\}$
Sample random secrets $a_i, b_i \leftarrow \mathbb{Z}_q$
ACSS(a_i, b_i) with randomness $(\hat{a}_i, \hat{b}_i)$;
$S := S \cup \{j\}$ when j-th ACSS terminates at node i

AGREEMENT PHASE:

if $|S| = n - t$ **then**
  Let $S_i := S$, invoke MVBA(S_i) with predicate $P(S_j, S)$
  ▷ S_j is the input value of some node j, S is node i's local variable defined in the Sharing Phase. $P(S_j, S)$ only returns 1 once $S_j \subseteq S$.
Let T be the output of the MVBA protocol
for each $j \in [n] \setminus T$ **do**
  Let $[\![a_j]\!]_i := 0$; $[\![\hat{a}_j]\!]_i := 0$
  Let $[\![b_j]\!]_i := 0$; $[\![\hat{b}_j]\!]_i := 0$
  Let $u_j := 1_{\mathbb{G}}$; $v_j := 1_{\mathbb{G}}$

RANDOMNESS EXTRACTION PHASE:

Wait for the agreement phase to terminate
Let $M, \tilde{M}$ be hyperinvertible matrix described in §4.3.
Let $[\![z_0, z_1, z_2, \ldots, z_t]\!]_i := M \cdot [\![a_1, a_2, \ldots, a_n]\!]_i$
Let $[\![z_{t+1}, z_{t+2}, \ldots, z_\ell]\!]_i := \tilde{M} \cdot [\![b_1, b_2, \ldots, b_n]\!]_i$
Let $[\![\hat{z}_0, \hat{z}_1, \hat{z}_2, \ldots, \hat{z}_t]\!]_i := M \cdot [\![\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_n]\!]_i$
Let $[\![\hat{z}_{t+1}, \hat{z}_{t+2}, \ldots, \hat{z}_\ell]\!]_i := \tilde{M} \cdot [\![\hat{b}_1, \hat{b}_2, \ldots, \hat{b}_n]\!]_i$
Let $z(\cdot), \hat{z}(\cdot)$ be the ℓ-degree polynomial defined in §4.3
Send $\langle \texttt{RANDEX}, [\![z(j)]\!]_i, [\![\hat{z}(j)]\!]_i \rangle$ to node $j, \forall j \in [n]$.
upon receiving $\langle \texttt{RANDEX}, [\![z(i)]\!]_j, [\![\hat{z}(i)]\!]_j \rangle$ from j **do**
  $K := K \cup \{(j, [\![z(i)]\!]_j)\}, R := R \cup \{(j, [\![\hat{z}(i)]\!]_j)\}$
Run OEC on the set K and R
Let $z(i) :=$ OEC(K) and $\hat{z}(i) :=$ OEC(R).

KEY DERIVATION PHASE:

Let $\pi_i := \{\mathsf{pok.Prove}(z(i), g, g^{z(i)}), \mathsf{pok.Prove}(\hat{z}(i), h, h^{\hat{z}(i)})\}$
Send $\langle \texttt{KEY}, g^{z(i)}, h^{\hat{z}(i)}, \pi_i \rangle$ to all
Let $[c_0, c_1, c_2, \ldots, c_t] := M \star [u_1, u_2, \ldots, u_n]$
Let $[c_{t+1}, c_{t+2}, \ldots, c_\ell] := \tilde{M} \star [v_1, v_2, \ldots, v_n]$
  ▷ $\star$ denotes inner product in the exponent. u_i, v_i are the Pedersen commitments of a_i, b_i in the ACSS respectively, as described in §4.1.
upon receiving $\langle \texttt{KEY}, g^{z(j)}, h^{\hat{z}(j)}, \pi_j \rangle$ from node j **do**
  Validate π_j and check whether $c(j) = g^{z(j)} h^{\hat{z}(j)}$
  if All the condition above are valid **then**
    $H := H \cup \{(j, g^{z(j)})\}$
    if $|H| \geq \ell + 1$ **then**
      Interpolate $g^{z(0)}$ and any missing $g^{z(j)}$
      output $z(i)$, $g^{z(0)}$, and $g^{z(j)}$ for each $j \in [n]$

4.2 Agreement Phase

During the agreement phase, nodes run an multi-valued validated Byzantine agreement (MVBA) protocol to agree on a subset of valid ACSS instances that terminated. More specifically, each node waits for $n - t$ ACSS instances to terminate locally. Let S_i be the set of first $n - t$ ACSS instances that terminates node i. Node i then inputs S_i to the MVBA protocol. Node i also maintains a set S of all ACSS instances that terminate at node i. Note that the S is ever growing. For any value S_j input to the MVBA by node j, node i uses the predicate $P(S_j, S)$ to check that $|S_j| \geq n - t$ and $S_j \subseteq S$, i.e., all ACSS instances in S_j terminated at node i.

Let T be the output of the MVBA protocol, $|T| \geq n - t$. Hence, T includes at least $n - 2t$ honest nodes. After the MVBA protocol outputs the set T, node i sets $[\![a_j]\!]_i, [\![b_j]\!]_i$, to be equal to 0 for each $j \in [n] \setminus T$. This implies that for each $j \in [n] \setminus T$, a_j and b_j are set to be equal to 0 as field elements.

4.3 Randomness Extraction Phase

Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be the vectors defined as below,

$$\boldsymbol{a} = [a_1, a_2, \ldots, a_n]; \text{ where } a_j = 0, \forall j \in [n] \setminus T$$
$$\boldsymbol{b} = [b_1, b_2, \ldots, b_n]; \text{ where } b_j = 0, \forall j \in [n] \setminus T$$

Let $[\![\boldsymbol{a}]\!]_i$ and $[\![\boldsymbol{b}]\!]_i$ be the vectors consisting of element-wise secret shares of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, respectively, held by node i. Each node i then locally compute the secret share of the vector $[z_0, z_1, z_2, \ldots, z_\ell]$ where the elements are defined as below

$$\begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_t \end{bmatrix} = \begin{bmatrix} 1 & \omega_1 & \ldots & \omega_1^{n-1} \\ 1 & \omega_2 & \ldots & \omega_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{t+1} & \ldots & \omega_{t+1}^{n-1} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} z_{t+1} \\ z_{t+2} \\ \vdots \\ z_\ell \end{bmatrix} = \begin{bmatrix} 1 & \omega_1 & \ldots & \omega_1^{n-1} \\ 1 & \omega_2 & \ldots & \omega_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{\ell-t} & \ldots & \omega_{\ell-t}^{n-1} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{5}$$

where the matrix is the hyperinvertible matrix.

Using the $(n, t+1)$ secret shares of $\boldsymbol{a}$ and $\boldsymbol{b}$, each node i locally computes the element wise $(n, t+1)$ secret shares of the vector $[z_0, z_1, z_2, \ldots, z_\ell]$, by applying the operations specified in equation (4) and equation (5) to their shares of the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, respectively.

