Practical Asynchronous Proactive Secret Sharing and Key Refresh

Christoph U. Günther
Institute of Science and Technology Austria
cguenthe@ist.ac.at

Sourav Das
University of Illinois at Urbana-Champaign
souravd2@illinois.edu

Lefteris Kokoris-Kogias
Institute of Science and Technology Austria & Mysten Labs
ekokoris@ist.ac.at

Abstract

With the emergence of decentralized systems, spearheaded by blockchains, threshold cryptography has seen unprecedented adoption. Just recently, the trustless distribution of threshold keys over an unreliable network has started to become practical. The next logical step is ensuring the security of these keys against persistent adversaries attacking the system over long periods of time.

In this work, we tackle this problem and give two practical constructions for Asynchronous Proactive Secret Sharing. Our first construction uses recent advances in asynchronous protocols and achieves a communication complexity of $O(n^3)$ where n is the total number of nodes in the network. The second protocol builds upon the first and uses sortition to drive down the communication complexity to $O(cn^2)$. Here, c is a tunable parameter that controls the expected size of the sharing committee chosen using the existing random coin.

Additionally, we identify security flaws in prior work and ensure that our protocols are secure by giving rigorous proofs. Moreover, we introduce a related notion which we term Asynchronous Refreshable Secret Sharing — a functionality that also re-randomizes the secret itself. Finally, we demonstrate the practicability of our constructions by implementing them in Rust and running large-scale, geo-distributed benchmarks.

1 Introduction

Threshold cryptosystems have seen a surge of interest in the last few years as they allow for reduced communication and verification cost of consensus protocols [2, 19, 36] as well as for protection against arbitrage and censorship attacks in blockchain systems [26, 30, 39]. These applications are only the beginning towards general multi-party computation incorporating threshold cryptography [28].

In threshold cryptosystems, a secret key is distributed amongst participants so that each node holds a share of the secret and a certain number, i.e., the *threshold*, of them are required to reconstruct the secret key. Recent research has been focused on generating and distributing a secret key in a trustless manner. This has resulted in relatively practical setup protocols called Asynchronous Distributed Key Generation (ADKG) [1, 15, 16, 27] that neither relies on trusted third-parties nor on tight synchrony bounds. However, setting up the shared secret key is only one part of a complete solution for a practical and secure threshold cryptosystem.

Crucially, most use cases of threshold cryptosystems are long-term. So a persistent adversary can slowly corrupt one node after another. Eventually, it will hold sufficiently many shares to compromise the secret. Prior work introduced a countermeasure in the form of *Proactive Secret Sharing (PSS)* [9, 22, 32, 34, 40]. In a PSS protocol, nodes re-randomize their shares while ensuring that the secret key stays the same. Thus, after re-randomization, all shares previously compromised by the adversary become worthless.

Unfortunately, most existing works either assume a (partially) synchronous network or scale too poorly for regular use. Additionally, some contain subtle flaws in their security proofs and thus do not guarantee that honest nodes receive unbiased shares (cf. §4.2).

Currently, the only practical way to achieve proactive security in an asynchronous networks is to repeatedly run an ADKG. This is suboptimal for two reasons: First, the shared secret changes on every execution which rules out applications that require a persistent shared secret key (e.g., threshold signatures). Second, ADKGs are not purpose-built for this task, as they do not use the common coin that nodes have access to from the initial sharing. To circumvent these challenges, we propose an *Asynchronous Proactive Secret Sharing (APSS)* protocol that is both practical and secure.

Our Approach. Our protocols work with any discrete logarithm-based threshold cryptosystem built using Shamir secret sharing [33]. Briefly, similar to some prior PSS schemes, nodes generate a random blinding polynomial $p(\cdot)$ with $p(0) = 0$, henceforth called a *zero polynomial*. Then, each node i receives a point $p(i)$, $i > 0$ and updates its share of the secret by adding $p(i)$ to it. In a network of $n = 3f + 1$ nodes, our protocol can tolerate up to f malicious nodes. Further-

more, it has a communication cost of $O(n^3)$ and can support secrets with a reconstruction threshold within $[f+2, 2f+1]$. We prove the security of our protocol in the Random Oracle Model, assuming the hardness of the Decisional Diffie-Hellman problem.

While a communication complexity for $O(n^3)$ is efficient enough for small to medium size networks (say $n \leq 64$), it is impractical when n is large. For such scenarios, we modify our protocol such that expensive parts of the protocol are executed by a small, randomly sampled committee of c nodes. This drives down the communication complexity to $O(cn^2)$ allowing for true scalability.

We want to note that, the reconstruction threshold $f+2$ is one-off from optimal threshold of $f+1$. Intuitively, this is because one point of the zero polynomial, $p(0) = 0$, is always known to the adversary.

In addition to APSS, we introduce the notion of *Asynchronous Refreshable Secret Sharing (ARSS)*. It is similar to APSS as it re-randomizes the shares, but it does not guarantee that the shared secret stays the same. Therefore, it can be viewed as executing an ADKG again to sample a fresh secret. Such a functionality is sufficient in scenarios where, e.g., the threshold cryptosystem is only used to power a common coin. Compared to ADKG, ARSS is more efficient since it can utilize the already existing shared randomness. Our ARSS protocol maintains the efficiency of the APSS schemes while achieving the optimal reconstruction threshold $f+1$.

Implementation and Evaluation. We implement both of our APSS constructions using Rust in a reusable and modular fashion. Our experimental evaluation running across geographically distributed Amazon EC2 instances demonstrates the practicability of both of our constructions. As part of this, we compare our APSS to the state-of-the-art ADKG [16] which — as expected — shows that our construction is more efficient and thus better suited for repeated use.

In summary, we make the following contributions:

- We give two practical APSS constructions that support high-thresholds. The second is the first (to our knowledge) with sub-cubic complexity.
- We identify subtle vulnerabilities in prior work and show how certain protocol-design techniques can be used to alleviate such issues.
- We provide a publicly-available Rust prototype of our protocols. It is modular by design, and we hope that parts of it, specifically the networking crate, will be useful in implementing asynchronous protocols in Rust.
- We benchmark the prototype across a large-scale, globally distributed network of nodes to demonstrate the practicability of our constructions.

Paper organization. The paper is organized as follows: First, we discuss related work in §2 and then cover preliminaries in §3. Then, we define polynomial generation — a core building block of our protocols — in §4 and give a cubic and a sub-cubic construction in §5. Using these, we design an APSS scheme in §6 and ARSS scheme in §7. We prove the correctness of our constructions in §8, benchmark them in §9 and conclude with §10.

Table 1: A selection of prior PSS work.

	Network Model	Fault Tolerance	Communication Cost (total)	Trustless Setup
Herzberg [22]	sync.	$n/2$	$O(n^3)$	✓
CHURP [29]	sync.	$n/2$	$O(n^3)$	✗
MPSS [32]	partial sync.	$n/3$	$O(n^4)$	✓
COBRA [34]	partial sync.	$n/3$	$O(n^3)$	✓*
Cachin et al. [9]	async.	$n/3$	$O(n^4)$	✗
Zhou et al. [40]	async.	$n/3$	$O(\exp(n))$	✓
Shanrang [35]	async.	$n/4$	$O(n^3 \log n)$	✗
Our APSS (§5.1)	async.	$n/3$	$O(n^3)$	✓*
Our APSS (§5.2)	async.	$n/3$	$O(cn^2)^\dagger$	✓*
Our ARSS	async.	$n/3$	$O(n^3)/O(cn^2)^{\dagger\ddagger}$	✓*

† $c \leq n$ is the size of the committee and a tunable parameter.
‡ Depending on whether basing it on §5.1 or §5.2, similar to our APSS.
* Can also use constructions requiring a one-time setup.

2 Related Work

Most prior works (cf. Table 1) have considered PSS in (partially) synchronous models (e.g., [22, 29, 32, 34]) and achieve at best $O(n^3)$ communication cost. In contrast, our protocols function in asynchronous networks while being more performant. Additionally, our second construction enjoys sub-cubic communication cost.

Only a few other asynchronous protocols exist [9, 35, 40]. However, compared to our work, they are either considerably less efficient [10, 40] or not optimally fault-tolerant [35].

The most similar works to ours are [22, 32, 34], which also generate a zero polynomial to update the shares held by honest nodes. However, they are either hard to prove secure or inefficient. The other major approach taken by prior work [9, 29, 35] is based on bivariate polynomials. Its core idea is that instead of generating zero polynomials, nodes secret share their current share and thereby implicitly define a bivariate polynomial which can then be used to calculate new shares.

One particular challenge we do not consider in this work is that of dynamically changing committees over time [29, 32, 34, 35, 40] as our primary focus is on improving the performance of proactive protocols. Nevertheless, combining our work with the approach taken by COBRA [34] should readily provide an efficient and provably-secure APSS that supports dynamic committees. We leave the actual composition and security proofs to future work.

Finally, two concurrent works on APSS have been released [24, 38]. Similarly to our work, they build on advances in asynchronous protocols design (i.e., efficient Multi-valued Byzantine agreement and Complete Secret Sharing). Yet, unlike our work, both are based on bivariate polynomials and only achieve a communication complexity of $O(n^3)$. In terms of concrete performance, DyCAPS [24] prototype is considerably slower than our cubic implementation (300 vs. 120 seconds for $n = 64$), and we cannot compare bandwidth consumption as they do not state it. Similarly, we cannot compare our work with [38] since it is focused on a batched setting and only gives a limited amount of benchmarks.

