Practical Asynchronous Distributed Key Generation: Improved Efficiency, Weaker Assumption, and Standard Model

Haibin Zhang
Beijing Insitute of Technology
Email: haibin@bit.edu.cn

Sisi Duan
Tsinghua University
Email: duansisi@tsinghua.edu.cn

Chao Liu
Quanzhou Normal University
Email: cliu717@163.com

Boxin Zhao
Zhongguancun Laboratory
Email: zhaobx@mail.zgclab.edu.cn

Xuanji Meng
Tsinghua University
Email: mxj21@mails.tsinghua.edu.cn

Shengli Liu
Shanghai Jiao Tong University
Email: slliu@sjtu.edu.cn

Yong Yu
Shaanxi Normal University
Email: yuyong@snnu.edu.cn

Fangguo Zhang
Sun Yat-sen University
Email: isszhfg@mail.sysu.edu.cn

Liehuang Zhu
Beijing Institute of Technology
Email: liehuangz@bit.edu.cn

***Abstract*—Distributed key generation (DKG) allows bootstrapping threshold cryptosystems without relying on a trusted party, nowadays enabling fully decentralized applications in blockchains and multiparty computation (MPC). While we have recently seen new advancements for asynchronous DKG (ADKG) protocols, their performance remains the bottleneck for many applications, with only one protocol being implemented (DYX+ ADKG, IEEE S&P 2022). DYX+ ADKG relies on the Decisional Composite Residuosity assumption (expensive to instantiate) and the Decisional Diffie-Hellman assumption, incurring a high latency (more than 100s with a failure threshold of 16). Moreover, the security of DYX+ ADKG is based on the random oracle model (ROM) which takes hash function as an ideal function; assuming the existence of random oracle is a strong assumption and up to now we cannot find any theoretically-sound implementation. Furthermore, the ADKG protocol needs public key infrastructure (PKI) to support the trustworthiness of public keys. The strong models (ROM and PKI) further limit the applicability of DYX+ ADKG, as they would add extra and strong assumptions to underlying threshold cryptosystems. For instance, if the original threshold cryptosystem works in the standard model, then the system using DYX+ ADKG would need to use ROM and PKI.**

In this paper, we design and implement a modular ADKG protocol that offers improved efficiency and stronger security guarantees. We explore a novel and much more direct reduction from ADKG to the underlying blocks, reducing both the computational overhead and communication rounds of ADKG in the normal case. Our protocol works for both the low-threshold and high-threshold scenarios, being secure under the standard assumption (the well-established discrete logarithm assumption only) in the standard model (no trusted setup, ROM, or PKI).

I. Introduction

Distributed key generation (DKG) allows a group of servers to jointly generate a public and private key pair such that each server obtains a system public key and a share of the secret key corresponding to the public key. The DKG mechanism can be used to avoid trusted setup in various threshold cryptosystems, such as threshold encryption [46], threshold signatures [8], [45], [10], threshold common coins [15], all of which are essential building blocks in modern blockchains, cryptocurrencies, and Byzantine fault-tolerant (BFT) protocols [41], [28], [34], [32], [48], [14].

While DKG has been extensively studied in synchronous environments, we still lack efficient asynchronous DKG (ADKG) protocols—which are increasingly needed in both partially synchronous and asynchronous BFT protocols such as SBFT [32], HotStuff [48], HoneyBadger [41], Dumbo [34], Narwhal&Tusk [24], and PACE [50]. Only until very recently, we have the first implementation of ADKG [27] (which we call DYX+ ADKG). The DYX+ ADKG protocol supports both low (regular) threshold ($t+1$ out of $n = 3t+1$) and high threshold (at least $2t + 1$ out of $n = 3t + 1$). Being able to support the high threshold is of particular importance, as popular BFT protocols such as SBFT [32] and HotStuff [48] and various decentralized crypto wallets require high thresholds. The DYX+ ADKG protocol, however, incurs a high latency (more than 100s for $t = 16$ for high threshold). Moreover, the protocol is based on strong assumptions and models: 1) it assumes both the hardness of Decisional Diffie-Hellman (DDH) problem and Decisional Composite Residuosity (DCR) problem [43] (more expensive to instantiate than conventional elliptic curve hardness problems), 2) relies on the random oracle model (ROM) by treating hash functions as ideal random oracles [5] (no secure instantiations in practice), and 3) needs to use PKI (public key infrastructure) to support the trustworthiness of public keys. Therefore, the resulting threshold cryptosystems using the DYX+ ADKG protocol would need all these assumptions. Consider the well-known threshold encryption scheme of Boneh, Boyen, and Halevi [9] under the Bilinear Diffie-Hellman (BDH) assumption in the standard model. If using DYX+ ADKG, then its security would additionally rely on the DCR assumption, the DDH assumption, ROM, and PKI—hindering the application and adoption of DYX+ ADKG.

A. Our Contributions

We design and implement a simple and efficient ADKG protocol that supports both the low threshold and the high threshold. The runtime of DYX+ ADKG is about 2x-4.6x that of our protocol. Moreover, our protocol relies on the classic discrete logarithm (DL) assumption only (weaker than the DDH assumption and cheaper to instantiate than the DCR assumption), and does not assume ROM or PKI, thereby improving the security guarantees of ADKG.

At the core of our protocol are 1) a new and direct reduction from ADKG to the underlying building block—a variant of asynchronous complete secret sharing (ACSS) protocols [19], and 2) a latency-optimized and computationally efficient ACSS construction supporting both the low threshold and the high threshold in the standard model (without using PKI or ROM).

First, we show a more efficient and direct reduction from (high-threshold) ADKG to (high-threshold) ACSS. Our reduction deviates from existing constructions, building on top of the PACE [50] and WaterBear [51] BFT framework. In particular, we first run n parallel regular ACSS or high-threshold ACSS (HACSS) instances to distribute the secret shares, and then run n parallel reproposable asynchronous binary Byzantine agreement (RABA) instances [50] based on local coins to agree on which (H)ACSS secrets are included in the agreed public/secret pairs. RABA protocols offer a fast path such that in the normal case our ADKG has fast termination.

Second, we build an efficient ACSS protocol with the homomorphic partial commitment property [27], where servers that terminate will output a commitment (e.g., g^s) of the secret shared (e.g., s) and the commitments are additively homomorphic. Our protocol works for both the low threshold and the high threshold. Our protocol relies on the standard discrete logarithm (DL) assumption only and incurs $O(\lambda n^3)$ communication (where λ is a security parameter), outperforming protocols of the same kind.

We summarize our contributions in the following:

- We propose a novel reduction approach to building an efficient ADKG protocol that is not only more efficient than DYX+ ADKG protocol but also relies on the standard DL assumption in the standard model (no ROM and no PKI). Besides, our ADKG protocol supports both the low and high thresholds and supports the field element as the secret key, so it is compatible with state-of-the-art threshold cryptosystems and distributed applications.
- We build the first high-threshold asynchronous complete secret sharing (HACSS) protocol that simultaneously satisfies 1) the homomorphic partial commitment property, 2) optimal resilience, 3) $O(\lambda n^3)$ communication, 4) security using the standard assumption (the DL assumption), and 5) security in the standard model (no ROM or PKI).
- We offer a formal definition of security for ADKG and provide a proof of correctness for our ADKG protocol.
- We implement and evaluate our ADKG protocol, showing our protocol is more efficient (2x-4.6x) than DYX+ protocol. We provide an open-source library for our protocol.

Concurrent work. Concurrent to our work, Das, Xiang, Kokoris-Kogias, and Ren (DXKR) provide a beautiful ADKG protocol [25] that offers improved performance to the DYX+ ADKG protocol. The DXKR protocol is more efficient than DYX+ ADKG protocol and appears more efficient than ours with the high threshold. However, the DXKR protocol relies on both ROM and PKI, while our protocol does not use ROM or PKI, thereby offering stronger security guarantees. Our protocol follows a path that is different from DYX+ ADKG and DXKR ADKG and is of independent interest. Besides, our ACSS and HACSS protocols do not rely on ROM, PKI, or follow the accusation paradigm (which can lead to extra rounds of communication). In addition, we contribute a new Golang library for both ADKG and ACSS to the community.

II. Related Work

Synchronous DKG. Unlike ADKG, synchronous DKG has been studied since 1990's and a considerable amount of synchronous DKG protocols have been proposed [44],[18],[29],[20],[31],[42],[35],[33].

Partially synchronous DKG. In partial synchronous settings, the protocol of Kate, Huang, and Goldberg [36] has $O(\lambda n^4)$ communication. The work [36] has recently been improved by a factor of $O(n/\log n)$ in computation at the cost of an $O(\log n)$ communication increase [47].

