Asynchronous Data Dissemination and its Applications

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

Zhuolun Xiang
University of Illinois at Urbana-Champaign
xiangzl@illinois.edu

Ling Ren
University of Illinois at Urbana-Champaign
renling@illinois.edu

ABSTRACT

In this paper, we introduce the problem of Asynchronous Data Dissemination (ADD). Intuitively, an ADD protocol replicates a message to all honest nodes in an asynchronous network, given that at least $t + 1$ honest nodes initially hold the message where t is the maximum number of malicious nodes. We design a simple yet efficient ADD protocol for n parties that is information theoretically secure, tolerates up to one-third malicious nodes, and has a communication cost of $O(n|M|+n^2)$ for replicating a message M.

We then use our ADD protocol to improve many important primitives in cryptography and distributed computing. For reliable broadcast, assuming the existence of collision resistance hash functions, we present a protocol with communication cost $O(n|M|+\kappa n^2)$ where κ is the size of the hash function output. This is an improvement over the best-known complexity of $O(n|M|+\kappa n^2 \log n)$ under the same setting. Next, we use our ADD protocol along with additional new techniques to improve the communication complexity of Asynchronous Verifiable Secret Sharing (AVSS) and Asynchronous Complete Secret Sharing (ACSS) with no trusted setup from $O(\kappa n^2 \log n)$ to $O(\kappa n^2)$. Furthermore, we use ADD and a publicly-verifiable secret sharing scheme to improve dual-threshold ACSS and Asynchronous Distributed Key Generation (ADKG).

1 INTRODUCTION

In this paper, we propose and study a new problem which we call *Asynchronous Data Dissemination* (ADD). Briefly, the goal is to replicate a data blob from a subset of honest nodes to *all* honest nodes, despite the presence of some malicious nodes. More specifically, in a network of n nodes where up to t nodes could be malicious, a subset of at least $t + 1$ honest nodes start with a common message blob M, and the goal is to have all honest nodes output M at the end of the protocol.

We will present a solution for the ADD problem with $n \geq 3t + 1$, where at least $t + 1$ honest nodes start with the message M. For a message M of size $|M|$, our protocol has a total communication cost of $O(n|M|+n^2)$. Moreover, our solution to the ADD problem is *information theoretically* secure i.e., it does not rely on any cryptographic assumption. Additionally, if we assume the existence of a collision-resistant hash function with output size κ, we can extend our ADD protocol to any $t \leq (1/2 - \epsilon)n$ for $\epsilon > 0$ with a total communication cost of $O(n|M|/\epsilon + n^2\kappa)$.

We then observe that ADD lies in the heart of many important problems, including asynchronous reliable broadcast (RBC) [14, 17], asynchronous verifiable secret sharing (AVSS) [4, 16], asynchronous complete secret sharing (ACSS) [55], dual-threshold ACSS [3, 37], and asynchronous distributed key generation (ADKG) [2, 37], as illustrated in Figure 1. Using our solution to ADD with $n \geq 3t + 1$ and some additional new techniques, we can construct solutions to the problems in Figure 1 with improved communication complexity or weaker assumptions.

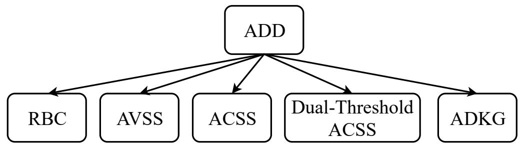

Figure 1: Illustration of the relationships between the problems in this paper. Asynchronous Data Dissemination (ADD) can be used to solve Asynchronous Reliable Broadcast (RBC), Asynchronous Verifiable Secret Sharing (AVSS), Asynchronous Complete Secret Sharing (ACSS), Dual-threshold ACSS and Asynchronous Distributed Key generation (ADKG).

Table 1: Comparison of protocols proposed in this paper with best known protocols realizing different primitives under different setup and cryptographic assumption. Here n is the number of nodes in the system and κ is the security parameter.

Scheme	Communication Cost (total)	Cryptographic Assumption	Setup	Reference
RBC extension	$O(n^2\|M\|)$	None	None	[14]
	$O(n\|M\|+\kappa n^2 \log n)$	Hash	None	[17]
	$O(n\|M\|+\kappa n^2)$	q-SDH+DBDH	Trusted	[43]
	$O(n\|M\|+\kappa n^2)$	Hash	None	**this work**
AVSS and ACSS	$O(\kappa n^2 \log n)$	DL+RO	PKI	[55]
	$O(\kappa n^2 \log n)$	DL+RO	None	[3]*
	$O(\kappa n^2)$	q-SDH + Hash	Trusted	[4]
	$O(\kappa n^2)$	DL+Hash	None†	**this work**
Dual Threshold ACSS*	$O(\kappa n^2 \log n)$	DL+RO	None	[3]
	$O(\kappa n^2)$	q-SDH + Hash	Trusted	[3]
	$O(\kappa n^2)$	DL+RO†	PKI	**this work**
ADKG	$O(\kappa n^3 \log n)$	DL+RO	None	[3, 37]
	$O(\kappa n^3)$	Strong RSA+DCR+RO	PKI	**this work**

† Our AVSS does not require PKI, but our ACSS does. Also, If we assume DBDH assumption, then we do not need RO in our dual-threshold ACSS.

* As presented, the ACSS scheme of [3] only supports uniform random secrets. All the dual-threshold ACSS scheme also only support uniform random secrets. But they can be extended to arbitrary secrets using techniques from [51].

Overview of our results. Table 1 compares our results with existing works. As mentioned, all our protocols are built on top of ADD with $n = 3t + 1$, which intuitively replicates a message M to all honest nodes, given the initial condition that at least $t + 1$ honest nodes have M as the input and rest of the nodes have $\perp$. To implement ADD efficiently, our protocol first leverages error-correcting code to encode M into n codewords, and then the i-th node is responsible for dispersing the i-th codeword. For any node i that has input $\perp$, it learns and disperses the i-th codeword after receiving $t + 1$ identical messages from the set of $t + 1$ honest nodes that have input M. After each node i disperses the i-th codeword, honest nodes repeatedly try to reconstruct the message M whenever at least $2t + 1$ codewords are received. The node outputs the reconstructed message if it matches $2t + 1$ codewords. The ADD protocol is information-theoretic, has a total communication cost of $O(n|M|+n^2)$, and tolerates $t < n/3$ faults.

The first primitive we can improve is reliable broadcast (RBC) [14, 17] for large messages, a fundamental primitive in asynchronous networks. Briefly, a reliable broadcast protocol implements a broadcast channel in an asynchronous network and ensures that all honest nodes in the network deliver the same message if any honest node delivers. RBC has been used to construct many higher-level protocols such as atomic broadcast [1, 26, 30, 33, 39, 42], asynchronous multi-party computation [38, 55], and asynchronous distributed key generation [2, 30, 37]. All of these involve RBC for long messages, typically of size $\Omega(n)$ where n is the total number of nodes in the protocol. The problem of RBC for long messages is called RBC *extensions*. The classic RBC protocol due to Bracha [14] solves 1-bit RBC with communication cost $O(n^2)$, and therefore $O(n^2|M|)$ for broadcasting a message M. Prior to our work, the most closely related and inspiring work is the asynchronous information dispersal (AVID) protocol due to Cachin and Tessaro [17]. Their protocol uses error-correcting codes and Merkle path proofs and has a total communication cost of $O(n|M|+\kappa n^2 \log n)$; it can be modified to solve RBC extension with the same total communication cost.

We show that we can combine Bracha's classic RBC protocol [14] with our ADD protocol to obtain an improved solution to RBC extension. The key observation is that running the Bracha's RBC protocol on the hash digest of M can establish exactly the initial condition for ADD that at least $t + 1$ honest nodes start with M and no honest node starts with any other message. As a result, after running our ADD protocol, every honest node unanimously outputs the message M. Compared to current best RBC extension protocol [17], we remove the $\log n$ term in the communication cost as we remove the Merkle path proofs.

Our ADD solution along with additional techniques also improves the communication costs of Asynchronous Verifiable Secret Sharing (AVSS) [4, 16] and its variants such as Asynchronous Complete Secret Sharing (ACSS) and Dual-Threshold ACSS [3, 37]. Typically, state-of-the-art protocols for these primitives use RBC extension as a black-box, but they also have other bottleneck steps in terms of communication cost. For example, without any trusted setup, the best known AVSS protocol [3] instantiated with polynomial commitment schemes from [15] has a communication cost of $O(\kappa n^2 \log n)$ due to two steps: a RBC on an $O(\kappa n)$ size message, and an all-to-all gossip of $O(\kappa \log n)$ size polynomial evaluation proofs. The polynomial evaluation proofs are needed because the RBC extension is executed without nodes checking the validity of the message. Hence, to reduce the total communication cost of AVSS it is not sufficient to use a more communication efficient black-box RBC extension protocol. Instead, as mentioned earlier, we open the black-box of RBC extension and propose an improved protocol that checks the validity of messages during RBC on the message digest followed by an ADD to replicate the message.

In summary, we make the following contributions.

- We introduce the Asynchronous Data Dissemination (ADD) problem for asynchronous networks where up to a threshold fraction of nodes could be malicious. We design an information-theoretically secure and efficient protocol to solve the ADD problem that tolerates up to 1/3 malicious nodes and has cost $O(n|M|+n^2)$ for a message M of size $|M|$. Assuming collision resistance hash function, we can extend this ADD protocol to tolerate up to 1/2 malicious nodes.
- We then use ADD to design an improved asynchronous reliable broadcast protocol for any message M with a communication cost of $O(n|M|+\kappa n^2)$. Our construction assumes the existence of collision-resistant hash functions of output size κ.
- Finally, we use our solution to ADD along with additional techniques to design asynchronous verifiable secret sharing (AVSS), asynchronous complete secret sharing (ACSS), dual-threshold ACSS, asynchronous distributed key generation, all with improved communication cost or weaker assumptions comparing to the state-of-the-art solutions.

Paper Organization. The rest of the paper is organized as follows. We describe our system model, introduce notations and provides some necessary background §2. In §3 we first formally introduce the problem of Asynchronous Data Dissemination (ADD) and describe our protocol for ADD that meets the desired communication cost. Next, in §4 we describe how we use ADD to implement reliable broadcast extensions. We then provide details construction of communication efficient AVSS ACSSand dual threshold ACSS in §5. Next in §6, we show how to modify our dual-threshold ACSS to improve the communication cost of asynchronous DKG §6. In §7 we describe the related work and conclude after a discussion in §8.

2 SYSTEM MODEL AND PRELIMINARIES

Cryptographic Assumptions. Let $\mathbb{G}$ be a group of order q where q is a prime number and $\mathbb{Z}_q$ be the field with integer operation modulo q. Throughout the paper, we use $\mathsf{hash}(\cdot)$ as a collision resistance hash function. We will use κ to denote the size of cryptographic objects, e.g., the length of the hash function output, the size of a ciphertext of an encryption scheme, or the size of an element in the group $\mathbb{G}$. These objects may slightly differ in size in practice. In that case, we assume they are on the same order, or one can interpret κ as the largest among them.

Network and Adversarial Assumptions. We consider an asynchronous network of $n = 3t + 1$ nodes where a malicious adversary can corrupt up to t nodes in the network. The corrupted nodes can deviate arbitrarily from the protocol. The remaining nodes are honest and strictly adhere to the protocol. We also assume that every pair of honest nodes have access to pairwise reliable and authenticated channels. The network is asynchronous, so the adversary can arbitrarily delay or reorder messages between honest nodes, but must eventually deliver every message.

Error Correcting Code. Our ADD protocol uses error correcting codes. For concreteness, we will use the standard Reed Solomon (RS) codes [49]. A (m, k) RS code in Galois Field $\mathbb{F} = \mathrm{GF}(2^a)$ with $m \leq 2^a - 1$, encodes k data symbols from $\mathrm{GF}(2^a)$ into codewords of m symbols from $\mathrm{GF}(2^a)$. Let $\mathsf{RSEnc}(M, m, k)$ be the encoding algorithm. Briefly, the RSEnc takes input a message M consisting of $k + 1$

data symbols, treats it as a polynomial of degree k and outputs m evaluation of the corresponding polynomial.

Let $\mathsf{RSDec}(k, r, T)$ be the RS error correction procedure. The RSDec takes as input a set of codeword symbols T (some of which may be incorrect), and outputs a degree k polynomial, by correcting up to r errors (incorrect symbols) in T. It is well-known that [40] RSDec can correct up to r errors in T and output the original message provided $|T| \geq k + 2r + 1$. Concrete instantiations of RSEnc include the Berlekamp-Welch algorithm [53], the algorithm due to Gao [31], etc.

3 ASYNCHRONOUS DATA DISSEMINATION

In this section, we formally define the problem of Asynchronous Data Dissemination (ADD). We then provide our solution to the problem, and finally analyze its correctness and performance.

3.1 Problem Statement

Formally, we define Asynchronous Data Dissemination as follows.

Definition 3.1 (Asynchronous Data Dissemination (ADD)). Given a network of n nodes, of which up to t could be malicious, let M be a data blob that is the input of at least $t + 1$ honest nodes. The remaining honest nodes start with $\perp$. A protocol Π solves Asynchronous Data Dissemination (ADD) if it ensures that every honest node eventually outputs M.

Here on, we refer to the honest nodes that start with the message M as the *sender* nodes and the honest nodes that start with $\perp$ as the *recipient* nodes.

