2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833584

Practical Asynchronous Distributed Key Generation

Sourav Das*, Thomas Yurek*, Zhuolun Xiang*, Andrew Miller*, Lefteris Kokoris-Kogias†, and Ling Ren*
*University of Illinois at Urbana-Champaign, †IST Austria
{souravd2, yurek2, xiangzl, soc1024, renling}@illinois.edu, ekokoris@ist.ac.at

***Abstract*—Distributed Key Generation (DKG) is a technique to bootstrap threshold cryptosystems without a trusted third party and is a building block to decentralized protocols such as randomness beacons, threshold signatures, and general multiparty computation. Until recently, DKG protocols have assumed the synchronous model and thus are vulnerable when their underlying network assumptions do not hold. The recent advancements in asynchronous DKG protocols are insufficient as they either have poor efficiency or limited functionality, resulting in a lack of concrete implementations.**

In this paper, we present a simple and concretely efficient *asynchronous* DKG (ADKG) protocol. In a network of n nodes, our ADKG protocol can tolerate up to $t < n/3$ malicious nodes and have an expected $O(\kappa n^3)$ communication cost, where κ is the security parameter. Our ADKG protocol produces a field element as the secret and is thus compatible with off-the-shelf threshold cryptosystems. We implement our ADKG protocol and evaluate it using a network of up to 128 nodes in geographically distributed AWS instances. Our evaluation shows that our protocol takes as low as 3 and 9.5 seconds to terminate for 32 and 64 nodes, respectively. Also, each node sends only 0.7 Megabytes and 2.9 Megabytes of data during the two experiments, respectively.

I. Introduction

A Distributed Key Generation (DKG) protocol enables a set of mutually distrustful nodes to jointly generate a public/private key pair. The private key is secret-shared among the nodes via a threshold secret sharing scheme and is never reconstructed or stored at a single node. The secret-shared private keys can later be used in a threshold cryptosystem, e.g., to produce threshold signatures [9], [31], to decrypt ciphertexts of threshold encryption [21], [42] or to generate common coins [12] for consensus [41], [29], [30], [18], [32].

The increasing demand for decentralized Byzantine Fault Tolerant (BFT) applications over the Internet revived interests in DKG protocols. Many state-of-the-art BFT protocols use threshold signatures to improve communication efficiency [57], [5], [44], [29] and/or threshold encryptions to prevent censorship [45], [22], [44], [36]. For asynchronous BFT protocols [45], [22], [27], [44], [36], [41] that assume no bounded message delay, shared randomness is required to circumvent the FLP impossibility [25]. All of these threshold cryptographic primitives require nodes to have secret shares of a private key. The naïve way to bootstrap them is to rely on a trusted dealer whose corruption will break the entire system. A DKG protocol is necessary to bootstrap the above threshold cryptographic primitives while avoiding any central trust or single point of failure.

Numerous DKG protocols are known when the underlying network is synchronous [50], [13], [26], [14], [31], [47], [37], [53], [35] (see §IX). In contrast, only a handful of recent works have looked into DKG for asynchronous networks, which we call *asynchronous* DKG (ADKG) [43], [4], [28], [19]. Kokoris et al. [43] presented the first ADKG protocol. Their construction uses n concurrent high-threshold asynchronous complete secret sharing (ACSS) schemes to construct an ADKG protocol that has an expected total communication cost of $O(\kappa n^4)$ and terminates in expected $O(n)$ rounds. Here κ is the security parameter. Recently, Abraham et al. [4] proposed a special-purpose of ADKG protocol with an expected total communication cost of $O(\kappa n^3 \log n)$, which is later improved to $O(\kappa n^3)$ by Gao et al. [28] and Das et al. [19]. We say these ADKG schemes are *special-purpose* because the distributed secret key is a group element rather than a field element (i.e., g^z rather than z), so they cannot be used in most off-the-shelf threshold encryption [21] or threshold signature protocols [9]. We summarize existing works in Table I.

Our results. In this paper, we design a new simple and concretely efficient ADKG 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 ADKG protocol achieves an expected communication cost of $O(\kappa n^3)$ and terminates in expected $O(\log n)$ rounds. Hence, our protocol improves upon the prior known general-purpose ADKG protocol of Kokoris-Kogias et al. [43] by a factor of n in communication and a factor of $n/\log n$ in expected runtime. For setup assumption, Kokoris-Kogias et al. [43] assumes Random Oracle (RO), and our protocol assumes RO and PKI (PKI needed only for our ACSS construction).

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 field of size q. Thus, our protocol is compatible with off-the-shelf discrete-logarithm based threshold cryptosystems [21], [9], [31].

Our protocol also supports any reconstruction threshold $\ell \in [t + 1, n - t]$, i.e., ℓ nodes are required to use the secret key z (e.g., to produce a threshold signature or decrypt a threshold encryption). To get this property efficiently, we design a new additively homomorphic high-threshold

© 2022, Sourav Das. Under license to IEEE.

Table I: Comparison of existing DKG protocols. We use $\mathcal{B}(L)$ and $\mathcal{R}$ to denote the communication cost of Byzantine broadcast of L-bit message and round complexity of Byzantine broadcast, respectively. We measure the computation cost in terms of number of group exponentiations.

	Network model	Fault Tolerance	Adaptive Adversary	Secret key from a Field?	High Threshold	Publicly Verifiable	Communication Cost (per node)	Computation Cost (per node)	Total Round Complexity	Cryptographic Assumption	Setup Assumption
Gennaro et al. [31]	sync.	1/2	✗	✓	✓	✗	$O(n \cdot \mathcal{B}(\kappa))$	$O(n^2)$	$O(\mathcal{R})$	DDH	CRS
Canneti et al. [13]	sync.	1/2	✓	✓	✓	✗	$O(n \cdot \mathcal{B}(\kappa))$	$O(n^2)$	$O(\mathcal{R})$	DDH	CRS
Fouque-Stern [26]	sync.	1/2	✗	✓	✓	✓	$O(\mathcal{B}(\kappa n))$	$O(n^2)$	$O(\mathcal{R})$	DCR	RO & PKI
Neji et al. [47]	sync.	1/2	✗	✓	✓	✗	$O(n \cdot \mathcal{B}(\kappa))$	$O(n^2)$	$O(\mathcal{R})$	DDH	CRS
Gurkan et al. [37]	sync.	$\log n$	✗	✗	✓	✓	$O(\log n \cdot \mathcal{B}(\kappa n) + n \cdot \mathcal{B}(\kappa))$	$O(n \log^2 n)$	$O(\mathcal{R} + \log n)$	SXDH	RO & PKI
Groth [35]	sync.	1/2	✗	✓	✓	✓	$O(\mathcal{B}(\kappa n))$	$O(n^2)$	$O(\mathcal{R})$	DDH	RO & PKI
Kate et al. [40]	partial sync.	1/3	✗	✓	✗	✗	$O(\kappa n^3)$	$O(n^3)$	$O(n)$	DDH	RO & PKI
Kokoris et al. [43]	async.	1/3	✗	✓	✓	✗	$O(\kappa n^3)$	$O(n^3)$	$O(n)$	DDH	RO
Abraham et al. [4], [28], [19]	async.	1/3	✗	✗	✗[†]	✗	$O(\kappa n^2)$	$O(n^2)$	$O(1)$	SXDH	RO & PKI
This work	async.	1/3	✗	✓	✓	✗	$O(\kappa n^2)$	$O(n^3)$[‡]	$O(\log n)$	DDH	RO & PKI

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

[‡] In our protocol, for reconstruction threshold of $t+1$, in the common case in practice, each node only incurs a quadratic computation (see §V)

ACSS scheme with an expected communication cost of $O(\kappa n^2)$. Our high-threshold ACSS assumes the hardness of Decisional Composite Residuosity (DCR) in the Random Oracle model and does not require a trusted setup. Our high-threshold ACSS scheme improves the communication cost by a factor of $\log n$ over the prior best scheme of [6]. This result may be of independent interest.

Evaluation. We implement our ADKG protocol and made our implementation available at https://github.com/sourav1547/adkg. Our implementation supports both `curve25519` and `bls12-381` elliptic curves and any reconstruction threshold in the range $[t+1, n-t]$. We evaluate with up to 128 nodes in geographically distributed Amazon EC2 instances. For a reconstruction threshold of $t+1$ with 32 nodes and either curve, our *single-thread* implementation takes about 3 seconds and each node sends 0.7 Megabytes of data. When the reconstruction threshold is $n-t$, for 32 nodes, our ADKG takes 38 and 41 seconds for `curve25519` and `bls12-381` elliptic curves, respectively, while each node sends approximately 4.2 Megabytes of data.

Paper Organization. The rest of the paper is organized as follows. In §II we describe our system model, define the ADKG problem, and present an overview of our ADKG protocol. We then describe preliminaries used in our protocol in §III. We present the detailed design of our ADKG protocol in §IV and analyze it in §V. In §VI we briefly describe how to extend our ADKG protocol to support a reconstruction threshold of up to $n-t$. We then describe our new additively homomorphic high-threshold ACSS scheme with quadratic communication cost in §VII. In §VIII we provide implementation details and our evaluation results. We discuss related work in §IX and conclude in §X.

II. System Model and Overview

A. *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 output of the hash function. 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 $\{a, a+1, \ldots, b\}$.

Threat model and network assumption. We consider a network of n nodes where every pair of nodes is connected via a pairwise authenticated channel. We consider the presence of a malicious adversary $\mathcal{A}$ that can corrupt up to t of the at least $3t+1$ nodes in the network. 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.

B. *Definition of ADKG*

As mentioned in §I, in this paper, we focus on ADKG for discrete logarithm-based cryptosystems such as ElGamal encryption [23] and BLS signatures [10], [9]. Our definition is inspired from the DKG definition from Gennaro et al. [31].

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 $y = h^z$, where h 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, ℓ)-threshold Shamir share [54] of the secret z, where ℓ shares are needed to use z. More precisely, let $p(\cdot) \in \mathbb{Z}_q[x]$ be a random polynomial of degree $\ell - 1$ such that $p(0) = z$. At the end of the DKG protocol, the i^{th} node outputs its share of the secret key $z_i = p(i)$, and every node outputs the public key $y = h^z$. A DKG protocol is called t-secure if the following *Correctness* and *Secrecy* properties hold in the presence of an adversary $\mathcal{A}$ that corrupts up to t nodes.

- **Correctness:**

(C1) All subsets of ℓ shares provided by honest parties define the same unique secret key z.

(C2) All honest nodes output the same public key $y = h^z$ where z is the unique secret guaranteed by (C1).

(C3) The secret key z is computationally indistinguishable from a uniformly random element in $\mathbb{Z}_q$.

Additionally, applications of DKG such as threshold signatures and threshold encryption require that in addition to y, threshold public keys of all nodes are also publicly known. So we add a fourth correctness requirement.