3 Preliminaries

3.1 Notation

We omit the security parameter κ for readability and leave it implicit. For an integer $x > 0$, let $[x]$ denote the set $\{1, \ldots, x\}$. $a \leftarrow A$ denotes that element a is chosen uniformly at random from the set A. Let $\mathbb{Z}_q$ be a field of size q and let $\mathbb{G}$ be a group of prime order q where the Decisional Diffie-Hellman (DDH) problem is assumed to be hard. Also, let $g, h \in \mathbb{G}$ be two uniformly random and independent generators of $\mathbb{G}$.

We denote vectors in bold (e.g., $\mathbf{v}$) and assume they are of the appropriate length. $\mathsf{idx}(\mathbf{v}) \subseteq [|\mathbf{v}|]$ returns the set of coordinates of a vector containing a value. When its meaning is clear from context, we sometimes abuse notation and apply idx to sets as well.

For any given $d < n$, our protocol uses $(n, d+1)$ Shamir secret sharing [33] to secret share elements in $\mathbb{Z}_q$. A $(n, d+1)$ Shamir secret sharing of any secret $s \in \mathbb{Z}_q$ implies that s is secret shared using a degree d polynomial, and can be reconstructed using $d+1$ valid shares.

Shared secrets are often used in threshold signatures (cf. Appendix B). Thus, we often call the secret shared value *the secret key* sk instead of *the secret* s. The corresponding public key is $\mathsf{pk} = g^s$. For readability, we sometimes omit passing the secret key to certain functions and write, e.g., $\mathsf{psign}_i(\cdots)$ instead of $\mathsf{psign}(\cdots, \mathsf{sk}_i)$.

3.2 Model

This paper focuses on threshold cryptosystems for discrete logarithm-based cryptosystems. We consider a network of n nodes, denoted with $\{1, 2, \ldots, n\}$, that jointly execute our protocols. The nodes are pairwise connected by private and authenticated channels. We consider the presence of a probabilistic polynomial time (PPT) static Byzantine adversary $\mathcal{A}$ that can corrupt up to $f < n/3$ nodes.

Proactive protocols focus on the long-time security of cryptosystems across multiple *epochs*. Epochs are demarked by executions of a proactive protocol, and $\mathcal{A}$ can corrupt different sets of nodes in each epoch. Unfortunately, APSS is impossible in a model where $\mathcal{A}$ is allowed to launch active attacks across epochs [3]. Such attacks are possible because private and authenticated channels are too weak to guarantee security in asynchronous systems. In particular, $\mathcal{A}$ can leak communication between two honest nodes in an epoch τ. Assume that, during epoch τ, two nodes i and j are honest, and i sends a message to j. $\mathcal{A}$ can delay this message until a later epoch $\tau' > \tau$ where j is corrupt and leak it. Similarly, if a node is corrupt in τ, $\mathcal{A}$ can use its key to sign messages belonging to phase $\tau' > \tau$ in advance. Then, in phase τ', even if the node is honest again, the adversary can use these messages to make the node behave in a Byzantine manner.

This paper assumes that $\mathcal{A}$ does not actively attack the system across multiple epochs. Note that $\mathcal{A}$ can still actively disrupt each protocol execution in isolation, and it can collect shares across epochs. Thus, we will focus on one execution of the proactive protocol.

The impossibility result can also be circumvented by assuming channels that ensure a message can be received in an epoch if and only if it has been sent in the same epoch. This approach is considered in [9], where they implement such strong channels using secure co-processors.

Notation	Description
n	Number of nodes
f	Number of corrupted nodes
$\mathbb{Z}_q$	Field of prime-order q
$\mathbb{G}$	DDH group of prime-order q
g, h	Random, independent generators of $\mathbb{G}$
d	Polynomial degree
$p(x)$	Degree-d Polynomial
a_i	Polynomial coefficients
$\mathbf{v}$	Feldman commitment to a polynomial
$\mathbf{w}$	Commitment to a polynomial's points $g^{p(i)}$, $i \in [n]$
s, sk	Secret/secret key
pk	Commitment to/public key of s, i.e., g^s
s_i, sk_i	Secret/secret key share of node i
pk_i	Public key of s_i, i.e., g^{u_i}
σ	Signature
σ_i	Partial signature
Σ	Set of partial signatures
Φ	Set of tuples $\langle i, \sigma_i \rangle$. Used as VABA value.
c	Committee size

Table 2: Notation

3.3 Asynchronous Proactive Secret Sharing

Definition 1. Consider a network of n nodes, where nodes hold a $(n, d+1)$ Shamir secret share of a secret $s \in \mathbb{Z}_q$. Let s_i be the secret share of s held by node i. Each node also knows the commitment g^s as well as g^{s_j} for every node $j \in [n]$.

An Asynchronous Proactive Secret Sharing (APSS) [12], parameterized by d, is a protocol among the n nodes to re-

randomize their $(n, d+1)$ shares of a secret s. Specifically, at the end of the protocol, nodes output a new $(n, d+1)$ sharing of the secret s. Nodes also output the corresponding set of commitments, i.e., each node i outputs $\tilde{s}_i$ and threshold public keys $g^{\tilde{s}_j}$ for each $j \in [n]$.

An APSS protocol must satisfy the following properties except for negligible probability:

- *Termination:* If all honest nodes start the APSS protocol, then all honest nodes will eventually terminate the protocol.
- *Correctness:* If the APSS protocol terminates at any honest node, then every honest node will eventually output their $(n, d+1)$ secret share of s. Say $\tilde{s}_i$ is the secret share of node i, then nodes also output the threshold public keys $g^{\tilde{s}_j}$ for every $j \in [n]$.
- *Secrecy:* The adversary learns no information about the secret s apart from what is revealed by the commitments.

3.4 Asynchronous Complete Secret Sharing

Asynchronous Complete Secret Sharing (ACSS) [4, 15, 16, 37] allows a dealer to share a secret $s \in \mathbb{Z}_q$ with other nodes using $(n, d+1)$ Shamir secret sharing scheme such that, eventually, all honest nodes will receive a share. Moreover, all shares are consistent with some degree-d polynomial $p(\cdot)$.

We extend this standard notion of ACSS and require that nodes additionally output a polynomial commitment to the underlying degree d polynomial $p(\cdot)$ where:

$$p(x) = s + a_1 x + a_2 x^2 + \cdots + a_d x^d \tag{1}$$

In our protocol, we use the well-known *Feldman commitment* [17], denoted by $\mathbf{v}$ and defined as:

$$\mathbf{v} = [g^s, g^{a_1}, g^{a_2}, \ldots, g^{a_d}] \tag{2}$$

Definition 2 (ACSS [15, adapted])**.** An ACSS scheme enables a *dealer* to share a secret $s \in \mathbb{Z}_q$ by distributing a random degree-d polynomial $p(\cdot)$ with $p(0) = s$ to all other nodes. Let $\mathsf{share}_i^g(s, d)$ be the functionality dealer uses to share the secret s. Here, g is a uniformly random generator of group $\mathbb{G}$. Once the share terminates, all nodes will output a share $s_i = p(i)$ and the Feldman commitment $\mathbf{v}$ to $p(\cdot)$ with base g.

An ACSS scheme must satisfy the following properties except for a negligible probability:

- *Termination:* If the dealer is honest, then every honest node will eventually terminate the share protocol. Moreover, if any honest node terminates the share protocol, then every other honest will eventually terminate share protocol.
- *Completeness:* If some honest node terminates share, then there exists a degree-d polynomial $p(\cdot)$ such that $p(0) = s'$ and each honest node i will eventually hold a share $s_i = p(i)$ and Feldman commitment $\mathbf{v}$ to $p(\cdot)$. Moreover, when the dealer is honest, $s' = s$.
- *Secrecy:* If the dealer is honest and no honest node has started participating in a reconstruction effort, then an adversary that corrupts at most d nodes has no information about s apart from what is revealed by $\mathbf{v}$.

Our prototype implementation uses a modified version of HAVEN [4] since it was the easiest to implement given the available libraries for Rust. It has a communication complexity of $O(n^2)$ but requires a one-time trusted setup. This setup can be avoided by using [16], which still enjoys the same communication complexity.

3.5 Validated Asynchronous Byzantine Agreement

Definition 3 (VABA [2, adapted])**.** VABA allows nodes to agree on a value M that satisfies a Boolean predicate $Q(\cdot)$. We require that the following properties hold against an adversary except for negligible probability:

- *Termination:* If all honest nodes start with valid values, then all honest nodes will eventually decide.
- *Validity:* If an honest nodes decides on a value M, then $Q(M) = \top$.
- *Agreement:* All honest nodes that terminate decide on the same value.

We implement the VABA protocol from [2] which has a communication cost of $O(|M|n^2 + n^2)$ and requires $O(1)$ rounds in expectation. However, to achieve this, their protocol requires a shared key with a threshold of $d = 2f+1$. With a lower threshold, i.e., $d < 2f+1$, their protocol's communication cost degrades to $O(|M|n^2 + n^3)$.

Therefore, [2] is unsuitable for a sub-cubic construction using a low threshold. We sketch an alternative VABA protocol in Appendix A. It is based on the common subset protocol of [5], has a communication cost of $O(n^2)$ but requires $O(\log n)$ rounds in expectation.