A simple but important observation is that to solve ADD in a network with t faults, the number of honest sender nodes must be at least $t + 1$. Otherwise, to any honest recipient node i, the set of honest senders that claim to start with M is indistinguishable from the set of malicious senders that behave honestly but claim to start with M'. This justifies the initial condition of at least $t + 1$ honest sender nodes. In addition, in all of the applications considered in this paper, we have $n \geq 3t+1$, which is optimal for an asynchronous network. We will focus on the case of $n = 3t + 1$ for convenience.

The simplest ADD protocol just has each honest sender multicast its input to all other nodes. A recipient node, upon receiving $t + 1$ identical copies of a message M, outputs M. Since we begin with $t + 1$ honest senders, every recipient node will eventually receive $t + 1$ identical messages. Furthermore, whenever an honest node receives $t + 1$ identical message, say M', this means that at least one honest sender sent M'. Since honest senders only send M, this implies that $M' = M$. The issue with this approach is that it is not communication efficient. Specifically, this approach has a total communication cost of $O(n^2|M|)$.

An alternate ADD protocol is where each recipient node request M from the sender nodes, to which an honest sender reply by sending M. Again, the issue with this approach is that malicious nodes may redundantly request M from all the honest senders. Since there are $t = \Theta(n)$ malicious nodes and each malicious node request M from all the sender nodes, this approach also has a total communication cost of $O(n^2|M|)$.

Algorithm 1 Pseudocode for node i in ADD for $n = 3t + 1$

```
1:  // encoding phase.
2:  input $M_i$: either $M_i = M$ or $M_i = \perp$
3:  if $M_i \neq \perp$ then
4:      Let $M' = [m_1, m_2, \ldots, m_n] := \mathsf{RSEnc}(M_i, n, t)$
5:  // dispersal phase
6:  if $M_i \neq \perp$ then
7:      $m_i^* := m_i$
8:      send $\langle \mathsf{DISPERSE}, m_j \rangle$ to node $j$ for every $j = 1, 2, \ldots, n$
9:  else
10:     upon receiving $t + 1$ identical $\langle \mathsf{DISPERSE}, m_i \rangle$ do
11:         $m_i^* := m_i$
12: // reconstruction phase
13: multi-cast $\langle \mathsf{RECONSTRUCT}, m_i^* \rangle$ to all nodes
14: if $M_i \neq \perp$ then
15:     output $M$ and return;
16: Let $T = \{\}$
17: For every $\langle \mathsf{RECONSTRUCT}, m_j^* \rangle$ received from node $j$, add $(j, m_j^*)$ to $T$
18: for $0 \leq r \leq t$ do        // online Error Correction
19:     Wait till $|T| \geq 2t + r + 1$
20:     Let $p_r(\cdot) = \mathsf{RSDec}(t, r, T)$
21:     if $2t + 1$ elements $(j, a) \in T$ satisfy $p_r(j) = a$ then
22:         output coefficients of $p_r(\cdot)$ as $M$ and return
```

3.2 Our Approach

We present two variants of our ADD protocol. The first variant requires $n \geq 3t + 1$ and is information-theoretically secure. The second variant of our ADD protocol requires $n \geq 2t+1$ but assumes the existence of a collision resistance hash function. As discussed earlier, since all the application of ADD we discuss in this paper require $n \geq 3t+1$ (due to reasons orthogonal to ADD) we will focus on the $n \geq 3t + 1$ variant of our ADD protocol. The $n \geq 2t + 1$ variant will be given in Appendix B.

Our ADD protocol has three phases: *Encoding, Dispersal,* and *Reconstruction.* Figure 2 illustrates our ADD protocol for a network of $n = 4$ nodes with $t = 1$. Also, we provide the pseudocode of the protocol for node i in Algorithm 1.

Encoding phase. In the encoding phase, every sender (who holds $M \neq \perp$) encodes M using a (n, t) Reed-Solomon code, i.e., using $\mathsf{RSEnc}(\cdot)$ described in §2. Line 2 – 4 in Algorithm 1 illustrates this. The encoded message $M' = \mathsf{RSEnc}(M)$ can be written as a vector $M' = [m_1, m_2, \ldots, m_n]$. Here each m_i is of size approximately $|m_i| = |M|/t$ because, after encoding,

$$|M'| = \frac{n|M|}{t} \Rightarrow \frac{|M'|}{n} = \frac{|M|}{t} \tag{1}$$

Dispersal phase. After encoding M into M', the senders start the dispersal phase (Line 5 – 10 in Algorithm 1). During the dispersal phase, every sender sends the message $\langle \mathsf{DISPERSE}, m_j \rangle$ to node j. A recipient node j, upon receiving $t + 1$ DISPERSE messages for the identical message m'_j sets m'_j as its local share m^*_j. Each honest sender node i, sets m_i as its local share m^*_i.

Reconstruction phase. During the reconstruction phase, every node i multi-casts $\langle \mathsf{RECONSTRUCT}, i, m^*_i \rangle$ to all other nodes. Then, every recipient node j, upon receiving enough shares, uses the standard *Online Error Correcting* (OEC) algorithm from [7] (line

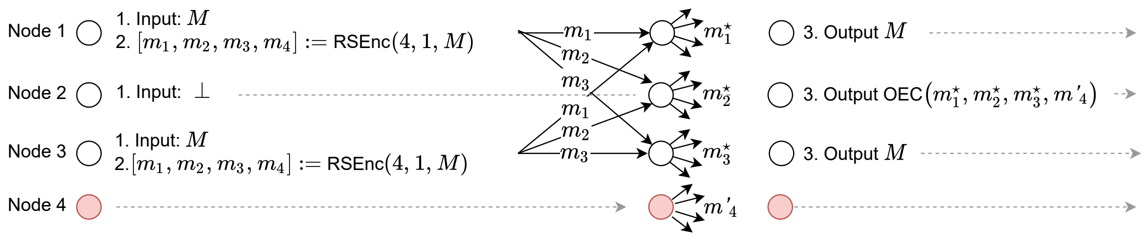

Figure 2: A example execution of ADD with 4 nodes $\{1, 2, 3, 4\}$ among which node 4 is malicious and node 2 does not start with M. As described, after the distribution phase, node 2 will receive $2 = t + 1$ identical copies of m_2 from node 1 and 4 and hence will output m_2 as m_2^*. As a result, after the reconstruction phase, node 2 will receive 3 correct chunks of M' which is sufficient for reconstructing M.

17 – 23 in Algorithm 1). Briefly, the OEC algorithm [7] performs up to t trials of reconstruction. In the r-th trial, the recipient waits until it receives RECONSTRUCT messages from $2t + r + 1$ nodes and tries to decode the message polynomial. If the reconstructed polynomial agrees with $2t + 1$ points in the RECONSTRUCT messages received so far, the recipient outputs the decoded message; otherwise, it waits for one more RECONSTRUCT message and tries again.

3.3 Analysis

Next, we will prove that our protocol solves the ADD problem. Towards this end, we will first prove that every honest node will hold the correct local share at the end of the dispersal phase. Next, we will argue that every honest node successfully reconstructs the message M. Recall that we refer to the nodes that start with the message M as the *sender* nodes and the honest nodes that start with $\perp$ as the *recipient* nodes.

Lemma 3.2. *After the dispersal phase each honest node j holds m_j^* where m_j^* is the jth coordinate of $M' =$ RSEnc(M, n, t).*

Proof. Recall that RSEnc encoding procedure is deterministic. If node j is a sender, then it trivially holds m_j^*. Thus, we focus on the case where node j is a recipient node.

Again since RSEnc encoding is deterministic, all honest senders compute the same M', and they all send m_j^* to node j. No honest node will send DISPERSE with any $m_j \neq m_j^*$. Since node j holds m_j only if it receives $t + 1$ identical DISPERSE messages for m_j, no honest node will hold $m_j \neq m_j^*$ at the end of the dispersal phase. Since there are at least $t + 1$ honest senders, and they all send m_j^* to node j, node j will eventually receive at least $t+1$ DISPERSE message for m_j^* and will hold m_j^* at the end of the dispersal phase. □

To argue that each honest recipient will eventually recover the message at the end of the reconstruction phase of our ADD protocol, we first prove the following Lemma about OEC. As we mention earlier, we tailor the Lemma for the specific case of $n = 3t + 1$, but it can be easily generalized (ref. Appendix B).

Lemma 3.3. *If at least $2t + 1$ correct codewords of the encoding of a message M are received, OEC will eventually reconstruct M.*

Proof. Let T_r be the set of received symbols up to iteration r (including iteration r).

First, we will argue that every honest node will eventually output a message. Let's focus on a single honest node, say i. Let's assume $\hat{r}_1$ corrupt nodes sent incorrect symbols to node i and $\hat{r}_2$ corrupt nodes did not send anything. Note that $\hat{r}_1 + \hat{r}_2 \leq t$.

Now consider the $(t - \hat{r}_2)^{\text{th}}$ iteration; since $\hat{r}_2$ nodes never sent any symbols to node i, i will receive $2t + (t - \hat{r}_2) + 1$ distinct symbols on the polynomial $p(\cdot)$ of which $\hat{r}_1$ symbols are incorrect, then

$$|T_{t-\hat{r}_2}| = 2t + (t - \hat{r}_2) + 1 \geq t + 2\hat{r}_1 + 1 \quad (2)$$

hence the algorithm RSDec will correct $\hat{r}_1$ errors and will return the polynomial $p_{t-\hat{r}_2}(\cdot)$ as the message during the $(t - \hat{r}_2)^{\text{th}}$ iteration.

For correctness, let's assume that node i outputs the polynomial $p_r(\cdot)$ as the message in the r^{th} iteration. Then, $p_r(\cdot)$ is consistent with $2t + 1$ points from T_r, of which at least $t + 1$ are from honest nodes. All these belong on both polynomial $p(\cdot)$, and $p_r(\cdot)$. Since, $p_r(\cdot)$ is a degree t polynomial and agrees on $t + 1$ points with $p(\cdot)$, $p_r(\cdot) = p(\cdot)$ as a polynomial as well. □

Lemma 3.4. *At the end of reconstruction phase of our ADD protocol, every honest node outputs M.*

Proof. Clearly, every honest sender outputs M. Thus, we again focus on honest recipient nodes.

From Lemma 3.2, at the end of the dispersal phase, every honest node holds the correct component of $M' =$ RSEnc(M, n, t). This implies that every recipient node will eventually receive RECONSTRUCT messages from all the honest nodes. Hence, due to guarantees of the OEC protocol, i.e., Lemma 3.3, this implies that all honest nodes will eventually output M. □

We will next argue about the total communication cost of our ADD protocol.

Lemma 3.5. *The total communication cost of our ADD protocol is $O(n|M|+n^2)$.*

Proof. Recall that $|M'| = (n/t)|M| = O(|M|)$. During the dispersal phase, each sender node sends a message of size $|M'|/n + O(1)$ to every other node. During the reconstruction phase, each node multi-casts a message of size $|M'|/n + O(1)$. Therefore, the total communication cost of the protocol is $n|M'|+O(n^2)$, which is the same as $O(n|M|+n^2)$. □

Using Lemma 3.2, 3.4 and 3.5, we have that Algorithm 1 solves ADD with a communication complexity of $O(n|M|+n^2)$.

Now let's analyze the computation cost of each node in Algorithm 1. The encoding step at every sender during the dispersal phase involves $O(n^2)$ operations (additions and multiplications) in $\mathbb{Z}_q$. The recipient nodes do nothing. During the reconstruction

Algorithm 2 Bracha's RBC [14]

```
// only broadcaster node
input $M$
send $\langle$PROPOSE, $M\rangle$ to all
// all nodes
upon receiving $\langle$PROPOSE, $M\rangle$ from the broadcaster do
    send $\langle$ECHO, $M\rangle$ to all
upon receiving $2t+1$ $\langle$ECHO, $M\rangle$ messages and not having sent a READY message do
    send $\langle$READY, $M\rangle$ to all
upon receiving $t+1$ $\langle$READY, $M\rangle$ messages and not having sent a READY message do
    send $\langle$READY, $M\rangle$ to all
upon receiving $2t+1$ $\langle$READY, $M\rangle$ messages do
    output $M$
```

phase, in the good case, each node needs to invoke the RSDec algorithm only once, i.e., only $r = 1$. Furthermore, the nodes need to perform the consistency check only once. However, each honest node would need to invoke RSDec t times in the worst case. Hence, the computation cost would be higher.

4 RELIABLE BROADCAST EXTENSION

In this section, we will describe how we can use our ADD protocol to design a protocol for reliable broadcast extensions. The problem of reliable broadcast was introduced by Bracha [14]. In the same paper, Bracha also provided a protocol for reliable broadcast of a single bit with total communication complexity of $O(n^2)$. Before we describe our solution, we define the problem of reliable broadcast.

Definition 4.1 (Reliable Broadcast). A protocol for a set of nodes $\{1, ..., n\}$, where a distinguished node called the broadcaster holds an initial input M of size $|M|$, is a reliable broadcast (RBC) protocol tolerating an adversary $\mathcal{A}$, if the following properties hold

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

Since our RBC extension protocol relies upon the RBC protocol due to Bracha [14], for completeness we will first describe the RBC protocol of Bracha [14] in Algorithm 2. The main idea of Bracha's RBC is to use quorum intersection (of ECHO messages) for agreement, and use vote amplification (of READY messages) for totality. However, the protocol needs to attach the input M in every vote message, leading to a high communication cost of $O(n^2|M|)$.