(C4) All honest nodes agree on and output the threshold public keys of all nodes. The threshold public key of node j is $y_j = h^{z_j}$.

- **Secrecy:** No information about the secret z can be learned by a computationally bounded adversary beyond the public key $y = h^z$.

We define the secrecy property in terms of *simulatability*: for every probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ that corrupts up to t nodes, there exists a PPT simulator $\mathcal{S}$, such that on input of a uniformly random element $y \in \mathbb{G}$, produces a view which is indistinguishable from $\mathcal{A}$'s view of a run of the ADKG protocol that ends with y as its public key output.

Remark. Our property C3 is slightly weaker than the property C3 defined in Gennaro et al. [31]. In particular, Gennaro et al. require that the secret key is uniformly random, whereas we only require the secret key to be computationally indistinguishable from uniform random.

C. Overview of our Protocol

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. Once secret-sharing finishes for $t+1$ nodes, nodes locally aggregate their 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.

Although the idea is simple, there are many challenges for this idea to work in an asynchronous network. The biggest challenge is to agree on which shares to aggregate for the final secret key z. It is well-known that reaching agreement under asynchrony requires randomness [25], often shared randomness [12], [46], [17]. However, existing efficient mechanisms to generate shared randomness assume threshold secret-shared keys, hence creating a circularity. The inefficiency or lack of generality of prior works often results from difficulties in tackling this circularity.

We address this circularity with a new approach illustrated in Figure 1. After nodes finish their secret-sharing step, we let each node i propose, using a reliable broadcast (RBC), a set of nodes that i believes performed the secret-sharing correctly. We refer to the set proposed by node i as the i^{th} *intermediate key set* and denote it using T_i. Then we run n concurrent asynchronous Byzantine binary agreement (ABA) instances. The i^{th} ABA instance uses the i^{th} intermediate key set T_i to generate shared randomness. The output of the i^{th} ABA instance decides whether or not the i^{th} intermediate key set should be included in the final key. Finally, once all ABA instances terminate, we use the approach from Neji et al. [47] to compute the final public key h^z.

However, two challenges remain for the above approach.
Challenge 1. Ensure all honest nodes receive all the shares required to generate shared randomness and to compute their share of the final secret key z.

By definition, an asynchronous verifiable secret sharing (AVSS) scheme allows a situation where enough, but not all, honest nodes receive their shares from the dealer. Such a situation usually arises when a malicious dealer sends valid shares to a subset of honest nodes, and the corrupted nodes also claim to have the shares so that all honest nodes terminate the sharing phase. Hence, with AVSS, it is possible that not all honest nodes received shares of every AVSS instance included in an intermediate key set T_i. Such situations prohibit nodes from aggregating AVSS instances in T_i, which is required for generating shared randomness and computing its share of the final secret key z.

We address this issue using two ideas. First, we use asynchronous complete secret sharing (ACSS) instead of AVSS. A crucial property of ACSS is that it ensures that if it terminates successfully at one honest node, then all honest nodes will eventually receive a valid share. Second, an honest node i participates in the reliable broadcast (RBC) of key set proposal T_j only after every ACSS instance in T_j has terminated at node i. This ensures if a RBC instance delivers T_j at any honest node, then every ACSS instance in T_j will eventually terminate at all honest nodes, which further ensures that honest nodes can construct the shared randomness for j^{th} ABA as well as the final secret key.
Challenge 2. Ensure all ABA instances terminate, even if some malicious nodes do not send their intermediate key sets.

There is a subtle liveness issue in the approach we described so far. If a malicious node j does not propose the j^{th} intermediate key set, then there is no shared randomness available for the j^{th} ABA to circumvent the FLP impossibility [25]. To resolve this issue, we make the crucial observation that the FLP impossibility only applies when the initial state is bivalent, i.e., honest nodes have different inputs. For a univalent initial state, i.e., all honest nodes have the same input, there is no impossibility, and agreement can

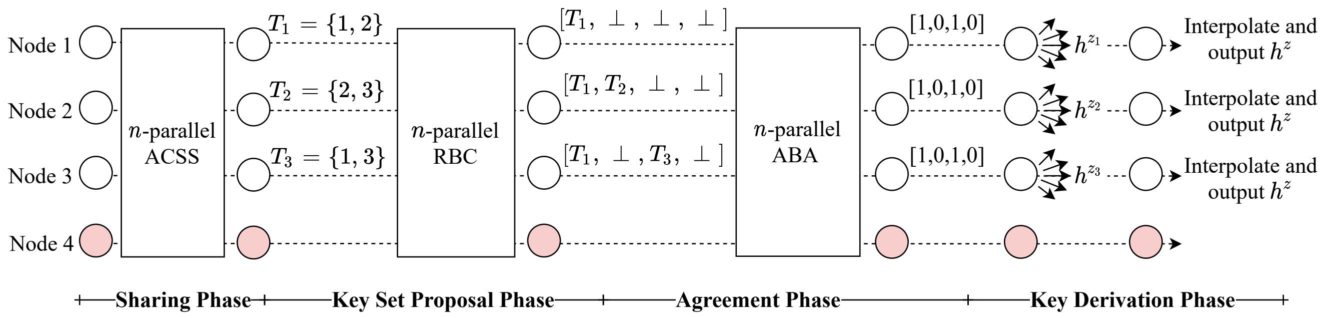

Figure 1: Overview of our ADKG protocol in a network of 4 nodes where node 4 is malicious. During the Sharing phase, each node secret shares a random secret using a ACSS protocol. During the Key Set Proposal phase, each node waits till $t + 1$ ACSS terminates locally. Let T_i be the set of $t + 1$ ACSS instances that terminated at node i. Node i then reliably broadcasts T_i. During the agreement phase, nodes agree on which nodes proposed a valid key set using n-parallel ABAs. Also, for i^{th} ABA nodes use the secrets of nodes in T_i to generate shared randomness. Finally, once all ABA instances terminate, nodes reconstruct the final public key during the Key Derivation phase.

terminate without any randomness. Therefore, we designed our protocol to ensure that if a malicious node does not propose an intermediate key set, it leads to a univalent initial state for ABA. In particular, in such a situation, all honest nodes input 0, and we need the ABA to output 0 without using randomness. We refer to this property as *Good-Case-Coin-Free*, and indeed, the ABA protocol due to Crain [17] (restated in Appendix B) has this property. Hence, either all honest nodes input 0 to ABA and it terminates deterministically or, thanks to ACSS, eventually all honest nodes receive the intermediate key set and thus can generate shared randomness to circumvent FLP.

III. Preliminaries

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

Table II: Notations used in the paper

Notation	Description
n	Total number of nodes
t	Maximum number of malicious nodes
$\mathbb{Z}_q$	Field of order q where q is prime
$\mathbb{G}$	Group of order q where DDH is assumed to be hard
g, h	Random and independent generators of $\mathbb{G}$
z, h^z	ADKG secret and public key
z_i	Secret share of z output by node i
h^{z_i}	Threshold public key of node i
ℓ	Reconstruction threshold of ADKG
κ	Security parameter
pk_i, sk_i	Public and secret keys of i^{th} node.
s_i	Secret chosen by i^{th} node during sharing phase
$p_i(\cdot)$	Polynomial chosen by i^{th} node to share s_i
$\boldsymbol{v}_i$	Feldman commitment of the polynomial $p_i(\cdot)$
$s_{i,j}$	$p_i(j)$, i.e., $p_i(\cdot)$ evaluated at j
$v_{i,j}$	Commitment of $s_{i,j}$ computed as $g^{s_{i,j}}$
T_i	Intermediate key set proposed by node i in RBC
T'_i	Indices of ACSS that terminates at node i

A. Validated Reliable Broadcast

Definition 1 (Reliable Broadcast [11])**.** A protocol for a set of nodes $\{1, ..., n\}$, where a distinguished node called the broadcaster holds an initial input M, is a reliable broadcast (RBC) protocol, if the following properties hold

- *Agreement:* If an honest node outputs a message M' and another honest node outputs M'', then $M' = M''$.
- *Validity:* If the broadcaster is honest, all honest nodes eventually output the message M.
- *Totality:* If an honest node outputs a message, then every honest node eventually outputs a message.

We will use the recent *validated* RBC protocol of [19]. For a message M, its communication cost is $O(n|M|+\kappa n^2)$ where $|M|$ is the size of M and κ is the output size of a collision-resistant hash function.

B. Asynchronous Complete Secret Sharing

Definition 2 (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 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 an adversary controlling up to t nodes:

- *Termination:*
  1) If the dealer L is honest, then each honest node will eventually terminate the Sh protocol.
  2) If an honest node terminates the Sh protocol, then every honest node will eventually terminate Sh.
  3) If all honest nodes start Rec, then each honest node will eventually terminate Rec.
- *Correctness:*
  1) If L is honest, then each honest node upon terminating Rec, outputs the shared secret s.

2) If some honest node terminates Sh, then there exists a fixed secret $s' \in \mathbb{Z}_q$, such that each honest node upon completing Rec, will output s'.

- *Secrecy:* If L is honest and no honest node has begun executing Rec, then an adversary that corrupts up to t nodes has no information about s.
- *Completeness:* If some honest node terminates Sh, then there exists a degree t polynomial $p(\cdot)$ over $\mathbb{Z}_q$ such that $p(0) = s'$ and each honest node i will eventually hold a share $s_i = p(i)$. Moreover, when L is honest $s' = s$.

We need to slightly relax the above standard secrecy notion: for a uniformly random $s \in \mathbb{Z}_q$, we allow the ACSS to reveal g^s for a random generator $g \in \mathbb{G}$. We prove in §V that revealing g^s does not affect the Secrecy property of our ADKG protocol.

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 (as defined in Completeness) for all i. Furthermore, these commitments are additively homomorphic across different ACSS instances.

We require the Homomorphic-Partial-Commitment property for two reasons: first, we need nodes to output the threshold public key h^{z_j} of each node; second, we need to aggregate commitments of distinct ACSS instances.

We observe that if an ACSS protocol outputs a Feldman commitment of the underlying polynomial, then it guarantees Homomorphic-Partial-Commitment. We briefly describe the Feldman polynomial commitment next.