Let $z(\cdot) \in \mathbb{Z}_q[x]$ be the polynomial of degree ℓ defined as,

$$z(x) = z_0 + z_1 x + z_2 x^2 + \cdots + z_\ell x^\ell \tag{6}$$

Each node has a $(n, t+1)$ share of every coefficient of the polynomial $z(\cdot)$. Each node i then locally computes $[\![z(j)]\!]_i$ for every other node j. We denote by $\boldsymbol{z}$ the vector $[z(1), z(2), \ldots, z(n)]$. In addition to computing

$[\![\boldsymbol{z}]\!]_i$, nodes additionally compute shares of the vector $\hat{\boldsymbol{z}} = [\hat{z}(1), \hat{z}(2), \ldots, \hat{z}(n)]$ in the same way using equation (4) and equation (5) where, instead of using $[\![\boldsymbol{a}]\!]_i$ and $[\![\boldsymbol{b}]\!]_i$, nodes use $[\![\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_n]\!]_i$ and $[\![\hat{b}_1, \hat{b}_2, \ldots, \hat{b}_n]\!]_i$, respectively. Recall from §4.1 that $\hat{a}_k$ and $\hat{b}_k$ for any $k \in T$ are the randomness used in the Pedersen commitment of a_k and b_k, respectively, and $\hat{a}_k = \hat{b}_k = 0$ for all $k \in [n] \setminus T$.

Node i then sends $\langle \texttt{RANDEX}, [\![z(j)]\!]_i, [\![\hat{z}(j)]\!]_i \rangle$ to node j. Upon receiving $\langle \texttt{RANDEX}, [\![z'(i)]\!]_j, [\![\hat{z}'(i)]\!]_j \rangle$ from node j, node i reconstructs $z(i)$ and $\hat{z}(i)$ using online error correction (see Appendix A for more details on online error correction).

4.4 Key Derivation Phase

During the key derivation phase, each node first computes the Pedersen commitments to $z(i)$ for each $i \in [n]$ using the publicly available information. Recall from sharing phase (§4.1) each for every ACSS j that terminates, node i outputs the Pedersen commitment to the corresponding secrets. Let u_j, v_j be the corresponding commitments, where

$$u_j = g^{a_j} h^{\hat{a}_j}; \quad v_j = g^{b_j} h^{\hat{b}_j}$$

Let $[c_0, c_1, c_2, \ldots, c_\ell]$ be the vector defined as below:

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_t \end{bmatrix} = \begin{bmatrix} 1 & \omega_1 & \ldots & \omega_1^{n-1} \\ 1 & \omega_2 & \ldots & \omega_2^{n-1} \\ \vdots & & & \\ 1 & \omega_{t+1} & \ldots & \omega_{t+1}^{n-1} \end{bmatrix} \star \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}$$

$$\begin{bmatrix} c_{t+1} \\ c_{t+2} \\ \vdots \\ c_\ell \end{bmatrix} = \begin{bmatrix} 1 & \omega_1 & \ldots & \omega_1^{n-1} \\ 1 & \omega_2 & \ldots & \omega_2^{n-1} \\ \vdots & & & \\ 1 & \omega_{\ell-t} & \ldots & \omega_{\ell-t}^{n-1} \end{bmatrix} \star \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Here the $\star$ operation denotes the inner product in the exponent, i.e.,

$$c_0 = \prod_{k \in [n]} u_k^{\omega_1^{k-1}}$$

We will now define the function $c(\cdot) : \mathbb{Z}_q \to \mathbb{G}$ as:

$$c(x) = \prod_{i=0}^{\ell} c_i^{x^i}$$

It is easy to see that $c_i = g^{z_i} h^{\hat{z}_i}$. Also, $c(i) = g^{z(i)} h^{\hat{z}(i)}$. Let $\boldsymbol{c}$ be the vector defined as:

$$\boldsymbol{c} = [c(1), c(2), \ldots, c(n)]$$

Let π_i be the non-interactive zero-knowledge proof of knowledge of $z(i)$ and $\hat{z}(i)$ with respect to $g^{z(i)}$ and $h^{\hat{z}(i)}$, respectively [50]. In particular, let

$$\pi_i = \left\{ \mathsf{pok.Prove}\left(z(i), g, g^{z(i)}\right); \mathsf{pok.Prove}\left(\hat{z}(i), h, h^{\hat{z}(i)}\right) \right\}$$

Each node i then sends $\langle \texttt{KEY}, g^{z(i)}, h^{\hat{z}(i)}, \pi_i \rangle$ to every node. Also, node i, upon receiving $\langle \texttt{KEY}, g^{z(j)}, h^{\hat{z}(j)}, \pi_j \rangle$ from node j checks whether π_j is a valid proof of knowledge. Node i additionally checks whether $c(j) = g^{z(j)} h^{\hat{z}(j)}$ or not. Upon receiving $\ell + 1$ valid KEY messages, a node can compute the public key $g^{z(0)}$ and the missing threshold public keys $g^{z(j)}$ for each $j \in [n]$ using Lagrange interpolation in the exponent.

4.5 Reducing Common-Case Computation

One way to compute $c(k)$ for any $k \in [n]$ is to first compute c_i for all $i \in [0, \ell]$ and then evaluate the function $c(\cdot)$ at k. It is easy to see that computing $c(k)$ for all $k \in [n]$ naïvely would require each node to perform $O(n^2)$ group exponentiations. Using number theoretic transform (NTT), a node can compute $c(k)$ for all k in $O(n \log n)$ group exponentiations as follows. Each node first uses NTT to compute c_i for all $i \in [0, \ell]$. Then, each node use NTT one more time to compute $c(k)$ for all $k \in [n]$, using c_i for all $i \in [0, \ell]$. We design an optimization that can further reduce it to $O(n)$ exponentiation in the fault-free case, i.e., without any byzantine behavior. Our approach also maintains the cost of $O(n \log n)$ group exponentiations in the presence of arbitrary faults.

During key derivation phase, every node upon receiving the message $\langle \texttt{KEY}, g^{z'(i)}, h^{\hat{z}'(i)} \rangle$ from node i, validates them for proof of knowledge. Upon successful validation, each node optimistically assumes that $z'(i) = z(i)$ and $\hat{z}'(i) = \hat{z}(i)$, i.e., they are valid evaluations of the polynomials $z(\cdot)$ and $\hat{z}(\cdot)$. Upon receiving $\ell + 1$ such KEY messages, the node optimistically compute $g^{z'(0)}$ and $h^{\hat{z}'(0)}$ by interpolating in the exponent. The node additionally computes c_0 and checks whether $g^{z'(0)} h^{\hat{z}'(0)} = c_0$. If the check passes, the node outputs $g^{z'(0)}$ as the ADKG public key. Otherwise, if the check fails, it falls back to the original protocol. Specifically, it finds the invalid KEY messages using the check we described in §4.4, removes them from the set of received KEY messages, and computes the correct ADKG public key as per §4.4.

To compute the threshold public keys, each node waits for all n KEY messages for a pre-specified time. Upon receiving all n KEY messages, nodes checks that the threshold keys included in the message lie on a degree ℓ polynomial using [16]. Otherwise, if the check fails or a node does not receive all n KEY messages within the pre-specified time, the node computes the threshold public keys of other nodes using NTT.

We will prove in §5.2 that this approach ensures correctness, i.e., nodes always output the correct ADKG public key and threshold public keys.

4.6 Reducing the Computation Cost of MVBA

We will now describe how to reduce the worst-case computation cost of the multi-valued validated byzantine agreement (MVBA) protocol of [22] from $O(n^3)$ group operations

to $O(n^2)$ group exponentiations. We will first briefly describe the MVBA protocol of Das et al. [22].