ADKG. In completely asynchronous environments, Kokoris-Kogias, Malkhi, and Spiegelman (KMS) provided the first ADKG protocol that has a total communication cost of $O(\lambda n^4)$ and expected round complexity of $O(n)$ [38].

Abraham et al. proposed the first ADKG protocol with $O(1)$ time and $O(\lambda n^3 \log n)$ communication [3]. Later and independently, Gao et al. [30] and Das, Xiang, and Ren [26] reduced the communication by a factor of $O(\log n)$. All these protocols [3], [30], [26] rely on a special publicly verifiable secret sharing (PVSS) scheme [35], so they only support the scenario where the secret key is a group element and cannot be used in the conventional threshold cryptosystems [46], [8], [45], [10], [15] where the secret key is a field element.

AVSS, ACSS, and HACSS. Our work is based on asynchronous complete secret sharing (ACSS) which compared to the conventional asynchronous verifiable secret sharing (AVSS), additionally ensures that if an ACSS protocol terminates at one server, then *all* correct servers will eventually receive valid shares. Earlier ACSS protocols proposed in the 1990s achieved unconditional security [19], [6], [17], yet at the expense of huge communication. The first practical ACSS scheme was achieved by Cachin et al. [13] (assuming the discrete logarithm assumption). Their protocol achieves an optimal message complexity of $O(n^2)$ and resilience of $n > 3t$, but has $O(\lambda n^3)$ communication complexity. Various ACSS protocols have been proposed to improve the communication complexity [49], [26], [4].

If an ACSS supports a high threshold of $2t+1$ out of $3t+1$, then we call it HACSS. The same paper of Cachin et al. [13] proposed the first dual-threshold ACSS with consensus threshold $t < n/4$ (not optimal) and privacy threshold $p < n/2$, with the same message and communication complexity as the above one. In KMS ADKG, the authors proposed the first HAVSS with optimal resilience. The construction comes with a price of $O(\lambda n^4)$ communication with 4 rounds of communication and the need for PKI. Alhaddad et al. reduced the communication complexity of HACSS to $O(\lambda n^2 \log n)$ without assuming PKI or trusted setup [4]. Das et al. further reduced the communication to $O(\lambda n^2)$ communication at the price of relying on ROM, PKI and non-standard assumptions [27].

III. System Model and System Overview

We consider distributed key generation in asynchronous environments making no timing assumptions on message processing or transmission delays. We assume $t \leq \lfloor \frac{n-1}{3} \rfloor$, which is optimal. For simplicity, we assume $n = 3t+1$.

Unlike prior constructions, our protocol in this paper does not rely on PKI (public key infrastructure). Namely, we neither use public-key encryption or digital signatures, nor rely on any properties that the model has. We only assume authenticated and private channels, the collision resistance property of hash functions, and the hardness of discrete logarithm. Also, our protocol works in the standard model, while prior constructions assume the random oracle model (ROM) treating hash functions as ideal random oracles.

Throughout the paper, λ denotes the security parameter, and $\mathsf{negl}(\lambda)$ refers to a negligible function vanishing faster than any inverse polynomial in the security parameter, "overwhelming" refers to $1 - \mathsf{negl}(\lambda)$ for a negligible function negl.

We define asynchronous distributed key generation (ADKG) as an interactive protocol that generates a key pair (pk, sk), where pk is the public key generated from ADKG protocol, and sk is a vector of n secret keys. Let $(\mathsf{transcript}, pk) \xleftarrow{\$} \mathsf{ADKG}(I, n)$, where n is the number of the servers, I denotes the indices of the faulty servers ($|I| \leq t$), and $\mathsf{transcript}$ represents the messages exchanged throughout the process. We consider a recover algorithm that takes as input secret shares submitted by any set of $p+1$ correct servers and outputs the secret key sk corresponding to pk. Here, p is the recovery threshold. If $p = t$, the threshold is a (regular) low threshold. If $p = 2t$, the threshold is a high threshold: Such a setting is needed for the popular BFT protocols (e.g., SBFT [32], HotStuff [48]). We consider the following properties of ADKG:

- **Termination.** All correct servers eventually terminate with probability 1.
- **Agreement.** All correct servers that terminate output the same ADKG public key pk.
- **Completeness.** Each correct server will output a secret key share sk_i.
- **Robustness.** There exists an efficient algorithm recover such that $sk \leftarrow \mathsf{recover}(\mathsf{ADKG}, sk_1, \ldots, sk_l)$, where the set of secret key shares contains shares from at least $p+1$ correct servers and sk is the unique key such that $pk \leftarrow \mathsf{KeyGen}(1^\lambda; sk)$.
- **Security preservation.** A threshold cryptographic scheme under ADKG retains all the properties of the standard scheme under the key generation (KeyGen) algorithm.

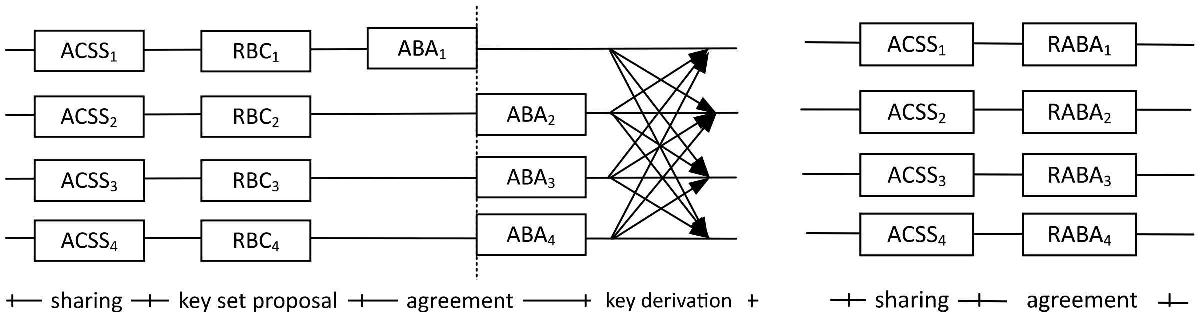
The formal definition of security preservation is described as below. We define a protocol $(\mathsf{transcript}, pk, sk, \mathsf{st}) \xleftarrow{\$} \mathsf{OracleDKG}(I, n)$ that knows the internal state of each server, including the internal state st of the adversary $\mathcal{A}$ and also the key pair (pk, sk). We define Game as a game containing the line $pk \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda; sk)$ (denoted line_{pk}) and where pk is the input to an adversary $\mathcal{A}$. Define $\mathsf{Game}'(\mathsf{line}, x)$ as Game but with line_{pk} replaced by line and $\mathcal{A}$ given x as input rather than pk. We go on to define $\mathsf{line}_{\mathsf{adkg}}$ as the line $(\mathsf{transcript}, pk, sk, \mathsf{st}) \xleftarrow{\$} \mathsf{OracleADKG}(I, n)$, and define $\mathsf{ADKG\text{-}Game} \leftarrow \mathsf{Game}'(\mathsf{line}_{\mathsf{adkg}}, \mathsf{st})$. We say ADKG preserves security for Game if $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{ADKG\text{-}Game}}(\lambda) \leq \mathsf{Adv}_{\mathcal{A}}^{\mathsf{Game}}(\lambda) + \mathsf{negl}(\lambda)$, for any probabilistic polynomial-time adversary $\mathcal{A}$.

Our definition of security preservation follows that of distributed key generation of Gurkan et al. [35]: we consider the security of ADKG and the underlying threshold cryptosystem altogether. We extent [35] to the asynchronous setting.

A. System Overview

Review of DYX+ ADKG. Existing ADKG protocols follow a common approach as follows: each of the n servers concurrently and independently runs an AVSS instance to share a random secret among all the servers; as long as $t+1$ AVSS instances terminate, all servers can locally add their shares as the secret key, because the added secret key contains at least a share from a correct server. The crux, however, is to agree on which AVSS shares to aggregate for the final secret key—a challenging task in asynchronous settings.

As shown in Figure 1a, DYX+ ADKG has four phases: sharing phase, key set proposal phase, agreement phase, and key derivation phase. In the sharing phase, each server uses an ACSS instance to secret share a random secret. In the key set proposal phase, each server waits for $t+1$ ACSS instances completed, and uses Byzantine reliable broadcast (BRB) [16], [11] to broadcast the key set of the indexes of the $t+1$ instances. In the agreement phase, servers agree on the valid key sets using n asynchronous binary Byzantine agreement (ABA) instances. As in [7], the phase consists of two subphases: ABA instances refrain from inputting 0 to any ABA until the *first* ABA terminates with 1. The random coins needed for the i-th ABA are generated from the aggregated secrets from the key set proposal phase; for ABA instances that have not received the corresponding key sets, servers rely on the good-case-coin-free property of ABA to guarantee termination [27], [23]. Finally, in the key derivation phase,



(a) DYX+ ADKG [27]. (b) Our ADKG.