Feldman polynomial commitment. The commitment to a random degree-d polynomial

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_dx^d$$

for uniformly random coefficients $a_k \in \mathbb{Z}_q$ for each $k \in [0, d]$, is a vector $\boldsymbol{v}$ computed as:

$$\boldsymbol{v} = [g^{a_0}, g^{a_1}, g^{a_2}, \ldots, g^{a_d}]$$

It is easy to see that given the Feldman commitment $\boldsymbol{v}$ of a polynomial $p(\cdot)$, we can compute $g^{p(i)}$, the commitment of $p(i)$ by interpolating $p(i)$ in the exponent. Also, given polynomial $p(\cdot)$ and $p'(\cdot)$, the commitments $g^{p(i)}$ and $g^{p'(i)}$ are additively homomorphic. Note that this commitment is not completely hiding as it leaks g^{a_k} for each $k \in [0, d]$. We show in §V that revealing g^{a_k} does not violate the secrecy property of our ADKG. Also, the size of the commitment is linear in d. Given a commitment $\boldsymbol{v}$ and a share s_i, a node checks whether $s_i = p(i)$ by checking whether

$$g^{s_i} = \prod_{k=1}^{d} v_k^{i^k} \tag{1}$$

In our paper, we use two different ACSS protocols; the ACSS scheme from Das et al. [19] which improves upon Yurek et al. [58], and our new ACSS scheme in §VII. We will incorporate a Feldman commitment into each of them to ensure the Homomorphic-Partial-Commitment property. For the ACSS scheme of [19], we can simply use Feldman commitment instead of Pedersen's commitment for the underlying polynomial. Our high-threshold ACSS in §VII outputs the Feldman commitment by construction.

C. Asynchronous Binary Agreement

Definition 3 (Asynchronous Binary Agreement)**.** A protocol for a set of nodes $\{1, ..., n\}$ each holding an initial binary input $b \in \{0, 1\}$, is an Asynchronous Binary Agreement (ABA) protocol, if the following properties hold under asynchrony

- *Agreement:* No two honest nodes output different values.
- *Validity:* If all honest nodes have the same input value, no honest node outputs a different value.
- *Termination:* Every honest node eventually outputs a value.

All ABA protocols rely on randomization to circumvent the FLP impossibility [25]. The most efficient approach is to use shared randomness, provided by a common coin protocol [12]. Our ADKG protocol requires an ABA protocol with the following additional property.

- *Good-Case-Coin-Free:* If all honest nodes input the same value to the ABA, then all honest nodes output without invoking the common coin.

The ABA protocol of Crain [17] has the Good-Case-Coin-Free property. It uses $O(n^2)$ expected communication and expected $O(1)$ rounds. For completeness, we provide the pseudocode of Crain's ABA and explain why it satisfies the Good-Case-Coin-Free property in Appendix B (Figure 5).

IV. Design

Our ADKG protocol has four phases: *Sharing, Key Set Proposal, Agreement* and *Key Derivation.* The first three phases have a similar structure where we run n concurrent instances of ACSS, RBC, and ABA, respectively, where each node initiates one instance of ACSS and RBC, and each ABA instance agrees on whether or not the corresponding RBC terminates. We refer to the ACSS (RBC or ABA) invoked by or associated with node i as the i^{th} ACSS (RBC or ABA). We give the pseudocode of our ADKG protocol in Algorithm 1 and describe each phase next.

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 ACSS protocol. In this section, we will focus on the case of $\ell = t + 1$ (refer to §II-B).

Algorithm 1 ADKG for node i

```
OUTPUT: $z_i, h^z, \{h^{z_j}\}$ for each $j \in [n]$
PUBLIC PARAMETER: $g, h, \{pk_j\}$ for each $j \in [n]$

SHARING PHASE:
1: Sample a random secret $s \leftarrow \mathbb{Z}_q$
2: ACSS($s$)

KEY SET PROPOSAL PHASE:
11: Let $S_i = \{\}; T'_i = \{\}$
12: upon termination of $j^{\text{th}}$ ACSS do
13:     Let $s_{j,i}$ be the share of node $i$ for $j^{\text{th}}$ ACSS.
14:     $S_i := S_i \cup \{s_{j,i}\}$
15:     $T'_i := T'_i \cup \{j\}$
16:     if $|T'_i| = t+1$ then
17:         $T_i := T'_i$
18:         RBC($T_i$)
19: Participate in $j^{\text{th}}$ RBC when $T_j \subseteq T'_i$

AGREEMENT PHASE:
21: upon termination of $j^{\text{th}}$ RBC do
22:     Let $T_j$ be the RBC output
23:     upon $T'_i \supseteq T_j$ do
24:         Input 1 to $j^{\text{th}}$ ABA      ▷ if it has not input any value
25:         $u_{i,j} := \sum_{k \in T_j} s_{k,i}$  ▷ share of key for $j^{\text{th}}$ ABA coin.
26:         Use $u_{i,j}$ for coin in $j^{\text{th}}$ ABA
27: $T := \{\}$
28: upon termination of $j^{\text{th}}$ ABA do
29:     if $j^{\text{th}}$ ABA outputs 1 then
30:         $T := T \cup T_j$
31:         Input 0 to all remaining ABAs

KEY DERIVATION PHASE:
41: Wait until all ABAs terminate
42: $z_i := \sum_{k \in T} s_{k,i}$     ▷ share of final secret key $z = \sum_{k \in T} s_k$
43: Let $\pi_i$ be the NIZK proof of $\log_g g^{z_i} = \log_h h^{z_i}$
44: send $\langle \texttt{KEY}, h^{z_i}, \pi_i \rangle$ to all
45: $H = \{\}$
46: upon receiving $\langle \texttt{KEY}, h^{z'_j}, \pi_j \rangle$ from node $j$ do
47:     Derive $g^{z_j}$ using the ACSS commitments
48:     if $\pi_j$ is a valid NIZK proof of $\log_g g^{z_j} = \log_h h^{z'_j}$ then
49:         $H \leftarrow H \cup \{(j, h^{z'_j})\}$
50:         if $|H| \geq \ell$ then
51:             Interpolate $h^z$ and any missing $h^{z_j}$ for each $j \in [n]$
52:             output $z_i$, $h^z$, and $h^{z_j}$ for each $j \in [n]$
```

A. Sharing Phase

During the sharing phase, each node i samples a uniformly random secret $s_i \in \mathbb{Z}_q$ and secret-shares it with all other nodes using an ACSS scheme (lines 1-2 in Algorithm 1). For $\ell = t+1$, we use the ACSS scheme from [19] but replace its Pedersen polynomial commitment with a Feldman polynomial commitment to achieve Homomorphic-Partial-Commitment (refer to §III-B). This simply requires using a zero-polynomial as the hiding polynomial in Pedersens' polynomial commitment.

Let $p_i(\cdot)$ be the degree-t polynomial where

$$p_i(x) = s_i + a_{i,1}x + a_{i,2}x^2 + \ldots + a_{i,t}x^t \qquad (2)$$

where $a_{i,k} \leftarrow \mathbb{Z}_q$ are chosen at random. Due to the completeness property of ACSS, once the i^{th} ACSS instance terminates, all honest nodes output an evaluation point on $p_i(\cdot)$. Each node additionally outputs $\boldsymbol{v}_i$, the Feldman commitment of $p_i(\cdot)$

$$\boldsymbol{v}_i = [g^{s_i}, g^{a_{i,1}}, g^{a_{i,2}}, \ldots, g^{a_{i,t}}] \qquad (3)$$

B. Key Set Proposal Phase

During the key set proposal phase, each node i maintains a set T'_i of terminated ACSS instances (lines 11-15). In particular, whenever the j^{th} ACSS terminates at node i, node i adds the index j to the set T'_i (line 15), and add its share $s_{j,i} = p_j(i)$ (of the secret chosen by node j) to the set S_i (line 14). Let T_i be the first $t+1$ ACSS instances that terminate at node i. Node i starts a RBC to broadcast T_i to all other nodes (lines 16-18). Intuitively, T_i is the proposal from node i for the set of nodes whose secrets are to be aggregated for the final secret key z.

One crucial point to note here is that each node i participates in the j^{th} RBC only after $T_j \subseteq T'_i$ (line 19), i.e., when all the ACSS instances specified in T_j have terminated at node i. Hence, if RBC(T_j) terminates, then at least $t+1$ honest nodes vouch that every ACSS instance in T_j has terminated. Thus, due to the Completeness property of ACSS, for every $k \in T_j$, the k^{th} ACSS instance will eventually terminate at all honest nodes, and every honest node will eventually receive a valid share of the secret s_k.

C. Agreement Phase

During the agreement phase, nodes try to agree on a subset of valid key set proposals. We then use the union of elements in these key set proposals to derive the final secret key. As described before, the key set proposal T_i by node i is valid if every ACSS instance in T_i has terminated. To agree on a subset of key set proposals, nodes start n concurrent ABA instances, where the j^{th} ABA instance seeks to decide whether or not the j^{th} key set proposal is valid. Node i inputs 1 to the j^{th} ABA instance if the j^{th} key set proposal RBC successfully terminated at node i (lines 21-24). Moreover, if any shared randomness is required for the j^{th} ABA instance, nodes use T_j to generate the shared randomness. More specifically, let u_j be the following value

$$u_j = \sum_{k \in T_j} s_k \qquad (4)$$

Then, during the j^{th} ABA instance, whenever a common coin is needed, nodes use the Diffie-Hellman threshold coin-tossing protocol due to [12] with shared secret key u_j (lines 25-26). Node i's share of u_j is $u_{i,j} = \sum_{k \in T_j} s_{k,i}$. We can then use the Homomorphic-Partial-Commitment property of our ACSS to allow each node to locally compute g^{u_j} and $g^{u_{i,j}}$ for all i and j to finish the coin tossing setup.

As mentioned in challenge 2 of §II, if the j^{th} node is malicious and does not reliably broadcast a key set, nodes will not have access to shared randomness in the j^{th} ABA. Fortunately, in this case, all honest nodes will input 0 to that ABA, and that ABA will terminate without using a common coin due to the Good-Case-Coin-Free property. For the bivalent case where honest nodes input different values to an ABA, at least one honest node inputs 1 to the ABA. It implies that at least one honest node has received the intermediate key set. By the Completeness of ACSS and Totality of RBC, all honest nodes will eventually receive the intermediate key set as well.

Finally, to ensure that not all ABAs terminate with 0, we will use the elegant idea from Ben-Or et al. [8]. Briefly, each honest node inputs 1 to all ABAs whose key set proposal phase terminates successfully and refrains from inputting 0 to any ABA until at least one ABA terminates with 1 (lines 29-31). Once an ABA terminates with 1, every node inputs 0 to all the remaining ABAs for which it has not input anything yet. Using an analysis similar to Ben-Or et al. [8], we show in Lemma 1 that at least one ABA will terminate with 1.

D. Key Derivation Phase

If the j^{th} ABA terminates with 1, we say the j^{th} key set proposal is *accepted*. Let T be the set of nodes that are included in at least one accepted key set proposal (lines 29-30). Note that $|T| \geq t+1$. We then use T to derive the final ADKG secret key z as

$$z = \sum_{j \in T} s_j \tag{5}$$

To compute the final ADKG public key h^z, each node i locally computes h^{z_i} where $z_i = \sum_{j \in T} s_{j,i}$ is its share of the secret key z. Each node i also computes a NIZK proof π_i that $\log_g g^{z_i} = \log_h h^{z_i}$ (line 43). Here $\log_g$ and $\log_h$ denotes the discrete logarithm with base g and h, respectively. We use the non-interactive variant of Chaum-Pedersen's protocol [16] for this purpose. Note that, for any given i, due to Homomorphic-Partial-Commitment of our ACSS, all honest nodes can compute g^{z_i} using the partial commitments of each ACSS instance included in T.