3.6 Other Primitives

Our APSS constructions also use well-known cryptographic building blocks, i.e., threshold signatures and non-interactive zero-knowledge proofs for the equality of discrete logarithms. For completeness, we describe them in Appendix B and Appendix C, respectively.

4 Random Polynomial Generation

As we outlined in §1, the core building of our proactive protocols is a random polynomial generation protocol. Specifically, nodes jointly generate a random polynomial $p(\cdot)$ of degree d such that $p(0) = 0$ (i.e., a zero polynomial) and each node i receives the point $p(i)$. A secure protocol for generating zero polynomial immediately implies APSS as each node i

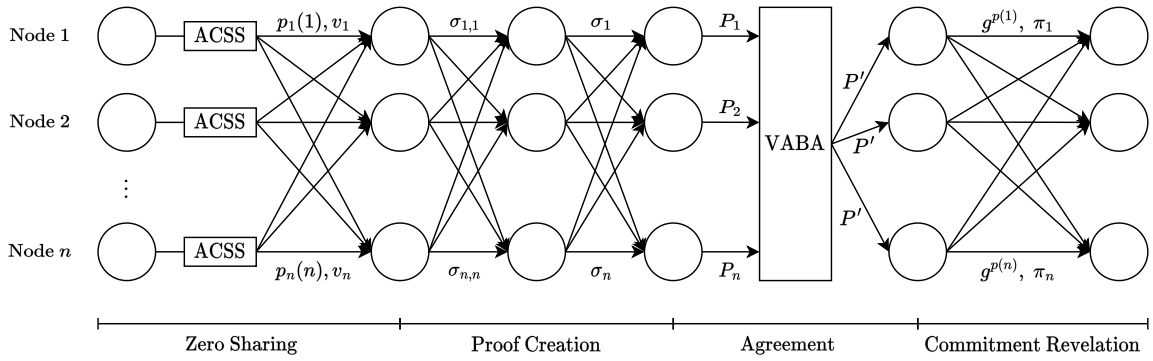

Figure 1: Execution of GenZeroPoly. Not all arrows are labelled for the sake of readability.

can re-randomize its share of secret s by adding $p(i)$ to it. Naturally, the security of the APSS protocol hinges on the security of the polynomial generation protocol.

However, some prior works [22, 34] on generating random zero polynomials are *insecure*. In particular, these works allow an adversary to bias the distribution of the new shares of honest nodes. Although such a bias does not pose an immediate threat, it might accumulate over time as an APSS is usually executed often (e.g., every day).

4.1 Random Zero Polynomial Generation

Definition 4 (Zero Polynomial Generation)**.** A zero polynomial generation protocol GenZeroPoly(d) allows nodes to jointly generate a uniformly random degree-d polynomial

$$p(x) = 0 + a_1 x + \cdots + a_d x^d$$

such that upon termination, each node i outputs $p(i)$ and the commitment vector $\mathbf{w} = [g^{p(1)}, g^{p(2)}, \ldots, g^{p(n)}]$.

A zero polynomial generation protocol needs to satisfy the following properties except for negligible probability:

- *Termination:* If all honest nodes start GenZeroPoly(d), then each honest node i will eventually output $p(i)$ and the commitment $\mathbf{w}$.
- *Completeness:* If all honest nodes start GenZeroPoly(d) and any honest node terminates, then there exists a degree-d polynomial $p(\cdot)$ with $p(0) = 0$ and each honest node i will eventually output a share $p(i)$ and commitment $\mathbf{w}$ as described above.
- *Secrecy:* A non-uniform PPT adversary that corrupts up to f nodes learns no information about the share of an honest node except whatever is revealed by the commitment $\mathbf{w}$.

4.2 Insecurity of Prior Work

The vulnerability in prior works bears a striking similarity with the vulnerability of distributed key generation protocols identified by Gennaro et al. [20]. To see this, we will briefly review the general approach for generating random zero-polynomials.

1. Each node proposes a random zero polynomial.
2. Nodes run a consensus protocol to agree on a subset Φ consisting of at least $f+1$ of proposals.
3. The random zero-polynomial $p(\cdot)$ is the sum of all the zero polynomials included in the set Φ.

Although this approach seems secure as Φ includes a polynomial generated by at least one honest node, this is not the case. We will highlight this using COBRA [34], a recently published APSS construction.

In COBRA, each node proposes a polynomial by secret sharing 0, encrypting every share and sending the vector of encrypted shares and a *commitment to the polynomial* to all nodes. Then, using a leader-based consensus protocol, $f+1$ of these proposals are chosen and summed up. The commitment is the problematic part, especially if it is instantiated with Feldman commitment, which is one possibility suggested in the COBRA paper.

The issue is that the adversary $\mathcal{A}$ sees all commitments before consensus on Φ is reached. This enables $\mathcal{A}$ to choose any arbitrary set of $f+1$ polynomials from the set of all polynomials such that the aggregated commitment $g^{p(i)}$ for an honest node i always ends with the bit 0. Now if $\mathcal{A}$ is the leader it can simply force this set to be selected. Since $g^{p(i)}$ is biased and always ends with bit 0, the share $p(i)$ is also biased. Intuitively, if the protocol would satisfy *Secrecy*, this event should only occur with probability $1/2$.

Getting ahead of ourselves, our protocol has an additional round of interaction to ensure that honest nodes receive uniformly random shares. More specifically, it is the *Commitment Revelation* phase in §5.

5 Polynomial Generation Design

We present two protocols that implements the GenZeroPoly primitive. The first protocol has a communication cost of $O(n^3)$ and the second one improves the communication cost to

Algorithm 1 GenZeroPoly$_i(d)$

Requires: Set-up secret sharing, i.e., node holds partial secret key sk_i and public keys pk as well as pk_j, $\forall j \in [n]$
Input: Polynomial degree d
Output: Polynomial share $p(i)$ and commitment $\mathbf{w}$
Local Variables: $\mathbf{V} = []; \mathbf{S} = []; \Sigma = \{\}; \Phi_i = \{\}; \mathbf{H} = []$

Join the VABA instance with predicate $Q(\mathsf{pk}, \cdot)$

// Zero Sharing
$\mathsf{ACSS.share}_i^h(0, d)$

// Proof Creation
upon Termination of ACSS from node j with share $s_{j,i}$ and Feldman commitment $\mathbf{v}_j = [h^{a_{0,j}}, \ldots, h^{a_{d,j}}]$ **do**
  if $\mathbf{v}_j[0] = h^0$ **then** *// Ensure that it's a zero polynomial*
    $\mathbf{S}[j] = s_{j,i}$; $\mathbf{V}[j] = \mathbf{v}_j$
    $\sigma_{i,j} = \mathsf{Sig.psign}(\langle \textsc{ShareOk}, j \rangle, \mathsf{sk}_i)$
    send $\langle \textsc{ShareOk}, \sigma_{i,j} \rangle$ **to** j
upon receiving $\langle \textsc{ShareOk}, \sigma_{j,i} \rangle$ from j **do**
  if $\mathsf{Sig.pverify}(\langle \textsc{ShareOk}, j \rangle, \sigma_{j,i}, \mathsf{pk}_j) = \top$ **then**
    $\Sigma = \Sigma \cup \{\sigma_{j,i}\}$
wait for $|\Sigma| \geq d + 1$ **and do**
  $\sigma_i = \mathsf{Sig.sign}(\Sigma)$
  $\Phi_i = \Phi_i \cup \{\langle i, \sigma_i \rangle\}$
  send $\langle \textsc{SharePoof}, \sigma_i \rangle$ **to all**
upon receiving $\langle \textsc{SharePoof}, \sigma_j \rangle$ from j **do**
  if $\mathsf{Sig.verify}(\langle \textsc{ShareOk}, j \rangle, \sigma_j, \mathsf{pk})$ $= \top$ **then**
    $\Phi_i = \Phi_i \cup \{\langle j, \sigma_j \rangle\}$

// Agreement
wait for $|\Phi_i| \geq f + 1$ **and do**
  Propose Φ_i to the VABA instance

// Commitment Revelation
upon Delivery of Φ from VABA and $\mathsf{idx}(\Phi) \subseteq \mathsf{idx}(\mathbf{S})$ **do**
  For all i, set $\mathbf{v}'[i] = \prod_{j \in \mathsf{idx}(\Phi)} \mathbf{V}[j][i]$
  $p(i) = \sum_{k \in \mathsf{idx}(\Phi)} \mathbf{S}[k]$
  $\pi_i \leftarrow \mathsf{DLEq.prove}(g^{p(i)}, h^{p(i)}, p(i))$
  send $\langle \textsc{Reveal}, g^{p(i)}, \pi_i \rangle$ **to all**
  upon receiving $\langle \textsc{Reveal}, g^{p'(j)}, \pi_j \rangle$ from j **do**
    Compute $h^{p(j)}$ by using $\mathbf{v}'$
    if $\mathsf{DLEq.verify}(g^{p'(j)}, h^{p(j)}, \pi_j) = \top$ **then**
      $\mathbf{H}[j] = g^{p'(j)}$
    wait for $|\mathbf{H}| > d$ **and do**
      Compute $\mathbf{w} = [g^{p(0)}, \ldots, g^{p(n)}]$ by using $\mathbf{H}$.
      output $p(i)$ and $\mathbf{w}$

$O(cn^2)$. This is at the cost of an additional failure probability of $e^{-\frac{1}{18}c}$. Here $c < n$ is a system parameter which can be tuned to achieve a trade-off between security and performance.