MVBA protocol of [22]. The protocol has three phases: Sharing, Key-proposal, and Agreement. During the sharing phase, each node secret shares a random value using an ACSS scheme. Each node i then waits for a set K_i of $t+1$ ACSS instances to terminate locally, and then reliably broadcasts K_i during the key-proposal phase. During the agreement phase, nodes run n parallel asynchronous binary agreement (ABA) protocol. For the i-th ABA, nodes aggregate the ACSS instances in K_i and use the aggregated secret to generate shared randomness [12]. Additionally, nodes aggregate the polynomial commitments of the ACSS instances in K_i, which require them to perform $O(n^2)$ group operations for each ABA. Since there are n such ABA instances, in the worst case, each node will need to perform $O(n^3)$ group operation.

Our approach. We reduce the computation cost of the agreement phase of their MVBA protocol to $O(n^2)$ group operations by adopting the distributed aggregation verification idea from [20]. The main idea is to have only one node perform the aggregation and let each node verify only one position of the aggregated commitment. Specifically, during the key-proposal phase, node i first aggregates the polynomial commitments of the ACSS instances in K_i, and then reliably broadcasts the tuple $(K_i, \boldsymbol{v}_i)$ where $\boldsymbol{v}_i$ is the aggregated commitment. Upon receiving the tuple $(K_i, \boldsymbol{v}_i)$ as the proposal during the i-th reliable broadcast, each node j, locally compute the correctly aggregated commitment $\tilde{\boldsymbol{v}}_i[j]$ using the publicly available information and checks whether $\boldsymbol{v}_i[j] = \tilde{\boldsymbol{v}}_i[j]$. A node j participates in the reliable broadcast only if this check is successful in addition to the checks in [22]. Lastly, when (if) the i-th key proposal reliable broadcast terminates, nodes use $\boldsymbol{v}_i$ to compute shared randomness for the i-th ABA instance.

It is easy to see that the for each ABA instance, every node needs to perform a linear number of group operations. Hence, each node will perform $O(n^2)$ group operations in total. Also, note that $\boldsymbol{v}_i$ is a commitment to a polynomial of degree t. Hence, $t+1$ distinct points uniquely determine the polynomial. Since the reliable broadcast successfully terminates only if at least $t+1$ honest nodes participate, this implies that $\boldsymbol{v}_i$ and $\tilde{\boldsymbol{v}}_i$ match in at least $t+1$ evaluation points and are commitments to the same polynomial.

5 Analysis

5.1 Security

Intuition. Intuitively, the security of our ADKG protocol follows from the fact that the randomness extraction phase outputs $\ell+1$ independent and uniformly random values, which are then used as the coefficients of a degree-ℓ polynomial. Since each node only learns one point on this polynomial, the ADKG secret key remains hidden from an adversary who learns at most ℓ points.

We will now formalize the above intuition using the real-ideal paradigm. More precisely, we will prove that for every PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ that, given g^z, $g^{[\![z]\!]}$, and $[\![z]\!]_i$ for each $i \in \mathcal{C}_1 \cup \mathcal{C}_2$ by the ideal functionality $\mathcal{F}_{\mathsf{ADKG}}$, produces a view such that the joint-distribution of $\mathcal{A}$'s view and honest parties' outputs in the ideal world is indistinguishable from that of the real world.

In order to prove security, we first prove the following important lemma.

Lemma 1. *Let $[\![z]\!]_i$ be the share of node i. These shares lie on a polynomial $z(x)$ of degree at most ℓ, i.e., $z(i) = [\![z]\!]_i$. Assuming the hardness of the discrete logarithm, all honest nodes output the same public key g^z and the same threshold public keys $g^{z(i)}$ of all nodes $i \in [n]$.*

Proof. The Termination property of the MVBA protocol of [22] ensures that the agreement phase terminates at all honest nodes. Moreover, the Agreement and External validity property ensures that all honest nodes agree on the MVBA output T, and $|T| \geq n-t$.

During the randomness extraction phase, each node i computes the tuple $[\![z(k)]\!]_i, [\![\hat{z}(k)]\!]_i$ for every $k \in [n]$ and sends them to node k using `RANDEX` message. The Correctness property of online error correction ensures that every honest node outputs the correct evaluation point on the polynomial $z(\cdot)$. Thus, any subset of $\ell+1$ valid shares uniquely identity $z = z(0)$.

The External validity property of the MVBA guarantees that every ACSS instance included in T will terminate at all honest nodes. Thus, by the Homomorphic-Partial-Commitment property of the ACSS scheme, each node will output the Pedersen commitment of every ACSS instance in T. Since multiplication by the Hyperinvertible matrix is deterministic, every node agrees on the Pedersen polynomial commitment to $z(\cdot)$.

Let $\boldsymbol{c} = [c(1), c(2), \ldots, c(n)]$ be the commitment vector where $c(i)$ is the Pedersen commitment to $z(i)$. Honest nodes can compute $\boldsymbol{c}$ using only publicly available information. During the key derivation phase, for any `KEY` message from node i, i,e., $\langle \mathtt{KEY}, g^{z'(i)}, h^{\hat{z}'(i)} \rangle$, the Proof-of-knowledge NIZK proof guarantees that node i knows $z'(i)$ and $\hat{z}'(i)$. Thus, by the binding property of the Pedersen commitment scheme, $z'(i) = z(i)$ and $\hat{z}'(i) = \hat{z}(i)$. Since each honest node only uses valid `KEY` messages to compute the ADKG public key, every honest node will output the same public key $g^{z(0)}$.

In addition to computing $g^{z(0)}$, nodes use the set of valid `KEY` messages to interpolate the threshold public keys of every node i. Hence, nodes agree on the threshold public key $g^{z(i)}$ of each node i. □

We describe the simulator $\mathcal{S}$ in Figure 3 and summarize it below. Let $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$ and $\mathcal{H} = [n] \setminus \mathcal{C}_1$. $\mathcal{S}$ samples a random element $\alpha \in \mathbb{Z}_q$ and sets $h = g^{\alpha}$. Then, $\mathcal{S}$ simulates the sharing and agreement phases of the ADKG protocol per the protocol specification. Specifically, for each node $i \in \mathcal{H}$, $\mathcal{S}$ samples

Inputs. $\mathcal{C}_1$ and $\mathcal{C}_2$.
Notations. Let $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$ and $\mathcal{H} = [n] \setminus \mathcal{C}_1$.