Finally, each node i sends $\langle \texttt{KEY}, h^{z_i}, \pi_i \rangle$ to all other nodes. Upon receiving ℓ valid $\langle \texttt{KEY}, h^{z_i}, \pi_i \rangle$ messages, a node can compute the public key h^z and the threshold public keys h^{z_j} for each $j \in [n]$ using Lagrange interpolation (line 49-51).

V. Analysis

A. Correctness

To argue Correctness, we will first argue that our ADKG protocol terminates at all honest nodes, and that upon termination, all honest nodes agree on the set of nodes whose inputs are included in the final secret key.

Lemma 1. *Algorithm 1 terminates at all honest nodes, and all honest nodes output the same set of nodes whose inputs are to be included in the final secret key.*

Proof: Since $n - t \geq 2t+1$, it is easy to see that every ACSS and RBC initiated by an honest node will terminate at all nodes. Now we show that all ABA instances terminate and at least one ABA instance terminates with 1.

We first argue that at least one ABA instance eventually terminates with 1, and every honest node eventually inputs to every ABA. The key set proposal RBC of all honest nodes will eventually terminate at all honest nodes. Hence, all honest nodes will input 1 to the corresponding ABA instances unless they have already input 0 to some ABA. Consider the first ABA instance to which all honest nodes input a value. If all honest nodes input 1, then this ABA terminates with 1 due to the Validity of ABA. Otherwise, if some honest node inputs 0 to this ABA, according to Algorithm 1, some other ABA has already terminated with 1. Hence, at least one ABA instance in terminates with 1. Then, all honest nodes input 0 to the rest of the ABA instances if they have not input any value already.

Now we show all ABA instances terminate. If all honest nodes input 0 to an ABA, then this ABA terminates due to the Good-Case-Coin-Free property of Crain's ABA [17]. Otherwise, if at least one honest node inputs 1, then due to Totality of RBC and Completeness of ACSS, eventually every honest node will have the key set to generate shared randomness. Hence, the ABA will eventually terminate. ■

Lemma 2 (Correctness C1 and C2)**.** *All subsets of $t+1$ shares of honest nodes define the unique secret key z. Furthermore, all honest nodes output the same public key $y = h^z$.*

Proof: Lemma 1 implies that all honest nodes agree on and output T. The Completeness of ACSS ensures that every $k \in T$ corresponds to a polynomial $p_k(\cdot)$ of degree at most t. Define $\hat{p}(\cdot)$ to be $\sum_{k \in T} p_k(x)$. Then, $z = \hat{p}(0)$, and for node j, its share of z is $z_j = \hat{p}(j)$. This implies every set of (at least) $t+1$ valid shares $\{j, z_j\}$ interpolates to the unique polynomial $\hat{p}(\cdot)$.

Next, we show that honest nodes can tell apart correct shares from incorrect ones. First, for each share z_j, the value g^{z_j} can be computed from output of the sharing phase:

$$g^{z_j} = g^{\sum_{i \in T} s_{i,j}} = \prod_{i \in T} g^{s_{i,j}} = \prod_{i \in T} \prod_{k=0}^{t+1} v_{i,k}^{j^k} \tag{6}$$

where $v_{i,k}$ is the k^{th} element of $\boldsymbol{v}_i$ and the last equality follows from the Homomorphic-Partial-Commitment property of the ACSS scheme. With g^{z_j}, when a node receives $\langle \texttt{KEY}, h^{z'_j}, \pi_j \rangle$ from node j, it can check whether $z_j = z'_j$ due to the Soundness property of the equality of discrete logarithm NIZK proof π_j.

1) For each honest node $i \in H$, $\mathcal{S}$ samples a uniformly secret $s_i \in \mathbb{Z}_q$. Follow the protocol for every honest node till (including) the agreement phase.
2) Let T be the set of ABA instances that terminates with 1. Let $s = \sum_{k \in T} s_k$.
3) Let $z_1, z_2, \dots, z_t$ be the shares of s held by the malicious nodes. $\mathcal{S}$ then extracts $z_1, z_2, \dots, z_t$ as follows.
   - For each honest node in $k \in T$, $\mathcal{S}$ already knows the corresponding shares held by malicious nodes. Let $s_{k,i}$ be the share of adversarial node i.
   - For each malicious node $k \in T$, $\mathcal{S}$ uses $n - t$ shares received during sharing phase to reconstruct the polynomial $p_k(\cdot)$. Evaluate $p_k(j)$ for each $j \in [t]$ to get the share $s_{k,j}$. Then $z_j = \sum_{k \in T} s_{k,j}$
4) Let $\hat{p}(\cdot)$ be a degree-t polynomial such that $\hat{p}(0) = z$ and $\hat{p}(j) = z_j$ for each $j \in [t]$. Compute $h^{\hat{p}(i)}$ for each $i \in [n]$. For each $i = [t+1, n]$ use Lagrange interpolation in the exponent to compute $h^{\hat{p}(i)}$.
5) For each $i \in [t+1, n]$, generate NIZK proof π_i for equality of discrete logarithm of false statements that $\log_g g^{z_i} = \log_h h^{\hat{p}(i)}$. $\mathcal{S}$ uses the simulator of Chaum-Pedersen's protocol as described in Appendix C.
6) For each honest node i, let π_i be the generated NIZK proof. $\mathcal{S}$ sends $\langle \texttt{KEY}, h^{\hat{p}(i)}, \pi_i \rangle$ to all nodes on behalf of node i.

Figure 2: Description of the ADKG secrecy simulator $\mathcal{S}$.

Thus, upon receiving ℓ valid KEY messages, each node interpolates them in the exponent to compute h^z. ■

Lemma 3 (Correctness C3)**.** *Assuming hardness of Decisional Diffie-Hellman, the secret key z is computationally indistinguishable from a uniformly random element in $\mathbb{Z}_q$.*

We will prove Lemma 3 when we prove the Secrecy of our ADKG protocol.

Lemma 4 (Correctness C4)**.** *All honest nodes agree on and output the threshold public keys of all nodes. The threshold public key of node j is $y_j = h^{z_j}$.*

Proof: From Lemma 2, for each j, every node can compute g^{z_j} as in equation (6). Also, during the key derivation phase, each node will eventually receive ℓ valid $\langle \texttt{KEY}, h^{z_j}, \pi_j \rangle$ and can efficiently validate them. Upon receiving ℓ valid h^{z_j} nodes interpolate in the exponent to compute h^{z_k} for the remaining nodes. ■

B. Secrecy

We prove Secrecy using *simulatability*. In particular, we prove that for every probabilistic polynomial-time (PPT) static adversary $\mathcal{A}$ that corrupts up to t nodes, there exists a PPT simulator $\mathcal{S}$ that takes as input a uniformly random element $y \in \mathbb{G}$ and produces a view that is indistinguishable from $\mathcal{A}$'s view of a run of the ADKG protocol that outputs y as its public key. We assume a static adversary who picks the set of nodes to corrupt upfront. Without loss of generality, let $[t]$ be the set of nodes $\mathcal{A}$ corrupts, and let $pk_1, pk_2, \dots, pk_t$ be their public keys.

Lemma 5 (Secrecy S1)**.** *Assuming hardness of Decisional Diffie-Hellman (DDH), a PPT adversary $\mathcal{A}$ that corrupts up to t nodes learns no information about the secret z beyond what is revealed from the public key $y = h^z$.*

We describe the simulator $\mathcal{S}$ in Figure 2 and summarize it below. Upon input h and the ADKG public key y, the simulator first simulates the sharing phase of our ADKG. For each honest node i, $\mathcal{S}$ samples a uniformly random secret s_i and secret shares it among all nodes using ACSS. $\mathcal{S}$ then runs the key set proposal and agreement phase for each honest node as per our protocol.

Once all n ABA instances terminate at any honest node, $\mathcal{S}$ sets T to be the set of nodes chosen for the final secret key, and s to be the accumulated secret:

$$s = \sum_{i \in T} s_k \tag{7}$$

Let $z_1, z_2, \dots, z_t$ be shares of s held by adversarial nodes. Then, $\mathcal{S}$ extracts them as follows. For each honest node $i \in T$, $\mathcal{S}$ already knows shares of s_i held by adversarial nodes. For all adversarial nodes $j \in T$, due to Completeness of ACSS, $\mathcal{S}$ already knows $n-t$ shares of s_j. Thus, $\mathcal{S}$ can reconstruct the polynomial $p_j(\cdot)$ and evaluates it at $1, 2, \dots, t$ to recover the corresponding shares. Once $\mathcal{S}$ recovers all the individual shares for all polynomials corresponding to all $k \in T$, $\mathcal{S}$ sums them up to get $z_1, z_2, \dots, z_t$.

Let $p(\cdot)$ be the aggregated polynomial such that $p(0) = s$ and $p(j) = z_j$ for $j \in [n]$. Let $\hat{p}(\cdot)$ be a degree-t polynomial such that $\hat{p}(0) = z = \log_h y$ and $\hat{p}(j) = p(j)$ for each $j \in [t]$. Note that $\mathcal{S}$ needs to ensure that the ADKG public key is $h^{\hat{p}(0)}$. However, the aggregated secret is $p(0)$. The $\mathcal{S}$ addresses this issues by deviating from the specified protocol in the following manner.

$\mathcal{S}$ first computes $h^{\hat{p}(i)}$ for $i \in [n]$. For each $j \in [t]$, $\mathcal{S}$ uses z_j to compute $h^{\hat{p}(j)}$. Then, for each remaining $j \in [t+1, n]$, $\mathcal{S}$ uses Lagrange interpolation in the exponent to compute $h^{\hat{p}(i)}$ as follows.

$$h^{\hat{p}(i)} = \prod_{j=0}^{t} h^{\gamma_j \cdot \hat{p}(j)} \tag{8}$$

where γ_j's are the appropriate Lagrange coefficients. Then, for each honest node j, i.e., for $j \in [t+1, n]$, $\mathcal{S}$ computes the NIZK proof π_j for equality of the discrete logarithm $\log_g g^{p(j)} = \log_h h^{\hat{p}(j)}$ for each $j \in [t+1, n]$. $\mathcal{S}$ uses the perfect zero-knowledge simulator of the Chaum-Pedersen protocol to generate π_j. Note that $\mathcal{S}$ here generates a proof of *false* statement. We provide more details on how $\mathcal{S}$ generates such a proof in Appendix C.

We next argue that if a PPT adversary $\mathcal{A}$ can distinguish the simulated view generated by $\mathcal{S}$ from its view in a real execution of the protocol, then we can use $\mathcal{A}$ to build

Given a DDH tuple (g, g^a, g^b, o) such that o is either g^{ab} or g^r for some random $r \in \mathbb{Z}_q$, our distinguisher simulates the view of $\mathcal{A}$ as follows. We write g^b as h.