Fig. 1: DYX+ ADKG vs. Our ADKG.

servers exchange commitments of secret shares and reconstruct the final public key [42].

DYX+ ADKG relies on various strong assumptions which we illustrate as follows:

- **Random oracle model (ROM) needed.** In the low threshold case, DYX+ ADKG implements the DXR ACSS scheme [26]. For the high threshold case, DYX+ ADKG implements DCR-ACSS. In both cases, zero-knowledge proofs and Fiat-Shamir heuristic are used to ensure provable security in the random oracle model (ROM). Additionally, the key derivation phase also uses a variant of Chaum-Pederson's protocol [22] to ensure share consistency—again requiring ROM.
- **PKI and strong cryptograhic assumptions needed.** The HACSS scheme used in DYX+ ADKG requires PKI. This is due to usage of publicly verifiable secret sharing (PVSS) scheme used in [26]. The HACSS scheme relies on the both the Decisional Composite Residuosity (DCR) assumption [43] (expensive to instantiate than elliptic curve based DL problems) and the DDH assumption (stronger than DL). Both assumptions are strong, adding additional requirements to the underlying threshold crytosystems. For instance, if we want to use the threshold encryption algorithm of Shoup and Gennaro [46] (based on the CDH assumption) with DYX+ ADKG, then we must additionally require PKI and the stronger DCR and DDH assumptions.

Our ADKG protocol. When designing our ADKG protocol, we have two goals in mind: 1) improved efficiency and 2) stronger security guarantees. Correspondingly, our resulting protocol makes ADKG more practical and more applicable.

Our protocol first benefits from a direct reduction from ADKG to ACSS or HACSS. As depicted in Figure 1b, our protocol has only two phases—a sharing phase (using ACSS or HACSS as usual) and an agreement phase (using reproposable ABA (RABA) [50] instead of using the conventional ABA). Our idea remains the same as prior works, yet being more terse. Namely, the sharing phase runs n parallel ACSS instances and the agreement phase agrees on which ACSS shares to aggregate for the final secret key. There are several challenges for this idea to work; moreover, we aim at building our ADKG protocol with stronger security guarantees—no ROM, PKI, or non-standard assumptions.

The first challenge is to ensure that ACSS and HACSS can be built efficiently and without the need of ROM, PKI, or non-standard assumptions. Recall we need an ACSS scheme with the homomorphic partial commitment property. In a sense, the property requires leaking g^s (where s is the shared secret) and g^s is additively homomorphic. Recall that in the DCR-ACSS of the DYX+ ADKG protocol, a standard ACSS scheme is used and a verifiable encryption is then applied to ensure the property. But we cannot use the same strategy, as doing so would necessarily use ROM.

To tackle the issue, we directly build a new HACSS scheme that natively has the homomorphic partial commitment property and avoids using non-standard cryptographic assumptions, ROM, or PKI. We develop the technique in Haven HACSS [4] yet leaking the commitment for the secret. We carefully design our protocol to ensure provable security in the sense of reduction-based modern cryptography.

Second, compared to the agreement phase in DYX+ ADKG, RABA in our agreement phase is fully parallelizable, reducing the running time of ABA phases in DYX+ ADKG. However, RABA requires using random coins as input. If we use the same approach as in DYX+ ADKG to derive random coins, then we would have to rely on ROM and stronger assumptions. We instead use the local coin based RABA in [51].

Last, in our approach, each server can obtain its private key by locally adding the shares agreed, and obtain the public key by aggregating the commitments of shares agreed. Namely, we do not need the key derivation phase in DYX+ ADKG. Hence, our approach gains in simplicity, modularity, and efficiency. But crucially, all these features benefit from a careful (and subtle) security proof (see Sec. VI).

IV. Building Blocks

A. ACSS and HACSS

An AVSS scheme is an interactive protocol between n servers and allows a dealer server to share a secret among all servers in such a way that they obtain consensus over the shared secret while also protecting the privacy of the secret until reconstruction time even in the presence of Byzantine servers. For dual-threshold AVSS, the number of servers required for reconstruction of the secret may be different from the number of faulty servers that the protocol can tolerate. For instance, a typical privacy threshold (called high threshold) is $2t+1$ out of $n=3t+1$ servers.

A (n, p, t) *dual-threshold asynchronous verifiable secret sharing* (DAVSS) protocol has two stages:

- **Sharing stage.** This stage begins when a special server (the "dealer") p_d, is activated on an input message of the form $(\text{ID}.d, \mathsf{in}, \mathsf{share}, s)$. Here, the value ID.$d$ is a tag identifying the session, and s is the dealer's secret. p_d begins the protocol to share s using ID.d. A server p_i has completed the sharing for ID.d when it generates a local output of the form $(\text{ID}.d, \mathsf{out}, \mathsf{shared})$.
- **Reconstruction stage.** After server p_i has completed the sharing stage, it may *start reconstruction* for ID.d when activated on a message $(\text{ID}.d, \mathsf{in}, \mathsf{reconstruct})$. Eventually, the server outputs $(\text{ID}.d, \mathsf{out}, \mathsf{reconstructed}, z_i)$, in which case we say that p_i *reconstructs* z_i for ID.d.

An (n, p, t)-DAVSS satisfies the following security properties with an adversary $\mathcal{A}$ controlling up to t servers.

- **Privacy.** If a correct dealer shared s using ID.d and at most $p - t$ correct servers started reconstruction for ID.d, then $\mathcal{A}$ has no information about s.
- **Liveness.** 1) If the dealer p_d is correct throughout the sharing stage, then with overwhelming probability all correct servers complete the sharing. 2) If some correct server completes the sharing for ID.d, then all correct servers complete the sharing for ID.d. 3) If all correct servers start reconstruction for ID.d, then with overwhelming probability every correct server p_i reconstructs some s_i for ID.d.
- **Correctness.** Once $p + 1$ correct servers have completed the sharing for ID.d, there exists a fixed value z such that the following holds with overwhelming probability: 1) if the dealer shared s using ID.d and is correct throughout the sharing stage, then $z = s$ and 2) if a correct server p_i reconstructs z_i for ID.d, then $z_i = z$.

A *high-threshold AVSS* (HAVSS) protocol is a (n, p, t)-DAVSS that supports any choice of $t < n/3$ and $p < n - t$.

For our purpose, we consider the following privacy notion, where the adversary learns the commitment g^s for the secret s.

- **Privacy.** If a correct dealer shared s using ID.d and at most $p - t$ correct servers started reconstruction for ID.d, then $\mathcal{A}$ has no information about s except for what is implied by the value $y = g^s$.

An HACSS is an HAVSS scheme that additionally ensures every correct server receives its share of the secret. Formally, an HACSS should additionally satisfy completeness:

- **Completeness.** If a correct server terminates the sharing protocol, then there exists a degree p polynomial $R(\cdot)$ such that $R(0) = s'$ and each correct server will hold a secret share $s'_i = R(i)$. If the dealer is correct, then $s' = s$.

If an HACSS scheme satisfies privacy and the commitments are additively homomorphic, we say it satisfies the *homomorphic partial commitment* property.

B. Commitment Schemes

In this work, we consider non-interactive commitment schemes for polynomials and vectors. For our purpose, we use Feldman commitment for the polynomial commitment and use Merkle tree for the vector commitment [40].

Let $\mathbb{G}$ be a cyclic group of a prime order q with a generator g. Given a polynomial $R(x)$ of degree p: $R(x) = r_0 + r_1 x + \cdots + r_p x^p$, where the coefficients $r_i \in \mathbb{Z}_q$, Feldman commitment for $R(x)$ is $(g^{r_0}, g^{r_1}, \cdots, g^{r_p})$. A Feldman commitment for $R(x)$ uniquely determines $R(x)$.

Vector commitments are succinct encodings of ordered lists in such a way that one can later open a value at a specific location [21], [39]. A *vector commitment scheme* $\mathcal{V} = (\text{VCom}, \text{VGen}, \text{VVerify})$ consists of three algorithms:

- $\text{VCom}(\vec{v}) \to C$ is given a vector $\vec{v} \in U^\ell$ where $\ell \le L$. It outputs a commitment string C.
- $\text{VGen}(\vec{v}, i) \to w_i$ is given a vector $\vec{v}$ and an index i. It outputs a witness string w_i.
- $\text{VVerify}(C, u, i, w) \to b$ takes as input a vector commitment C, an element $u \in U$, an index i, and a witness string w. It outputs a Boolean value b that only equals 1 if $u = \vec{v}[i]$ and w is the corresponding witness.