We present our RBC extension protocol in Algorithm 3 where we highlight the changes from Bracha's RBC in blue. The core idea in our RBC extension protocol is to run the Bracha-style reliable broadcast protocol only on the hash of the message M, and then replicate the message M using our ADD protocol. The previous best RBC due to Cachin and Tessaro [17] also runs RBC on the hash digest, but their protocol requires attaching Merkle path proofs in the messages during dispersal and reconstruction, which inevitably incurs a cost of $O(n|M|+\kappa n^2 \log n)$. In contrast, our ADD protocol

Algorithm 3 RBC extension for a long message M

```
// only broadcaster node
input $M$
send $\langle$PROPOSE, $M\rangle$ to all
// all nodes
upon receiving $\langle$PROPOSE, $M\rangle$ from the broadcaster do
    let $h :=$ hash($M$)
    send $\langle$ECHO, $h\rangle$ to all
upon receiving $2t+1$ $\langle$ECHO, $h\rangle$ messages and not having sent a READY message do
    send $\langle$READY, $h\rangle$ to all
upon receiving $t+1$ $\langle$READY, $h\rangle$ messages and not having sent a READY message do
    send $\langle$READY, $h\rangle$ to all
upon receiving $2t+1$ $\langle$READY, $h\rangle$ messages do
    if received $\langle$PROPOSE, $M\rangle$ and $h =$ hash($M$) then
        ADD($M$)
    else
        ADD($\perp$)
```

removes such Merkle path proof and only incurs a communication cost of $O(n|M|+n^2)$ for replicating the message, leading to a total communication cost of $O(n|M|+\kappa n^2)$ for RBC extension.

More specifically, the broadcaster of our RBC extension protocol first multi-cast M to all other nodes. Every honest node upon receiving the message from the leader, first participates in a Bracha-style RBC on h, the hash of the message M. Once the RBC terminates, i.e., the node receives $2t+1$ READY messages for some h, the node inputs M to the ADD protocol if $h = \text{hash}(M)$. Otherwise, the node inputs $\perp$ to the ADD protocol if $h \neq \text{hash}(M)$. Recall that in Bracha's RBC, a node outputs M when receiving $2t + 1$ READY messages for M, which implies at least $t + 1$ honest nodes have received M from the broadcaster before sending ECHO for M. Similarly, in our RBC extension protocol, when a node receives $2t+1$ READY messages for some hash h, at least $t + 1$ honest nodes have received the message M such that $\text{hash}(M) = h$. Moreover, the agreement property of the Bracha-style RBC guarantees that no two honest nodes will agree on different hashes, and thus any honest node that receives $M' \neq M$ from the broadcaster will input $\perp$ to the ADD. Therefore, the initial condition of ADD is met in our construction. Hence, the guarantees of our ADD protocol ensure the desired properties of the RBC extension protocol.

Remark. Our RBC extension construction first uses a Bracha-style RBC on a short digest to set up the initial condition of ADD, and then executes ADD. We remark that this structure will repeatedly appear in all the protocols we design in this paper.

4.1 Analysis

First, we show that running Bracha's RBC on the hash h sets up the initial condition for ADD.

LEMMA 4.2. *If any honest node executes* ADD(M), *then there at least $t + 1$ nodes that receive M from the broadcaster and multi-cast* $\langle$ECHO, $h\rangle$ *where* $h = \text{hash}(M)$.

PROOF. An honest node i executes ADD(M) only upon receiving $\langle$READY, $h = \text{hash}(M)\rangle$ messages from a quorum Q_2 of $2t + 1$ nodes.

Since, there are at most t malicious nodes in the system, at least $t+1$ of the nodes in Q_2 are honest. This implies that at least one honest node receives a $\langle \texttt{ECHO}, h \rangle$ message from a quorum Q_1 of $2t+1$ nodes and then multi-casts $\langle \texttt{READY}, h \rangle$ message. Again, at least $t+1$ nodes of Q_1 are honest. These at least $t+1$ honest nodes receive M from the broadcaster and multi-cast $\langle \texttt{ECHO}, h \rangle$ messages. □

Lemma 4.3. *If any honest node executes* ADD(M)*, then no honest node executes* ADD(M') *for any $M' \neq M$ and $M' \neq \perp$, and eventually all honest nodes execute* ADD(M) *or* ADD($\perp$) *among which at least $t+1$ honest nodes execute* ADD(M).

Proof. Let node i executes ADD(M), then by Lemma 4.2, there exits a quorum Q_h of honest nodes who sent $\langle \texttt{ECHO}, h \rangle$ message where $h = \text{hash}(M)$. Now, for the sake of contradiction assume that an honest node j executes ADD(M') for some $M' \neq M$, then by Lemma 4.2, there exist a quorum $Q_{h'}$ of honest nodes of size $t+1$ that sent $\langle \texttt{ECHO}, h' \rangle$ message where $h' = \text{hash}(M')$. This is impossible as there are at most $2t+1$ honest nodes and an honest node sends ECHO message at most once.

Since at least $t+1$ honest nodes multi-cast $\langle \texttt{READY}, h \rangle$, these messages eventually reach all honest nodes, and all $2t+1$ honest nodes will multi-cast $\langle \texttt{READY}, h \rangle$ according to the protocol. Hence, eventually all honest nodes receive $2t+1$ $\langle \texttt{READY}, h \rangle$ messages and executes ADD(M) or ADD($\perp$). By Lemma 4.2, at least $t+1$ honest nodes that receive M from the broadcaster will executes ADD(M) according to the protocol. □

Now, we can prove the properties of RBC extension are satisfied.

Lemma 4.4 (Agreement and Totality). *If an honest node outputs M, then no honest node will output $M' \neq M$, and every honest node will eventually output M.*

Proof. If an honest node outputs M, then some honest node must execute ADD(M). No honest node executes ADD(M') for any $M' \neq M$, otherwise according to Lemma 4.3 the initial condition for ADD is met for message M' and all honest nodes will output M'. Since some honest node executes ADD(M), by Lemma 4.3, all honest nodes will execute ADD(M) or ADD($\perp$), among which at least $t+1$ honest nodes receiving M from the broadcaster that will execute ADD(M). Hence, it is a execution of our ADD with the required initial condition, by Lemma 3.4, every honest node will output M. □

Lemma 4.5 (Validity). *When the broadcaster is honest and has an input M, then every honest will eventually output M.*

Proof. When the broadcaster is honest, it will send $\langle \texttt{PROPOSE}, M \rangle$ to all nodes. As a result, $2t+1$ honest node will multi-cast $\langle \texttt{ECHO}, h \rangle$ to all other nodes. As a result, every honest node will eventually multi-cast $\langle \texttt{READY}, h \rangle$ message to others. Since, there are at least $2t+1$ honest nodes, every honest node will start ADD with M, i.e., ADD(M). Since ADD ensures correctness and totality for any number of senders greater than t, this implies that every honest node will output M. □

Next, we will analyze the message and communication complexity of the protocol.

Lemma 4.6. *Assuming existence of collision resistance hash functions, Algorithm 3 solves RBC extension with message complexity of $O(n^2)$ and communication complexity of $O(n|M|+\kappa n^2)$, where κ is the size of the output of the hash function.*

Proof. Recall from §3, the message complexity of our ADD protocol for a linear size message is $O(n^2)$. In addition to running our ADD protocol, the leader multi-cast a single PROPOSE to all other nodes. Moreover, every honest in our RBC extension protocol multi-casts a single ECHO message and a single READY message. Hence, the total message complexity is $O(n^2)$.

For message M, its proposal has a communication cost of $O(n|M|)$. The multi-cast of ECHO and READY message has a communication cost of $O(n\kappa)$ for each node; hence a total communication cost is $O(\kappa n^2)$. From Lemma 3.5, the communication cost of ADD is $O(n|M|+n^2)$. Combining all these costs, we get that the total communication cost of our RBC extension protocol is $O(n|M|+\kappa n^2)$. □

Combining the above lemmas, we get the following theorem.

Theorem 4.7. *In an asynchronous network of n nodes where $t < n/3$, assuming existence of collision resistance hash functions, Algorithm 3 solves RBC extension with message complexity $O(n^2)$ and communication complexity $O(n|M|+\kappa n^2)$ where κ is the size of the output of the hash function.*

4.2 Applications of Reliable Broadcast

Reliable broadcast has been used as a crucial primitive in many protocols ranging from atomic broadcast [26, 30, 33, 39, 42], state machine replication [22, 54], asynchronous verifiable secret sharing [3, 4, 17, 55], asynchronous multi-party computation [38, 55], asynchronous distributed key generation [2, 37], etc.. Almost all these protocols involve reliable broadcast of messages of size at least linear in the number of participants in the network.

Direct improvements. For some of the above applications such as atomic broadcast and asynchronous MPC, our improvements to RBC extension directly improves them. For example, HoneyBadger BFT [42] uses $O(n)$ reliable broadcast of $O(n)$ size messages. Hence, their protocol has a total cost of $O(n^3 \log n)$ per block. With our RBC extension, the communication cost of HoneyBadger BFT will reduce to $O(n^3)$. Similarly, both Dumbo [33] and Aleph [30] use RBC to gossip $O(n)$ size messages for every block, hence we immediately reduce their communication cost by a $\log n$ factor as well.

Other cases. However, for some other primitives, simply replacing the RBC extension protocol with our improved one does not immediately reduce the asymptotic communication cost of the overall protocol because there are often other bottleneck steps in the protocol. For example, hbACSS [55] is an ACSS scheme with $O(\kappa n^2 \log n)$. But the cost arises from an RBC extension of message of size $O(\kappa n \log n)$, thus we can not simply use our improved RBC extension to improve the communication cost. Similarly, Haven [3] is a dual-threshold ACSS protocol for uniform secrets that has a communication of $O(\kappa n^2 \log n)$. These cost arise from two sources: a reliable broadcast of $O(\kappa n)$ size message as well as a step where nodes gossip zero-knowledge proofs of size $O(\kappa \log n)$. In the next section, we introduce additional techniques to obtain protocols for these primitives with communication complexity of $O(\kappa n^2)$.

5 ASYNCHRONOUS (DUAL THRESHOLD) VERIFIABLE SECRET SHARING

In this section, we will provide a protocol for asynchronous verifiable secret sharing that does not require any trusted setup and has a communication complexity of $O(\kappa n^2)$. Prior to our work, the best-known protocol for AVSS in the same threat model had a communication cost of $O(\kappa n^2 \log n)$ [17] and a communication cost of $O(\kappa n^2)$ has only been achieved with a trusted setup [4]. Before we dive into our solution, we will formally define the problem of asynchronous verifiable secret sharing and its desired properties. Our definitions are borrowed from the definitions of [45].

5.1 Definitions and Preliminaries

For all these primitives, we assume a finite field $\mathbb{F}$ of order q. Let κ be the security parameter and $\mathsf{negl}(\kappa)$ is a negligible function in λ.

Definition 5.1 (Asynchronous Verifiable Secret Sharing). Let $(\mathsf{Sh}, \mathsf{Rec})$ be a pair of protocols in which a dealer L shares a secret s using Sh. We say that $(\mathsf{Sh}, \mathsf{Rec})$ is a t-resilient AVSS scheme if the following properties hold with probability $1 - \mathsf{negl}(\kappa)$:

- **Termination:**
  1. If the dealer L is honest, then each honest node will eventually terminate the Sh protocol.
  2. If an honest node terminates Sh protocol, then each honest node will eventually terminate Sh.
  3. If all honest node start Rec, then each honest node will eventually terminate Rec.
- **Correctness:**
  1. If L is honest, then each honest node upon terminating Rec, outputs the shared secret s.
  2. If L is corrupt and some honest node terminates Sh, then there exists a fixed secret $s' \in \mathbb{F}$, such that each honest node upon completing Rec, will output s'.
- **Secrecy:** If L is honest and no honest node has begun executing Rec, then an adversary that corrupts up to t nodes has no information about s.

Note that, by definition of the Termination property, AVSS schemes allow a situation where despite finishing the sharing phase and entering the reconstruction phase, an honest node does not receive its share from the dealer. Such a situation usually arises when the dealer is malicious and only sends the valid shares to a subset of honest nodes, yet the corrupted nodes also claim to have the shares so that all honest nodes terminate the sharing phase.

To accommodate for this scenario, a stronger primitive called *Asynchronous Complete Secret Sharing* (ACSS) is defined as follows. An ACSS is an AVSS scheme that additionally ensures that at the end of sharing phase, every honest node receives its shares of a consistent secret. This holds even if the dealer is corrupt.

Definition 5.2 (Asynchronous Complete Secret Sharing). An Asynchronous Complete Secret Sharing protocol is an AVSS protocol that additionally satisfy the following completeness property.

- **Completeness:** If some honest node terminates Sh, then there exists a degree t polynomial $p(\cdot)$ over $\mathbb{F}$ such that $p(0) = s'$ and each honest node i will eventually hold a share $s_i = p(i)$. Moreover, when L is honest $s' = s$.

We also provide a protocol that improves the best known dual-threshold ACSS scheme. A dual-threshold ACSS scheme is formally defined as follows, and sometimes referred to as high-threshold ACSS as well.

Definition 5.3 (Dual-threshold ACSS). A (n, ℓ, t) dual-threshold ACSS with n nodes is an ACSS scheme with the additional property that the reconstruction threshold ℓ can be any value in the range $[t + 1, n - t]$.