1. Send $\mathcal{C}_1, \mathcal{C}_2$ to $\mathcal{F}_{\mathsf{ADKG}}$ and receive $g, g^z, g^{[\![z]\!]}$, and $[\![z]\!]_i$ for each $i \in \mathcal{C}$.
2. Sample a uniformly random $\alpha \in \mathbb{Z}_q$ and sets $h = g^\alpha$.
3. For each node $i \in \mathcal{H}$, sample uniformly secrets $a_i, b_i \in \mathbb{Z}_q$. Follow the protocol for every honest node during the sharing phase and the agreement phase. Let T be the output of the agreement phase.
4. Let $z'(x)$ be the resulting polynomial of degree ℓ specified by T. Then, during the randomness extraction phase, each node j possess $[\![z'(i)]\!]_j$ for all $i \in [n]$.
5. Let $z(x)$ be the polynomial of degree ℓ such that $z(0) = z$ and $z(i) = [\![z]\!]_i$ for each $i \in \mathcal{C}$.
6. For each node $i \in \mathcal{C}$, compute $[\![z(i)]\!]_j$ for all $j \in \mathcal{H}$ such that $[\![z(i)]\!]_k = [\![z'(i)]\!]_k$ for all $k \in \mathcal{C}_1$. Also, for each $i \in \mathcal{C}$, compute the corresponding Pedersen commitment randomness $[\![\hat{z}(i)]\!]_j$ for all $j \in \mathcal{H}$ such that the underlying Pedersen commitments do not change. Use knowledge of α to compute these values.
7. On behalf of each node $j \in \mathcal{H}$, sends $\langle \texttt{RANDEX}, [\![z(i)]\!]_j, [\![\hat{z}(i)]\!]_j \rangle$ to node i for all $i \in \mathcal{C}$.
8. For each honest node $j \in [n] \setminus \mathcal{C}$, compute $g^{z(j)}$ by interpolating in the exponent. Additionally, compute $h^{\hat{z}(j)}$ as $g^{z'(j)} h^{\hat{z}(j)} / g^{z(j)}$. Generate NIZK proof π_j for proof of knowledge of $g^{z(j)}$ and $h^{\hat{z}(j)}$ using the NIZK simulator. Finally, send $\langle \texttt{KEY}, g^{z(j)}, h^{\hat{z}(j)}, \pi_j \rangle$.

Figure 3: Description of the ADKG simulator $\mathcal{S}$.

uniformly random secrets a_i, b_i and secret shares them using the ACSS protocol. $\mathcal{S}$ then runs the agreement phase for each honest node. Let T be the output of the MVBA protocol.

For each node $i \in [n]$, let $z'(i), \hat{z}'(i) \in \mathbb{Z}_q$ be its share and for Pedersen commitment computed as per §4.3. Then, by construction, after applying the randomness extractor to the secret shares received during the sharing phase, each node $j \in \mathcal{H}$ possesses $[\![z'(i)]\!]_j$ and $[\![\hat{z}'(i)]\!]_j$. Also, let $c_{i,j}$ be the corresponding Pedersen commitment, i.e., $c_{i,j} = g^{[\![z'(i)]\!]_j} h^{[\![\hat{z}'(i)]\!]_j}$.

Next comes the key step of our simulator. Let $z(x)$ be the polynomial of degree ℓ such that $z(0) = z$ and $z(i) = [\![z]\!]_i$ for each $i \in \mathcal{C}$. For each $i \in \mathcal{C}$, $\mathcal{S}$ computes $[\![z(i)]\!]_j$ for all $j \in \mathcal{H}$ such that $[\![z(i)]\!]_k = [\![z'(i)]\!]_k$ for all $k \in \mathcal{C}_1$. $\mathcal{S}$ computes these values by interpolating a polynomial of degree t that evaluates to $z(i)$ at 0 and to $[\![z(i)]\!]_k$ at k for all $k \in \mathcal{C}_1$. For each $i \in \mathcal{C}, j \in \mathcal{H}$, $\mathcal{S}$ also computes the corresponding Pedersen commitment randomness $[\![\hat{z}(i)]\!]_j$ as:

$$[\![\hat{z}(i)]\!]_j = \left([\![z'(i)]\!]_j - [\![z(i)]\!]_j\right) \alpha^{-1} + [\![\hat{z}'(i)]\!]_j$$

This ensures $c_{i,j} = g^{[\![z(i)]\!]_j} h^{[\![\hat{z}(i)]\!]_j}$. Then, on behalf of each node $j \in \mathcal{H}$, $\mathcal{S}$ sends $\langle \texttt{RANDEX}, [\![z(i)]\!]_j, [\![\hat{z}(i)]\!]_j \rangle$ to each node $i \in \mathcal{C}$.

Next, for each node $j \in [n] \setminus \mathcal{C}$, $\mathcal{S}$ does the following: (i) It ignores the RANDEX messages sent to it; (ii) It computes $g^{z(j)}$ by interpolating with $g^{z(i)}$ for each $i \in \mathcal{C}$ and $g^{z(0)}$ in the exponent; (iii) It computes $h^{\hat{z}(j)}$ as $g^{z'(j)} h^{\hat{z}'(j)} / g^{z(j)}$; (iv) It generates NIZK proof π_j for proof of knowledge of $g^{z(j)}$ and $h^{\hat{z}(j)}$ using the NIZK simulator; (v) Lastly, it sends $\langle \texttt{KEY}, g^{z(j)}, h^{\hat{z}(j)}, \pi_j \rangle$ on behalf of node j.

Theorem 1 (Security)**.** *The joint distribution of the honest parties' output and $\mathcal{A}$'s view is identically distributed in the ideal world and the real protocol execution.*

Proof. We first observe that $\mathcal{A}$'s shares of the secret key, the threshold public keys, and the ADKG public key in the simulated world match those output by the $\mathcal{F}_{\mathsf{ADKG}}$. Then, we will prove that $\mathcal{A}$'s view is identically distributed in the real protocol execution and the simulated world through a sequence of hybrids.

Hybrid 0. This corresponds to the real-world execution.

Hybrid 1. Same as Hybrid 0 except that the common random string element h is sampled as g^α for a known uniform random $\alpha \in \mathbb{Z}_q$. Hybrid 1 is indistinguishable from Hybrid 0 as the distribution of h is identical in both hybrids.

Hybrid 2. Same as Hybrid 1, except that we simulate the NIZK proofs of equality of discrete logarithms used during the key-derivation phase. Since Schnorr's protocol [50] for proving the knowledge of discrete logarithm is perfect zero-knowledge, Hybrid 2 is identically distributed as Hybrid 1.

Hybrid 3. This corresponds to the simulated world.

The differences between Hybrid 3 from Hybrid 2 are that the randomness extraction messages of all honest parties are generated by steps 5 and 6 of Figure 3, and the KEY messages as generated by step 7 of Figure 3.

Hybrid 3 is identically distributed as Hybrid 2 due to the perfect hiding of the Pedersen commitment scheme, the perfect secrecy of Shamir secret sharing, and the fact that the output of the randomness extractor is uniformly random. The perfect hiding property of the Pedersen commitment scheme reveals no information about the underlying message. The security of the $(n, t+1)$ Shamir secret sharing scheme ensures that less than or equal to t shares reveal no information about the remaining shares. Thus, when we replace the output of the randomness extraction phase with uniformly random shares received from $\mathcal{F}_{\mathsf{ADKG}}$ (while ensuring consistency of RANDEX and KEY messages sent by honest nodes), Hybrid 3 maintains the same distribution as Hybrid 2.