1) For each honest node i, i.e., for each $i \in [t+1, n]$, sample a uniformly random $r_i \in \mathbb{Z}_q$.
2) Implicitly set $a + r_i$ as the ACSS secret of node $i \in [t+1, n]$ and run sharing, key set proposal phase of our ADKG protocol. In particular, for node i, sample uniformly random $a_{i,j} \in \mathbb{Z}_q$ for each $j \in [t]$. Let $p_i(\cdot)$ be a degree-t polynomial such that $p_i(0) = a + r_i$ and also $p_i(j) = a_{i,j}$ for each $j \in [t]$.
3) Compute the Feldman polynomial commitment to $p_i(\cdot)$ as follows.
   - Let $v_{i,j} = g^{a_{i,j}}$ for each $j \in [t]$. Set $v_{i,0} = g^{a+r_i}$.
   - Use Lagrange interpolation to compute $v_{i,j}$ for each $j \in [t+1, n]$, i.e.,
   $$v_{i,j} = \prod_{k=0}^{t} g^{\gamma_k a_{i,k}}$$
   here $\gamma_{j,k} = \prod_{j \neq k} \frac{j-l}{j-k}$ is the k^{th} Lagrange coefficient.
   - For each $j \in [t]$, use $p_i(j)$ as the secret share of node j for the secret $a + r_i$.
   - Run the ACSS step for every honest dealer.
4) For any honest node i, let T_i be the intermediate key set proposed by node i. $\mathcal{D}$ uses the NIZK simulator of equality of discrete logarithm protocol for generating required proofs during the coin-tossing protocol for the i^{th} ABA instance.
5) $\mathcal{D}$ waits till all ABA instances terminate at any honest nodes. Also, let T be chosen at the end of the agreement phase. Let $s = \sum_{k \in T} s_k$.
6) For every adversarial node $j \in T$, $\mathcal{D}$ extracts the secrets s_j as described in Figure 2. Let $Q \subset T$ be the set of malicious nodes in T. Then, let $u = \sum_{k \in Q} s_k$. Also, let $w = |T| - |Q|$, i.e., the number of honest node included in T. Then, the s can be written as:
$$s = u + w \cdot a + \sum_{k \in T \setminus Q} r_k \tag{9}$$
7) $\mathcal{D}$ extracts u as in Figure 2 and uses $o^w h^u h^{\sum_{k \in T \setminus Q} r_k}$ as the final public key. For each honest node, $\mathcal{D}$ uses the NIZK simulator of dleq protocol for generating required proofs during the key derivation phase.

Figure 3: Description of DDH distinguisher $\mathcal{D}$

a distinguisher $\mathcal{D}$, given in Figure 3, to break the DDH assumption.

The distinguisher $\mathcal{D}$ gets an DDH tuple (g, g^a, g^b, o) as input where either $o = g^{ab}$ or $o = g^r$ for a random $r \in \mathbb{Z}_q$. $\mathcal{D}$ then runs our ADKG protocol with the adversary $\mathcal{A}$ where $\mathcal{D}$ emulates the honest nodes. At the end of the ADKG protocol, $\mathcal{A}$ outputs a bit β which is $\mathcal{A}$'s guess of whether the ADKG transcript is identical to the simulated transcript or the transcript of a real execution of the protocol. $\mathcal{D}$ then outputs β as its guess of whether $o = g^{ab}$ or $o = g^r$.

Next we prove the following claims about the transcript of the interaction of $\mathcal{A}$ with $\mathcal{D}$.

Claim 1. *If $o = g^{ab}$, the distribution of transcript generated due to $\mathcal{A}$'s interaction with $\mathcal{D}$ is identical to the distribution generated during a real execution of the protocol.*

Proof: When $o = g^{ab}$ then
$$\begin{aligned} o^w h^u h^{\sum_{k \in T \setminus Q} r_k} &= g^{abw} h^u h^{\sum_{k \in T \setminus Q} r_k} \\ &= h^{aw+u+\sum_{k \in T \setminus Q} r_k} \end{aligned} \tag{10}$$

Since $s = aw + u + \sum_{k \in T \setminus Q} r_k$ and the NIZK simulator of dleq is perfect, the distribution generated by $\mathcal{D}$ is identical to the real-world execution of our ADKG protocol. ■

Claim 2. *If $o = g^r$ for a random $r \in \mathbb{Z}_q$ the distribution of transcript generated due to $\mathcal{A}$'s interaction with $\mathcal{D}$ is identical to the distribution generated by $\mathcal{S}$.*

Proof: When $o = g^r$ then
$$\begin{aligned} o^w h^u h^{\sum_{k \in T \setminus Q} r_k} &= g^{rw} h^u h^{\sum_{k \in T \setminus Q} r_k} \\ &= h^{r'w} h^u h^{\sum_{k \in T \setminus Q} r_k}; \qquad (r' = rb^{-1}) \\ &= h^{r'w+u+\sum_{k \in T \setminus Q} r_k} \end{aligned} \tag{11}$$

Since r' is uniformly random and independent of w, u and r_k's, $s = r'w + u + \sum_{k \in T \setminus Q} r_k$ is uniformly random. Hence, the distribution generated by $\mathcal{D}$ is identical to the distribution generate by $\mathcal{S}$. ■

Proof of Lemma 5: From Claim 1 and 2, if $\mathcal{A}$ distinguishes these two distributions with probability $1/2 + p$, then $\mathcal{D}$ will distinguish the DDH tuple with the same probability. Thus, assuming the hardness of DDH, the view generated by the $\mathcal{S}$ is indistinguishable from the view of the actual protocol. ■

Proof of Lemma 3: It follows from the description of the distinguisher $\mathcal{D}$ and the proof of Lemma 5 that assuming hardness of DDH, the ADKG public key h^z is indistinguishable from a uniformly random element in $\mathbb{G}$. Since h is fixed and independent of z, this implies that z is indistinguishable from a random element in $\mathbb{Z}_q$. ■

C. Performance

Lemma 6. *The expected total communication cost of our ADKG protocol is $O(n^3)$.*

Proof: Let C_{ACSS} be the communication cost of one ACSS instance. Let C_{ABA} be the expected communication cost of one ABA instance. Let $C_{\text{RBC}}(L)$ be the communication cost of an RBC protocol on a message of size L. Then, $C_{\text{ACSS}} = O(\kappa n^2)$ ([58], §VII), $C_{\text{ABA}} = O(\kappa n^2)$ [17], [12], and $C_{\text{RBC}}(L) = O(nL + \kappa n^2)$ [19]. The expected communication cost of our ADKG protocol is $n \cdot (C_{\text{ACSS}} + C_{\text{RBC}}(n) + C_{\text{ABA}} + O(\kappa n^2) = O(\kappa n^3)$. ■

Remark. Note that although the expected latency for n parallel instances of ABA to terminate is $O(\log n)$ rounds [7], the expected communication cost is $n \cdot C_{\text{ABA}} = O(\kappa n^3)$.

Lemma 7. *The expected computation cost per node in our ADKG protocol is $O(\kappa n^3)$, measured in number of elliptic curve exponentiations.*

We prove Lemma 7 in Appendix A.

Lemma 8. *Our ADKG protocol terminates in $O(\log n)$ rounds in expectation.*

Proof: The sharing, key set proposal, and key derivation phase of our ADKG protocol require $O(1)$ rounds [19]. Although a single ABA instance terminates in $O(1)$ rounds in expectation, it takes $O(\log n)$ rounds in expectation for all n parallel instances to terminate [7]. Thus, our ADKG protocol runs in $O(\log n)$ rounds in expectation. ■

Remark. Although, in theory, our ADKG protocol may take $O(\log n)$ rounds in expectation, in the common case in practice, our protocol terminate within much fewer rounds. This is due to the property of Crain's ABA [17] that when all honest nodes input the same value to an ABA instance, that ABA instance will terminate within two iterations, without using any common coin. For the same reason, in the common case in practice, each node only incurs $O(n^2)$ computation cost (as opposed to $O(n^3)$ in the worst-case) because nodes need not compute the intermediate threshold keys for generating common coins.

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

Theorem 1 (ADKG). *In a network of $n \geq 3t+1$ nodes where up to t nodes could be malicious, assuming hardness of Decisional Diffie-Hellman, Algorithm 1 implements an ADKG protocol with expected communication cost of $O(\kappa n^3)$, expected computation cost of $O(n^3)$ per node and expected $O(\log n)$ rounds (κ is the security parameter).*

VI. High-threshold ADKG

So far, we have discussed our ADKG protocol for a threshold of $\ell = t+1$, i.e., the final secret key is secret shared among nodes using a $(n, t+1)$ threshold secret sharing. However, many applications [12], [57], [36] require the secrer key to be shared by a $(n, n-t)$ threshold secret sharing. Here on, we will refer to an ADKG protocol that shares the secret using a threshold of $\ell > t+1$ as a *high-threshold* ADKG. In this section, we describe how to extend our ADKG protocol to support high-threshold.

Design. The only change we need to get a high-threshold ADKG is to use a high-threshold ACSS scheme with the properties specified in §III-B in the sharing phase. The rest of the protocol can proceed exactly as in §IV.

But designing an efficient high-threshold ACSS scheme with our desired properties turns out to be challenging. The prior best known high-threshold ACSS with these properties is due to [43]. However, their ACSS has a communication cost of $O(\kappa n^3)$, which is too costly. The high-threshold ACSS protocols of [6] and [19] have communication costs of $O(\kappa n^2 \log n)$ and $O(\kappa n^2)$, respectively, but do not provide the required Homomorphic-Partial-Commitment property. We design a new high-threshold ACSS that adds the Homomorphic-Partial-Commitment property to [19] while retaining its $O(\kappa n^2)$ communication complexity. We provide more details on our high-threshold ACSS in §VII.

Analysis. Since the only change we introduce is to use a high-threshold ACSS, the correctness of our high-threshold ADKG follows directly from the correctness analysis in §V. For secrecy, we need to ensure that given g^s for a secret s, the high-threshold ACSS scheme is simulatable.

VII. High-threshold ACSS

This section describes a new high-threshold ACSS scheme that adds the Homomorphic-Partial-Commitment property to [19] while retaining its $O(\kappa n^2)$ communication cost.

Briefly, we need *verifiable encryption of discrete logarithms*, i.e., a CPA-secure encryption scheme that allows an encrypter to prove in zero-knowledge about the correct encryption of discrete logarithms of a known value. We use the scheme due to Fouque [26] which assumes the hardness of Decisional Composite Residuosity (DCR) [49].