The advantage of dual-threshold ACSS with $\ell > t$ is that, the protocol can ensure secrecy of the secret even when the adversary corrupts more than t but less than ℓ nodes in the system. However, to ensure termination, we need $\ell \leq n - t$. When $n = 3t + 1$, ℓ can be as large as $n - t = 2t + 1$.

Remark. By definition, ACSS solves AVSS as ACSS additionally requires every honest node to hold the corresponding share once the sharing phase terminates, and Dual-threshold ACSS solves ACSS as it additionally ensures a higher threshold $t < \ell \leq n - t$ for the adversary to break the secrecy and learn some information about the secret.

5.2 Asynchronous Verifiable Secret Sharing

Our construction of AVSS has a total communication cost of $O(\kappa n^2)$ and does not require any trusted setup. Briefly, our AVSS protocol has a similar structure as our RBC extension protocol. We use a Bracha's RBC on the digest of a Pedersens' polynomial commitment to set up the initial condition for ADD; we then invoke ADD to replicate the polynomial commitment. The AVSS algorithm is given in Algorithm 4.

Our AVSS scheme is inspired by the VSS scheme of [47] and crucially uses Pedersen's polynomial commitment scheme that provides two interfaces: $\mathsf{PedPolyCommit}$ and $\mathsf{PedEvalVerify}$. Let pp be the public parameters. We describe its interfaces next and provide the details in Appendix C.

- $\mathsf{PedPolyCommit}(pp, p(\cdot), t, n) \rightarrow \boldsymbol{v}, \boldsymbol{s}, \boldsymbol{r}$: The $\mathsf{PedPolyCommit}(\cdot)$ algorithm takes as input a t-degree polynomial $p(\cdot)$ and outputs three vectors each consisting of n elements. The vector $\boldsymbol{v}$ is the commitment to the polynomial and vectors $\boldsymbol{s}$ and $\boldsymbol{r}$ consists of shares of every nodes.
- $\mathsf{PedEvalVerify}(pp, \boldsymbol{v}, i, s_i, r_i) \rightarrow 0/1$: The $\mathsf{PedEvalVerify}(\cdot)$ algorithm takes as input a commitment $\boldsymbol{v}$ to a polynomial $p(\cdot)$, a tuple (i, s_i, r_i) and outputs 1 only if $p(i) = s_i$, and 0 otherwise.

During the sharing phase of our AVSS, the dealer L with the secret s first samples a random t-degree polynomial $p(\cdot)$ such that $p(0) = s$. The dealer then calls $\mathsf{PedPolyCommit}(\cdot)$ to obtain a commitment vector $\boldsymbol{v}$ as well as share vectors $\boldsymbol{s}, \boldsymbol{r}$ (each of size $O(n\kappa)$). Then, L sends node j $\boldsymbol{v}$ and the j^{th} element of vector $\boldsymbol{s}$ and $\boldsymbol{r}$ using private messages. In particular, L sends $\langle \mathtt{PROPOSE}, \boldsymbol{v}, \boldsymbol{s}[j], \boldsymbol{r}[j] \rangle$ to node j.

Upon receiving $\langle \mathtt{PROPOSE}, \boldsymbol{v}, s_j, r_j \rangle$ from the dealer, node j checks whether (j, s_j, r_j) is a valid share or not using $\mathsf{PedEvalVerify}(\cdot)$. Upon successful verification, j computes $h = \mathsf{hash}(\boldsymbol{v})$ and multicasts $\langle \mathtt{ECHO}, h \rangle$ message to all. The remaining steps of our AVSS sharing phase is identical to our RBC extension protocol.

We remark the latter part of the sharing phase (starting from line 201) is almost an RBC extension protocol. But there is a minor but

Algorithm 4 AVSS

PUBLIC PARAMETER: pp of Pedersen's polynomial commitment

```
SHARING PHASE:
    // As dealer $L$ with input $s$:
101: Sample a $t$-degree random polynomial $p(\cdot)$ such that $p(0) = s$
102: $\boldsymbol{v}, \boldsymbol{s}, \boldsymbol{r} \leftarrow$ PedPolyCommit($pp, p(\cdot), t, n$)
103: for $i = 1, 2, ..., n$ do
104:     send ⟨PROPOSE, $\boldsymbol{v}$, $\boldsymbol{s}[i]$, $\boldsymbol{r}[i]$⟩ to node $i$
    // As node $i$:
201: upon receiving ⟨PROPOSE, $\boldsymbol{v}, s_i, r_i$⟩ do
202:     if PedEvalVerify($pp, \boldsymbol{v}, i, s_i, r_i$) then
203:         let $h$ := hash($\boldsymbol{v}$)
204:         send ⟨ECHO, $h$⟩ to all (if haven't yet)
205: upon receiving $2t + 1$ ⟨ECHO, $h$⟩ messages do
206:     send ⟨READY, $h$⟩ to all (if haven't yet)
207: upon receiving $t + 1$ ⟨READY, $h$⟩ messages do
208:     send ⟨READY, $h$⟩ to all (if haven't yet)
209: upon receiving $2t + 1$ ⟨READY, $h$⟩ messages do
210:     if received ⟨PROPOSE, $\boldsymbol{v}, s_i, r_i$⟩ and $h$ = hash($\boldsymbol{v}$) then
211:         ADD($\boldsymbol{v}$)
212:     else
213:         ADD($\perp$)

RECONSTRUCTION PHASE:
    // every node $i$ after finishing the sharing phase
301: if PedEvalVerify($pp, \boldsymbol{v}, s_i, r_i$) then
302:     send ⟨RECONSTRUCT, $s_i, r_i$⟩ to all
303: upon receiving ⟨RECONSTRUCT, $s_j, r_j$⟩ from node $j$ do
304:     if PedEvalVerify($pp, \boldsymbol{v}, s_j, r_j$) then
305:         $T = T \cup \{s_j\}$
306:         if $|T| \geq t + 1$ then
307:             output $s$ using Lagrange interpolation and return
```

important difference. In RBC extension, a node echoes whatever it receives from the broadcaster; here in AVSS, however, every honest node must verify the Petersen's commitment it receives from the dealer. Therefore, although it may be helpful to think of the sharing phase as an RBC extension to aid understanding, we have to repeat those steps in Algorithm 4 for rigor. The same situation occurs in subsequent sections as well.

In the reconstruction phase, each node that received a valid share from L during the sharing phase multi-casts ⟨RECONSTRUCT, i, s_i, r_i⟩ to all other nodes. Each node upon receiving ⟨RECONSTRUCT, j, s_j, r_j⟩ from node j, checks whether (j, s_j, r_j) is valid using PedEvalVerify($\cdot$). Upon receiving $t + 1$ valid shares, a node reconstructs the secret using Lagrange interpolation.

We prove our AVSS protocol guarantees termination, correctness, and secrecy against all PPT adversaries.

Lemma 5.4 (Termination). *Assuming a collision resistant hash function, the AVSS scheme in Algorithm 4 guarantees termination.*

Proof. From Lemma 4.6 and 3.4, whenever an honest node terminates the sharing phase, each honest node will eventually terminate the sharing phase. Furthermore, from Lemma 4.4 and the fact that nodes only send ECHO messages if their PedEvalVerify($\cdot$) is successful (line 202), we get that if the sharing phase terminates at an honest node, then PedEvalVerify($\cdot$) was successful for at least $t + 1$ honest node. Without loss of generality, let S be this set of honest nodes. Then, Lemma C.1 implies that the share of nodes in S are sufficient to reconstruct the secret s. During the reconstruction phase, since each honest node will eventually receive RECONSTRUCT message from all nodes in S, each honest node will eventually output a secret. Hence, the AVSS scheme ensures termination. □

Lemma 5.5 (Correctness). *Assuming a collision resistant hash function, the AVSS scheme in Algorithm 4 guarantees correctness.*

Proof. When the dealer is honest, the secret reconstructed using the shares of only honest nodes is clearly equal to s. Additionally, note that during the reconstruction phase, an honest node only accepts shares for which PedEvalVerify($\cdot$) is successful. Hence, the uniqueness property (Theorem C.3) of the Pedersen's VSS protocol implies that whenever an honest node outputs a secret s', $s' = s$.

When the sharing phase terminates for a malicious dealer, from Lemma 3.4, every honest nodes outputs the same commitment vector $\boldsymbol{v}$. Additionally, Lemma 4.4 together with the fact that nodes only send ECHO messages if their PedEvalVerify($\cdot$) is successful (line 202), implies that the PedEvalVerify($\cdot$) associated with $\boldsymbol{v}$ was successful for at least $t + 1$ honest node. By Lemma C.1, these honest nodes can recover an appropriate secret. Thus, using the same argument as above, every honest node reconstructs the same secret. □

Observe that the view of an adversary in our AVSS scheme is identical to the view of the adversary in Pedersen's VSS [47]. Hence, the secrecy of our AVSS protocol follows from Theorem C.4.

5.3 Asynchronous Complete Secret Sharing

We can also extend our AVSS scheme to ensure the completeness guarantees described in definition 5.2 using the *encrypt-then-disperse* technique from [38, 55]. In doing so, the total communication cost increases only by a small constant factor. Since our technique of extending our AVSS scheme to a ACSS scheme uses techniques [55] in a straightfoward fashion, we will only describe this part briefly below and refer the reader to [55].

During the sharing phase, the dealer additionally computes a ciphertext vector $\boldsymbol{c}$ that consists of encryptions of shares (s_i, r_i) of each node i.

$$\boldsymbol{c} = \{c_1, \ldots, c_n\} = \{\mathsf{Enc}_{pk_1}(s_1, r_1), \ldots \mathsf{Enc}_{pk_n}(s_n, r_n)\} \qquad (3)$$

Here, pk_i is the public key of node i, $\mathsf{Enc}_{pk_i}(x)$ denotes a CPA secure public key encryption of x using public key pk_i.

The dealer then also sends $\boldsymbol{c}$ along with the PROPOSE message. Note that since the dealer is sending $\boldsymbol{c}$ anyway with the proposal, the dealer no longer needs to send the plaintext shares. Upon receiving $\boldsymbol{c}$, each node decrypts its plaintext share using its secret key and validates it using PedEvalVerify($\cdot$). As in [55], if the decrypted share turns out to be invalid, then the node uses it as evidence to implicate the dealer. When the dealer is determined to be faulty, each node enters a share recovery phase, which ensures that each node receives its share. It is important to note that during both the leader implication step and the share recovery step, each node only multi-casts $O(\kappa)$ bits to all other nodes.

Now we give a proof sketch to this ACSS protocol. Termination and correctness of our ACSS scheme follow directly from the termination and correctness guarantees of our AVSS scheme. For secrecy, note that when the dealer is honest, the share recovery phase is not invoked. Hence, when the dealer is honest, before any honest node starts reconstruction of the secret, the encryption hides the shares and the view of the adversary in our ACSS scheme is computationally indistinguishable to to the view of the adversary in AVSS. Hence ACSS also ensures secrecy.

Lastly, completeness is clear when the dealer is honest. When the dealer is malicious, completeness follows from the fact that a single genuine blame is sufficient to initiate the recovery protocol. Hence, the recovery protocol ensures that every honest node reconstructs the committed polynomial and the corresponding secret.

5.4 Dual-threshold ACSS

We will first describe a dual-threshold ACSS for uniform secrets. The term "uniform secret" [3, 21, 51] refers to the fact that the dealer can only share uniformly random secrets. In particular, given a group $\mathbb{G}$ of a prime order q with a randomly chosen generator g_1, the schemes allow the dealer to share a secret of the form g_1^s where $s \in \mathbb{Z}_q$ is chosen uniformly at random. However, using techniques from [51], we can extend the scheme to allow the dealer to share an arbitrary secret in $\mathbb{Z}_q$.

Our dual-threshold ACSS scheme in Algorithm 5 is also publicly-verifiable, i.e., any external verifier, who need not be a participant, can check that the dealer L acted honestly without learning any information about the shares or the secret.

Before we provide details of our dual-threshold ACSS, we want to discuss why we cannot extend our ACSS scheme described in §5.3 to tolerate a higher reconstruction threshold. More concretely, what would go wrong if $p(\cdot)$ is a ℓ-degree polynomial for $\ell > t + 1$? The issue is that, in our ACSS protocol, it is possible that the sharing phase terminates, and only $t + 1$ honest nodes received their shares but all honest nodes initiate the recovery protocol. In such situation, if the reconstruction threshold $\ell > t + 1$, there are not enough valid shares to recover $p(0)$.

Next, we describe our dual-threshold ACSS protocol and summarize it in Algorithm 5.

For any given $n \geq 3t + 1$, our scheme can tolerate any reconstruction threshold ℓ in the range $t + 1 \leq \ell \leq n - t$. Our dual-threshold ACSS scheme internally uses a PVSS scheme. For concreteness, we will use the PVSS scheme from Scrape [21], which is secure assuming the existence of a Random Oracle and the hardness of the Decisional Diffie-Hellman problem. If one wishes to avoid the Random Oracle, one can use the recent PVSS scheme from [25], which is secure assuming the hardness of Bilinear decisional Diffie-Hellman (DBDH) problem.

We briefly summarize the interface of PVSS. Let pp be the public parameters.

- PVSS.Share$(pp, s, \ell, n) \longrightarrow \boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}$: For a uniform random $s \in \mathbb{F}$, the vector $\boldsymbol{v}$ is a commitment to a degree-ℓ random polynomial with $p(0) = s$. The vector $\boldsymbol{c}$ consists of encrypted shares of each node. The vector $\boldsymbol{\pi}$ consists of non-interactive zero-knowledge proofs that each encryption in $\boldsymbol{c}$ is a correct encryption of shares of s.