Lastly, consider the joint distribution of honest nodes output and view of $\mathcal{A}$'s view in both the real world and the ideal world. In both worlds, the view of the $\mathcal{A}$, in particular the threshold public keys, uniquely determines the honest parties' outputs. In the real execution, Lemma 1 guarantees that the honest parties output the same threshold public keys and their secret shares are consistent with the threshold public keys. In

the ideal world, $\mathcal{S}$ ensures that the threshold public keys in $\mathcal{A}$'s view and the shares of nodes in $\mathcal{C}$ match the $\mathcal{F}_{\mathsf{ADKG}}$ output. Therefore, the fact that Hybrid 0 is identically distributed as Hybrid 3 also implies that the joint distribution of the honest parties' output and the $\mathcal{A}$'s view is identically distributed in the ideal world and the real protocol execution. □

5.2 Analysis of Optimization in §4.5

It is easy to see that the in the optimistic case, an honest node will perform only $O(n)$ group exponentiations. Now, we will illustrate that if an adversary $\mathcal{A}$ can violate correctness, i.e., make any honest node output $g^{z'(0)}$ for $z'(0) \neq z(0)$, we can use $\mathcal{A}$ to build an adversary $\mathcal{A}_{\text{DL}}$ that breaks the discrete logarithm assumption.

$\mathcal{A}_{\text{DL}}$ upon receiving the discrete logarithm tuple (g,h), runs the ADKG simulator up until (including) the randomness extraction phase of the ADKG protocol. This implies that $\mathcal{A}_{\text{DL}}$ knows both $z(0)$ and $\hat{z}(0)$. Let $g^{z'(0)}$ and $h^{\hat{z}'(0)}$ be the ADKG public key and the associated randomness, output by any honest node. Also, let α be such that $h = g^{\alpha}$. Then, by construction:

$$\begin{aligned} & c_0 = g^{z(0)}h^{\hat{z}(0)} = g^{z'(0)}h^{\hat{z}'(0)} \\ \implies & z(0) + \alpha\hat{z}(0) = z'(0) + \alpha\hat{z}'(0) \\ \implies & \alpha = \frac{z'(0) - z(0)}{\hat{z}(0) - \hat{z}'(0)} \end{aligned} \tag{7}$$

Since we assume $z(0) \neq z'(0)$, equation (7) is well defined.

For every `KEY` message from node i with valid proof-of-knowledge proof, $\mathcal{A}_{\text{DL}}$ uses the proof-of-knowledge extractor to extract $z'(i)$ and $\hat{z}'(i)$. $\mathcal{A}_{\text{DL}}$ then computes $z'(0)$ and $\hat{z}'(0)$ by interpolation and computes α, the discrete logarithm of h with respect to g, using equation (7).

Finally, let $\tilde{z}(\cdot)$ be the polynomial of degree ℓ, such that a honest node output $g^{\tilde{z}(k)}$ for every $k \in [n]$ as the ADKG public key. Then, $\tilde{z}(\cdot) = z(\cdot)$ as they agree on all n points, including $n - t \geq \ell$ points sent by honest nodes.

5.3 Performance

Lemma 2. *The expected total communication cost of our ADKG protocol is $O(\kappa n^3)$.*

Proof. The sharing phase consists of $O(n)$ ACSS instances, as for any given $\ell \in [t, n-t-1]$, each node need to share at most two uniform random secrets. Each of which has a communication cost of $O(\kappa n^2)$. The MVBA protocol from Das et.al. [22] has cost $O(\kappa n^3)$. In the randomness extraction phase, every node sends an $O(\kappa)$-size message to every other node, which has cost $O(\kappa n^2)$ in total. In the key derivation phase, every node broadcasts an $O(\kappa)$-size message, which has cost $O(\kappa n^2)$ in total. Therefore, the total communication cost of our ADKG protocol is $O(\kappa n^3)$. □

Lemma 3. *The expected computation cost per node in our ADKG protocol is $O(n^2)$, measured in the number of elliptic curve exponentiations.*

Proof. Each node incurs the computation cost of one ACSS dealer and $n-1$ ACSS non-dealer node. During the agreement phase each node incurs the computation cost of one RBC broadcaster, $n-1$ RBC non-broadcaster node, and the computation cost of n parallel ABA instances.

In each ACSS instance, each node needs to perform $O(n)$ exponentiations, hence a total of $O(n^2)$ exponentiations in the sharing phase. Also, in the MVBA protocol, each node incurs a computation cost of $O(n^2)$ group exponentiations (ref. §4.6). During the randomness extraction phase, in the worst case, each node incurs $O(n^3 \log n)$ computation cost. However, these costs are due to Reed-Solomon decoding and do not involve any elliptic curve operations and hence are not a bottleneck. Finally, during the key derivation phase, each node needs to perform $O(n \log n)$ group exponentiations in the worst case. □

Lemma 4. *Our ADKG protocol terminates in $O(\log n)$ rounds in expectation.*

Proof. The ACSS has expected $O(1)$ round latency, the MVBA protocol has expected $O(\log n)$ round latency [22], and rest of the ADKG protocol has constant steps. Therefore the expected latency of the protocol is $O(\log n)$ rounds. □

Remark. Like [22], our protocol will also terminate in $O(1)$ rounds in the common case where there is synchrony and no failures. We refer the reader to [22] for more details.

Combining all of the above, we get the following theorem.

Theorem 2 (ADKG)**.** *In a network of $n \geq 3t+1$ nodes where a PPT adversary $\mathcal{A}$ corrupts up to t nodes and additionally observes shares of $\ell - t$ additional nodes, assuming hardness of discrete logarithm, Algorithm 1 implements a high-threshold ADKG protocol with expected communication cost of $O(\kappa n^3)$, expected computation cost of $O(n^2)$ per node and expected $O(\log n)$ rounds (κ is the security parameter).*

6 Extension and Other Applications

As we mention in §1, the core component – specifically, the first three phases, sharing, agreement, and randomness extraction – of our high-threshold ADKG protocol can be distilled as a mechanism to secret share a random polynomial of degree ℓ. Here on, we will refer to this as the *distributed polynomial sampling*. A distributed polynomial sampling protocol for a network of n nodes $\{1, 2, \ldots, n\}$ guarantees that after the protocol execution, each node i outputs $z(i)$, which is some uniformly random degree-ℓ polynomial $z(\cdot)$ evaluated at i. We will illustrate below that it has other applications besides ADKG.

6.1 Asynchronous Random Double Sharing

Our distributed polynomial sampling protocol above also implies an asynchronous protocol to generate *double sharings* of random values, which means generating a degree-t sharing and a degree-$2t$ sharing of some random secret z. Consider the secret z and let $\ell = 2t$. After the randomness extraction phase, each node i holds $z(i)$ where $z(\cdot)$ is a polynomial of degree $2t$. Moreover, as a result of multiplying $[\![\boldsymbol{a}]\!]_i$ with the hyperinvertible matrix, node i also receives a share of z on a degree-t polynomial. Now, we show an application of our double sharing protocol.

Secure multi-party computation (MPC) allows different parties to jointly evaluate a function over their inputs while keeping the inputs secret [56]. Typically the function is expressed as an arithmetic circuit that contains addition and multiplication gates, and the parties compute the function by evaluating additions and multiplications of the secret shares of their input values. While the addition of secret shares is straightforward, multiplication is more involved. One approach for secure multiplication of secret shared values is using double sharing of random values [19].

Briefly, the secure multiplication protocol proceeds in an offline-online manner [40]. During the offline phase, nodes prepare double shares of random values via our double sharing protocol above, i.e., nodes receive secret shares of a random field element z using both degree t and degree $2t$ polynomials, denoted $[\![z]\!]^t$ and $[\![z]\!]^{2t}$, respectively. Note that the offline phase is independent of the values that will be multiplied later in the online phase.