5.1 Cubic Polynomial Generation

Our cubic polynomial generation protocol follows the high-level idea where each node shares a zero polynomial and then nodes agree on $f + 1$ of them. The resulting zero polynomial is the sum of these $f + 1$ polynomials. Our protocol supports any degree d such that $d \in [f + 1, n - f - 1]$ and assumes that nodes already start with a $(n, d + 1)$ Shamir secret sharing of a secret. Also, every node knows the threshold public keys of every other node (cf. §3.3). We illustrate an overview of our protocol in Figure 1 and summarize it in Algorithm 1.

Algorithm 2 VABA Predicate for GenZeroPoly

function $Q(\mathsf{pk}, \Phi_i)$
  if $|\Phi| < f + 1$ **then**
    return $\perp$
  for $\langle i, \sigma_i \rangle \in \Phi_i$ **do**
    if $\mathsf{verify}(\langle \textsc{ShareOk}, i \rangle, \sigma_i, \mathsf{pk}) = \perp$ **then**
      return $\perp$
  return $\top$

5.1.1 Design

Our protocol has four phases: *Zero Sharing*, *Proof Creation*, *Agreement*, and *Commitment Revelation*.

Zero Sharing. During Zero Sharing phase, each node i samples a random polynomial $p_i(\cdot)$ with $p_i(0) = 0$. Node i then shares $p_i(\cdot)$ using $\mathsf{share}_i^h(0, d)$ (cf. §3.4), where $h \neq g$ is a uniformly random generator of $\mathbb{G}$. Recall from Definition 2, during $\mathsf{share}_i^h(0, d)$, every node $j \in [n]$ will receive its share $p_i(j)$ and $\mathbf{v}_i$, the Feldman commitment to $p_i(\cdot)$. To ensure that $p_i(\cdot)$ is a zero polynomial, every other node checks whether $\mathbf{v}_i[0] = h^0 = 1$.

Proof Creation. Once $\mathsf{share}_k^h(0, d)$ for any $k \in [n]$ terminates at node i, it sends the message $\langle \textsc{ShareOk}, \sigma_{i,k} \rangle$ to node k. Here, $\sigma_{i,k}$ is a partial signature on the message $\langle \textsc{ShareOk}, k \rangle$.

Each node then waits for $d + 1$ partially signed ShareOk messages. Upon receiving $d + 1$ valid ShareOk messages, it combines them to create a threshold signature σ_i on the message $\langle \textsc{ShareOk}, i \rangle$. Node i then multicasts the message $\langle \textsc{SharePoof}, \sigma_i \rangle$ to all other nodes.

Agreement. Every node i keeps track of a set Φ_i and when it receives a valid SharePoof message from node j, it adds the tuple $\langle j, \sigma_j \rangle$ Φ_i. When $|\Phi_i| > f$, it proposes Φ_i to the running validated asynchronous Byzantine agreement (VABA) instance. Φ_i indirectly represents the set of $f + 1$ polynomials that node i wants to sum to the zero polynomial $p(\cdot)$.

During the VABA protocol, nodes use the validity predicate $Q(\cdot)$ specified in Algorithm 2. Intuitively, this validity check ensures that at least one honest node has received a share of all the proposed polynomials.

Commitment Revelation. Let Φ be the output of the VABA protocol. Let $p(\cdot)$ be the polynomial defined as the sum of the polynomials corresponding to Φ. Once all the ACSS instances in Φ terminate at node i, it computes its share, $p(i)$, by adding its share of the polynomials in Φ.

Algorithm 3 Sub-cubic version of Algorithm 1

Requires: Set-up secret sharing, i.e., node holds partial secret key sk_i and public keys pk as well as pk_j, $\forall j \in [n]$
Input: Polynomial degree d
Output: Polynomial share $p(i)$ and commitment $\mathbf{w}$
Local Variables: $\mathbf{V} = []; \mathbf{S} = []; \Sigma = \{\}; \Phi = \{\}; \mathbf{H} = []; C = \{\}$

// ...
// *Zero Sharing*
1: Collaboratively sign $\mathbf{msg_{com}}$ producing $\sigma_{\mathbf{com}}$
2: Sample $C \leftarrow [n]$ using $\sigma_{\mathbf{com}}$
3: **if** $i \in C$ **then**
4: $\quad \mathsf{share}_i^h(0,d)$

// *Proof Creation*
5: **upon** Termination of ACSS from node $j \in C$ with share $s_{j,i}$ and commitment $\mathbf{v}_j$ **do**
// ...

// *Agreement*
6: **wait for** $|\Phi_i| \geq \lfloor c/2 \rfloor + 1$ **and do**
7: $\quad$ Propose Φ_i to the VABA instance
// ...

Nodes compute the commitment $\mathbf{w} = [g^{p(1)}, \ldots, g^{p(n)}]$ using an additional round of interaction. Each node i locally computes $g^{p(i)}$ and a non-interactive zero-knowledge proof π_i that proves $\log_g g^{p(i)} = \log_h h^{p(i)}$. Then, it multicasts $\langle \textsc{Reveal}, g^{p(i)}, \pi_i \rangle$ to others. Upon receiving such a $\langle \textsc{Reveal}, g^{p(k)}, \pi_k \rangle$ from node k, node i locally computes $h^{p(k)}$ using the Feldman commitments from the Zero Sharing phase and verifies $g^{p(k)}$ using π_k and $h^{p(k)}$. Upon receiving $d+1$ valid Reveal messages, node i computes all the missing $g^{p(k)}$ by interpolating in the exponent.

5.1.2 Analysis

We will prove in §8 that Algorithm 4 securely implements a protocol for generating random zero-polynomial, as per Definition 4. We will also demonstrate the Algorithm 4 has a communication cost of $O(n^3)$.

Observation 1. Modifying this construction to support any arbitrary fixed value $p(0)$ or even a uniformly random one is straight-forward.

Observation 2. We stated Algorithm 1 in a way that is suitable for APSS. It generates a polynomial with degree equal to the degree of the threshold key. However, for other applications, it might be desirable to generate a polynomial of a different degree. This is indeed possible, i.e., a threshold key of degree $d \geq f$ can generate a zero polynomial of degree $d' > f$.

5.2 Sub-cubic Polynomial Generation

While the $O(n^3)$ communication complexity of Algorithm 1 matches existing solutions, reducing the cost further would be desirable — especially for large-scale deployments. Algorithm 1 is cubic because of the Zero Sharing phase, where nodes execute n parallel ACSS, one for sharing secret of zero-polynomial chosen by each node. Since each ACSS instance has a communication complexity of $O(n^2)$, the total communication cost of Zero Sharing phase is $O(n^3)$.

To break the $O(n^3)$ barrier and achieve a sub-cubic communication complexity, we allow for some additional, yet configurable, failure probability. The core idea is that only nodes in a small, randomly sampled committee [6, 14, 21] act as a dealer in the Zero Sharing phase. All other nodes only act as receivers.

More concretely, let $c < n$ be the size of the committee. Then, our protocol has a sub-cubic communication complexity of $O(cn^2)$ and it is secure as long as an absolute majority of committee members is honest, i.e., $\lfloor c/2 \rfloor + 1$. Thus, it incurs an additional failure probability of at most $e^{-c/18}$ as we will prove in Lemma 5. The value of c can be varied to achieve a trade-off between the failure probability and performance.

5.2.1 Design

The sub-cubic protocol works similarly to the cubic one described in Section 5.1 with only two phases differing slightly. We describe the necessary changes in these phases below and state the changes to the Algorithm 1 in Algorithm 3.

Zero Sharing. First, nodes decide on the committee. To this end, they collaboratively produce a signature on a predetermined message $\mathbf{msg_{com}}$. Using the signature's randomness, a committee of c nodes is randomly sampled from $[n]$ without replacement. Then, only these nodes share a zero polynomial.

Proof Creation & Agreement. Here, two natural changes are necessary. First, nodes only accept zero polynomials from committee members. Second, the size of Φ_i needs to be changed from $f+1$ to $\lfloor c/2 \rfloor + 1$. This is because at most c nodes will execute the Zero Sharing phase.

5.2.2 Analysis

We analyze the performance and security of Algorithm 3 in §8.2. In short, it has a communication complexity of $O(cn^2)$ and is secure as long as the majority of the committee members are honest. Per Lemma 5, the probability of this not being the case is bounded by $e^{-c/18}$.

Suppose that there is no honest majority in the committee. Let us consider in what ways the adversary $\mathcal{A}$ can break the protocol. It can violate *Termination* by not performing the Zero Sharing phase and thereby hindering any progress. Further, $\mathcal{A}$ can break *Secrecy* since it might know all $\lfloor c/2 \rfloor + 1$

chosen polynomials. However, it cannot violate *Completeness*. So if an honest node terminates, all honest nodes will eventually receive a zero polynomial.

Looking ahead, when we use the GenZeroPoly in an APSS protocol, such a failure is not catastrophic as it does not corrupt or directly leak the secret s. However, if the adversary hinders the protocol from making progress, getting the protocol unstuck might require manual intervention.