Algorithm 5 Dual-threshold ACSS for uniform secrets

PUBLIC PARAMETERS: $pp := (n, t, \ell, g_0, g_1, \{pk_i\})$ for $i = 1, 2, \ldots, n$
PRIVATE PARAMETERS: Node i has secret key sk_i i.e., $pk_i = g_1^{sk_i}$

```
SHARING PHASE:
     // As dealer L with input s:
101: c, v, π ← PVSS.Share(pp, s, ℓ, n)
102: for i = 1, 2, ..., n do
103:     send ⟨PROPOSE, c, v, π⟩ to node i
     // As receiver i:
201: upon receiving ⟨PROPOSE, c, v, π⟩ do
202:     if PVSS.Verify(pp, ℓ, c, v, π) then
203:         Let h := hash(v‖c)
204:         send ⟨ECHO, h⟩ to all (if haven't yet)
205: upon receiving 2t + 1 ⟨ECHO, h⟩ messages do
206:     send ⟨READY, h⟩ to all (if haven't yet)
207: upon receiving t + 1 ⟨READY, h⟩ messages do
208:     send ⟨READY, h⟩ to all (if haven't yet)
209: upon receiving 2t + 1 ⟨READY, h⟩ messages do
210:     if received ⟨PROPOSE, c, v, π⟩ and h = hash(v‖c) then
211:         ADD(v‖c)
212:     else
213:         ADD(⊥)

RECONSTRUCTION PHASE:
     // every node i with key pk_i, sk_i
301: s̃_i := c[i]^{sk_i}; π̃ := dleq.Prove(sk_i, g_1, pk_i, c[i], s̃_i);
302: send ⟨RECONSTRUCT, s̃_i, π̃_i⟩ to all
303: upon receiving ⟨RECONSTRUCT, s̃_j, π̃_j⟩ from node j do
304:     if dleq.Verify(π̃, g_1, pk_j, c[j], s̃_j) then
305:         T = T ∪ {s̃_j}
306:         if |T| ≥ ℓ then
307:             g_1^s := PVSS.Recon(T)
308:             output g_1^s and return
```

- PVSS.Verify$(pp, \ell, \boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}) \longrightarrow 0/1$: The PVSS.Verify function takes in the tuple $(\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi})$ and outputs 1 only if $\boldsymbol{v}$ is a commitment to a ℓ-degree polynomial $p(\cdot)$. Each component of $\boldsymbol{c}$ is a valid encryption of shares of $p(0)$.

Our dual-threshold ACSS make use of a non-interactive protocol for checking equality of discrete logarithm. In particular, given a group $\mathbb{G}$ of prime order q, two uniformly random generators $g_0, g_1 \in \mathbb{G}$ and a tuple (g_0, x, g_1, y), a prover $\mathcal{P}$ wants to prove to a PPT verifier $\mathcal{V}$, in zero-knowledge, that there exists an witness α such that $x = g_0^\alpha$ and $y = g_1^\alpha$. We describe the interfaces of a protocol that achieve this functionality next and provide the detailed protocol in Appendix D.

- dleq.Prove$(\alpha, g_0, x, g_1, y) \longrightarrow \pi$: Given tuple (g_0, x, g_1, y) and α where $\alpha = \log_{g_0} x = \log_{g_1} y$, dleq.Prove outputs an non-interactive zero-knowledge proof π that such an α exists.
- dleq.Verify$(\pi, g_0, x, g_1, y) \longrightarrow 0/1$: Given a proof π and a tuple (g_0, x, g_1, y), dleq.Verify outputs 1 if $\log_{g_0} x = \log_{g_1} y$, and 0 otherwise.

The dealer L with a uniform random secret $s \in \mathbb{F}$, first computes the PVSS shares for the secret using PVSS.Share. Let $(\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi})$

be the three vectors output by the PVSS.Share$(\cdot)$. Next, L multi-casts $\langle \texttt{PROPOSE}, \boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi} \rangle$ to all nodes. Each node upon receiving the proposal from L validates shares of every node using PVSS.Verify. Upon successful verification, each node multi-cast $\langle \texttt{ECHO}, \mathsf{hash}(\boldsymbol{v} \| \boldsymbol{c}) \rangle$ to all other nodes. After which the protocol has a structure similar to our AVSS protocol.

In the reconstruction phase, each node decrypts its share using its secret key and generates a non-interactive zero-knowledge proof of correct decryption using dleq.Prove algorithm. Let $\tilde{s}_i$ and $\tilde{\pi}_i$ be the decrypted share and its correctness proof of node i, respectively. Node i then multi-casts $\langle \texttt{RECONSTRUCT}, i, \tilde{s}_i, \tilde{\pi}_i \rangle$ to all nodes. Each node upon receiving a RECONSTRUCT message validates it for correctness using the dleq.Verify algorithm. Finally, after receiving ℓ valid RECONSTRUCT message, each honest node reconstructs h^s using Lagrange interpolation.

Dual-threshold ACSS for arbitrary secrets Although our dual-threshold ACSS in Algorithm 5 only supports uniform secrets, it can be extended to arbitrary secrets $z \in \mathbb{Z}_q$ using techniques from [51]. Moreover, z need not be uniformly distributed over $\mathbb{Z}_q$. Briefly, to share a secret $z \in \mathbb{Z}_q$, the dealer first runs the sharing phase of Algorithm 5 for a random secret g_1^s along with a RBC on $z \cdot h^{-s}$. Upon reconstructing h^s, each honest node can use this to recover z. The security of this approach require the DDH assumption. We will refer the reader to [51] for more details on this approach.

We next analyze our dual-threshold ACSS protocol.

Lemma 5.6 (Termination). *Our dual-threshold ACSS protocol in Algorithm 5 guarantees termination for any $\ell \le n - t$.*

Proof. When the dealer is honest, then the termination property of our dual-threshold ACSS follows directly from the completeness property of the dleq.Prove protocol and the fact the PVSS.Verify check will be successful at every honest node.

Alternatively, when the dealer is malicious, and the sharing phase terminates at an honest node., then a similar argument as Lemma 5.4 implies that every honest node will eventually terminate the sharing phase. Furthermore, from Lemma 4.2, the PVSS.Share check successful in at least $t + 1$ honest node. Hence, except with probability $1 - \binom{2t+1}{t+1}\frac{1}{q^{t+1}}$, $\boldsymbol{v}$ is a commitment to a degree ℓ polynomial. Furthermore, all these nodes check that shares of every other node encrypted as per the protocol specification. Hence, except with negligible probability, $\boldsymbol{c}$ is generated as per the protocol specification. Also, our ADD protocol guarantees that every node will eventually output $\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}$, hence, each honest will eventually receive their encrypted shares and will multi-cast it during the reconstruction phase. For any given $\ell \le n - t$, since there are at least ℓ honest nodes in the system, this implies that during the reconstruction phase, each honest node will receive at least $n - t$ valid decrypted shares and will terminate the Reconstruction phase of the protocol. □

Lemma 5.7 (Correctness). *Assuming* dleq.Prove *ensures soundness, our dual-threshold ACSS protocol guarantees correctness for any $\ell \le n - t$.*

Proof. During the reconstruction phase of the protocol, the soundness guarantee of dleq.Prove ensures that each honest node only accepts valid decrypted shares. Furthermore, all of these points lie on a fixed degree-ℓ polynomial, which gets finalized whenever the sharing phase terminates at an honest node. Also, any set of ℓ valid shares will result in the same output. □

Lemma 5.8 (Completeness). *The dual-threshold ACSS in Algorithm 5 ensures completeness.*

Proof. When the dealer is honest, completeness follows from the completeness property of dleq.Prove, i.e., an honest prover can always convince an honest verifier. When the dealer is malicious, at the end of sharing phase, from the guarantees of our ADD protocol, every honest node will eventually receive $\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}$. Furthermore, Lemma 4.4 implies that at least $t + 1$ nodes validated $\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}$ for correctness. Hence, by soundness guarantees of dleq.Prove, $\boldsymbol{c}$ consists of correct encryptions of shares of each node. Putting all these together, we get that Algorithm 5 guarantees completeness. □

Lastly, when the secret is a uniform random element $g_1^s \in \mathbb{G}$, IND1-Secrecy property (Definition E.2) of Scrape's PVSS implies that to an adversary that corrupts up to ℓ, assuming hardness of DDH, g_1^s is indistinguishable from a randomly chosen element. Hence, when we use it as a one time pad in the transformation of [51], our dual-threshold ACSS ensures secrecy.

Remark. Unlike existing dual-threshold ACSS such as [3, 17, 37], our dual-threshold ACSS ensures secrecy even in scenarios where adversary corrupts more than t nodes during both sharing and reconstruction phase. However, for termination of the sharing phase, we require that adversary corrupts only up to t nodes during the sharing phase and $n - \ell$ nodes during the reconstruction phase.

6 ASYNCHRONOUS DISTRIBUTED KEY GENERATION

In this section, we present a protocol for asynchronous distributed key generation (ADKG). Briefly, the goal of the ADKG protocol is to generate a public key of the form g^s where g is a generator of an appropriate group, s is a secret, and nodes hold threshold shares of the secret s. These shares can be used as secret keys for applications such as threshold signatures. One crucial property we want from the DKG protocol is that an adversary that corrupts up to t nodes in the system does not learn any information about the secret except the public key g^s. We refer readers to [32, 37] for more detailed definition of the (asynchronous) distributed key generation.

Prior to our work, the most efficient ADKG protocol that achieves these properties is due to Kokoris et al. [37]. In particular, Kokoris et al. provide a generic technique to convert a dual-threshold ACSS scheme with some additional properties (which we describe later) to an ADKG. Here on we will refer to this transformation as the KMS transformation. The KMS transformation uses n distinct invocation of the dual-threshold ACSS, where each node acts as a dealer in one invocation. Thus, if we instantiate the KMS transformation with the dual-threshold ACSS from [3], we get an ADKG protocol with communication cost of $O(\kappa n^3 \log n)$.

The additional properties the KMS transformation needs from a dual-threshold ACSS are:

(1) **Homomorphism.** For two secrets u and v, possibly shared by two different dealers, let u_i and v_i be the shares held by node i,

then there is an efficient homomorphic operation $\boxplus$ such $u_i \boxplus v_i$ is the i^{th} share of the secret $u + v$.
(2) **Knowledge soundness.** If the sharing phase terminates for a secret $s \in \mathbb{Z}_q$, then the dealer knows s.

The KMS transformation needs these properties because it will aggregate secrets from multiple dealers. Hence, it naturally needs homomorphism. It also requires knowledge soundness since we do not want a malicious node to select a secret that depends on the secrets of honest dealers.

Note that our dual-threshold ACSS scheme in Algorithm 5 do not satisfy homomorphism. Next, we introduce some modification to Algorithm 5 to achieve both homomorphism and knowledge soundness while retaining the communication cost of $O(\kappa n^2)$. After that, we can plug it into the KMS transformation to get an ADKG scheme with communication cost of $O(\kappa n^3)$.

Briefly, we need *verifiable encryption of discrete logarithms*, i.e., a CPA secure encryption scheme that allows an encrypter to prove in zero-knowledge about the correct encryption of discrete logarithm of a known value. We know of such an instantiation by Camenisch and Shoup [18]. Their construction assumes the hardness of strong RSA [5, 29] and DCR [44]. Next, we will introduce the verifiable encryption of discrete logarithm problem and its interface.

6.1 Verifiable Encryption of Discrete Logs

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

The protocol due to Camenisch and Shoup [18] for verifiable encryption of discrete logarithm is a "Σ-protocol" and is zero-knowledge and knowledge sound. In particular, the protocol has the following interfaces.

- CS.KeyGen(1^κ) $\to (pk, sk)$. The key generation algorithm outputs a public-private key pair for the encryption scheme.
- CS.Encrypt(pk, α) $\to c$: Given a public key pk, a message α, CS.Encrypt computes the public key encryption of α.
- CS.Decrypt(sk, c) $\to \alpha$: The CS.Decrypt function decrypts a ciphertext c using the secret key sk and outputs the message α.
- CS.EncAndProve(pk, α, g) $\to (c, x, \pi)$: CS.EncAndProve function encrypts a uniformly randomly chosen message α using the public-key pk, compute g^α, and creates a non-interactive zero-knowledge proof of knowledge π of the statement that the $\mathcal{P}$ knows α such that c = CS.Encrypt(pk, α) and $x = g^\alpha$.
- CS.VerifyDLog(pk, g, x, c, π) $\to 0/1$. Given the tuple (pk, g, x, c, π), the CS.VerifyDLog($\cdot$) outputs 1 if there exists α such that $x = g^\alpha \wedge c$ = CS.Encrypt(pk, α). Note that CS.VerifyDLog needs to do these checks without using the secret key or the underlying message. The proof π assists in that.

6.2 Design and Analysis

We summarize the modifications to our dual-threshold ACSS to achieve the properties required for the KMS transformation in Algorithm 6. The main difference between Algorithm 5 and Algorithm 6 is the way the dealer encrypts the shares of each node computes the corresponding zero-knowledge proofs, etc.