We need the binding property of vector commitments:

- **Binding.** No probabilistic polynomial-time adversary can compute a vector commitment C, a position i, two elements u and v, and two witnesses w_1 and w_2 such that $\text{VVerify}(C, u, i, w_1) = \text{VVerify}(C, v, i, w_2) = 1$ with non-negligible probability.

C. RABA

This work requires the use of a special asynchronous binary Byzantine agreement (ABA)—RABA (reproposable ABA), a notion that is proposed by Zhang and Duan [50]. We first review ABA and then introduce RABA.

Asynchronous (binary) Byzantine agreement (ABA). An ABA protocol is specified by *propose* and *decide*. Each server proposes an initial binary value for consensus and servers will decide on some value.

- **Validity.** If all correct servers *propose* v, then any correct server that terminates *decides* v.
- **Agreement.** If a correct server *decides* v, then any correct server that terminates *decides* v.
- **Termination.** Every correct server eventually *decides* some value.
- **Integrity.** No correct server *decides* twice.

RABA. In contrast to conventional ABA protocols, where servers can vote once only, RABA allows servers to change their votes. Formally, a RABA protocol tagged with a unique identifier id is specified by *propose*$(id, \cdot)$, *repropose*$(id, \cdot)$, and *decide*$(id, \cdot)$, with the input domain being $\{0, 1\}$. For our purpose, RABA is "biased towards 1." Each server can propose a vote v at the beginning of the protocol. Each server can propose a vote only once. A correct server that proposed 0 is allowed to change its mind and repropose 1. A server that proposed 1 is not allowed to repropose 0. If a server reproposes 1, it does so at most once. A server terminates the protocol identified by id by generating a decide message. RABA (biased toward 1) satisfies the following properties:

- **Validity**: If all correct servers *propose* v and never *repropose* $\bar{v}$, then any correct server that terminates *decides* v.
- **Unanimous termination**: If all correct servers *propose* v and never *repropose* $\bar{v}$, then all correct servers eventually terminate.
- **Agreement**: If a correct server *decides* v, then any correct server that terminates *decides* v.
- **Biased validity**: If $t+1$ correct servers *propose* 1, then any correct server that terminates *decides* 1.
- **Biased termination**: Let Q be the set of correct servers. Let Q_1 be the set of correct servers that propose 1 and never repropose 0. Let Q_2 be correct servers that propose 0 and later repropose 1. If $Q_2 \neq \emptyset$ and $Q = Q_1 \cup Q_2$, then each correct server eventually terminates.
- **Integrity**: No correct server decides twice.

Here validity is modified to accommodate the RABA syntax. Integrity ensures RABA decides once only. Meanwhile, unanimous termination and biased termination are introduced to help achieve RABA termination. External operations are needed to force the protocol to meet these termination conditions. Last, biased validity in RABA requires that if $t+1$ servers, not simply all correct servers, propose 1, then a correct server that terminates decides 1.

V. Our HACSS and ACSS Protocols

In this section, we show how to build our ACSS and HACSS protocols. Both protocols are based on the discrete logarithm (DL) assumption and rely on authenticated and private channels only. Namely, they do not assume PKI or ROM. Both protocols achieve a communication complexity of $O(\lambda n^3)$. Our ACSS protocol can be viewed as a special case of HACSS, so we focus on HACSS.

A. Our HACSS

Our HACSS protocol follows the Haven framework due to AlHaddad, Varia, and Zhang [4] but differs from it in significant manners. The two instantiations in Haven (based on KZG commitments [37] and Bulletproofs [12]) perfectly hide the secret key. They cannot be directly used in threshold crytosystems, as we require the underlying HACSS to satisfy the homomorphic partial commitment property. We therefore propose a new HACSS protocol exposing g^s where s is the secret to be shared. Our construction is devised to satisfy provable security, where an efficient simulator can simulate the view of the adversary and the adversary can learn no information about s except for what is implied by the value g^s. Crucially, we need to avoid using PKI and ROM. We compare our HACSS scheme with existing ones in Table I.

HACSS proceeds in two phases: a sharing phase in which the dealer distributes shares of a secret s (**Algorithm 1**), and a reconstruction phase in which the servers collectively reconstruct the secret (**Algorithm 2**).

Sharing phase. The sharing phase of our HACSS has three stages, following the same communication pattern as Bracha's

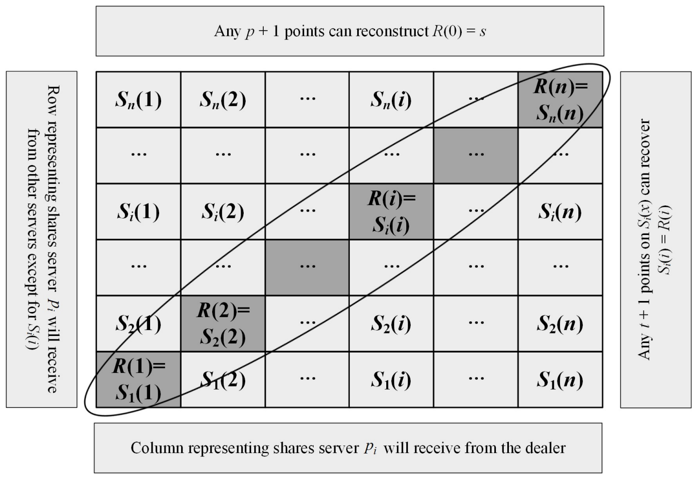

Any $p+1$ points can reconstruct $R(0) = s$
Row representing shares server p_i will receive from other servers except for $S_i(i)$
Any $t+1$ points on $S_i(x)$ can recover $S_i(i) = R(i)$
$S_n(1)$
$S_n(2)$
$\cdots$
$S_n(i)$
$R(n)= S_n(n)$
$S_i(1)$
$S_i(2)$
$R(i)= S_i(i)$
$S_i(n)$
$S_2(1)$
$R(2)= S_2(2)$
$S_2(i)$
$S_2(n)$
$R(1)= S_1(1)$
$S_1(2)$
$S_1(i)$
$S_1(n)$
Column representing shares server p_i will receive from the dealer

Fig. 2: Illustration of our initial work.

BRB [11]. The dealer p_d transmits an $O(n^2)$-size send message to all servers. Then each server broadcasts an $O(n)$-size echo and ready messages. Hence, our has $O(n^2)$ messages and $O(\lambda n^3)$ communication.

The initial work of the dealer p_d is shown in lines 1-16 and in Figure 2.

- In lines 2-4, the dealer first randomly samples a degree-p *recovery polynomial* R such that $R(0) = s$. Then the dealer computes a Feldman polynomial commitment for R.
- In line 5, we let G denote g^s.
- In lines 6-9, the dealer first samples n degree-t *share polynomials* $S_1, \ldots, S_n$. It holds $S_i(i) = R(i)$, but the n share polynomials are otherwise uniformly sampled. Then the dealer computes polynomial commitments of all S_i.
- In lines 11-12, the dealer forms a vector $\boldsymbol{y}_i^S$ containing n evaluations (in a transposed order). The vector contains the evaluation of one point on each share polynomial $S_1, \ldots, S_n$. In total, the vector contains n^2 points.
- In lines 13-14, the dealer forms the *root commitment* C, a vector commitment to all of the polynomial commitments. Abusing notation, we assume each polynomial commitment contains the witness to its own inclusion in C. We will ignore the witness when running PVerify for simplicity.
- In lines 15-16, the dealer sends to server p_i the root commitment, all $n+1$ polynomial commitments, one evaluation on everybody's share polynomial.

Once a server p_i receives the send message from the dealer, it verifies if the message is "consistent" (lines 17-20):

- All polynomial commitments link back to the root commitment.
- All polynomial evaluations received are verifiably part of S_j.
- The recovery and share polynomials are equal at $R(i) = S_i(i)$.