6 Asynchronous Proactive Secret Sharing

We now use the random zero polynomial generation protocols from §5 to implement an asynchronous proactive secret sharing (APSS) protocol. In particular, nodes jointly generate a random zero polynomial $p(\cdot)$. Each node i then adds $p(i)$ to its share of the secret key sk_i. Let $\widetilde{\mathsf{sk}}_i$ be the new share of node i, i.e., $\widetilde{\mathsf{sk}}_i = \mathsf{sk}_i + p(i)$. This corresponds to adding the polynomial $p(\cdot)$ to the polynomial implicitly defined by the existing secret sharing. Clearly, the secret key sk stays the same as $\mathsf{sk} + p(0) = \mathsf{sk} + 0 = \mathsf{sk}$.

One important subtlety one needs to consider while using GenZeroPoly for APSS protocol is the following. Consider an honest node i that just computed its new share $\widetilde{\mathsf{sk}}_i$. Node i needs to delete its old share sk_i; otherwise, the adversary $\mathcal{A}$ could corrupt it in the future and learn sk_i. However, once node i deletes sk_i, it can no longer participate in the VABA protocol used in the GenZeroPoly protocol. In the worst case, if a large fraction of honest nodes stop participating in the VABA protocol, as they delete their old share, the GenZeroPoly protocol might fail to terminate at remaining honest nodes.

To ensure this does not happen, GenZeroPoly must fulfill a stronger form of termination [11]: A node must be able to stop participating without hindering the progress of other nodes. We say *gracefully exiting* the protocol.

Algorithms 1 and 3 are strongly terminating if the ACSS and VABA sub-protocols are. Concretely, in case of our prototype, HAVEN [4] is strongly terminating by default, and [2] only requires sending an additional message at the end. Similarly, [16] and Appendix A can be modified to be so as well.

The properties of the APSS scheme follow directly from GenZeroPoly, which yields the following Theorem:

Theorem 1 (APSS)**.** *Algorithm 4 fulfills Definition 1. It inherits its communication- and round complexity from the* GenZeroPoly *construction that is being used.*

7 Asynchronous Refreshable Secret Sharing

By definition, proactive secret sharing requires that the long-term secret stays the same. However, for many applications (e.g., shared randomness) this is both unnecessary, as they do not require the same long-term key to function, and perilous, as a momentary compromise of the long-term key leads to system-wide compromise ad infinitum.

Algorithm 4 APSS

Requires: Set-up secret sharing, i.e., node holds partial secret key sk_i and public keys pk as well as $\{\mathsf{pk}_j\}_{j\in[n]}$.

Output: Re-randomized secret sharing, i.e., new secret key $\widetilde{\mathsf{sk}}_i$ and updated public keys $\left\{\widetilde{\mathsf{pk}}_j\right\}_{j\in[n]}$.

1: $(p(i), \mathbf{w}) \leftarrow \mathsf{GenZeroPoly}_i(d)$
2: $\widetilde{\mathsf{sk}}_i = \mathsf{sk}_i + p(i)$.
3: **for** $j \in [n]$ **do**
4: $\quad \widetilde{\mathsf{pk}}_j = \mathsf{pk}_j \cdot \mathbf{w}[j]$
5: Gracefully exit $\mathsf{GenZeroPoly}_i(d)$ and delete sk_i.
6: **output** $\widetilde{\mathsf{sk}}_i$ and $\left\{\widetilde{\mathsf{pk}}_j\right\}_{j\in[n]}$

To avoid this danger, protocols can generate a new key from scratch by running an ADKG every epoch, but this does not take advantage of the fact that there is already a threshold key. To provide a simpler and more efficient solution, we instead propose a natural relaxation of APSS, which we call *Asynchronous Refreshable Secret Sharing (ARSS)*. Such a protocol re-randomizes the shares *and the secret itself*.

The cubic and the sub-cubic APSS construction can be easily modified into an ARSS. Specifically, during the Zero-sharing Phase, nodes share a uniformly random polynomial instead of sharing a polynomial with the constant term being 0. We do not provide an extensive algorithm due to space constraints.

ARSS has an additional benefit over APSS; we can prove it secure for even $d = f+1$, whereas APSS requires $d > f+1$. Intuitively, this is because, in APSS, the adversary always knows that $p(0) = 0$, whereas $p(0)$ in ARSS is uniformly random.

8 Analysis of Zero Polynomial Generation

8.1 Cubic Polynomial Generation

We will now prove that Algorithm 1 realizes the GenZeroPoly functionality, by proving about each property of Definition 4.

Lemma 1 (Termination)**.** *Algorithm 1 fulfills Termination as in Definition 4.*

Proof. By assumption, $n-f$ honest nodes act as a dealer and share 0 using an ACSS. Therefore, by ACSS *Termination*, all honest nodes will eventually receive their share of these $n-f$ zero polynomials. Hence, all honest nodes will reply to each honest dealer with a SHAREOK message containing a valid threshold signature. Thus, every honest dealer will eventually know $n-f > d$ valid partial signatures which it combines to create a proof, i.e, a threshold signature. Since there are $n-f$ honest dealers, each honest node will eventually receive at least the required $f+1$ valid proofs which it will input to the VABA protocol. Thus, all honest nodes eventually start the

VABA with a valid value and, by VABA *Termination*, will agree on some valid value.

Since the value is valid according to the predicate Q and by *Unforgeability* of the signature scheme, all chosen ACSS instances have terminated for at least $d+1-f \geq 1$ honest node(s). Consequently, by ACSS *Completeness* and the check in Line 4, every honest will eventually hold a share and commitment in base h for all chosen zero polynomials.

Lastly, each honest node i of which $n-f > d$ share a valid commitment $g^{p(i)}$. Thus, every node can compute $\mathbf{w}$ by interpolating the exponent. □

Lemma 2 (Completeness)**.** *Algorithm 1 fulfills Completeness as in Definition 4.*

Proof. All honest nodes agree on the same zero polynomial $p(\cdot)$ and its corresponding Feldman commitment $\mathbf{v}$ in base h. By VABA *Validity* and *Agreement*, all honest nodes decide on the vector of proofs. Each proof attests, by *Unforgeability* of the signature scheme and by the validity predicate Q, that the corresponding ACSS has terminated in at least one honest node. Therefore, by ACSS *Completeness*, all honest nodes will eventually receive their corresponding shares and Feldman commitments to the agreed on polynomials. Finally, by the check in Line 4, all chosen polynomials must be zero polynomials. Therefore, the sum of the chosen polynomials is a zero polynomial.

Furthermore, all nodes also agree on $\mathbf{w}$. This follows directly the *Completeness* and *Soundness* of DLEq. □

Lemma 3 (Secrecy)**.** *In the Random Oracle Model, under the Decisional Diffie-Hellman assumption, Algorithm 1 fulfills Secrecy as in Definition 4.*

Our proof of Lemma 3 uses the following observation:

Observation 3. A polynomial of degree d is uniquely defined given any set $d+1$ unique coefficients and evaluation points. In particular, given any set containing a total $d+1$ coefficients and evaluation points, one can compute all the coefficients by solving the system of linear equations given in equation 3

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_i & x_i^2 & \cdots & x_i^d \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{pmatrix} = \begin{pmatrix} p(x_1) \\ p(x_2) \\ \vdots \\ p(x_i) \end{pmatrix} \tag{3}$$

We now prove a claim that we will use to prove Lemma 3.

Claim 1. *In the Random Oracle Model, under the Decisional Diffie-Hellman assumption, $d-f$ coefficients of the polynomial generated by Algorithm 1 are distributed uniformly in $\mathbb{Z}_q$. Furthermore, these coefficients are unknown to the adversary apart from what is revealed by* $\mathbf{w}$.

We prove Claim 1 by simulation, i.e., we describe a simulator, $\mathcal{S}$, on input the generators g,h and a set of $d-f$ commitments to coefficients $M = \{(i, g^{a'_i})\}$, interacts with an adversary $\mathcal{A}$, that corrupts up to f nodes. The simulator produces a view such that the Feldman commitment $\mathbf{v}$ to $p(\cdot)$ fulfills $\mathbf{v}[i] = g^{a'_i}$ for every $i \in M$. Also, the view of the $\mathcal{A}$ in the simulated protocol is indistinguishable from the view of the $\mathcal{A}$ in the real protocol.

Description of the Simulator. Let $\mathcal{C}$ be the set of corrupt nodes where, without loss of generality, $|\mathcal{C}| = f$ and let $\mathcal{H} = [n] \setminus \mathcal{C}$ be the set of honest nodes. $\mathcal{S}$ first simulates all honest nodes according to Algorithm 1 until all nodes agree on the set of chosen polynomials Φ (i.e., until Line 21). Since $|\mathcal{C}| = f < d$, Φ contains at least one zero polynomial of an honest node. Thus, $\mathcal{A}$ does not know the $p(\cdot)$, which is the sum of these polynomials, in its entirety.

Note that $\mathcal{S}$ can compute all agreed upon polynomials $p_i(\cdot)$, $i \in \mathsf{idx}(\Phi)$. If $i \in \mathcal{H}$, then $\mathcal{S}$ sampled the polynomial itself and knows it. Otherwise, if $i \in \mathcal{C}$, *Completeness* of the ACSS ensures that $\mathcal{S}$ learns enough shares to get the polynomial by interpolation. Therefore, $\mathcal{S}$ can also compute the sum $p(\cdot) = \sum_{i \in \mathsf{idx}(\Phi)} p_i(\cdot)$. Clearly, with overwhelming probability, $p(\cdot)$ will not contain the desired coefficients a'_i at the appropriate positions. Hence, $\mathcal{S}$ needs to diverge from the protocol at this point.