Algorithm 6 Homomorphic dual-threshold ACSS for ADKG

```
PUBLIC PARAMETER: $\{pk_i\}$ for $i = 1, 2, \ldots, n$ public keys of each node as per encryption scheme of [18].
INPUT: $n, t, \ell$

SHARING PHASE:
    // As dealer $L$ with a uniform random input $s$:
101: Sample a $k$-degree random polynomial $p(\cdot)$ such that $p(0) = s$
102: Let $v_j, c_j, \pi_j \leftarrow$ CS.EncAndProve($pk_j, g, p(j)$) for $j = 1, 2, \ldots, n$.
103: Let $\boldsymbol{v} = \{v_1, v_2, .., v_n\}$, $\boldsymbol{c} = \{c_1, c_2, .., c_n\}$, and $\boldsymbol{\pi} = \{\pi_1, \pi_2, .., \pi_n\}$.
104: for $i = 1, 2, ..., n$ do
105:     send ⟨PROPOSE, $\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}$⟩ to node $i$
    // As receiver $i$:
201: upon receiving ⟨PROPOSE, $\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}$⟩ do
202:     Sample a random code word $y^\perp \in C^\perp$ and check whether
             $\prod_{k=1}^{n} v_k^{x_k^\perp} = 1_{\mathbb{G}}$
203:     if CS.VerifyDLog($\boldsymbol{v}[j], \boldsymbol{c}[j], \boldsymbol{\pi}[j]$) is valid for all $j$ then
204:         let $h :=$ hash($\boldsymbol{v} \| \boldsymbol{c}$)
205:         send ⟨ECHO, $h$⟩ to all (if haven't yet)
    // the rest of the sharing phase is identical to line 205 to 213 of Algorithm 5.

RECONSTRUCTION PHASE:
    // every node $i$ with key $pk_i, sk_i$
301: $\tilde{s}_i :=$ CS.Decrypt($sk_i, \boldsymbol{c}[i]$)
302: send ⟨RECONSTRUCT, $\tilde{s}_i$⟩ to all
303: upon receiving ⟨RECONSTRUCT, $\tilde{s}_i$⟩ from node $i$ do
304:     if $\boldsymbol{v}[i] = g^{\tilde{s}_i}$ then
305:         $T = T \cup \{\tilde{s}_i\}$
306:         if $|T| \geq \ell$ then
307:             output $s$ using Lagrange interpolation and return
```

During the sharing phase, to share a uniform random secret $s \in \mathbb{Z}_q$, the dealer L samples a random ℓ-degree polynomial $p(\cdot)$ such that $p(0) = s$. Then for each $j \in \{1, 2, \ldots, n\}$, L computes $(v_j, c_j, \pi_j) \leftarrow$ CS.EncAndProve($g, pk_j, p(j)$). Let

$$\boldsymbol{v} = \{v_1, \ldots, v_n\}; \boldsymbol{c} = \{c_1, \ldots, c_n\}; \text{ and } \boldsymbol{\pi} = \{\pi_1, \ldots, \pi_n\} \quad (4)$$

We will refer to $\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}$ as the commitment, encryption and proof vector, respectively. Observe that the commitment vector is same as the commitment vector of PVSS.Verify. Then the dealer multi-casts the message ⟨PROPOSE, $\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}$⟩ to all other nodes.

Each node i upon receiving ⟨PROPOSE, $\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}$⟩ checks while $\boldsymbol{v}$ is a commitment to a polynomial of degree at most ℓ using step (2) of PVSS.Verify. Then, for each tuple (v_j, c_j, π_j) node i checks whether c_j is an encryption of $\log_g v_j$ or not using CS.VerifyDLog. If all the validation checks pass, node i computes h = hash($\boldsymbol{v} \| \boldsymbol{c}$) and multi-casts the ⟨ECHO, h⟩ to all other nodes. The rest of the sharing phase then has the same structure as in our Algorithm 5.

During the reconstruction phase, each node i decrypt $c[i]$ to recover its share $\tilde{s}_i$ i.e., $\tilde{s}_i :=$ CS.Decrypt($sk_i, c[i]$)). Node i then multi-casts $\tilde{s}_i$ to all other nodes as ⟨RECONSTRUCT, $i, \tilde{s}_i$⟩ message. A node j upon receiving ⟨RECONSTRUCT, $i, \tilde{s}_i$⟩ from node j, checks whether $\boldsymbol{v}[j]$ equals $g^{\tilde{s}_j}$, and accepts it only if the check pass. Lastly, after receiving ℓ or more valid RECONSTRUCT messages node j reconstructs the secret using Lagrange interpolation.

Now let's look at the properties ensured by Algorithm 6. The termination of our modified dual-threshold ACSS follows from Lemma 5.6 and completeness property of the CS.EncAndProve. Similarly, the completeness property follows from the soundness guarantee of CS.EncAndProve and that ADD ensures that all honest nodes eventually receive $\boldsymbol{v}\|\boldsymbol{c}$. The correctness property follows directly from Lemma 5.7.

For secrecy, we argue that, for a uniform secret s, an adversary learns nothing about s beyond whatever is revealed by the public key g^s assuming the hardness of Computational Diffie-Helmann. Note that this secrecy is slightly weaker than that the one Definition 5.1, which does not leak any information about s. But this is the natural definition of secrecy for DKG and is sufficient for applications such as [10, 13, 27].

Lemma 6.1. *For a uniformly random s, given g^s, assuming hardness of Strong RSA and DCR, and the existence of a Random Oracle, there exists an PPT simulator that can simulate the view of all static PPT adversaries.*

We only provide a proof sketch for Lemma 6.1. Let $\mathcal{A}$ be a PPT adversary that corrupts up to ℓ nodes. Without loss of generality, let $\mathcal{A}$ corrupts the first ℓ nodes. Let pk_i for $i = 1, 2, \ldots, n$ be the public keys of the nodes. Given g^s for a random secret s, the simulator $\mathcal{S}$ chooses ℓ random points $s_i \in \mathbb{Z}_q$ for $i = 1, 2, \ldots, \ell$ and sets $v_i = g_1^{s_i}$. For $\ell < i \leq n$, the $\mathcal{S}$ constructions v_i using Lagrange interpolation in the exponent. $\mathcal{S}$ then encrypts the share of the each node $j \in \{1, 2, \ldots, \ell\}$ and for the remaining nodes, $\mathcal{S}$ uses the CPA security of CS.Encrypt and send encryptions of all zero string. Next, $\mathcal{S}$ uses the zero-knowledge simulator of the CS.EncAndProve to construct the proofs π_j for $\ell < j \leq n$. It is easy to see that the view of $\mathcal{A}$ in its interaction with the simulator is computationally indistinguishable from its view in the actual protocol.

Now, let's analyze the communication cost of our modified dual-threshold ACSS. Observe that each $\boldsymbol{v}, \boldsymbol{c}$ and $\boldsymbol{\pi}$ are κn bits long. Hence, the communication cost of the sharing phase in $O(\kappa n^2)$. Similarly, during the reconstruction phase, each node multi-cast $O(\kappa)$ bits, the reconstruction phase has a communication cost of $O(\kappa n^2)$. Hence, the total communication cost is $O(\kappa n^2)$. Since KMS transformation involves n invocation of dual-threshold ACSS we get a ADKG protocol with communication cost of $O(\kappa n^3)$.

7 RELATED WORK

To the best of our knowledge, the ADD problem has not been studied before. This may be in part because, despite being a simple primitive, its applications are not immediately apparent. In hindsight, the biggest conceptual barrier for us in this work was to realize the usefulness of ADD as opposed to designing protocols to solve it. Even then, for many of the applications we have identified for ADD in this work, we had to introduce additional techniques to address other efficiency bottlenecks in them.

Reliable broadcast. The problem of reliable broadcast (RBC) was introduced by Bracha [14]. In the same paper, Bracha provided an RBC protocol for a single bit with a communication cost of $O(n^2)$, thus $O(n^2|M|)$ for $|M|$ bits using a naïve approach. Almost two decades later, Cachin and Tessaro [17] improved the cost to $O(n|M|+\kappa n^2 \log n)$ assuming a collision-resistant hash function with κ being the output size of the hash. Hendricks et al. in [35] propose an alternate RBC extension protocol with a communication cost of $O(n|M|+\kappa n^3)$ using a novel erasure coding scheme where each element of a codeword can be verified for correctness.

Recently, assuming a trusted setup phase, hardness of q-SDH [11, 12] and Decisional bilinear Diffie-Hellman (DBDH) assumption [13], Nayak et al. [43] reduced the communication cost to $O(n|M|+\kappa n^2)$. Our RBC protocol achieves the best of both, i.e., it only assumes existence of collision resistant hash function, does not require a setup, and has a communication cost of $O(n|M|+\kappa n^2)$.

Asynchronous VSS/CSS. The problem of asynchronous verifiable secret sharing has been studied for decades in many different settings [3, 4, 6, 7, 16, 19, 20, 37, 45, 55]. The information-theoretically secure schemes [19, 20, 24, 45, 46] mostly have high communication cost or sub-optimal fault tolerance. Using cryptographic assumption such as collision resistance hash function and Decisional Diffie-Hellman assumption, Cachin et al. [16] proposed a AVSS scheme with communication cost of $O(n^3\kappa)$, and later improved to $O(\kappa n^2)$ by Backes et al. in [4] assuming a trusted setup phase and the hardness of the q-SDH. Very recently, Alhaddad et al. [3] proposed a ACSS scheme for secrets chosen at random with a total communication cost of $O(\kappa n^2 \log n)$.

Some works focused on improving the amortized communication cost of AVSS for many secrets [3, 17, 55], such as amortized cost of $O(n^2\kappa)$ per secret in Cachin et al. [17], and $O(n\kappa)$ per secrets in both hbACSS [55] and Haven [3].

Dual-threshold ACSS. The problem of dual-threshold AVSS was introduced by Cachin and Tessaro [17] where they provide a dual-threshold for $t < n/4$ and $\ell < n/2$ with communication cost of $O(\kappa n^3)$. Only recently, Kokoris et al. [37] proposed the first dual-threshold ACSS protocol for $t < n/3$ and $\ell < 2n/3$ with a communication cost of $O(\kappa n^3)$ and Alhaddad et al. [3] improved the communication cost to $O(\kappa n^2 \log n)$. If we assume a trusted setup and use the polynomial commitment scheme of [36], then the total communication cost can be improved to $O(\kappa n^2)$. Moreover, during the sharing phase, all of the above mentioned schemes provide secrecy only against an adversary that corrupts up to t nodes. Contrary to the existing schemes, our dual-threshold ACSS always ensures secrecy against an adversary that corrupts up to ℓ nodes.

Asynchronous Distributed Key Generation. There are relatively fewer works on asynchronous DKG [2, 20, 30, 37]. The ADKG construction of Canetti and Rabin [20] uses n^2 AVSSand is hence inefficient. Kokoris et al, [37] uses n homomorphic dual-threshold ACSS with reconstruction threshold of $2t + 1$ to design an ADKG scheme. Their original paper [37] uses a $O(\kappa n^3)$ dual-threshold ACSS protocol and hence incur an expected cost of $O(\kappa n^4)$. Using our quadratic homomorphic dual-threshold ACSS, we obtain a ADKG with expected communication cost of $O(\kappa n^3)$. Two recent works [2, 30] propose weaker variant of the ADKG with limited applications. These schemes have total communication costs of $O(\kappa n^3 \log n)$ and they can be improved to $O(\kappa n^3)$ using our RBC extension in a black box manner.

8 DISCUSSION

Concrete communication cost of ADD. Although reducing the communication cost by a factor of $\log n$ is interesting from a theoretical point of view, it is equally important from a practical point of view primitives to make the hidden constants small. Indeed, this is the case with the primitives we construct in this work. In particular, our ADD with $n = 3t + 1$ has a concrete communication cost, $B_{\text{ADD}} = 6n|M| + 2n^2$. The factor 6 is due to the increase in the size of the message due to RSEnc and the fact that ADD has two rounds of communication. If we substitute B_{ADD} in our RBC extension protocol we get a communication cost of $n|M| + 2\kappa n^2 + B_{\text{ADD}}$. Here the $n|M|$ accounts for the communication cost the dealer incurs during multi-cast of M to all nodes. The $2\kappa n^2$ is the cost of Bracha's RBC on the $\mathsf{hash}(M)$.

A note on the κ. Although for notational convenience, we have used a single parameter κ for all cryptographic primitives, it is worth noting that for the same level of security, the sizes of group or field elements vary considerably depending upon the underlying cryptographic assumptions. One needs to factor this in when comparing concrete performance. For example, SHA256 outputs are 32 bytes and provide 128 bits of security. The elliptic curve group ed25519 has 32-byte group elements and also provides approximately 128 bits. However, for 128 bits of security, RSA group elements need to be 384 bytes long. As a result, our ADKG construction is not as efficient as our other constructions. We leave it to future work to try to remove RSA from our ADKG construction for better concrete efficiency.

Limitations of ADD. One limitation of ADD is that it may increase the number of rounds needed by two. For example, our RBC extension protocol requires five rounds whereas both [14] and [16] require only three rounds. We leave reducing the round complexity of our RBC extension as future work.

Another limitation of using ADD is the added computation it introduces. These costs are due to encoding and decoding of the message. Additionally, in the presence of malicious nodes, each honest node may have to try decoding t times. Since $t = \Theta(n)$, each honest node possibly needs to perform $O(n^2 \log n)$ extra field operations. The exact overhead can be tricky to calculate since prior protocols such as [16] involve more hash computation.