If all checks pass, the server p_i sends an echo message to each server p_m containing the root commitment C and two pieces of information about server m's share polynomial: its

Algorithm 1 Sharing phase of HACSS, for server p_i and tag ID.d

```
upon receiving (ID.$d$, in, share, $s$):    ▷ send stage; only if $p_i$ is the dealer $p_d$
  randomly choose recovery polynomial $R$ of degree $p$ s.t. $R(0) = s$
                                            ▷ concretely, $R(x) = r_0 + r_1 x + \cdots + r_p x^p$ where $r_0 = s$
  $\hat{R} \leftarrow \mathrm{PCom}(R, p)$    ▷ make polynomial commitment for $R(x)$; concretely, $\hat{R} = (g^{r_0}, g^{r_1}, \cdots, g^{r_p})$
  $G \leftarrow g^s$                          ▷ let $G$ denote $g^s$
  for $j \in [1..n]$ do                       ▷ $j$ denotes the row of the matrix
    randomly choose share polynomial $S_j$ of degree $t$ s.t. $S_j(j) = R(j)$
                                            ▷ concretely, $S_j(x) = s_{j,0} + s_{j,1}x + \cdots + s_{j,t}x^t$ s.t. $S_j(j) = R(j)$
    $\hat{S}_j \leftarrow \mathrm{PCom}(S_j, t)$    ▷ concretely, $\hat{S}_j = (g^{s_{j,0}}, g^{s_{j,1}}, \cdots, g^{s_{j,t}})$
  $\hat{\boldsymbol{S}} \leftarrow (\hat{S}_1, \hat{S}_2, \ldots, \hat{S}_n)$
  for $i \in [1..n]$ do                       ▷ evaluate and create witnesses ($n^2$ points in total))
    $\boldsymbol{y}_i^S \leftarrow [S_j(i)]$ for $j \in [1..n]$    ▷ one point on each $S_j$
  $C \leftarrow \mathrm{VCom}(\hat{R}, \hat{S}_1, \hat{S}_2, \ldots, \hat{S}_n)$    ▷ root vector commitment
  append to $\hat{R}$ and each $\hat{S}_j$ a witness of inclusion in $C$ at the right location
  for $i \in [1..n]$ do
    send (ID.$d$, send, set$_i$) to server $p_i$, where set$_i = \{C, G, \hat{R}, \hat{\boldsymbol{S}}, \boldsymbol{y}_i^S\}$

upon receiving (ID.$d$, send, set$_j$) from $p_d$ for the first time:    ▷ echo stage
  if $\hat{R}$ and all $\hat{\boldsymbol{S}}$ are in $C$ at the expected locations then
    if $\mathrm{PVerify}(\hat{S}_j, S_j(i), t) = 1$ for all $j \in [1..n]$ then    ▷ concretely, verify if all $g^{S_j(i)} = \prod_{k=0}^{t} (\hat{S}_j(k))^{i^k}$
      if $S_j(j) = R(j)$ for all $j \in [1..n]$ then    ▷ concretely, verify if all $\prod_{k=0}^{t} (\hat{S}_j(k))^{j^k} = \prod_{k=0}^{p} (\hat{R}(k))^{j^k}$
        for $m \in [1..n]$ do                 ▷ server $i$ sends message to each server $p_m$
          send (ID.$d$, echo, info$_{i,m}$) to $p_m$, where info$_{i,m} = \{C, G, \hat{S}_m, S_m(i)\}$

upon receiving (ID.$d$, echo, info$_{m,i}$) from $p_m$ for the first time:    ▷ ready stage
  if $\hat{S}_i$ is in $C$ at location $i$ and $\mathrm{PVerify}(\hat{S}_i, S_i(m), t) = 1$ then
    if not yet sent ready and received $2t + 1$ valid echo with $C$ and $G$ then
      send (ID.$d$, ready, $C$, $G$) to all servers

upon receiving (ID.$d$, ready, $C$, $G$) from $p_m$ for the first time:
  if not yet sent ready and received $t + 1$ ready with this $C$ and $G$ then
    send (ID.$d$, ready, $C$, $G$) to all servers    ▷ amplification step

  if received $2t + 1$ ready with this $C$ and $G$ then    ▷ delivery
    wait to receive $t + 1$ valid echo with this $C$ and $G$
    interpolate $S_i$ from the $t + 1$ valid $S_i(m)$ in the received echo
    compute $S_i(i)$
    output (ID.$d$, out, shared)
```

commitment $\hat{S}_m$ and the evaluation at one point $S_m(i)$ to help p_m interpolate the polynomial. *The step is vital to achieve the "completeness" property for our HACSS protocol.* The remainder of the protocol proceeds as in Bracha's broadcast. Upon receiving $2t + 1$ valid echo messages with the same C, a server broadcasts a ready message. If a server receives $t + 1$ ready message with the same C, and if a server has not broadcast a ready message, then it broadcasts a ready message.

For our construction, we use Merkle tree as the underlying vector commitment. The Merkle tree root element is a vector of $O(n)$ group elements and a Merkle tree proof has $O(\lambda n \log n)$ communication. The total communication incurred by Merkle tree in send is bounded by $O(\lambda n^2 \log n)$, while the communication in send is $O(\lambda n^3)$.

Reconstruction phase. If a server p_i completes the sharing and starts the reconstruction stage, then this server knows the share polynomial S_i (and a witness that it links back to some agreed root commitment C). The server sends all servers an evaluation of $S_i(i)$ and together the witness linking S_i to C. Each server verifies the evaluation and interprets this point as $R(i)$. Given $p + 1$ valid messages from other servers, p_i interpolates the corresponding points and recovers the secret.

B. Low-Threshold ACSS

Our technique works for the case of the single-threshold ACSS, where the privacy threshold equals the threshold t. To do this, one just needs to choose a recovery polynomial of degree t, while the rest of algorithms remain the same as those of our HACSS.

	resilience	communication	rounds	no PKI?	no ROM?	crypto assumption	homomorphic partial commitment?
CKLS [13]	$n > 4f$	$O(\lambda n^3)$	3	✓	✓	DL	✗
KMS [38]	$n > 3f$	$O(\lambda n^4)$	4	✗	✓	DL	✓
AVZ [4]	$n > 3f$	$O(\lambda n^2 \log n)$	3	✓	✗	DL	✗
DYX+ [27]	$n > 3f$	$O(\lambda n^2)$	3	✗	✗	DCR + DDH	✓
Our HACSS	$n > 3f$	$O(\lambda n^3)$	3	✓	✓	DL	✓

TABLE I: Comparison of HACSS protocols. HACSS without the homomorphic partial commitment property cannot be used for ADKG: we include them just for a comparison.

Algorithm 2 Reconstruction phase of HACSS, for server p_i and tag ID.d

```
1: upon receiving (ID.$d$, in, reconstruct):
2:     for $j \in [1, n]$ do
3:         send (ID.$d$, share, $C$, $G$, $\hat{S}_j$, $S_j(j)$) to $p_j$
4: upon receiving (ID.$d$, share, $C$, $G$, $\hat{S}_m$, $S_m(m)$):
5:                                                   ▷ from $p_m$
6:     if $\hat{S}_m$ in $C$ and PVerify$(\hat{S}_m, S_m(m), t) = 1$ then
7:         if received $p+1$ valid share with the same $C$ and $G$ then
8:             interpolate $R$ from the $p+1$ valid points
9:             output (ID.$d$, out, reconstructed, $R(0)$)
```

C. Analysis

We show the proof for our HACSS protocol and the proof easily follows for our low-threshold ACSS protocol.

Theorem 1. *Assuming the DL assumption and a secure vector commitment (that can be realized using Merkle tree), our HACSS protocol is a high-threshold ACSS achieving liveness, correctness, completeness, and privacy. Besides, our HACSS protocol achieves $O(n^2)$ message complexity and $O(\lambda n^3)$ communication complexity.*

Proof. We begin with the liveness properties.

Liveness-1. If the dealer p_d is correct, then p_d will send all servers the same root commitment and polynomial commitments. Hence, each server receives one point on each share polynomial. After receiving the send message, all checks will pass for each server. Hence, the correct servers can echo these points. The echo message will be accepted and each server will be able to send ready messages and interpolate their own share polynomial.

Liveness-2. (Liveness-2 corresponds to the usual "agreement" property of BRB. But the proof is trickier than that for Bracha's broadcast.) Assume that a correct server p_i has completed the sharing for ID.d. We show that another correct p_j will also complete the sharing.

First, as p_i completed the sharing, it must have received $2t+1$ ready messages with the same root commitment C. At least $t+1$ ready messages are from correct servers. Due to line 29 (the amplification step), this will cause all correct servers to send ready messages if they have not done so. Hence, p_j will eventually receive $2t+1$ ready messages with root commitment C, thereby satisfying the conditional on line 30.

Second, as p_i completed the sharing, p_i must have sent a ready message in line 26 or line 29. Note that the condition for line 29 cannot be satisfied until $t+1$ servers sent a ready due to line 26, at least one of whom must be correct (say, server p_m). Therefore, server p_m must have observed $2t+1$ echo messages that are consistent and linked to the same root commitment C. Hence, we know that at least $t+1$ of those echo message senders are correct. Thus, they will also send consistent echo messages to server p_j. So p_j can complete the wait step on line 31. Meanwhile, the echo messages have sufficient information for p_j to compute lines 32-33 and complete the sharing.