During the online phase, nodes perform the secure multiplication. Given degree-t secret sharing of x and y, i.e., $[\![x]\!]^t$ and $[\![y]\!]^t$, nodes obtain degree-t secret share of xy as follows. Each node i locally multiplies its shares $[\![x]\!]^t_i$ and $[\![y]\!]^t_i$ to get the share $[\![xy]\!]^{2t}_i$. Nodes than publicly reconstruct the value $xy+z$ by revealing $[\![xy]\!]^{2t}_i + [\![z]\!]^{2t}_i$. Upon reconstructing $xy+z$, nodes compute their share $[\![xy]\!]^t$ of the multiplication as:

$$[\![xy]\!]^t_i = xy + z - [\![z]\!]^t_i \tag{8}$$

Security analysis. Our security analysis will be based on the simulation argument, i.e., we will illustrate that, given the adversarial shares, a simulator, $\mathcal{S}$, can simulate the rest of the protocol transcript. Without loss of generality, let us assume the adversary $\mathcal{A}$ corrupts the first t nodes. Given the adversarial shares, $\mathcal{S}$ samples random values for z_i for each $i \in [0,\ell]$. $\mathcal{S}$ then uses these values to compute $z(k)$ for each $k \in [n]$. Then, for each $k \in [n]$, $\mathcal{S}$ uses $z(k)$ and shares of adversarial nodes to compute $[\![z(k)]\!]_j$ for every $j \in [t+1,n]$.

6.2 Proactive Secret Sharing

In proactively secure systems, nodes periodically refresh their secrets to prevent attacks against long-term adversaries [10]. Our distributed polynomial sampling protocol will also be useful in proactively secure systems such as CHURP [42] and COBRA [55] where nodes want to refresh their secrets using possibly high-degree polynomials. In particular, nodes will sample the coefficients $z_1, z_2, \ldots, z_\ell$ and set $z_0 = 0$. Each node i then locally adds $z(i)$ to their old share to get the newly updated share.

We want to note that, although our distributed polynomial sampling is secure against an adversary that corrupts up to t nodes during the share refresh protocol, the distributed polynomial sampling can be used to re-randomize secrets shared using degree ℓ polynomial. More specifically, the refresh protocol is secure only if the adversary corrupts at most t nodes during the refresh protocol; but once the refresh protocol terminates, the adversary can learn up to ℓ new shares.

7 Implementation and Evaluation

7.1 Implementation Details

We have implemented a prototype of our ADKG protocol in `python 3.7.13` on top of the open-source asynchronous DKG implementation of [22]. Our implementation supports any reconstruction threshold $\ell \geq t$.

We use `rust` libraries for elliptic curve operations and `asyncio` for concurrency, though our prototype only runs on a single processor core. We use the low-threshold ACSS protocol from [21, §5.3]. Our implementation uses the asynchronous binary agreement protocol from [18] and reliable broadcast protocol from [21].

In our implementation, we use both the `curve25519` and `bls12381` elliptic curves. We use the Ristretto group over `curve25519` implementation from [2] and the `bls12381` implementation from Zcash [31] (with a `python` wrapper around each) for primitive elliptic curve operations. Note that `bls12381` supports pairing, so our implementation can be used for pairing-based threshold cryptosystems such as [8]. However, a downside of pairing friendly curves is that they are less efficient for applications that do not need them, in terms of both communication and computation costs. For example, a group element in `curve25519` is 32 bytes, whereas group elements in `bls12381` are 48 and 96 bytes. Furthermore, our micro-benchmark illustrates that a group exponentiation in `bls12381` is 6× slower than that of `curve25519`.

MVBA implementation. We reuse the MVBA protocol implementation from [22]. As a result, we inherit the optimization that the shared randomness for an asynchronous binary agreement (ABA) protocol is computed only if it is needed by the ABA instance [22, Appendix A]. We also merge the sharing phase of the MVBA protocol with the sharing phase of our high-threshold ADKG protocol. More specifically, in the sharing phase of the MVBA protocol, nodes need to secret share a random secret using low-threshold ACSS. Since the MVBA protocol of [22] also uses the low-threshold ACSS of [21], the communication pattern of their sharing phase is

Table 3: Evaluation of high-threshold ADKG schemes. Average runtime is measured as the average time difference between the start of the ADKG and the time a node output keys. Bandwidth usage is the amount of data a node sends during the ADKG protocol.

Scheme	Curve	ℓ	Avg. runtime (in seconds)				Bandwidth usage/ node (in Megabytes)			
			$n=16$	$n=32$	$n=64$	$n=128$	$n=16$	$n=32$	$n=64$	$n=128$
Das et al. [22]	curve25519	t	1.10	2.96	9.56	39.56	0.17	0.68	2.78	11.24
	bls12381	t	1.32	3.16	10.66	42.28	0.17	0.73	2.96	11.98
	curve25519	$2t$	10.64	37.76	152.56	—	0.95	4.20	18.97	—
	bls12381	$2t$	11.50	40.43	164.37	—	0.97	4.13	19.29	—
This work	curve25519	$2t$	1.29	3.19	12.48	46.55	0.20	0.82	3.32	13.10
	bls12381	$2t$	1.47	4.03	14.50	50.35	0.21	0.84	3.48	13.44

identical to ours. Thus, we merge the ACSS instances into a single ACSS instance that secret shares three random secrets.

7.2 Evaluation Setup

We evaluate our ADKG implementation with a varying number of nodes: 16, 32, 64, 128. For a given $n \geq 3t+1$, we evaluate with a reconstruction threshold of $\ell = 2t$. Note that the bandwidth usage of our algorithm is identical for any $\ell \in [t, n-t-1]$; the impact of ℓ on computation is also insignificant. So our experimental results are representative of any reconstruction threshold.

We run all nodes on Amazon Web Services (AWS) *t3a.medium* virtual machines (VM) with one node per VM. Each VM has two vCPUs and 4GB RAM and runs Ubuntu 20.04. We place nodes evenly across eight AWS regions: Canada, Ireland, North California, North Virginia, Oregon, Ohio, Singapore, and Tokyo. We create an overlay network among nodes where all nodes are pair-wise connected, i.e., they form a complete graph.

Baselines. The only existing asynchronous DKG implementation is for the protocol [22]. Thus, we only compare with their protocol.

7.3 Evaluation Results

With our evaluation, we aim to demonstrate that our high-threshold ADKG protocol is significantly more scalable than the prior-best high-threshold ADKG protocol.

Runtime. We measure the time difference between the start of the ADKG protocol and when a node outputs the shared public key and its secret share. We then average this time across all nodes to compute the runtime of our ADKG protocol. We report the results in Table 3.

For $\ell = 2t$ and `curve25519` as the elliptic curve, our high-threshold ADKG protocol takes approximately 12.48 seconds for 64 nodes, which is only 8.2% that of the high-threshold ADKG protocol of [22], for the same experimental setup. Also, our high-threshold ADKG protocol introduces only 30% additional overhead compared to their low-threshold ADKG protocol.

Bandwidth usage. We measure bandwidth usage as the amount of bytes sent by a node in the entire ADKG protocol. We report bandwidth usage per node in Table 3. Consistent with the analysis from §5, the bandwidth usage of our protocol increases quadratically with the number of nodes.