9 CONCLUSION

In this paper we have introduced the problem of asynchronous data dissemination (ADD), which seeks to replicate a data blob M from a subset of honest subset of honest nodes to *all* honest nodes, despite the presence of some malicious nodes. We have presented an ADD protocol for n parties with a communication cost of $O(n|M| + n^2)$. We then used our ADD protocol to improve the communication cost or trust assumption of RBC extension, AVSS, ACSS, dual-threshold ACSS, and ADKG. All our constructions have a common structure, consisting of a Bracha's RBC on a hash of a message M and an ADD for replicating a long message M.

We believe ADD can be useful in other applications that we did not study in this paper, e.g., in improving the communication cost of recent randomness beacon protocols for both synchronous [25] and asynchronous networks [8]. Generally speaking, ADD will be useful in protocols that involve distribution of long common messages across all nodes. These messages include but are not limited to blocks in blockchain protocols, polynomial commitments, encrypted shares, NIZK proofs, etc..

ACKNOWLEDGMENTS

The authors would like to thank Vinith Krishnan, Andrew Miller, and Tom Yurek for many helpful discussions related to the paper.

REFERENCES

[1] Ittai Abraham, TH Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2019. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 317–326.

[2] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. 2021. Reaching Consensus for Asynchronous Distributed Key Generation. *arXiv preprint arXiv:2102.09041* (2021).

[3] Nicolas Alhaddad, Mayank Varia, and Haibin Zhang. 2021. High-Threshold AVSS with Optimal Communication Complexity. Cryptology ePrint Archive, Report 2021/118. (2021). https://eprint.iacr.org/2021/118.

[4] Michael Backes, Amit Datta, and Aniket Kate. 2013. Asynchronous computational VSS with reduced communication complexity. In *Cryptographers' Track at the RSA Conference*. Springer, 259–276.

[5] Niko Barić and Birgit Pfitzmann. 1997. Collision-free accumulators and fail-stop signature schemes without trees. In *International conference on the theory and applications of cryptographic techniques*. Springer, 480–494.

[6] Soumya Basu, Alin Tomescu, Ittai Abraham, Dahlia Malkhi, Michael K Reiter, and Emin Gün Sirer. 2019. Efficient Verifiable Secret Sharing with Share Recovery in BFT Protocols. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2387–2402.

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

[8] Adithya Bhat, Nibesh Shrestha, Aniket Kate, and Kartik Nayak. 2020. RandPiper–Reconfiguration-Friendly Random Beacons with Quadratic Communication. (2020).

[9] George Robert Blakley. 1979. Safeguarding cryptographic keys. In *1979 International Workshop on Managing Requirements Knowledge (MARK)*. IEEE, 313–318.

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

[11] Dan Boneh and Xavier Boyen. 2004. Short signatures without random oracles. In *International conference on the theory and applications of cryptographic techniques*. Springer, 56–73.

[12] Dan Boneh and Xavier Boyen. 2008. Short signatures without random oracles and the SDH assumption in bilinear groups. *Journal of cryptology* 21, 2 (2008), 149–177.

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

[14] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (1987), 130–143.

[15] Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. 2017. Proofs-of-delay and randomness beacons in ethereum. *IEEE Security and Privacy on the blockchain (IEEE S&B)* (2017).

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

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

[18] Jan Camenisch and Victor Shoup. 2003. Practical verifiable encryption and decryption of discrete logarithms. In *Annual International Cryptology Conference*. Springer, 126–144.

[19] Ran Canetti. 1996. *Studies in secure multiparty computation and applications*. Ph.D. Dissertation. Citeseer.

[20] Ran Canetti and Tal Rabin. 1993. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 42–51.

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

[22] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. 173–186.

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

[24] Ashish Choudhury. 2020. *Optimally-resilient unconditionally-secure asynchronous multi-party computation revisited.* Technical Report. Cryptology ePrint Archive, Report 2020/906, 2020. https://eprint. iacr. org ....

[25] Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. 2021. SPURT: Scalable Distributed Randomness Beacon with Transparent Setup. Cryptology ePrint Archive, Report 2021/100. (2021). https://eprint.iacr.org/2021/100.

[26] Sisi Duan, Michael K Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2028–2041.

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

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

[29] Eiichiro Fujisaki and Tatsuaki Okamoto. 1997. Statistical zero knowledge protocols to prove modular polynomial relations. In *Annual International Cryptology Conference*. Springer, 16–30.

[30] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. 2019. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 214–228.

[31] Shuhong Gao. 2003. A new algorithm for decoding Reed-Solomon codes. In *Communications, information and network security*. Springer, 55–68.

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

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

[34] Somayeh Heidarvand and Jorge L Villar. 2008. Public verifiability from pairings in secret sharing schemes. In *International Workshop on Selected Areas in Cryptography*. Springer, 294–308.

[35] James Hendricks, Gregory R Ganger, and Michael K Reiter. 2007. Verifying distributed erasure-coded data. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. 139–146.

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

[37] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. 2020. 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*. 1751–1767.

[38] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. 2019. 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*. 887–903.

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

[40] Florence Jessie MacWilliams and Neil James Alexander Sloane. 1977. *The theory of error correcting codes*. Vol. 16. Elsevier.

[41] Robert J. McEliece and Dilip V. Sarwate. 1981. On sharing secrets and Reed-Solomon codes. *Commun. ACM* 24, 9 (1981), 583–584.

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

[43] Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. 2020. Improved Extension Protocols for Byzantine Broadcast and Agreement. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

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

[45] Arpita Patra, Ashish Choudhary, and C Pandu Rangan. 2009. Efficient statistical asynchronous verifiable secret sharing with optimal resilience. In *International Conference on Information Theoretic Security*. Springer, 74–92.

[46] Arpita Patra, Ashish Choudhury, and C Pandu Rangan. 2015. Efficient asynchronous verifiable secret sharing and multiparty computation. *Journal of Cryptology* 28, 1 (2015), 49–109.

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

[48] David Pointcheval and Jacques Stern. 1996. Security proofs for signature schemes. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 387–398.

[49] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

[50] Alexandre Ruiz and Jorge L Villar. 2005. Publicly verifiable secret sharing from Paillier's cryptosystem. In *WEWoRC 2005–Western European Workshop on Research in Cryptology*. Gesellschaft für Informatik eV.

[51] Berry Schoenmakers. 1999. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Annual International Cryptology Conference*. Springer, 148–164.

[52] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[53] Lloyd R Welch and Elwyn R Berlekamp. 1986. Error correction for algebraic block codes. (Dec. 30 1986). US Patent 4,633,470.

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

[55] Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. 2021. hbACSS: How to Robustly Share Many Secrets. (2021).

A THRESHOLD SECRET SHARING

A (n, k) threshold secret sharing scheme allows a secret $s \in \mathbb{Z}_q$ to be shared among n nodes such that any k of them can come together to reconstruct the original secret, but any subset of $k-1$ shares cannot be used to reconstruct the original secret [9, 52]. We use the common Shamir secret sharing [52] scheme , where the secret is embedded in a random degree $k-1$ polynomial in the field $\mathbb{Z}_q$ for some prime q. Specifically, to share a secret $s \in \mathbb{Z}_q$, a polynomial $p(\cdot)$ of degree $k-1$ is chosen such that $s = p(0)$. The remaining coefficients of $p(\cdot)$, $a_1, a_2, \cdots, a_t$ are chosen uniformly randomly from $\mathbb{Z}_q$. The resulting polynomial $p(x)$ is defined as:

$$p(x) = s + a_1x + a_2x^2 + \cdots + a_{k-1}x^{k-1}$$

Each node is then given a single evaluation of $p(\cdot)$. In particular, the i^{th} node is given $p(i)$ i.e., the polynomial evaluated at i. Observe that given $t+1$ points on the polynomial $p(\cdot)$, one can efficiently reconstruct the polynomial using Lagrange Interpolation. Also note that when s is uniformly random in $\mathbb{Z}_q$, s is information theoretically hidden from an adversary that knows any subset of $k-1$ or less evaluation points on the polynomial other than $p(0)$ [52].

B ADD FOR HIGH THRESHOLD

Recall from §3 that it is impossible to solve ADD for $n < 2t+1$. In this section we will describe how to extend our solution to ADD with $n = 3t+1$ to a threshold of the form $t = (1/2-\epsilon)n$ for any $\epsilon > 0$. Briefly, to extend our results from §3 to an arbitrary $\epsilon > 0$, we use a larger m in the RSEnc and a collision resistant hash function. The detailed changes in our original protocol is as follows.

During the encoding phase, to protect against an adversary corrupting up to $(1/2-\epsilon)n$ nodes, each sender encodes the message M with $m \geq (k+1)/(2\epsilon)$ (due to reasons to be described later). Let $M' = \mathsf{RSEnc}(M, m, k)$ be the encoded message.

Then, during the dispersal phase, each sender sends the i^{th} component of M', m_i to node i. Note that size of m_i is

$$|m_i| = \frac{m|M|}{nk} \geq \frac{(1+k)|M|}{2\epsilon nk}$$

Let $h = \mathsf{hash}(p(\cdot))$ i.e., hash of the coefficients of the original polynomial $p(\cdot)$. Also, let $\alpha := (m/n)$.

For $0 \le r \le t$:

1. Let T_r be the subset of shares received till iteration r. Wait until $|T_r| \ge \alpha(n - t + r + 1)$.
2. Then run $\mathsf{RSDec}(k; r; T_r)$, and let $p_r(\cdot)$ be the output polynomial.
3. If $\mathsf{hash}(p_r(\cdot)) = h^*$, output $p_r(\cdot)$. Otherwise, proceed to the next iteration.

Figure 3: Hash based Online Error Correction (HOEC).

Furthermore, each sender additionally multi-casts the cryptographic digest, i.e., hash of message of $h = \mathsf{hash}(M)$ to all other nodes. Each, recipient upon receiving $t + 1$ identical hash h sets it as h^*. The recipient uses the h^* during the reconstruction phase.

Similar to our 1/3rd ADD, during the reconstruction phase each node i, multi-casts m_i^* to all other nodes. The procedure to recover the message using the received reconstruction message differ from OEC. In particular, every node instead runs the hash based OEC that we design and refer to as the HOEC algorithm. We summarize it in Figure 3 and describe it next.

The HOEC algorithm is iterative can take up to $t + 1$ iterations. In iteration r, each recipient node i waits to receive $n - t + r + 1$ `RECONSTRUCT` messages. For every received message of the form $(\texttt{RECONSTRUCT}, \ell, m_\ell)$ received till iteration r, node i adds (ℓ, m_ℓ) to a set T_r. Then, the recipient uses the RSDec algorithm on T_r to recover the original message. Let $p_r(\cdot) := \mathsf{RSDec}(t, r, T_r)$ be the output of the decoding algorithm, then the recipient outputs the coefficients of $p_r(\cdot)$ as the final message only if $\mathsf{hash}(p_r(\cdot))$ matches h^*, the hash received during the dispersal phase. Here $\mathsf{hash}(p_r(\cdot))$ denotes the hash of the coefficients of the polynomial $p_r(\cdot)$. If the hashes do not match, the recipient starts the next iteration.

The guarantees provided by HOEC is very similar to the guarantees of OEC. But for completeness, we state it below.

Lemma B.1. *For any (m, k), let $(a_1, ..., a_n)$ be the output of the encoding $\mathsf{RSEnc}(M, m, k)$ of a message M, where $m = (k + 1)/(2\epsilon)$. Assuming a collision resistant hash function, if an honest node i receives at least $n - t$ correct symbols of $\mathsf{RSEnc}(M, m, k)$ where $t = (1/2 - \epsilon)n$ for $\epsilon > 0$, HOEC ensures that node i eventually outputs M.*

Proof. Similar to our proof of OEC first we will argue that every honest node will eventually output a polynomial and we will treat the corrupt nodes that sent correct symbols to n_i as honest.

Let's focus on a single honest node, say n_i. Let $\mathcal{A}$ corrupts $\hat{r}$ nodes who either sends incorrect symbols or do not send any symbols to n_i. Note that $\hat{r} \le t$. Also, for any given m, each node is responsible for sending $\alpha = m/n$ symbols to n_i. Hence, an adversary controlling $\hat{r}$ nodes manages $\hat{r}\alpha(1/2 - \epsilon)$ symbols.

Let's assume $\hat{r}_1$ corrupt nodes sent incorrect symbols to n_i and $\hat{r}_2$ corrupt nodes did not send anything. Note that $\hat{r}_1 + \hat{r}_2 = \hat{r}$. Now consider the $(t - \hat{r}_2)^{\text{th}}$ iteration; since $\hat{r}_2$ nodes never sent any symbols to n_i, n_i will receive $\alpha(n - t + t - \hat{r}_2)$ distinct symbols on the polynomial $p(\cdot)$ of which $\alpha\hat{r}_1$ symbols are incorrect. Let $\mathcal{I}_{t-\hat{r}_2}$ be the set of received symbols in iteration $t - \hat{r}_2$, then

$$|\mathcal{I}_{t-\hat{r}_2}| = \alpha(n - t + t - \hat{r}_2) = \alpha(n - \hat{r}_2) \tag{5}$$

Now consider $k + 2\hat{r}_1\alpha + 1$. When $m = (k + 1)/(2\epsilon)$, we have

$$\begin{aligned} k + 2\hat{r}_1\alpha + 1 &= k + 2(\hat{r} - \hat{r}_2)\alpha + 1 \\ &\le (k + 1) + 2(t - \hat{r}_2)\alpha \\ &= (k + 1) + (1 - 2\epsilon)n\alpha - 2\hat{r}_2\alpha \\ &= (k + 1) + (n - \hat{r}_2)\alpha - 2\epsilon n\alpha - \hat{r}_2\alpha \\ &= (k + 1) + (n - \hat{r}_2)\alpha - (k + 1) - \hat{r}_2\alpha \\ &= (n - \hat{r}_2)\alpha - \hat{r}_2\alpha \le (n - \hat{r}_2)\alpha \end{aligned} \tag{6}$$

the last inequality holds due to the fact that $\hat{r}_2 \ge 0$.