Liveness-3. Suppose all correct servers start reconstruction for ID.d. First, note that there are at least $n - t \geq p + 1$ correct servers. As these servers completed the sharing, they each have a share polynomial that can be linked back to the common root commitment C. Hence, these servers can construct an evaluation at $S_i(i)$ that will be accepted by others. Once $p+1$ such points are received, each correct server can verify the correctness of points and recover a secret.

Privacy. To formally prove the privacy property of our HACSS protocol, we need to build a simulator $\mathcal{S}$ that simulates the view of adversary using g^s for some unknown s. The simulation is technically non-trivial, but it is simpler than the one we will present for our ADKG protocol in Sec. VI. One can similarly build a simulator based on our ADKG simulator, so we simply omit the proof here.

Correctness-1. Assume that a correct dealer shared a secret s. Then, the share polynomial evaluations at all $\{S_i(i)\}_{i\in[1..n]}$ lie on a degree p polynomial that will recover s. Due to the Liveness-2 property, if a correct server completes the sharing, correct servers will have a common root commitment C. Due to the (perfect) binding property of vector commitment and Feldman commitment, it must hold $z = s$.

Correctness-2. If a correct server p_i reconstructs z_i, it must have received at least $p+1$ valid shares with the same vector commitment C. Any set of $p+1$ points uniquely determine a polynomial of degree p. As we have agreed on the commitment C, the commitments for different subsets of points must be the same due to the binding property of vector commitment. As Feldman commitment is also binding, the reconstruction is therefore unique.

Completeness. This is implied by the correctness proof and the liveness-2 proof. Indeed, once a correct server completes the sharing, each correct server p_i will receive at least $t+1$ valid echo messages allowing each correct server to obtain its share polynomial S_i and therefore obtain $R(i) = S_i(i)$.

Efficiency. The protocol achieves a message complexity of $O(n^2)$, just as in Bracha's BRB. In the send stage the dealer sends n messages of size $O(\lambda n^2)$, and in the other two stages

Algorithm 3 ADKG, for server p_i and tag ID

```
1: upon initialization
2:     $sk_i$; $pk$    ▷ the secret key, the public key of server $p_i$
3:     $\mathsf{cs} \leftarrow \emptyset$    ▷ the index set of agreed HACSS instances
4:     select random $s_i$    ▷ the HACSS secret shared by $p_i$
5:     $G_i \leftarrow g^{s_i}$    ▷ the commitment of $s_i$
6:     $HACSS.share(i, \mathsf{in}, \mathsf{share}, s_i)$    ▷ server $i$ shares $s_i$
7: upon (ID.$j$, out, shared)    ▷ the sharing phase of $p_j$'s HACSS completes
8:     if RABA$_j$ has not been started then
9:         propose 1 for RABA$_j$
10:    else
11:        repropose 1 for RABA$_j$
12: upon receiving (ID.$d$, out, shared) from $n - t$ HACSS instances
13:     for RABA instances that have not been started do
14:         propose 0
15: upon RABA$_j$ decides 1
16:     $\mathsf{cs} \leftarrow \mathsf{cs} \cup \{j\}$
17: upon all $n$ RABA instances decide
18:     wait until (ID.$j$, out, shared) for all $j \in \mathsf{cs}$
19:         output $sk_i \leftarrow \sum_{j \in \mathsf{cs}} s_j^i$; $pk \leftarrow \prod_{j \in \mathsf{cs}} G_j$
```

each server broadcasts n messages of size $O(\lambda)$. Therefore, our HACSS has $O(\lambda n^3)$ communication. □

VI. Our ADKG Protocol

We present our ADKG protocol in **Algorithm 3**. Our ADKG uses HACSS and RABA. In particular, we use the *HACSS.share* and *HACSS.reconstruct* primitives of HACSS, and *propose*, *repropose* and *decide* primitives of RABA [50], [51]. To avoid trusted setup, we use the Quadratic RABA from local coins [51].

Concretely, our ADKG protocol consists of n parallel HACSS instances and n parallel RABA instances. In the HACSS phase, each server p_i HACSS.shares a secret s_i for the i-th HACSS instance. If p_i delivers a secret share s_j^i from j-th HACSS instance, it proposes 1 for RABA$_j$. Upon delivery of $n - t$ HACSS instances, p_i immediately proposes 0 for all RABA instances that have not been started. If p_i later delivers a secret share from some HACSS$_j$, it has proposed 0 for RABA$_j$, and has not terminated RABA$_j$, then it reproposes 1 for RABA$_j$. Let cs be the set of indexes where RABA$_j$ decides 1. When all RABA instances terminate and all HACSS$_i$ ($i \in \mathsf{cs}$) instances are delivered, p_i locally adds the shares with indexes in cs (i.e., $\sum_{j \in \mathsf{cs}} s_j^i$) as the secret key share, and locally aggregates the corresponding commitments as the public key.

We first prove termination needed to prove other properties.

Theorem 2. *All correct servers eventually terminate.*

Proof. If all correct servers are activated on HACSS for ID, they will disperse their secret shares. Eventually, all correct servers will complete at least $2t+1$ HACSS instances. Hence, all correct servers will propose 0 for all RABA instances that have not been started. We distinguish three cases and show for each case all RABA instances will terminate.

We first consider case 1, where all correct servers propose 1 for an RABA. In this case, according to unanimous termination, the RABA instance eventually terminates.

We then consider case 2, where all correct servers propose 0. In this case, we further distinguish two sub-cases: 1) If they never repropose 1, the RABA instance eventually terminates due to unanimous termination. 2) If some servers repropose 1, then these servers must have delivered the corresponding secret shares. According to the agreement property (liveness 2) of HACSS, all correct servers will complete the HACSS instances and repropose 1. The protocol will terminate due to biased termination.

Finally, we consider case 3, where some correct servers propose 0 and some other correct servers propose 1. The case is similar to Case 2-2. Due to the agreement property of HACSS, correct servers will eventually repropose 1 and the RABA instance will terminate. Therefore, the protocol will eventually terminate. □

We are now ready to prove that each correct server will have the same set cs.

Lemma 3. *If any correct server outputs cs, then each correct server outputs cs.*

Proof. We consider the case where a correct server p_j delivers a set cs. We assume the corresponding set of secret shares is K: the i-th element in K, $K[i]$, may be empty or s_i^j (a secret share sent by p_i), depending on if the corresponding RABA instance RABA$_i$ decides 0 or 1, where $i \in [1..n]$. The index set cs contains the indexes $i \in [1..n]$ where $K[i]$ is non-empty.

We just need to show that each server p_k will output a set K' with a corresponding cs' such that $\mathsf{cs} = \mathsf{cs}'$. If p_j outputs a set K, then all RABA instances either decide 0 or 1. According to Lemma 2, we know all RABA instances must terminate. Due to the agreement property of RABA, these RABA instances decide the same values for p_k. Therefore, all RABA instances for p_k will terminate and decide the same values as p_j. Thus, p_k will have the same subset cs. □

We now prove a crucial theorem on completeness.

Theorem 4. *The set cs containing at least $t + 1$ non-empty values will be output.*

Proof. For simplicity, we assume $n = 3t + 1$. Conditioned on termination for all RABA instances (shown in Lemma 2), we now bound the number of RABA instances that decide 1 which corresponds to the number of non-empty elements. The proof is almost identical to the proof for efficiency in PACE BFT [50].

According to the biased validity property, a RABA instance RABA$_i$ will definitely decide 1, if $t + 1$ or more correct servers propose 1. We just need to bound the number of RABA instances where less than $t + 1$ correct servers propose 1.

A crucial observation is that a correct server will propose 1 for at least $2t+1$ RABA instances, a fact guaranteed by HACSS. All correct servers will input 1 for $(2t+1)(2t+1)$ for all RABA instances. There are at most $(3t+1)(2t+1)$ inputs for correct servers. Hence, the total number of the 0 input from all correct servers for all RABA instances is at most $(3t+1)(2t+1)-(2t+1)(2t+1)=2t^2+t$. Thus, the number of RABA instances that decide 0 is bounded by $\frac{2t^2+t}{t+1}<\frac{2t^2+2t}{t+1}=2t$. Thus, the number of RABA instances that decide 1 is at least $t+1$. $\square$

We now prove the robustness property.