$\mathcal{A}$ only learns further information about $p(\cdot)$ in the form of the commitments sent using REVEAL messages. Note that it cannot interpolate the coefficients of $p(\cdot)$ directly as it only knows (accounting for the fixed $p(0) = 0$) $f+1 < d+1$ points. Therefore, $\mathcal{S}$ still has some wiggle-room to find a "fake" polynomial $\bar{p}(\cdot)$ such that (i) $\bar{p}(0) = 0$, (ii) $\bar{p}(i) = p(i)$ for $i \in \mathcal{C}$ and (iii) $\bar{p}$ contains the coefficients a'_i at the appropriate places. In other words, $\bar{p}(\cdot)$ must be consistent with what the adversary has seen so far *and* include the coefficients at the appropriate places. Since the constraints (i)-(iii) encompass $f+1$ points and $d-f$ coefficients, Observation 3 tells us that there exists a unique degree-d polynomial satisfying them. As $\mathcal{S}$ is only given the commitments to the coefficients, it cannot compute $\bar{p}(\cdot)$ itself but only $g^{\bar{p}(\cdot)}$ which is sufficient for the simulation.

Last, $\mathcal{S}$ must simulate the *Commitment Revelation* phase of the protocol. To this end, it computes the commitments $g^{\bar{p}(i)}$ to the honest nodes' "fake" share and simulates the corresponding proof π_i.

Proof. We reduce $\mathcal{A}$ distinguishing between (i) a real execution Algorithm 1 and (ii) interacting with $\mathcal{S}$ to the hardness of the DDH problem. Let $\mathcal{A}_{\text{DDH}}$ be a DDH distinguisher that uses $\mathcal{A}$'s guess (cf. Figure 3)

$\mathcal{A}_{\text{DDH}}$ receives a generator h and DDH challenge $(h^\alpha, h^\beta, h^\gamma)$ where $\gamma = \alpha\beta$ or $\gamma \leftarrow \mathbb{Z}_q$ as inputs. It embeds the challenge into every random polynomial sampled by an honest node. More specifically, in Step 3, for every $p_i(\cdot)$, $i \in \mathcal{H}$ it sets $d-f$ coefficients to a random multiple of α (i.e., $h^{\alpha a'_{i,j}}$

Input: Generators g, h and set of coefficients $\left\{(i, g^{a'_i})\right\}$.

1. Simulate each node in $\mathcal{H}$ according to the protocol until Line 21.
2. Compute all polynomials $p_i(\cdot) \in \Phi$. Trivially, if $i \in \mathcal{H}$, $p_i(\cdot)$ is already known. Otherwise, if $i \in \mathcal{C}$, $\mathcal{S}$ knows at least $d'+1$ secret shares $p_i(j), j \in \mathcal{H}$ which it can interpolate to yield $p_i(\cdot)$.
3. Let $p(\cdot) = \sum_{i \in \mathsf{id} \times \Phi} p_i(\cdot)$ and compute all points $p(i)$, $i \in \mathcal{C}$ known to the adversary.
4. Solve the system of linear equations specified by conditions (i)-(iii) to get $g^{\bar{p}(\cdot)}$ and use this to compute $g^{\bar{p}(i)}$ for $i \in \mathcal{H}$.
5. For each honest node $i \in \mathcal{H}$, simulate the proof π_i for the statement $\log_h h^{p(i)} = \log_g g^{\bar{p}(i)}$.
6. Send $\langle \textsc{Reveal}, g^{\bar{p}(i)}, \pi_i \rangle$ to $j \in \mathcal{C}$.

Figure 2: Simulator $\mathcal{S}$ for Algorithm 1.

while defining the generator $g = h^\beta$. Then, for each corrupt node $j \in \mathcal{C}$, it randomly samples a point $p_i(j) \leftarrow \mathbb{Z}_q$, which it sends to node j during the zero sharing phase.

Then $\mathcal{A}_{\text{DDH}}$ runs the protocol until agreement on the set of polynomials is reached. Let $\Phi \subseteq [n]$ be the set nodes whose polynomial has been chosen. Note that the ACSS sub-protocol must simulatable given only the points $p_i(j)$, $j \in \mathcal{C}$ and $\mathbf{v}_i$. This is the case for the constructions mentioned so far.

Then, $\mathcal{A}_{\text{DDH}}$ computes the Feldman commitment $\mathbf{v}$ to the polynomial $p(\cdot) = \sum_{i \in \Phi} p_i(\cdot)$ as described in Step 5. Crucially, as part of this, it calculates $h^{\gamma a'_{i,j}}$. Depending on the DDH challenge γ, there are two possibilities:

- $\gamma = \alpha\beta$: Then $h^{\gamma a'_{i,j}} = g^{\alpha a'_{i,j}}$ which corresponds to the correct $\mathbf{v}$ being commitment to $p(\cdot)$ in base g. Therefore, $\mathbf{v}$ is as in a real execution of the protocol.
- $\gamma \leftarrow \mathbb{Z}_q$: Then, with non-negligible probability, $\gamma \neq \alpha\beta$ and it follows that the commitments in base h and g are independent of each other. Hence, $\mathbf{v}$ is distributed identically to one produced by $\mathcal{S}$ on input of randomly distributed coefficients.

Let b be $\mathcal{A}$'s guess on whether it thinks this is a real execution or not. $\mathcal{A}_{\text{DDH}}$ outputs b.

It follows that the reduction $\mathcal{A}_{\text{DDH}}$ correctly answers the DDH challenge if the adversary correctly distinguishes a real from a simulated protocol execution except for some negligible loss in tightness (e.g., when programming the random oracle fails or the simulated execution happens to coincide with the real one).

□

Proof of Lemma 3. Assume that the adversary knows the share of an honest node. Hence, it knows $f+2$ distinct points of the polynomial. Therefore, by Observation 3, at most $d-f-1$ coefficients may be chosen arbitrarily. However, this contradicts Claim 1 as $\mathcal{A}$ would be able to distinguish a simulated view from a real one. □

Input: Generator h and DDH instance $\left(h^\alpha, h^\beta, h^\gamma\right)$ where $\gamma = \alpha\beta$ or $\gamma \leftarrow \mathbb{Z}_q$
Output: Guess b with $b = \top$ indicating that $\gamma = \alpha\beta$.

1. Define generator $g = h^\beta$.
2. Choose a random subset $J \subset [d]$ with $|J| = d - f$.
3. For each $i \in \mathcal{H}$, sample the ACSS polynomial $p_i(\cdot)$ with Feldman commitment $\mathbf{v}_i$:
   (a) Sample $a'_{i,j} \leftarrow \mathbb{Z}_q$, $j \in J$ and compute $(h^\alpha)^{a'_{i,j}}$ which implicitly sets the j-th coefficient to $a_{i,j} = \alpha a'_{i,j}$.
   (b) Sample a set of points $Y_i = \left\{(j, r_j) | j \in \mathcal{C}, r_j \leftarrow \mathbb{Z}_q\right\}$ for the corrupt nodes.
   (c) Compute the Feldman commitment $\mathbf{v}_i$ in base h to the polynomial $p_i(\cdot)$ that is defined by the coefficients $a_{i,j}$ and points $\{(0,0)\} \cup Y_i$.
4. Using Y_i and $\mathbf{v}_i$, perform the rest of the protocol up to Line 21. Let $\Phi \subseteq [n]$ be the set of nodes whose polynomial has been chosen.
5. Compute the Feldman commitment $\mathbf{v}$ to $p(\cdot)$ in base g.
   (a) For all $i \in \mathcal{C} \cap \Phi$, compute $p_i(\cdot)$ by interpolation and use it to convert the corresponding Feldman commitments $\mathbf{v}_i$ to base g.
   (b) Using $p_i(\cdot)$, $i \in \mathcal{C} \cap \Phi$ and Y_i, $i \in \mathcal{H} \cap \Phi$, compute $p(i)$ for all corrupt nodes $i \in \mathcal{C}$.
   (c) Compute $d-f$ commitments to coefficients a_j, $j \in J$ of $p(\cdot)$ by $\mathbf{v}[j] = \prod_{i \in \mathcal{H} \cap \Phi} (h^\gamma)^{a'_{i,j}} \prod_{i \in \mathcal{C} \cap \Phi} \mathbf{v}_i[j]$.
   (d) Compute the coordinates missing from $\mathbf{v}$ by solving the system of linear equations defined by the above commitments to coefficients and $p(i)$, $i \in \mathcal{C}$.
6. For each honest node $i \in \mathcal{H}$, perform the rest of the protocol.
   (a) Evaluate $\mathbf{v}$ at position i in the exponent yielding $g^{p(i)}$.
   (b) Simulate the NIZK proof π_i by programming the random oracle and then send the $\textsc{Reveal}$ message to the adversary.
7. Output $\top$ iff the adversary guesses that this is a real execution.

Figure 3: Reduction of distinguishing $\mathcal{S}$ from a real executions to the difficulty of DDH.