A. Verifiable Encryption of Discrete Logs

The problem of verifiable encryption of discrete logarithms involves three parties: a prover $\mathcal{P}$, a verifier $\mathcal{V}$, and a receiver $\mathcal{R}$. The receiver $\mathcal{R}$ has a public-private key pair (pk, sk). Let $\mathbb{G}$ be an appropriate group and let $g \in \mathbb{G}$ be a random generator of $\mathbb{G}$. Given (g, x, c, pk), the prover $\mathcal{P}$ wants to convince the verifier $\mathcal{V}$ that c is an public key encryption of α under the public key pk such that $g^\alpha = x$ and $\mathcal{P}$ knows α.

Fouque and Stern's protocol [26] for verifiable encryption of discrete logarithm is a "Σ-protocol" and is zero-knowledge and knowledge sound. The protocol has the following interfaces.

- $\mathsf{KeyGen}(1^\kappa) \rightarrow (pk, sk)$. KeyGen algorithm outputs a public-private key pair for the encryption scheme.
- $\mathsf{Decrypt}(sk, c) \rightarrow \alpha$: Given a ciphertext c and a secret key sk, Decrypt decrypts c using sk and outputs the message.
- $\mathsf{EncAndProve}(pk, g, \alpha) \rightarrow (c, x, \pi)$: EncAndProve function encrypts a uniformly random message α using the public-key pk to get c, computes $x = g^\alpha$, and creates a NIZK proof of knowledge π that the encryptor knows α such that $\alpha = \mathsf{Decrypt}(pk, c)$ and $x = g^\alpha$.
- $\mathsf{VerifyDLog}(pk, g, x, c, \pi) \rightarrow 0/1$. Given (pk, g, x, c, π), the $\mathsf{VerifyDLog}(\cdot)$ outputs 1 if π is a valid proof that there exists α such that $\alpha = \mathsf{Decrypt}(pk, c)$ and $x = g^\alpha$. Note that the proof π needs to be verifiable without access to the secret key or the underlying message α.

B. Design and Analysis

Our high-threshold ACSS is given in Algorithm 2. The main difference between our Algorithm 2 and the high-

Algorithm 2 Homomorphic high-threshold ACSS

```
PUBLIC PARAMETER: $n, t, \ell, g, \{pk_i\}$ for $i = 1, 2, \ldots, n$

SHARING PHASE:
   // As dealer $L$ with a uniform random input $s$:
 1: Sample a $(\ell - 1)$-degree random polynomial $p(\cdot)$ such that $p(0) = s$
 2: Let $v_j, c_j, \pi_j \leftarrow \mathsf{EncAndProve}(pk_j, g, p(j))$ for each $j \in [n]$.
 3: Let $\boldsymbol{v} = \{v_1, v_2, .., v_n\}$, $\boldsymbol{c} = \{c_1, c_2, .., c_n\}$, and $\boldsymbol{\pi} = \{\pi_1, \pi_2, .., \pi_n\}$.
 4: $\mathsf{RBC}(\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi})$ with predicate $P(\cdot)$.
 5: procedure $P(\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi})$        ▷ predicate for node $i$ during RBC
 6:     if $\boldsymbol{v}$ is commitment of a polynomial of degree $\leq \ell - 1$ then
 7:         if $\mathsf{VerifyDLog}(pk_j, g, \boldsymbol{v}[j], \boldsymbol{c}[j], \boldsymbol{\pi}[j])$ is valid $\forall j$ then
 8:             return True
 9:     return False

RECONSTRUCTION PHASE:
   // every node $i$ with key $pk_i, sk_i$
10: $\tilde{s}_i := \mathsf{Decrypt}(sk_i, \boldsymbol{c}[i])$
11: send $\langle \texttt{RECONSTRUCT}, \tilde{s}_i \rangle$ to all
12: upon receiving $\langle \texttt{RECONSTRUCT}, \tilde{s}_j \rangle$ from node $j$ do
13:     if $\boldsymbol{v}[j] = g^{\tilde{s}_j}$ then
14:         $T = T \cup \{\tilde{s}_j\}$
15:         if $|T| \geq \ell$ then
16:             output $s$ using Lagrange interpolation and return
```

threshold ACSS of [19] is the way the dealer encrypts the shares and computes the corresponding NIZK proofs.

During the sharing phase, to share a uniformly random secret $s \in \mathbb{Z}_q$, the dealer L samples a random $(\ell-1)$ degree polynomial $p(\cdot)$ such that $p(0) = s$. Then for each $j \in [n]$, L computes $(v_j, c_j, \pi_j) \leftarrow \mathsf{EncAndProve}(pk_j, g, p(j))$. Let

$$\boldsymbol{v} = \{v_1, \ldots, v_n\}, \boldsymbol{c} = \{c_1, \ldots, c_n\}, \text{ and } \boldsymbol{\pi} = \{\pi_1, \ldots, \pi_n\}.$$

We will refer to $\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}$ as the commitment, encryption and proof vector, respectively. Then, the dealer sends the tuple $(\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi})$ using a validated RBC protocol (e.g., [19]). In a validated RBC protocol, nodes participate only if the predicate $P(\cdot)$ returns true. In our case, this requires (i) $\boldsymbol{v}$ is a commitment to a polynomial of degree at most $\ell-1$; and (ii) for each tuple (v_j, c_j, π_j), $\mathsf{VerifyDLog}(pk_j, g, v_j, c_j, \pi_j) = 1$, i.e., c_j is an encryption of $\log_g v_j$ under pk_j, the public key of node j. For checking the degree of the polynomial commitment $\boldsymbol{v}$, we use the approach from [15].

During the reconstruction phase, each node i decrypts $\boldsymbol{c}[i]$ to recover its share $\tilde{s}_i := \mathsf{Decrypt}(sk_i, c[i])$. Node i then multi-casts $\tilde{s}_i$ to all other nodes. A node j, upon receiving $\tilde{s}_i$ from node i, checks whether $\boldsymbol{v}[i]$ equals $g^{\tilde{s}_i}$. After receiving ℓ or more valid shares, a node reconstructs the secret using Lagrange interpolation.

We now analyze our ACSS scheme in Algorithm 2. Termination of our ACSS follows from the Termination property of RBC and Completeness property of the EncAndProve. Similarly, the Completeness of our ACSS follows from the Soundness of EncAndProve and Totality of RBC. Correctness of our ACSS follows directly from the Soundness of the underlying zero-knowledge protocols. Furthermore, our ACSS has the Homomorphic-Partial-Commitment property because the dealer reliably broadcasts the Feldman commitment of the polynomial. We next provide a proof sketch for Secrecy.

Lemma 9. *For a uniformly random s, given g^s, assuming hardness of Decisional Composite Residuosity and the existence of a Random Oracle, there exists a PPT simulator that can simulate the view of any static PPT adversary.*

Proof Sketch: Let $\mathcal{A}$ be a static PPT adversary that corrupts up to ℓ nodes. Without loss of generality, let $\mathcal{A}$ corrupts the first ℓ nodes. Let pk_i for $i \in [n]$ be the public key of node i. Given g^s for a random secret s, the simulator $\mathcal{S}$ chooses $\ell - 1$ random points $s_i \in \mathbb{Z}_q$ for each $i \in [\ell - 1]$ and sets $v_i = g^{s_i}$. For each $i \in [\ell, n]$, $\mathcal{S}$ constructs v_i using Lagrange interpolation in the exponent. $\mathcal{S}$ then encrypts the share of the each node $j \in [\ell - 1]$. For each node $j \in [\ell, n]$, $\mathcal{S}$ uses encryptions of 0 as the encryption of shares of node j. Note that due to CPA security of the encryption scheme, the encryptions of 0 are indistinguishable from encryption of actual shares of nodes in $[\ell, n]$. Next, $\mathcal{S}$ uses the zero-knowledge simulator of the EncAndProve to construct the proofs π_j for each $j \in [\ell, n]$. It is easy to see that the view of $\mathcal{A}$ in its interaction with the $\mathcal{S}$ is computationally indistinguishable from its view in the actual protocol. ■

Now, let us analyze the communication cost of our high-threshold ACSS. Observe that each $\boldsymbol{v}, \boldsymbol{c}$ and $\boldsymbol{\pi}$ are κn bits long. Hence, using the RBC protocol due to [19], the communication cost of the sharing phase in $O(\kappa n^2)$. During the reconstruction phase, each node multi-casts $O(\kappa)$ bits, so the reconstruction phase has a communication cost of $O(\kappa n^2)$. Hence, the total communication cost is $O(\kappa n^2)$.

VIII. Implementation and Evaluation

A. Implementation Details

We have implemented a prototype of our ADKG protocol for any reconstruction threshold $\ell \in [t + 1, n - t]$ using `python` 3.7.6 on top of the open-source hbACSS library [3].

We use `Rust` libraries for elliptic curve operations and `asyncio` for concurrency, though our prototype only runs on a single processor core. For $\ell = t + 1$, we use the ACSS protocol from [19]. For $\ell \geq t + 1$, we implement the ACSS protocol from §VII. Here on, we refer to the former as the low threshold ACSS and the latter as the high threshold ACSS. We use the python `phe` library with default parameters for DCR operations [20]. We also implement Crain's ABA protocol [17] and Das et al.'s RBC [19].

In our implementation, we use both the `curve25519` and `bls12-381` elliptic curves. We use the Ristretto group over `curve25519` implementation from [2] and the `bls12-381` implementation from Zcash [34] (with a `python` wrapper

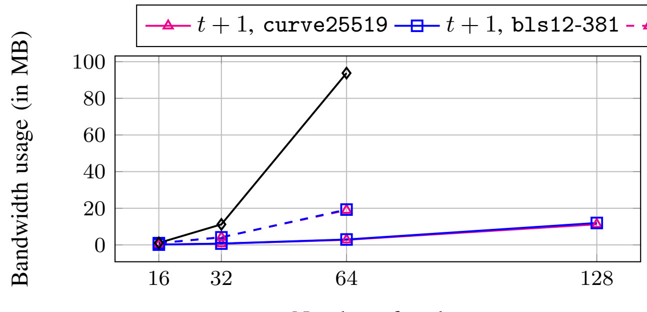

Figure 4: Bandwidth usage, amount of data sent by a node during ADKG protocol.

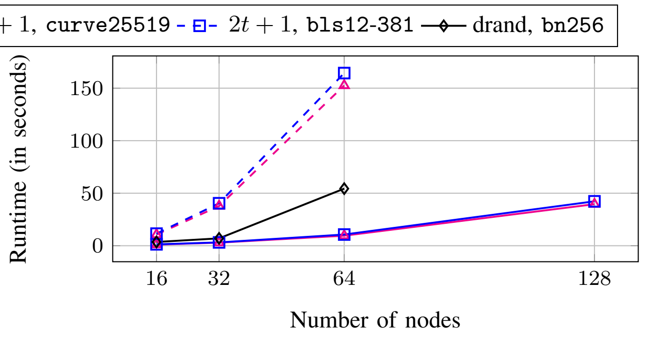

Figure 5: Average runtime as measured the average time difference between the start of the ADKG and the time a node output keys.

around each) for primitive elliptic curve operations. Note that `bls12-381` supports pairing, so our implementation can be used for pairing-based threshold cryptosystems such as [9]. 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 `bls12-381` are 48 and 96 bytes. Furthermore, our micro-benchmark illustrates that a group exponentiation in `bls12-381` is $6\times$ slower than that of `curve25519`.