Theorem 5. *Our ADKG protocol achieves robustness.*

Proof. At the end of ADKG protocol, it holds that if $i \in \mathsf{cs}$, then p_i has successfully performed the dealing of s_i under our HACSS protocol. For each such dealing of s_i, each correct server p_j holds shares s_i^j. According to the completeness property of HACSS, these shares s_i^j can interpolate to a unique polynomial with constant coefficient s_i. For any set S of $p+1$ shares, $s_i=\sum_{j\in S} l_j \cdot s_i^j$, where l_j are the Lagrange interpolation coefficients for the set S. We know each correct server p_j obtains their ADKG key sk_j as $sk_j=\sum_{i\in\mathsf{cs}} s_i^j$. Hence, for the set S of shares, we have

$$sk=\sum_{i\in\mathsf{cs}} s_i=\sum_{i\in\mathsf{cs}}(\sum_{j\in S} l_j\cdot s_i^j)=\sum_{j\in S} l_j\cdot(\sum_{i\in\mathsf{cs}} s_i^j)=\sum_{j\in S} l_j sk_j.$$

Namely, all ADKG secret keys $\{sk_j\}$ correspond to the unique secret key sk. Meanwhile, it is easy to distinguish valid shares from invalid shares based on the polynomial commitments broadcast. Therefore, the ADKG secret key sk can be reconstructed in an efficient manner using interpolation. Hence, we have completed the proof for robustness. $\square$

Finally, we prove that our ADKG protocol satisfies security preservation. Following Gurkan et al. [35], we just need to prove that our ADKG protocol satisfies the *key expressability* property. We review the definition of key-expressable DKG. For a simulator Sim, we define as $(\mathsf{transcript}, pk, \alpha, pk_2, sk_2) \leftarrow \mathsf{SimDKG}(\mathsf{Sim}, I, n)$ a run of the ADKG protocol, where all correct servers are controlled by Sim which takes as input a public key pk_1 and outputs α, pk_2, and sk_2. We say a DKG is key-expressable if there exists such a simulator Sim such that 1) $(\mathsf{transcript}, pk)$ is distributed identically to the output of $\mathsf{ADKG}(I,n)$, 2) (pk_2, sk_2) is a valid key pair, and 3) $pk=f(\alpha,pk_1,pk_2)=\alpha pk_1+pk_2$.

Theorem 6. *Our ADKG protocol satisfies security preservation.*

Proof. We present a simulator $\mathcal{S}$ that takes as input pk_1, and when the ADKG outputs pk_1, outputs α and sk_2 such that $pk=\alpha pk_1+pk_2$. Suppose $pk_1=g^s$ for some *unknown* secret s. The simulator $\mathcal{S}$ runs the ADKG with an adversary $\mathcal{A}$ that corrupts at most t servers. Due to Theorem 4, we know $\mathsf{cs} \geq t+1$. Hence, $|\mathsf{cs}|$ contains at least one correct server. Let $|\mathsf{cs}|=b$. Let I_C and I_M denote the set of correct servers and faulty servers in cs, respectively. Clearly, we have $\mathsf{cs}=I_C\cup I_M$, $|I_C|\geq 1$, and $|I_M|\leq t$.

At a high level, the simulation works as follows. First, it is easy to simulate the behavior of corrupted servers in cs and knows the secrets of corrupted servers distributed, as the simulator knows enough points on the polynomials shared by the corrupted servers. Second, it is also easy to simulate the view of adversary on behalf of correct servers, as the simulator controls all correct servers. But for one of the correct servers, say, p_k, the simulator has to change the value broadcasted by p_k to "hit" pk_1. While the simulator (in asynchronous environments) cannot enforce the final output to be exactly pk_1, it can, however, generate the output public key that has a known relation with its input public key pk_1, thereby completing the proof.

We distinguish three cases where p_i serves as an HACSS dealer. For $p_i \in I_C/\{k\}$, the simulator $\mathcal{S}$ can easily generate the messages transmitted to the faulty servers in I_M, as $\mathcal{S}$ can itself generate the random secret shared for these correct servers. In particular, $\mathcal{S}$ follows the ADKG protocol and generates the messages on behalf of correct servers.

For $p_i \in I_M$, the simulator $\mathcal{S}$ can also perfectly simulate the messages transmitted to faulty servers. When the ADKG protocol terminates, we know the HACSS process initiated by $p_i \in I_M$ completes. Due to the set agreement property of the asynchronous common subset approach, we know all correct servers include p_i in their agreed subset. Hence, due to the liveness property of the HACSS, all correct servers must have obtained their secret shares corresponding to s_i. As the simulator controls at least $n-t\geq p+1$ correct servers and the recovery threshold for the recovery polynomial is exactly p, the simulator can interpolate the $p+1$ shares to obtain s_i. Thus, the simulator $\mathcal{S}$ can perfectly simulate the view of the adversary and hence the transcript of ADKG for all $p_i \in I_M$.

Now we examine the case of p_k (for $i=k$). Given an cs set with b servers, we assume w.l.o.g. the identities of faulty servers and correct servers in cs are $\{1,\ldots,t\}$ and $\{t+1,\ldots,b\}$, respectively. We also assume $k\in\{t+1,\ldots,b\}$. (It is possible that less than t servers are faulty in cs, but this does not cause any problems for the simulation.)

Below we show how $\mathcal{S}$ would simulate the transcript for the case where p_k serves as the HACSS dealer. We describe the simulation step by step. The steps are depicted in Figure 3.

Step 1: We first consider the send stage of HACSS. We need to first simulate the polynomial commitment for the unknown recovery polynomial $R(x)$ where $R(0)=s$ and $g^s=pk_1$. The simulator $\mathcal{S}$ chooses p points (j,z_j) for $j\in[1..p]$ where $z_j \xleftarrow{\$} \mathbb{Z}_q$. Namely, we randomly choose $R(k)=z_k$, for $k\in[1..p]$.

The simulator interpolates $(pk_1, g^{z_1},\ldots,g^{z_p})$ for the set $\{0,1,\ldots,p\}$ to obtain $\hat{R}=(\hat{R}[0],\hat{R}[1],\ldots,\hat{R}[p])$, where $\hat{R}[0]=pk_1=g^s, \hat{R}[1]=g^{r_1},\ldots,\hat{R}[p]=g^{r_p}$. In particular, for $j\in[1..p]$, $\hat{R}[j]=g^{r_j}=\prod_{k=0}^{p}(g^{R(k)})^{\lambda_{jk}}$, where λ_{jk} are coefficients such that $r_j=\sum_{k=0}^{p}\lambda_{jk}R(k)$. Note the above process is different from the conventional "interpolation in the

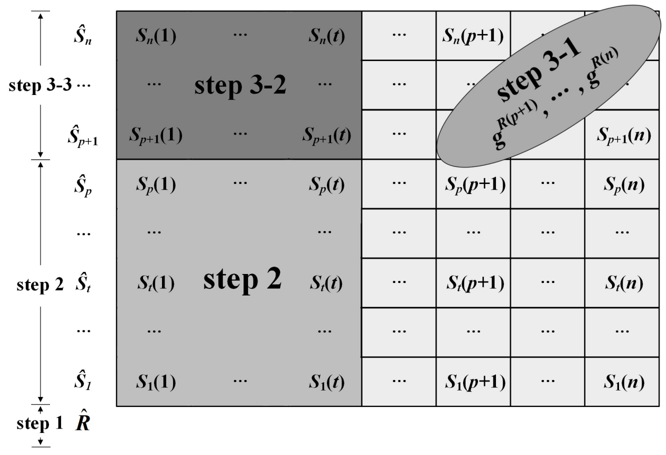


Fig. 3: Step-by-step simulation for the send stage.

exponent," as now we are not evaluating a new point in the exponent but computing the coefficients in the exponent.

We then simulate all the commitments for the n share polynomials and all secret shares distributed to (at most) t faulty servers $[1..t]$ (which contains at most $t \times n$ points). For the commitments for share polynomials, we distinguish two cases: the commitments for S_j where $j \in [1..p]$, and the commitments for S_j for the remaining $j \in [p+1..n]$.

Step 2: For the first case, the simulator $\mathcal{S}$ chooses p random polynomials subject to $S_j(j) = R(j)$. With the p share polynomials fixed, it is easy to first compute the commitment for each S_j, i.e., $\hat{S}_j$, and then directly compute $p \times t$ secret share values $S_j(i)$, where $j \in [1..p]$ and $i \in [1..t]$.