Lemma 4 (Performance)**.** *Algorithm 1 has a communication complexity of $O(n^3)$ and takes $O(1)$ rounds.*

Proof. Let C_{ACSS} and C_{VABA} be the expected communication costs of the ACSS and VABA protocols, respectively. Using constructions listed in Section 3, set $C_{\text{ACSS}} = O(n^2)$ and $C_{\text{VABA}} = O(n^3)$. The expected communication cost is then in $O(n \cdot C_{\text{ACSS}} + 2n^2 + C_{\text{VABA}})$ which is $O(n^3)$. Both

sub-protocols require expected $O(1)$ rounds and hence Algorithm 1 takes require expected $O(1)$ rounds. □

Combining all the above we get the following Theorem.

Theorem 2 (Zero Polynomial Generation)**.** *In the Random Oracle Model, under the Decisional Diffie-Hellman assumption, Algorithm 1 fulfills Definition 4 and has a communication complexity of $O(n^3)$ and takes $O(1)$ rounds.*

8.2 Sub-cubic Polynomial Generation

Since our sub-cubic protocol is very similar to the cubic one, the analysis is analogous, and only requires that sampled committee consists of an honest majority. Thus, in Lemma 5 we analyze this probability.

Lemma 5. *For a committee of size c implicitly sampled by* sample()*, a strict majority of the committee members is honest except with probability of at most $e^{-c/18}$.*

Proof. We are sampling c out of n nodes without replacement and more than $2/3$ of the nodes are honest which constitutes a hypergeometric distribution.

Let H be the random variable equal to the number of honest nodes in the committee. Then, using the well known tail bound [23], the probability that there is no absolute majority of honest nodes is

$$\Pr[H \leq c/2] = \Pr[H \leq (2/3 - 1/6)c] \tag{4}$$

$$\leq e^{-2c/6^2} = e^{-c/18} \tag{5}$$

□

As we illustrate in Figure 4, the upper bound for failure probability with $c = 64$ and $c = 128$ is approximately, 2.86% and 0.08%, respectively.

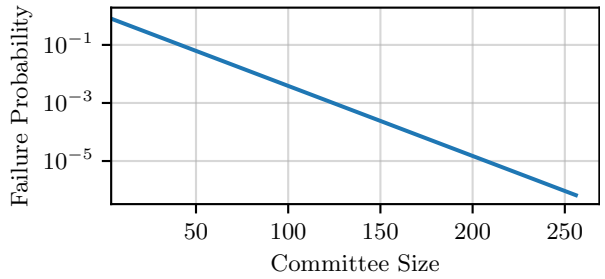

Figure 4: Relation between the failure probability and committee size (i.e., $e^{-c/18}$ as Lemma 5). Note logarithmic scaling.

Using Lemma 5 and Theorem 2, we get the following theorem.

Theorem 3. *Let c be the size of the committee. In the Random Oracle Model, under the Decisional Diffie-Hellman assumption, Algorithm 3 fulfills Definition 4 with a failure probability of at most $e^{-c/18}$. It has a communication complexity of $O(cn^2)$ and takes either $O(1)$ or $O(\log c)$ rounds depending on whether a high-threshold secret key is used.*

Proof. The protocol fails if there is no strict majority of honest nodes in the committee. From Lemma 5 this happens with probability of at most $e^{-c/18}$. Under this assumption the proof is analogous to the proof of Theorem 2.

The communication complexity of $O(cn^2)$ follows from the fact that ACSS is only executed c times.

Note that when the secret key is shared using a $(n, d+1)$ Shamir secret sharing scheme where $d+1 = 2f+1$, our protocol terminates in expected $O(1)$ rounds using the VABA protocol from [2]. With $d+1 < 2f+1$, the round complexity is $O(\log c)$, as we need to use the protocol from Section A. □

9 Implementation and Evaluation

We give a prototype implementation* of our APSS protocols, both cubic, and sub-cubic. It is based on the BLS12-381 curve, requires a one-time setup, tolerates $f < n/3$ corruptions, and supports secrets with a reconstruction threshold of $2f+1$. We use HAVEN [4] and [2] as sub-protocols for ACSS and VABA, respectively. Since the curve is pairing friendly, we chose to use BLS threshold signatures [8] and KZG polynomial commitments [25]. The latter needs a one-time setup that outputs $g^{\tau}, g^{\tau^2}, \ldots, g^{\tau^k}$ where $\tau \leftarrow \mathbb{Z}_q$ and k is the polynomial degree.

We utilize asynchronous Rust with `tokio` providing the asynchronous runtime and `blstrs` for BLS12-381 implementation. Nodes communicate via TCP sockets and a custom networking crate ensures a reliable exchange of messages by transparently handling retries and caching of messages. This networking crate is modular by design and self-contained. Thus, we hope that it might prove useful to other researchers wanting to implement an asynchronous protocol in Rust.

Through our experiments, we measure the *runtime* — the time a node takes to complete one invocation of the APSS protocol; and, the *bandwidth consumption* — the amount of data each node sends during the APSS protocol. As a baseline, we use the naive solution to APSS, i.e., generating a new secret by running an ADKG. Our experiments aim to demonstrate that (i) our deterministic construction (§5.1) outperforms the state-of-the-art ADKG constructions by virtue of using a simple black-box consensus instead of multiple binary agreements; and (ii) that our committee-based construction (§5.2) allows not only for lower runtime and bandwidth consumption but also for better scalability by varying the size of the committee.

We evaluate our protocol using Amazon Web Services and evenly distribute the nodes across eight global AWS regions: Northern California, Oregon, Ohio, Northern Virginia, Canada, Ireland, Singapore, and Tokyo.

*Available at https://github.com/ISTA-SPiDerS/apss

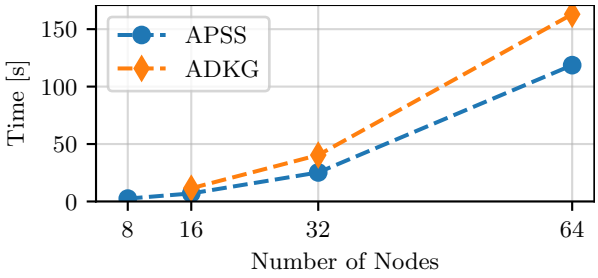

Figure 5: Average runtime of the APSS and ADKG protocols.

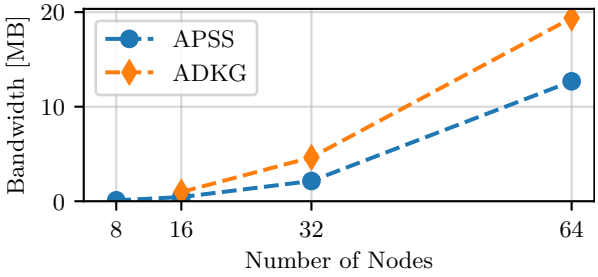

Figure 6: Average bandwidth consumption of one node during the APSS and ADKG protocols.

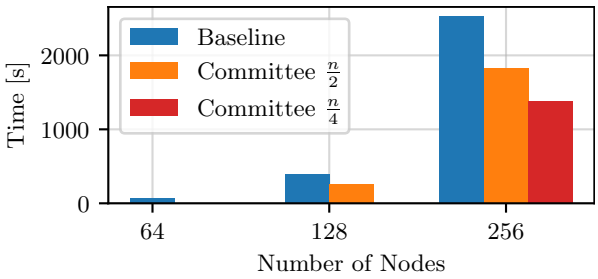

Figure 7: Average runtime of the APSS protocol for various committee sizes.

9.1 Evaluation of the Cubic Protocol

We evaluate our APSS protocol based on §4 with 8, 16, 32, and 64 nodes. Each node runs on a `t3a.medium` virtual machine (2 vCPUs, 4 GB RAM). Our results are averaged using ten runs of each experiment.

To demonstrate (i), we compare our prototype to a state-of-the-art ADKG protocol [16]. Their prototype also supports BLS12-381 and has been benchmarked on exactly the same AWS configuration.

We illustrate the average runtime and average bandwidth consumption of a node in Figure 5 and Figure 6, respectively. Even with 64 nodes, our prototype finishes in 2 minutes and consumes roughly 12.5 Megabytes (MB) of bandwidth. Compared to the ADKG, it is approximately 40 seconds (25% reduction) faster and saves 7 MB of bandwidth (35% reduction). Note that the runtime cannot be reliably compared with [16] as the prototypes are implemented in different programming languages. The bandwidth consumption, however, can be compared as (compressed) elliptic curve points, scalars, and

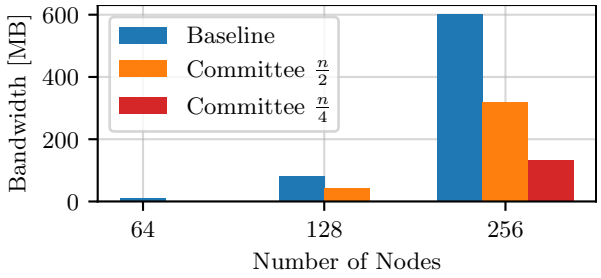

Figure 8: Average bandwidth consumption of one node during the APSS protocol for various committee sizes.

hashes make up the bulk of the communication. We believe that these improvements clearly showcase the practicality of our APSS which uses an efficient Byzantine agreement protocol leveraging the availability of a strong common coin.