We improve the bandwidth usage and runtime per node of our ADKG protocol under the common case, i.e., when all nodes are honest and the network has a small delay. We implement the following optimizations: (1) The data dissemination step of the RBC protocol [19, Algorithm 3], which involves error correction and two rounds of communication, can be omitted unless some nodes trigger it. This reduces the bandwidth usage by approximately 50% in the common case; (2) Most of the ABA instances terminate without a coin in the common case; hence, we never explicitly compute the threshold keys for those ABA instances. We observe that this optimization reduces the runtime by about 65% for our ADKG implementation with $\ell = t+1$ in the common case.

B. 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 two reconstruction thresholds: $t+1$ and $2t+1$. 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 different AWS regions: Canada, Ireland, N. California, N. 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. Since no implementation of any ADKG exists, we only compare with the synchronous DKG of Gennaro et al. [31] as implemented in Drand [1]. To our knowledge, Drand is the only DKG protocol in use today. Other implementations of synchronous DKG focus on the cryptography part and do not implement the networking part. We evaluate Drand with its default reconstruction threshold of $n/2+1$. We could only run Drand with up to 64 nodes; Drand with 128 nodes keeps aborting in our experiments.

C. Evaluation Results

With our evaluation we aim to demonstrate that our ADKG protocol scales well with the number of nodes and has reasonable runtime and bandwidth usage.

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 Figure 5.

For $\ell = t+1$, our ADKG protocol takes approximately 10 seconds for 64 nodes, which is only 19% of Drand. When $\ell = 2t+1$, however, our ADKG protocol takes much longer: 160 seconds for 64 nodes, about $3\times$ of Drand.

Upon close inspection, we found that when $\ell = t+1$, various miscellaneous steps account for most of the runtime, whereas when $\ell = 2t+1$, Paillier operations dominate the runtime. This is confirmed in Table III. Specifically, one instance of high threshold ACSS requires $O(n)$ Paillier operations per node, resulting in $O(n^2)$ Paillier operations per node in the sharing phase of our high-threshold ADKG.

Running time of sharing phase. We measure the per node running time of the sharing phase of our ADKG protocol. More specifically, we first measure the running time of the dealer and non-dealer nodes separately. We then calculate the running time of sharing phase as the sum of one dealer node's running time and the running time of n non-dealer node. We report the results in Table III. Observe that when $\ell = t+1$, even with 128 nodes, the running time of the sharing phase is less than 7% of the total running time of our ADKG protocol. However, with $\ell = 2t+1$, for 64 nodes, the computation time of ACSS step is more than 80% of the total running time.

Table III: Computation cost of ACSS phase measured in time taken (in seconds) with varying number of nodes.

		Time taken (in seconds)			
Elliptic Curve	ℓ	$n = 16$	$n = 32$	$n = 64$	$n = 128$
`curve25519`	$t + 1$	0.01	0.02	0.07	0.19
`bls12-381`	$t + 1$	0.07	0.21	0.71	2.52
`curve25519`	$2t + 1$	8.50	32.53	127.54	504.47
`bls12-381`	$2t + 1$	8.96	34.43	134.66	531.22

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 Figure 4. Consistent with the analysis from §V, the bandwidth usage of our protocol increases quadratically with the number of nodes.

Our bandwidth usage is significantly lower than Drand. Using the 64 nodes experiment, for example, each node in Drand sends 93.8 Megabytes of data; In our ADKG, when $\ell = t + 1$, each node only sends 2.96 Megabytes, which is only $1/30^{\text{th}}$ of Drand; when $\ell = 2t + 1$, each node sends 19.2 Megabytes, still only $1/5^{\text{th}}$ of Drand. We note that the higher bandwidth usage for high-threshold again comes from the high-threshold ACSS scheme.

We also note that, although in `bls12-381` 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, such as DCR group elements, field integers, and hashes.

IX. Related Work

Starting from the seminal work of Pedersen [50], numerous works have studied the problem of Distributed Key Generation with various cryptographic assumptions, network conditions and with other properties [13], [14], [26], [31], [47], [37], [35], [53], [39], [40], [43], [4], [28], [19]. 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 [50], [26], [13], [14], [31], [47], [37], [35], [53]. Pedersen proposed the first DKG protocol [50] using a verifiable secret sharing. Gennaro et al. [31] showed that the Pedersen protocol allows an attacker to bias the public-key distribution and proposed a scheme without this issue but at a higher cost. Neji et al. [47] proposed a simple mechanism to mitigate the bias-attack illustrated by [31]. We adopted their idea, but we found the proof presented in their original paper [47] skipped some details of the simulator. In this paper, we present a new proof of secrecy.

Canetti et al. [13] presented extended Gennaro et al. [31] to be secure against an adaptive adversary. Fouque and Stern [26] used publicly verifiable secret sharing (PVSS) instead of VSS to make the protocol non-interactive. Gurkan et al. [37] designed a PVSS-based DKG protocol with a linear size public-verification transcript. However, their protocol can only tolerate $O(\log n)$ faulty nodes. Moreover, in their protocol, the secret key is a group element instead of a field element. As a result, their protocol is incompatible with off-the-shelf threshold signature or encryption schemes. Very recently, Groth [35] designed a new DKG protocol based on a new PVSS scheme; the protocol is non-interactive, assuming the existence of a broadcast channel. Moreover, the secret key in his protocol is a field element.

Asynchronous DKG. Only a handful of works studied the DKG problem in partially synchronous or asynchronous networks [40], [43], [4], [28], [19]. Kate et al. [40] extended Pedersen's DKG to a partially synchronous network. The protocol has $O(\kappa n^4)$ total communication cost, tolerates up to one-third malicious nodes, and relies on synchrony for termination. Tomescu et al. [56] lowered the computational cost of Kate et al. [40] by a factor of $O(n/\log n)$ at a logarithmic increase in communication cost.

Kokoris et al. [43] designed the first asynchronous DKG scheme with a total communication cost of $O(\kappa n^4)$ and an expected round complexity of $O(n)$. Abraham et al. [4] proposed an ADKG protocol with a communication cost of $O(\kappa n^3 \log n)$. Gao et al. [28] and Das et al. [19] gave two methods to lower the communication cost of [4] to $O(\kappa n^3)$. Since Abraham et al. uses the PVSS scheme of Gurkan et al. [37], all three constructions [4], [28], [19] inherit the limitation that the secret key is a group element and the ADKG is not compatible with off-the-shelf threshold encryption or signature schemes.

The setup phase of Aleph's randomness beacon [27] used different sources of coins for different ABA instances. Their work inspired the key set proposal phase of our design. But note that Aleph's setup phase is not a ADKG protocol and our ADKG protocol differs significantly from it.

DKG implementations. The increasing popularity of threshold signatures has led to many DKG implementations [53], [52], [48], [38], [1], [33], [55]. All these implementations assume synchronous networks.

X. Conclusion

In this paper, we presented a simple and concretely efficient asynchronous distributed key generation protocol for discrete logarithm based threshold cryptosystem. In a network of n nodes, our ADKG protocol incurs a communication cost of $O(\kappa n^3)$ and terminates in expected $O(\log n)$ rounds. Our protocol uses many fundamental asynchronous primitives such as ACSS, RBC, threshold common coin, and ABA in a modular way. As a result, an improved protocol for these primitives, especially high-threshold ACSS, would immediately improve our ADKG protocol. We formally prove the security and correctness of our ADKG protocol. We provide a prototype implementation and evaluate our prototype atop up to 128 geographically distributed nodes to illustrate the practicality of our ADKG protocol.

ACKNOWLEDGMENTS

The authors would like to thank Amit Agarwal, Adithya Bhat, Kobi Gurkan, Dakshita Khurana, Nibesh Shrestha, and Gilad Stern for the helpful discussions related to the paper. Also, the authors would like to thank Sylvain Bellemare for helping with the hbACSS codebase and Nicolas Gailly for helping with running the Drand experiments.

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] "hbacss," 2021, https://github.com/tyurek/hbACSS.

[4] I. Abraham, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu, "Reaching consensus for asynchronous distributed key generation," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 363–373.

[5] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous byzantine agreement," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 337–346.

[6] N. Alhaddad, M. Varia, and H. Zhang, "High-threshold avss with optimal communication complexity," in *International Conference on Financial Cryptography and Data Security*. Springer, 2021, pp. 479–498.

[7] M. Ben-Or and R. El-Yaniv, "Resilient-optimal interactive consistency in constant time," *Distributed Computing*, vol. 16, no. 4, pp. 249–262, 2003.

[8] M. Ben-Or, B. Kelmer, and T. Rabin, "Asynchronous secure computations with optimal resilience," in *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, 1994, pp. 183–192.

[9] A. Boldyreva, "Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme," in *International Workshop on Public Key Cryptography*. Springer, 2003, pp. 31–46.

[10] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *International conference on the theory and application of cryptology and information security*. Springer, 2001, pp. 514–532.

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

[12] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[13] R. Canetti, R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Adaptive security for threshold cryptosystems," in *Annual International Cryptology Conference*. Springer, 1999, pp. 98–116.

[14] J. Canny and S. Sorkin, "Practical large-scale distributed key generation," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2004, pp. 138–152.

[15] I. Cascudo and B. David, "Scrape: Scalable randomness attested by public entities," in *International Conference on Applied Cryptography and Network Security*. Springer, 2017, pp. 537–556.

[16] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *Annual International Cryptology Conference*. Springer, 1992, pp. 89–105.

[17] T. 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.

[18] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a dag-based mempool and efficient bft consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 34–50.

[19] S. Das, Z. Xiang, and L. Ren, "Asynchronous data dissemination and its applications," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[20] C. Data61, "Python paillier library," https://github.com/data61/python-paillier, 2013.

[21] Y. G. Desmedt, "Threshold cryptography," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 449–458, 1994.

[22] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2028–2041.

[23] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE transactions on information theory*, vol. 31, no. 4, pp. 469–472, 1985.

[24] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *Conference on the theory and application of cryptographic techniques*. Springer, 1986, pp. 186–194.

[25] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[26] P.-A. Fouque and J. Stern, "One round threshold discrete-log key generation without private channels," in *International Workshop on Public Key Cryptography*. Springer, 2001, pp. 300–316.

[27] A. Gągol, D. Leśniak, D. Straszak, and M. Świętek, "Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 214–228.

[28] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Efficient asynchronous byzantine agreement without private setups," *arXiv preprint arXiv:2106.07831*, 2021.

[29] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *International conference on financial cryptography and data security*. Springer, 2022.