Step 3: What about S_j for $j \in [p+1..n]$? Note that the simulator does not know $R(j)$ for $j \in [p+1..n]$. In this case, the simulator performs the following steps:

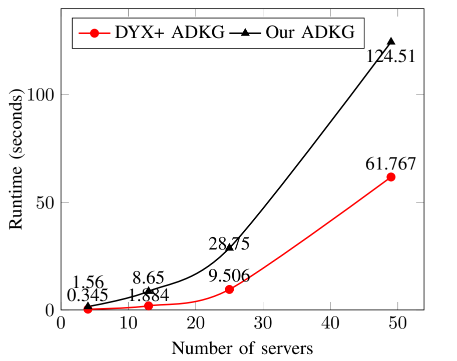
1) *Step 3-1:* $\mathcal{S}$ first uses $\hat{R}$ to obtain $g^{R(j)}$ for $j \in [p+1..n]$. Equivalently, $\mathcal{S}$ obtains $g^{S_j(j)} = g^{R(j)}$ for $j \in [p+1..n]$.
2) *Step 3-2:* $\mathcal{S}$ then chooses random points $S_j(i)$ for $j \in [p+1..n]$ and $i \in [1..t]$.
3) *Step 3-3:* For each $j \in [p+1..n]$, $\mathcal{S}$ directly computes $g^{S_j(i)}$ for $i \in [1..t]$ and then interpolates them and $g^{S_j(j)}$ to obtain the commitment of S_j for $j \in [p+1..n]$.

Note the simulation for the two cases is quite different. In particular, when simulating for the first case, we randomly choose the share polynomials such that the polynomials are subject to $S_j[j] = R(j)$. However, in the second case, we first randomly select t points for each share polynomial.

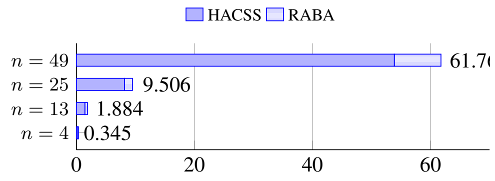
Step 4: In the echo stage, the simulator $\mathcal{S}$ knows the share polynomials S_j for $j \in [1..t]$ (and in fact knows S_j for $j \in [1..p]$) and easily simulates all shares sent to the faulty servers. (Note according to our protocol, the simulator does not need to send shares related to $S_j(i)$ for $j \in [t+1..n]$.)

Step 5: In the ready stage, the simulator $\mathcal{S}$ can easily simulate all messages transmitted. The same holds for the RABA phase.

Finally, when the ADKG terminates, the simulator $\mathcal{S}$ obtains the public key $pk = y_1 \cdots y_b$. Then the simulator $\mathcal{S}$ extracts $pk_2 = \prod_{j=1, j\neq k}^{b} y_j$ and $sk_2 = \sum_{j=1, j\neq k}^{b} s_j$. Clearly, we have $pk = pk_1 \cdot pk_2$ and $sk = sk_1 + sk_2$. The simulator outputs $(1, pk_2, sk_2)$. Thus, our ADKG protocol is key-expressable, which completes the proof of security preservation. □



(a) Average runtime for different network sizes.



(b) Runtime breakdown for our ADKG protocol.

Fig. 4: Evaluation and comparison.

VII. Implementation and Evaluation

Implementation. We implement[1] our ADKG protocol using Golang. We use gRPC as the communication library. For the HACSS protocol, we implement the one in Sec. V. For the RABA protocol, we use the Quadratic RABA proposed in [51] and implemented in [2]. Our library contains about 8,000 LOC in total.

For the HACSS protocol, we use Merkle tree as the vector commitment. As we commented earlier, doing so would not incur more communication. We use HMAC for the authenticated channels. We use CBC and HMAC to instantiate the authenticated and private channels. We use Dedis Kyber crypto library for all public-key cryptographic operations [1].

As part of our library, we include a visualized interface for our protocol to facilitate its adoption (detailed in our library).

Evaluation. We evaluate our ADKG protocol implementation on Amazon EC2 using up to 49 virtual machines (VM) evenly distributed in four EC2 regions. Each VM is a *t3a.medium* instance (with two virtual CPUs and 4GB RAM) running ubuntu 20.04.

We vary the network sizes to evaluate our protocol and compare the performance with DYX+ ADKG [27] using curve 25519. In each experiment, we use $n = 3t+1$ servers in total. We evaluate the ADKG protocols using the high threshold of $2t+1$ and report the average latency (runtime) for each n.

As shown in Figure 4a, our ADKG protocol consistently outperforms DYX+ ADKG. The runtime of DYX+ ADKG is about 2x-4.6x that of our ADKG protocol. The latency

[1]https://github.com/fififish/hacss

difference becomes comparably smaller as n increases. We did not evaluate the protocols for a larger n, as DYX+ ADKG library cannot complete as n further increases (which is consistent with the result in their paper [27] that does not report results for larger n's).

We also show a runtime breakdown of our ADKG protocol in Figure 4b. According to the results, the clear bottleneck is HACSS. The result justifies our design that aims at reducing the overall steps of ADKG. Also, we find that a faster HACSS would directly imply a more efficient ADKG protocol.

VIII. Conclusion

In this paper, we design and implement a simple and efficient ADKG protocol that improves both the efficiency and security guarantees of the-state-of-the-art DYX+ ADKG. In particular, the latency of DYX+ ADKG is about 2x-4.6x that of our protocol; unlike DYX+ ADKG requiring the both DDH and DCR assumptions, ROM, and PKI, our protocol relies on the classic and weaker discrete logarithm assumption only, and avoids ROM or PKI. We also build an efficient HACSS protocol with stronger security guarantees (the standard assumption and the standard model). Last, we contribute an open-source library for our HACSS and ADKG protocols.

References

[1] Dedis kyber go crypto library. https://github.com/dedis/kyber.

[2] WaterBear bft library. https://github.com/fififish/waterbear.

[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] Nicolas Alhaddad, Mayank Varia, and Haibin Zhang. High-threshold avss with optimal communication complexity. *IACR Cryptol. ePrint Arch.*, 2021:118, 2021.

[5] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[6] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 52–61, New York, NY, USA, 1993. Association for Computing Machinery.

[7] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *PODC*, pages 183–192. ACM, 1994.

[8] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *PKC*, 2003.

[9] Dan Boneh, Xavier Boyen, and Shai Halevi. Chosen ciphertext secure public key threshold encryption without random oracles. In *Cryptographers' Track at the RSA Conference*, pages 226–243. Springer, 2006.

[10] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *Journal of cryptology*, 17(4):297–319, 2004.

[11] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[12] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018.

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

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

[15] 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.

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

[17] Ran Canetti. *Studies in secure multiparty computation and applications*. PhD thesis, Citeseer, 1996.

[18] 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.

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

[20] 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.

[21] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *Public Key Cryptography*, volume 7778 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2013.

[22] David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *Annual international cryptology conference*, pages 89–105. Springer, 1992.

[23] 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.

[24] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

[25] Sourav Das, Zhuolun Xiang, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling. Cryptology ePrint Archive, Paper 2022/1389, 2022.

[26] 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*, pages 2705–2721, 2021.

[27] 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.

[28] Sisi Duan, Michael K Reiter, and Haibin Zhang. BEAT: Asynchronous bft made practical. In *CCS*, pages 2028–2041. ACM, 2018.

[29] 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.

[30] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Efficient asynchronous byzantine agreement without private setups. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 246–257. IEEE, 2022.

[31] 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.

[32] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *DSN*, pages 568–580, 2019.

[33] Jens Groth. Non-interactive distributed key generation and key resharing. *Cryptology ePrint Archive*, 2021.

[34] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *CCS*, 2020.

[35] Kobi Gurkan, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Aggregatable distributed key generation. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, 2021.

[36] Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed key generation in the wild. *Cryptology ePrint Archive*, 2012.

[37] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.

[38] Eleftherios Kokoris-Kogias, Alexander Spiegelman, and Dahlia Malkhi. 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.

[39] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 499–517. Springer, 2010.

[40] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

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

[42] 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.

[43] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, 1999.

[44] 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.

[45] Victor Shoup. Practical threshold signatures. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*, 2000.

[46] Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *J. Cryptol.*, 15(2):75–96, January 2002.

[47] Alin Tomescu, Robert Chen, Yiming Zheng, Ittai Abraham, Benny Pinkas, Guy Golan Gueta, and Srinivas Devadas. Towards scalable threshold cryptosystems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 877–893. IEEE, 2020.

[48] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *38th ACM symposium on Principles of Distributed Computing (PODC)*, 2019.

[49] Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. hbacss: How to robustly share many secrets. *NDSS*, 2022.

[50] Haibin Zhang and Sisi Duan. Pace: Fully parallelizable bft from reproposable byzantine agreement. *ACM CCS*, 2022.

[51] Haibin Zhang, Sisi Duan, Boxin Zhao, and Liehuang Zhu. Waterbear: Practical asynchronous bft matching security guarantees of partially synchronous bft. Cryptology ePrint Archive, Paper 2022/021, 2022.