9.2 Evaluation of the Sub-cubic Protocol

The previous experiments covered small to medium-sized networks. Our second set of experiments demonstrates that our sub-cubic protocol (cf. §5.2) supports large-scale deployments as mentioned in (ii). In these experiments, since nodes need to perform more computation, due to a large number of nodes, we run them on more powerful `t3a.xlarge` VMs (4 vCPUs, 16 GB RAM). Also, due to the longer running times, we repeated these experiments only three times.

To form a baseline, we measured the runtime of the cubic protocol with 64, 128, and 256 nodes. Then, we performed tests with committees of size 64 and 128.[†] The former degrades the protocol's security by at most ca. 2.86% and the latter by at most 0.08%. Recall that such a failure only hinders the re-randomization from taking place and it never directly leaks or corrupts the key

We present the runtime and bandwidth consumption in Figures 7 and 8, respectively. The baseline measurement shows that the cubic protocol with 256 nodes takes almost 40 minutes to complete and has a bandwidth consumption of 600 MB. Across all nodes, this adds up to 150 GB. A committee of size 128, which only introduces at most an additional failure probability of 0.08%, reduces this to 30 minutes and roughly 320 MB of communication on average. This sums up to 80 GB in total.

Since the ACSS is responsible for most of the prototype's communication, the sub-cubic protocol scales very well in terms of bandwidth. For $n = 64, 128, 256$ and $c = 64$, the prototype requires only $12.5, 43, 132$ MB of communication per node, respectively. For $n = 256$, this constitutes a savings of roughly 470 MB (79% reduction). The execution time behaves similarly though the effect is not as drastic. In the extreme case $n = 256$ and $c = 64$, the runtime is 19 minutes

[†]The committee was sampled through a VRF election and not a sortition without replacement, but this would have no effect on the performance of the system.

faster (45% reduction). In general, larger committees will also enjoy such scaling effects. Especially the resulting bandwidth savings make the sub-cubic protocol more practical for regular usage in large-scale networks.

10 Conclusion

We have presented two Asynchronous Proactive Secret Sharing protocols. Similarly to prior work, the first has a communication complexity of $O(n^3)$. The second improves upon the first and drives the communication complexity down to $O(cn^2)$ where $c < n$ is the size of a randomly sampled committee. Furthermore, we introduce Asynchronous Refreshable Secret Sharing and explain how our two constructions can be modified to fulfill this new notion.

We give rigorous security proofs and implement our protocols in Rust. Benchmarks of our prototype show that our protocols are practical for regular use. Especially the sub-cubic construction scales well even for large number of nodes.

Acknowledgments

Work of Sourav Das is supported by a Chainlink Labs Ph.D. fellowship.

References

[1] 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.

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

[3] Andreea B. Alexandru, Erica Blum, Jonathan Katz, and Julian Loss. State machine replication under changing network conditions. Cryptology ePrint Archive, Paper 2022/698, 2022. https://eprint.iacr.org/2022/698.

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

[5] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 183–192, 1994.

[6] Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. In *Theory of Cryptography Conference*, pages 353–380. Springer, 2020.

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

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

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

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

[11] 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.

[12] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201. IEEE, 2005.

[13] David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *Annual International Cryptology Conference*, pages 89–105. Springer, 1992.

[14] Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp. In *DISC*, 2020.

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

[16] 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.

[17] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438. IEEE, 1987.

[18] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[19] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *International Conference on Financial Cryptography and Data Security*. Springer, 2022.

[20] 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.

[21] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

[22] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *annual international cryptology conference*, pages 339–352. Springer, 1995.

[23] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[24] Bin Hu, Zongyang Zhang, Han Chen, You Zhou, Huazu Jiang, and Jianwei Liu. Dycaps: Asynchronous proactive secret sharing for dynamic committees. Cryptology ePrint Archive, Paper 2022/1169, 2022. https://eprint.iacr.org/2022/1169.

[25] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *International conference on the theory and application of cryptology and information security*, pages 177–194. Springer, 2010.

[26] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. Calypso: Private data management for decentralized ledgers. page 586–599. VLDB Endowment, dec 2020.

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

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

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

[30] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2016.

[31] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with t< n/3, o (n2) messages, and o (1) expected time. *Journal of the ACM (JACM)*, 62(4):1–21, 2015.

[32] David A Schultz, Barbara Liskov, and Moses Liskov. Mobile proactive secret sharing. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 458–458, 2008.

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

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

[35] Yunzhou Yan, Yu Xia, and Srinivas Devadas. Shanrang: Fully asynchronous proactive secret sharing with dynamic committees. *Cryptology ePrint Archive*, 2022.

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

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

[38] Thomas Yurek, Zhuolun Xiang, Yu Xia, and Andrew Miller. Long live the honey badger: Robust asynchronous dpss and its applications. Cryptology ePrint Archive, Paper 2022/971, 2022. https://eprint.iacr.org/2022/971.

[39] Haoqian Zhang, Louis-Henri Merino, Vero Estrada-Galinanes, and Bryan Ford. F3b: A low-latency commit-and-reveal architecture to mitigate blockchain front-running, 2022.

[40] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. Apss: proactive secret sharing in asynchronous systems. *ACM Transactions on Information System Security*, 8:259–286, August 2005.

A Low-Threshold Byzantine Agreement

As stated in Section 3.5, low-threshold, sub-cubic polynomial generation cannot use the VABA construction due to Abraham et al. [2] since it degrades to cubic communication complexity when used with a low-threshold key. An alternative construction that fulfills validated Byzantine Agreement as in Definition 3 can be realized by adapting Asynchronous Common Subset (ACS) protocols.

For example, consider the ACS due to Ben-Or et al. [5]. On a high level, each node in the committee performs a validated reliable broadcast (RBC) of the proof that it executed ACSS correctly. Additionally, for each RBC, an Asynchronous Binary Byzantine Agreement (ABBA) is run that signifies whether the RBC has terminated. Efficient RBC constructions (e.g.,n [15]) have a communication complexity of $O(n^2)$ in our setting. Furthermore, efficient low-threshold ABBA protocols (e.g., [31]) incur a cost of $O(n^2)$. Thus, the total expected communication cost of the protocol remains $O(cn^2)$.

B Threshold Signatures

A threshold signature scheme allows multiple nodes to jointly sign a message where the signing key is a shared secret. Let pk be the shared public key, sk_i and pk_i be the secret key share and corresponding public key of node i.

Definition 5. A threshold signature scheme Sig is defined by four algorithms psign, pverify, sign and verify.

- $\mathsf{psign}_i(m) \to \sigma_i$ is parameterized by the node i's secret key sk_i, takes a message m and outputs a partial signature σ_i.
- $\mathsf{pverify}(m, \sigma_i, \mathsf{pk}_i) = \top/\bot$ takes a message, a partial signature, and a node's public key pk_i and outputs whether the verification succeeded.
- $\mathsf{sign}(\Sigma) \to \sigma$ outputs a threshold signature σ if it is given $|\Sigma| \geq d+1$ correct partial signatures for the same message.
- $\mathsf{verify}(m, \sigma, \mathsf{pk}) = \top/\bot$ takes a message, threshold signature, and public key and outputs whether the verification succeeded.

We require that the following properties hold nodes except for negligible probability:

- *Correctness:* For every message m and node i it must hold that $\mathsf{pverify}(m, \mathsf{psign}(m, \mathsf{sk}_i), \mathsf{pk}_i) = \top$. Furthermore, given a set Σ with size $|\Sigma| \geq d+1$ of distinct such partial signatures, it must hold that $\mathsf{verify}(m, \mathsf{sign}(\Sigma), \mathsf{pk}) = \top$.
- *Unforgeability:* psign must be *existentially unforgeable under a chosen message attack (EUF-CMA)*. Furthermore, without knowledge of $d+1$ partial signatures, sign must be EUF-CMA secure as well.

For certain applications, such as creating shared randomness, it is useful for the signatures to be unique.

- *Uniqueness:* If $\mathsf{verify}(m, \sigma, \mathsf{pk}) = \top$ and $\mathsf{verify}(m, \sigma', \mathsf{pk}) = \top$, then $\sigma = \sigma'$.

Given a shared secret, standard discrete logarithm-based signature schemes such as BLS [8] can be turned into threshold signature schemes [7].

C Equality of Discrete Logarithms

As part of our constructions, we need zero-knowledge proofs for statements of the form $\log_g g^x = \log_h h^x$. That is, that discrete logarithm of two numbers relative to two bases is identical.

Chaum and Pedersen [13] devised a Σ-protocol that allows one to prove such statements. It provides the well-known properties *Completeness*, *Knowledge Soundness* and *Special Honest Verifier Zero-Knowledge* under the discrete logarithm assumption. Note that the latter implies that one can simulate proofs for false statements. Furthermore, in the random oracle model, by the the Fiat-Shamir heuristic [18], the protocol can be turned into a non-interactive zero-knowledge (NIZK) proof. We call this non-interactive version DLEq in our protocols and define the syntax below.

Definition 6 (Equality of Discrete Logarithms)**.** The protocol DLEq has two functions:

- $\mathsf{prove}(a, b, x) \to \pi$ which given two elements $a, b \in \mathbb{G}$ and exponent $x \in \mathbb{Z}_q$ outputs a proof π.
- $\mathsf{verify}(a, b, \pi) = \top/\bot$ which given two elements a, b and a proof π outputs whether π proves $\log_g a = \log_h b$.