Our bandwidth usage is significantly lower than the high-threshold ADKG protocol of [22]. Using the 64 nodes experiment, with `curve25519` as the elliptic curve, each node in [22] sends 18.9 Megabytes of data; whereas, in our protocol, each node only sends 3.32 Megabytes, which is only about 18% of the bandwidth usage of [22].

We also note that, although in `bls12381` group elements are 16 bytes longer than in `curve25519`, this does not noticeably affect the total protocol bandwidth usage due to the comparable costs of other data, field integers, and hashes.

Remark. Note that the evaluation results reflect the common-case performance of both protocols, which have the same asymptotic $O(n^2)$ computation cost per node. Our evaluation demonstrates that our protocol has significantly smaller constants in the computation cost compared to Das et al. [22] for the common case. For the worst-case computation cost, compared to [22], our protocol improves the per node worst-case computation cost from $O(n^3)$ group operations to $O(n^2)$ group exponentiations (with further optimization using multi-exponentiations [43]). Concretely, our worst-case computation cost is similar to our common-case computation cost.

8 Related Work

Numerous works have studied the problem of Distributed Key Generation with various cryptographic assumptions, network conditions and with other properties [3, 14, 15, 21, 26, 27, 29, 32, 34, 36, 37, 39, 44, 49]. We will roughly categorize prior works into two categories based on the network assumption: *Synchrony* and *Asynchrony*.

Synchronous DKG. DKG in the synchronous network has been studied for decades [14, 15, 26, 29, 32, 34, 44, 47, 49, 52]. The first DKG protocol [47] was proposed by Pedersen which was later shown, by Gennaro et al. [29], to allow an attacker to bias the public-key [29]. Gennaro et al. also proposed an expensive approach to fix this issue, which was later improved

by Neji et al. [44]. Canetti et al. [14] extended Gennaro et al. [29] to be adaptively secure. Gurkan et al. [34] designed a publicly verifiable secret sharing (PVSS) based DKG protocol with a linear size public-verification transcript but with only $O(\log n)$ fault-tolerance and has the secret key as a group element. Based on a new PVSS scheme, Groth [32] designed a new DKG protocol that is non-interactive, assumes the existence of a broadcast channel, and has the secret key as a field element. Recently, Shrestha et al. [52], proposed a new DKG protocol with a communication cost of $O(\kappa n^3)$ that does not rely on a Byzantine broadcast channel.

Asynchronous DKG. Our protocol is closely related to the protocol of [22] and follows the same high-level idea. Each node shares a secret via n parallel instances of ACSS and then agrees on a large set of finished ACSS instances for aggregating the corresponding shares to obtain the final key set. Naturally, the high-threshold ADKG protocol of [22] relies on a high-threshold ACSS and this is the primary source of inefficiency in their high-threshold ADKG. More concretely, in terms of computation cost, their high-threshold ACSS is 2 to 3 orders of magnitude more expensive than the low-threshold counterpart (500 seconds compared to 0.2 seconds). Their ACSS also results in a high-threshold ADKG with $6\times$ or more communication cost.

Our new construction circumvents the bottleneck of having the expensive high-threshold ACSS by solving the high-threshold ADKG based on the following insight. The problem of ADKG with reconstruction threshold ℓ is nothing but sampling a random polynomial of degree ℓ such that every node learns only one evaluation point on the polynomial. With this insight, the new goal is to sample $\ell+1$ random coefficients, which is equivalent to sampling a random polynomial of degree ℓ. Thus, we design a protocol to sample $\ell+1$ coefficients in a secret shared manner using only low-threshold ACSS instances. As a result, compared to [22], our protocol only relies on the low-threshold ACSS schemes and is almost as efficient as the low-threshold ADKG protocol.

A handful of other works has also studied the DKG problem in partially synchronous or asynchronous networks [3, 21, 27, 37, 39]. For partial synchrony, the protocol of Kate et al. [37] has $O(\kappa n^4)$ total communication cost and one-third resilience. Tomescu et al. [54] improves Kate et al. [37] by a factor of $O(n/\log n)$ in computation at the cost of a logarithmic increase in communication.

For asynchrony, the first DKG scheme by Kokoris et al. [39] has a total communication cost of $O(\kappa n^4)$ and expected round complexity of $O(n)$. Abraham et al. [3] proposed an ADKG protocol with a communication cost of $O(\kappa n^3 \log n)$, and was later improved to $O(\kappa n^3)$ by Gao et al. [27] and Das et al. [21], respectively. However, since Abraham et al. use the PVSS scheme of Gurkan et al. [34], all three constructions [3, 21, 27] of ADKG are not compatible with off-the-shelf threshold encryption or signature schemes, as they inherit the limitation that the secret key is a group element.

DKG implementations. There have been many synchronous DKG implementations [1, 30, 35, 45, 48, 49, 53], but the only asynchronous DKG implementation is due to [22].

9 Conclusion

We presented a simple and concretely efficient protocol for high-threshold asynchronous distributed key generation for discrete logarithm based threshold cryptosystems. At the core of our protocol is a novel mechanism to sample a polynomial of any specified degree in a distributed manner such that each node learns only one evaluation point on the polynomial. Our distributed polynomial sampling protocol uses low-threshold asynchronous complete secret sharing, and multi-valued validate byzantine agreement protocol in a modular way. As a result, an improved protocol for these primitives would immediately improve our distributed key generation protocol.

In a network of n nodes, our protocol improved the worst-case computation cost by a factor of n while maintaining the expected communication cost and expected round complexity of $O(\kappa n^3)$ and $O(\log n)$, respectively. Finally, we provide a prototype implementation and evaluate our prototype using up to 128 geographically distributed nodes to illustrate the efficiency of our protocol.

In this paper, we prove the security of our ADKG protocol in the stand-alone setting. However, note that our simulator is straight-line, i.e., it does not rewind the adversary. Hence, we believe it is possible to extend our security proof to the Universal composability (UC) framework [13]. We leave the formal security proof in the UC framework to future work.

Acknowledgments

The authors would like to thank Amit Agarwal, Andrew Miller, and Tom Yurek for the helpful discussions related to the paper. This work is funded in part by a VMware early career faculty grant, a Chainlink Labs Ph.D. fellowship, the National Science Foundation, and the Austrian Science Fund (FWF) F8512-N.

References

[1] Drand - a distributed randomness beacon daemon, 2020. https://github.com/drand/drand.

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

[3] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation.

In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 363–373, 2021.

[4] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

[5] Nicolas Alhaddad, Mayank Varia, and Haibin Zhang. High-threshold avss with optimal communication complexity. In *International Conference on Financial Cryptography and Data Security*, pages 479–498. Springer, 2021.

[6] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In *Theory of Cryptography Conference*, pages 213–230. Springer, 2008.

[7] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 52–61, 1993.

[8] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.

[9] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[10] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 88–97, 2002.

[11] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.

[12] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[13] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[14] Ran Canetti, Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Adaptive security for threshold cryptosystems. In *Annual International Cryptology Conference*, pages 98–116. Springer, 1999.

[15] John Canny and Stephen Sorkin. Practical large-scale distributed key generation. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 138–152. Springer, 2004.

[16] Ignacio Cascudo and Bernardo David. Scrape: Scalable randomness attested by public entities. In *International Conference on Applied Cryptography and Network Security*, pages 537–556. Springer, 2017.