[30] R. Gelashvili, L. Kokoris-Kogias, A. Spiegelman, and Z. Xiang, "Brief announcement: Be prepared when network goes bad: An asynchronous view-change protocol," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 187–190.

[31] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," *Journal of Cryptology*, vol. 20, no. 1, pp. 51–83, 2007.

[32] N. Giridharan, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Bullshark: Dag bft protocols made practical," *arXiv preprint arXiv:2201.05677*, 2022.

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

[34] J. Grigg and S. Bowe, "zkcrypto/pairing," https://github.com/zkcrypto/pairing.

[35] J. Groth, "Non-interactive distributed key generation and key resharing." *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 339, 2021.

[36] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous bft protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 803–818.

[37] K. Gurkan, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu, "Aggregatable distributed key generation," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 147–176.

[38] T. Hanke, M. Movahedi, and D. Williams, "Dfinity technology overview series, consensus system," *arXiv preprint arXiv:1805.04548*, 2018.

[39] A. Kate and I. Goldberg, "Distributed key generation for the internet," in *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 2009, pp. 119–128.

[40] A. Kate, Y. Huang, and I. Goldberg, "Distributed key generation in the wild." *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 377, 2012.

[41] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is dag," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 165–175.

[42] E. Kokoris-Kogias, E. C. Alp, L. Gasser, P. Jovanovic, E. Syta, and B. Ford, "Calypso: private data management for decentralized ledgers," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 586–599, 2020.

[43] E. Kokoris Kogias, D. Malkhi, and A. 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*, 2020, pp. 1751–1767.

[44] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited," in *Proceedings of the 39th Symposium on Principles of Distributed Computing*, 2020, pp. 129–138.

[45] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 31–42.

[46] A. Mostéfaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous binary byzantine consensus with t< n/3, o (n2) messages, and o (1) expected time," *Journal of the ACM (JACM)*, vol. 62, no. 4, pp. 1–21, 2015.

[47] W. Neji, K. Blibech, and N. Ben Rajeb, "Distributed key generation protocol with a new complaint management strategy," *Security and communication networks*, vol. 9, no. 17, pp. 4585–4595, 2016.

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

[49] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.

[50] T. P. Pedersen, "A threshold cryptosystem without a trusted party," in *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, 1991, pp. 522–526.

[51] D. Pointcheval and J. Stern, "Security proofs for signature schemes," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1996, pp. 387–398.

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

[53] P. Schindler, A. Judmayer, N. Stifter, and E. R. Weippl, "Ethdkg: Distributed key generation with ethereum smart contracts." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 985, 2019.

[54] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[55] H. Stamer, "Distributed privacy guard," 2018, https://github.com/gnosis/dkg.

[56] 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)*, 2020, pp. 877–893.

[57] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, “Hotstuff: Bft consensus with linearity and responsiveness,” in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM, 2019, pp. 347–356.

[58] T. Yurek, L. Luo, J. Fairoze, A. Kate, and A. Miller, “hbacss: How to robustly share many secrets,” in *Proceedings of the 29th Annual Network and Distributed System Security Symposium*, 2022.

Appendix A. Proof of Lemma 7

Proof of Lemma 7: Each node incurs the computation cost of one ACSS dealer and $n-1$ ACSS non-dealer node. During the key-set proposal phase, each node incurs the computation cost of one RBC broadcaster and $n-1$ RBC non-broadcaster node. During the agreement phase, in addition to the computation cost of n parallel ABA instances, each node needs to derive n different sets of threshold keys. Finally, during the key-derivation phase, each node verifies $O(n)$ shares and interpolate $O(n)$ threshold keys.

Thus, in our ADKG protocol, each node incurs $O(n^2)$ elliptic curve exponentiations except for the key set proposal phase and the key derivation step of the agreement phase [19], [12], [17]. During the key set proposal 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. Furthermore, each node incurs $O(n^3)$ elliptic curve operations to derive the intermediate threshold keys for the agreement phase. ■

Appendix B. ABA of [17]

In this section we briefly describe the ABA protocol of Crain ([17], Figure 3). The ABA protocol of Crain [17] improves the round complexity of Mostefaoui et al. [46], from 13 rounds per epoch to 8 rounds per epoch, assuming the common-coin uses a single round. If no honest node has terminated so far, the probability of termination in an epoch is $1/2$ in both protocols, assuming the common-coin is unbiased. Each message sent by a node in [17] contains a single bit and a tag. Crain’s ABA incurs a expected communication cost of $O(n^2)$ and terminates in $O(1)$ rounds in expectation.

As we describe earlier, in addition to the standard ABA properties, we crucially rely on the following property of Crain’s ABA.

Good-Case-Coin-Free: If all honest nodes input the same value to the ABA, then all honest nodes output without invoking the common coin.

Algorithm 3 `BV_Broadcast`(v)
```
1: $bin\_values \leftarrow \emptyset$
2: send $\texttt{BVAL}(v)$ to all
3: return $bin\_values$    ▷ $bin\_values$ has not necessarily reached its final value when returned
4: upon receiving $\texttt{BVAL}(v)$ do
5:     if $\texttt{BVAL}(v)$ received from $t+1$ different nodes then
6:         send $\texttt{BVAL}(v)$ to all (if haven't done already)
7:     if $\texttt{BVAL}(v)$ received from $2t+1$ different nodes then
8:         $bin\_values \leftarrow bin\_values \cup \{v\}$
```

Algorithm 4 `SBV_Broadcast`(v)
```
1: $bin\_values \leftarrow \texttt{BV\_Broadcast}(v)$
2: wait until $bin\_values \neq \emptyset$
3: send $\texttt{AUX}(w)$ for $w \in bin\_values$ to all    ▷ $bin\_values$ has not necessarily reached its final value when returned
4: wait until $\exists$ a set $view$ such that (i) $view \subseteq bin\_values$; and
        (ii) contained in $\texttt{AUX}(\cdot)$ messages received from $n-t$ nodes;
5: return $(view, bin\_values)$
```

We restate Crain’s ABA in Algorithm 5. Here we briefly explain why it has the Good-Case-Coin-Free property. When all honest nodes have the same input value v, it is not hard to verify that for the first round $r=1$, $view[1,0], view[1,1]$ and $view[1,2]$ will all equal to $\{v\}$. Hence, every honest node will decide v in line 12 without invoking the common coin. We also need to argue that every honest node knows that no other honest nodes need the common coin, so that it does not need to participate in the common coin to help other honest nodes. This is guaranteed by `SBV_Broadcast`, which ensures that if an honest node has $\{v\}$ as the output of `SBV_Broadcast`, then no honest node will have $\{v'\}$ where $v \neq v'$ as the output. Therefore, if the condition on line 11 gets satisfied at any honest node, then no honest node will enter the conditional branch on line 14, i.e., no honest node will need the common coin.

Appendix C. Zero knowledge Proof of Equality of Discrete Logarithm

Our ADKG protocol has a step that requires nodes to produce zero-knowledge proofs about the equality of discrete logarithms for a tuple of publicly known values. In particular, given a group $\mathbb{G}$ of prime order q, two uniformly random generators $g, h \leftarrow \mathbb{G}$ and a tuple (g, x, h, y), a prover $\mathcal{P}$ wants to prove to a probabilistic polynomial time verifier $\mathcal{V}$, in zero-knowledge, the knowledge of a witness α such that $x = g^\alpha$ and $y = h^\alpha$.

We will use the Chaum-Pedersen "Σ-protocols" [16], which assumes the hardness of the Discrete Logarithm problem.

Protocol for equality of discrete logarithm. For any given tuple (g, x, h, y), the Chaum-Pedersen protocol proceeds as

Algorithm 5 ABA protocol of [17] Figure 3 (Restated)

```
1: **input:** $v$
2: $est \leftarrow v; r \leftarrow 0$
3: **while** true **do**
4:     $r \leftarrow r + 1$
5:     $(view[r, 0], bin\_values[r]) \leftarrow$ SBV_Broadcast$(est)$
6:     **send** AUXSET$[r](view[r, 0])$ to all
7:     **wait** until $\exists$ a set $view[r, 1]$ such that
         (i) $view[r, 1] \subseteq bin\_values$; and
         (ii) contained in AUXSET$(\cdot)$ messages received from $n - t$ nodes;
8:     **if** $view[r, 1] = \{w\}$ **then** $est \leftarrow w$
9:     **else** $est \leftarrow \perp$
10:    $view[r, 2] \leftarrow$ SBV_Broadcast$(est)$
11:    **if** $view[r, 2] = \{v\}$, $v \neq \perp$ **then** decide$(v)$; $est \leftarrow v$
12:    **else** Coin$() = sign(pk, \#ABA\|\#round)$
13:    **if** $view[r, 2] = \{v, \perp\}$ **then** $est \leftarrow v$
14:    **if** $view[r, 2] = \{\perp\}$ **then** $est \leftarrow$ Coin$()$
```

follows.

1) $\mathcal{P}$ samples a random element $\beta \leftarrow \mathbb{Z}_q$ and sends (a_1, a_2) to $\mathcal{V}$ where $a_1 = g^\beta$ and $a_2 = g^\beta$.
2) $\mathcal{V}$ sends a challenge $e \leftarrow \mathbb{Z}_q$.
3) $\mathcal{P}$ sends a response $z = \beta - \alpha e$ to $\mathcal{V}$.
4) $\mathcal{V}$ checks whether $a_1 = g^z x^e$ and $a_2 = h^z y^e$ and accepts if and only if both the equality holds.

This protocol guarantees completeness, knowledge soundness, and zero-knowledge. The knowledge soundness implies that if $\mathcal{P}$ convinces the $\mathcal{V}$ with non-negligible probability, there exists an efficient (polynomial time) extractor that can extract α from the prover with non-negligible probability.

This above protocol can be made non-interactive in the Random Oracle model using the Fiat-Shamir heuristic [24], [51]. In our ADKG, we use the non-interactive variant of the protocol. For any given tuple (g, x, h, y) where $x = g^s$ and $y = h^s$, dleq.Prove(s, g, x, h, y) generates the non-interactive zero proof π. The proof π is $O(\kappa)$ bits long. Given a proof π and (g, x, h, y), dleq.Verify(π, g, x, h, y) verifies the proof.

Simulating a proof without the secret. We will use programmability of random oracle to generate an convincing NIZK proof without having access to the corresponding secret. Furthermore, we use the same approach to generate NIZK proof for false statements. Given the tuple (g, g^x, h, h^y), the simulator works as follows.

1) Sample uniformly random $z, c \in \mathbb{Z}_q$.
2) Compute $a_1 = g^z g^{x \cdot c}$ and $a_2 = h^z h^{y \cdot c}$.
3) Set $c = \mathsf{hash}(a_1, a_2)$.
4) Output $\pi = (a_1, a_2, z)$

Note that the distribution of proof generated by the simulator is identical to the distribution of proof of a correct statement generated by an honest prover during a real execution of the dleq protocol.