[17] Tyler Crain. A simple and efficient asynchronous randomized binary byzantine consensus algorithm. *arXiv preprint arXiv:2002.04393*, 2020.

[18] Tyler Crain. Two more algorithms for randomized signature-free asynchronous binary byzantine consensus with $t < n/3$ and $o(n^2)$ messages and $o(1)$ round expected termination. *arXiv preprint arXiv:2002.08765*, 2020.

[19] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Advances in Cryptology-CRYPTO 2007: 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007. Proceedings 27*, pages 572–590. Springer, 2007.

[20] Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. Spurt: Scalable distributed randomness beacon with transparent setup. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2502–2517. IEEE, 2022.

[21] Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[22] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2518–2534. IEEE, 2022.

[23] Yvo G Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–458, 1994.

[24] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[25] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[26] Pierre-Alain Fouque and Jacques Stern. One round threshold discrete-log key generation without private channels. In *International Workshop on Public Key Cryptography*, pages 300–316. Springer, 2001.

[27] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. *arXiv preprint arXiv:2106.07831*, 2021.

[28] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *International conference on financial cryptography and data security*. Springer, 2022.

[29] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.

[30] GNOSIS. Distributed key generation, 2020. https://www.nongnu.org/dkgpg/.

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

[32] Jens Groth. Non-interactive distributed key generation and key resharing. *IACR Cryptol. ePrint Arch.*, 2021:339, 2021.

[33] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 803–818, 2020.

[34] Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 147–176. Springer, 2021.

[35] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

[36] Aniket Kate and Ian Goldberg. Distributed key generation for the internet. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 119–128. IEEE, 2009.

[37] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed key generation in the wild. *IACR Cryptol. ePrint Arch.*, 2012:377, 2012.

[38] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: private data management for decentralized ledgers. *Proceedings of the VLDB Endowment*, 14(4):586–599, 2020.

[39] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1751–1767, 2020.

[40] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 887–903, 2019.

[41] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, pages 129–138, 2020.

[42] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. Churp: dynamic-committee proactive secret sharing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2369–2386, 2019.

[43] Bodo Möller. Algorithms for multi-exponentiation. In *International Workshop on Selected Areas in Cryptography*, pages 165–180. Springer, 2001.

[44] Wafa Neji, Kaouther Blibech, and Narjes Ben Rajeb. Distributed key generation protocol with a new complaint management strategy. *Security and communication networks*, 9(17):4585–4595, 2016.

[45] Orbs Network. Dkg for bls threshold signature scheme on the evm using solidity, 2018. https://github.com/orbs-network/dkg-on-evm.

[46] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*, pages 129–140. Springer, 1991.

[47] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 522–526. Springer, 1991.

[48] Philipp Schindler. Ethereum-based distributed key generation protocol, 2020. https://github.com/PhilippSchindler/ethdkg.

[49] Philipp Schindler, Aljosha Judmayer, Nicholas Stifter, and Edgar R Weippl. Ethdkg: Distributed key generation with ethereum smart contracts. *IACR Cryptol. ePrint Arch.*, 2019:985, 2019.

[50] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252. Springer, 1989.

[51] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[52] Nibesh Shrestha, Adithya Bhat, Aniket Kate, and Kartik Nayak. Synchronous distributed key generation without broadcasts. *Cryptology ePrint Archive*, 2021.

[53] Heiko Stamer. Distributed privacy guard, 2018. https://github.com/gnosis/dkg.

[54] A. Tomescu, R. Chen, Y. Zheng, I. Abraham, B. Pinkas, G. Gueta, and S. Devadas. Towards scalable threshold cryptosystems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 877–893, 2020.

[55] Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira, and Alysson Bessani. Cobra: Dynamic proactive secret sharing for confidential bft services. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1528–1528. IEEE Computer Society, 2022.

[56] Avi Wigderson, MB Or, and S Goldwasser. Completeness theorems for noncryptographic fault-tolerant distributed computations. In *Proceedings of the 20th Annual Symposium on the Theory of Computing (STOC'88)*, pages 1–10, 1988.

[57] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019.

[58] Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. hbacss: How to robustly share many secrets. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium*, 2022.

[59] Thomas Yurek, Zhuolun Xiang, Yu Xia, and Andrew Miller. Long live the honey badger: Robust asynchronous dpss and its applications. *Cryptology ePrint Archive*, 2022.

[60] Jiaheng Zhang, Tiancheng Xie, Thang Hoang, Elaine Shi, and Yupeng Zhang. Polynomial commitment with a {One-to-Many} prover and applications. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2965–2982, 2022.

A Online Error Correction

Our protocol uses the Online-Error-Correction (OEC) protocol introduced by Ben-Or, Canetti, and Goldreich [7]. The OEC takes a set T consisting of tuples (j, a_j) where j is an index $j \in [n]$ and a_j is a fragment of a Reed-Solomon codeword. The OEC algorithm then tries to decode a message M such that Reed-Solomon encoding of M matches with at least $2t+1$ elements in T. More specifically, the OEC algorithm performs up to t trials of reconstruction, and during the r-th trial, it uses $2t+r+1$ elements in T to decode. If the reconstructed message M' whose encoding matches with at least $2t+1$ tuples in T, the OEC algorithm successfully outputs the message; otherwise, it waits for one more fragment and tries again. We summarize the OEC algorithm in Algorithm 2. The OEC algorithm is error-free and information-theoretically secure against any adversary that corrupts up to t fragments among a total of $n \geq 3t+1$ fragments.

Algorithm 2 Online Error-correcting (OEC)

1: **Input:** T *// T consisting of tuples (j, a_j) where $j \in [n]$ and a_j is a fragment*
2: **for** $0 \leq r \leq t$ **do** *// online Error Correction*
3: Wait till $|T| \geq 2t+r+1$
4: Let $M := \mathsf{RSDec}(t+1, r, T)$
5: Let $T' := \mathsf{RSEnc}(M, m, t+1)$
6: **if** $2t+1$ fragments in T' match with T **then**
7: return M

B Asynchronous Complete Secret Sharing

We describe the Pedersen commitment based ACSS protocol due to [21], which improves upon the ACSS scheme of Yurek et al. [58]. Here, we will only describe part of the sharing phase of the protocol, as this is what is needed to understand our protocol. During the sharing phase, the dealer chooses two random polynomials $a(x)$ and $\hat{a}(x)$ of degree t where:

$$a(x) = a_0 + a_1 x + a_2 x^2 + \cdot + a_t x^t \tag{9}$$

$$\hat{a}(x) = \hat{a}_0 + \hat{a}_1 x + \hat{a}_2 x^2 + \cdot + \hat{a}_t x^t \tag{10}$$

Let $a_0 = m$. The dealer then reliably broadcasts $c_i = g^{a_i} h^{\hat{a}_i}$ for each $i \in [0, t]$. Additionally, the dealer sends the tuple

$a(i), \hat{a}(i)$ to node i. Node i, upon receiving α, β from the dealer, checks the validity of the received message by checking whether:

$$g^{\alpha} h^{\beta} = \prod_{k=0}^{t} (c_k)^{i^k} \tag{11}$$

If the above check passes, node i sends a vote for the dealer. Otherwise, node i multi-casts a complaint against the dealer and triggers the fallback protocol.