Equation (6) implies that $|\mathcal{I}_{t-\hat{r}_2}| \ge k + 2\hat{r}_1\alpha + 1$, hence, the algorithm RSDec will correct $\alpha\hat{r}_1$ errors and will return the polynomial $p_{t-\hat{r}_2}(\cdot)$ during the $(t - \hat{r}_2)^{\text{th}}$ iteration.

Next, for correctness, let's assume that n_i outputs the polynomial $p_r(\cdot)$ in the r^{th} iteration. Then, $\mathsf{hash}(p_r(\cdot)) = h = \mathsf{hash}(p(\cdot))$. Hence, by collision resistance property of the hash function, we get $p_r(\cdot) = p(\cdot)$ as polynomials. □

Note that Lemma 3.2 also holds for $h = \mathsf{hash}(M)$ in our high-threshold ADD protocol. Hence, at the end of the dispersal phase of our high-threshold ADD, every recipient node will output the correct hash h. We will next argue that during the reconstruction phase of both ADD and high-threshold ADD every honest recipient outputs M. Combining this with Lemma B.1 we get,

Lemma B.2. *Assuming $\mathsf{hash}(\cdot)$ is a collision resistant hash function, at the end of the reconstruction phase of our high-threshold ADD every honest node outputs M.*

We will next analyze the communication complexity of our high-threshold ADD.

Lemma B.3. *Assuming the existence of collision resistant hash function, for any $\epsilon > 0$, in a network of n nodes where up to $(1/2 - \epsilon)$ fraction of nodes could be malicious, our high-threshold ADD has a total communication cost of $O(n|M|/\epsilon + n^2\kappa)$. Here κ is the size of the output of the hash function.*

Proof. During the dispersal phase, each sender sends a message of size $O(|M'|/n + \kappa)$ to every other node. Hence, the total communication cost of every sender is $O(|M'| + n\kappa)$. Since there are $\Theta(n)$ senders, the total communication cost in the dispersal phase is $O(n|M'| + n^2\kappa)$. During the reconstruction phase, each nodes sends a message of size $O(|M'|/n)$ to every other node. Hence, the total communication cost during the reconstruction phase is $O(n|M'|)$

Since in our high-threshold ADD $|M'| = O(|M|/2\epsilon)$, it has a total communication cost of $O(n|M|/\epsilon + n^2\kappa)$. □

C PEDERSEN'S VSS [47]

Let κ be the security parameter. Let $\mathbb{G}$ be a cyclic abelian group of prime order q and $\mathbb{Z}_q$ the group of integer modulo q. Let $g_0, g_1 \leftarrow \mathbb{G}$ be two uniform and independent element from $\mathbb{G}$. Before we describe Pedersen's VSS scheme, we will first briefly describe the commitment scheme for a arbitrary secret $s \in \mathbb{Z}_q$. To commit to a secret s, the committer samples a random $r \in \mathbb{Z}_q$ and computes

$$\mathsf{commit}(s, r) = v = g_0^s g_1^r$$

Let s be the secret a node (the dealer) and let g_0, g_1 two uniformly random and independent generators of a group $\mathbb{G}$.

$\mathsf{PedPolyCommit}(g_0, g_1, s) \rightarrow \{\boldsymbol{s}, \boldsymbol{c}, \boldsymbol{r}\}$

(1) Sample $r \leftarrow \mathbb{Z}_q$ and let $v_0 = \mathsf{commit}(s, r)$.
(2) Sample $a_k, b_k \in Z_q$ for $k = 1, 2, ..., t$ and let:

$$p(x) = s + a_1 x + \ldots + a_t x^t; \text{ and}$$

$$\phi(x) = r + b_1 x + \ldots + b_t x^t$$

(3) Let $v_k = \mathsf{commit}(a_k, b_k)$ for $k = 1, 2, ..., t$,
(4) Let $\boldsymbol{v} = \{v_0, v_1, \ldots, v_t\}$, $\boldsymbol{s} = \{p(1), p(2), \ldots, p(n)\}$, and let $\boldsymbol{r} = \{\phi(1), \phi(2), \ldots, \phi(n)\}$.
(5) **return** $\{\boldsymbol{v}, \boldsymbol{s}, \boldsymbol{r}\}$

$\mathsf{PedEvalVerify}(g_0, g_1, \boldsymbol{v}, i, s_i, r_i) \rightarrow 0/1$:

(1) If

$$\mathsf{commit}(s_i, r_i) = g_0^{s_i} g_1^{r_i} = \prod_{j=0}^{t} v_j^{i^j}$$

return 1 otherwise **return 0.**

Given PedPolyCommit and PedEvalVerify be the polynomial commitment and evaluation scheme, respectively. The VSS scheme is defined as below.

$\mathsf{VSS.Share}(s, g_0, g_1, n, t)$:

(1) Let $\{\boldsymbol{v}, \boldsymbol{s}, \boldsymbol{r}\} := \mathsf{PedPolyCommit}(g_0, g_1, s)$
(2) **Broadcast** $\boldsymbol{v}$ to all nodes and **send** $\boldsymbol{s}[i], \boldsymbol{r}[i]$ to node i

$\mathsf{VSS.Verify}(g_0, g_1, \boldsymbol{v}, s_i, r_i) \rightarrow 0/1$:

(1) **Output** $\mathsf{PedEvalVerify}(g_0, g_1, \boldsymbol{v}, i, s_i, r_i)$.

Let T be the set of valid shares s_k where $|T| = t + 1$, then

$\mathsf{VSS.Recon}(\{s_k\}_{k \in T}) \rightarrow s$:

(1) Output

$$\sum_{k \in T} s_k \cdot \mu_k = p(0) = s \tag{7}$$

where $\mu_k = \prod_{j \neq k} \frac{j}{j-k}$ for $k \in T$ are Lagrange coefficients.

Figure 4: Pedersen's VSS scheme [47]

To reveal such a commitment later, the committer reveals (s, r) and the verifier checks whether $g_0^s g_1^r$ is equal to v or not. We refer to the reveal procedure as:

$$\mathsf{reveal}(v) := s, r \text{ such that } v = g_0^s g_1^r$$

Pedersen [47] illustrates that the commitment scheme described above information theoretically hides s and binds s to v for a computationally bounded prover, assuming the prover does not know the discrete logarithm of g_1 with respect to g_0, i.e., the prover can not efficiently compute $\log_{g_0} g_1$. We summarize the VSS scheme from [47] in Figure 4.

Observe that the commitment to the polynomial $p(\cdot)$ that embeds the secret s is linear in the number of nodes. Moreover, given the linear size commitment and a tuple (s_k, t_k), one can efficiently verify (without any extra information) whether s_k is equal to $p(k)$ or not. We will crucially use these properties to design our AVSS scheme with a total communication cost of $O(\kappa n^2)$.

Next, we briefly summarize the properties of the VSS scheme described in Figure 4. Informally, Lemma C.1 states that once the VSS.Share step terminated correctly, any set of $t + 1$ nodes can combine their shares to recover the secret. Theorem C.3 states that any subset of $t + 1$ nodes will reconstruct the same secret.

LEMMA C.1 (LEMMA 4.2 OF [47]). *Let $S \subset \{1, 2, \ldots, n\}$ be a set of $t + 1$ nodes such that the verification was successful for these $t + 1$ nodes. Then these $t+1$ nodes can find a pair (s', t') such that $v = g_0^{s'} g_1^{t'}$.*

Definition C.2 (Uniqueness). For all subsets S_1 and S_2 of $\{1, 2, \ldots, n\}$ of size k such that all nodes in S_1 and S_2 accepted their shares in the verification protocol described above. Let s_i be the secret computed by the participants in S_i, then $s_1 = s_2$.

THEOREM C.3 (THEOREM 4.2 OF [47]). *Under the assumption that the dealer can not find $\log_{g_0} g_1$ except with negligible probability in $|q|$, the verification protocol satisfies uniqueness.*

Let S be the subset of nodes with $|S| \leq t$, let view_S be the internal state of nodes in S and messages sent and received by nodes in S. Then, the next theorem ensures formally states the secrecy property of the VSS.

THEOREM C.4 (THEOREM 4.4 OF [47]). *For all adversary $\mathcal{A}$, for any subset $S \subset [n]$ of size t and view_S, for all $s \in \mathbb{Z}_q$*

$$\Pr[\mathcal{A} \text{ has secret} \mid \mathsf{view}_S] = \Pr[\mathcal{A} \text{ has secret}]$$

D ZERO KNOWLEDGE PROOF OF EQUALITY OF DISCRETE LOGARITHM

The dual-threshold AVSS protocol has a step that requires nodes to produce zero-knowledge proofs about equality of discrete logarithms for a tuple of publicly known values. In particular, given a group $\mathbb{G}$ of prime order q, two uniformly random generators $g_0, g_1 \leftarrow \mathbb{G}$ and a tuple (g_0, x, g_1, y), a prover $\mathcal{P}$ wants to prove to a probabilistic polynomial time (PPT) verifier $\mathcal{V}$, in zero-knowledge, the knowledge of a witness α such that $x = g_0^\alpha$ and $y = g_1^\alpha$.

Throughout this paper, we will use the Chaum-Pedersen "Σ-protocols" [23], which assumes the hardness of the Decisional Diffie-Hellman (DDH) problem, and can be made non-interactive using the Fiat-Shamir heuristic [28].

Decisional Diffie–Hellman assumption. Given a group $\mathbb{G}$ with generator $g \in \mathbb{G}$ and uniformly random samples $a, b, c \leftarrow \mathbb{Z}_q$, the Decisional Diffie–Hellman (DDH) hardness assumes that the following two distributions D_0, D_1 are computationally indistinguishable: $D_0 = (g, g^a, g^b, g^{ab})$ and $D_1 = (g, g^a, g^b, g^c)$.

Protocol for equality of discrete logarithm. For any given tuple (g_0, x, g_1, y), the Chaum-Pedersen protocol proceeds as follows.

(1) $\mathcal{P}$ samples a random element $\beta \leftarrow \mathbb{Z}_q$ and sends (a_1, a_2) to $\mathcal{V}$ where $a_1 = g_0^\beta$ and $a_2 = g_1^\beta$.
(2) $\mathcal{V}$ sends a challenge $e \leftarrow \mathbb{Z}_q$.
(3) $\mathcal{P}$ sends a response $z = \beta - \alpha e$ to $\mathcal{V}$.
(4) $\mathcal{V}$ checks whether $a_1 = g_0^z x^e$ and $a_2 = g_1^z y^e$ and accepts if and only if both the equality holds.

As mentioned, this protocol can be made non-interactive in the Random Oracle model using the Fiat-Shamir heuristic [28, 48]. This protocol guarantees completeness, knowledge soundness, and zero-knowledge. The knowledge soundness implies that if $\mathcal{P}$ convinces the $\mathcal{V}$ with non-negligible probability, there exists an efficient (polynomial time) extractor that can extract α from the prover with non-negligible probability.

In our dual-threshold ACSS, we use the non-interactive variant of the protocol described above and denote it $\mathsf{dleq}(\cdot)$. In particular, for any given tuple (g_0, x, g_1, y) where $x = g_0^s$ and $y = g_1^s$, $\pi \leftarrow \mathsf{dleq.Prove}(s, g_0, x, g_1, y)$ generates the proof π. Given the proof π and (g_0, x, g_1, y), $\mathsf{dleq.Verify}(\pi, g_0, x, g_1, y)$ verifies the proof.

E PUBLICLY VERIFIABLE SECRET SHARING

We restate this section from [25]. Our (n, ℓ) dual-threshold ACSS scheme for $n \geq 3t + 1$ and $t < \ell \leq n - t$ crucially rely on on a (n, ℓ) publicly verifiable secret sharing (PVSS). In particular, we use the PVSS scheme from Scrape [21], which is an improvement over the Schoenmakers scheme [51]. The scheme allows a node (dealer) to share a secret $s \in \mathbb{Z}_q$ among n nodes, such that any subset of at least ℓ nodes can reconstruct g_1^s. Here, g_1 is a random generator of $\mathbb{G}$. Additionally, any subset of ℓ or less nodes, can not learn any information about the secret s.

The reconstruction threshold ℓ is chosen in a way such that valid contribution from at least ℓ nodes are required to recover g_1^s.

A key property of a PVSS scheme is that, not only the recipients but any third party (with access to recipients' public keys) can verify, even before the reconstruction phase begins, that the dealer has generated the shares correctly without having plaintext access to the shares.

The PVSS scheme of Scrape [21] is non-interactive in the random oracle model and has three procedures: PVSS.Share, PVSS.Verify, and PVSS.Recon. A node (dealer) with public-private key pair pk, sk, uses PVSS.Share to share a secret s, other nodes or external users use PVSS.Verify to validate the shares, and PVSS.Recon is used to recover h^s. We describe them in detail in Figure 5.

The verification procedure of Scrape's PVSS uses properties of error correcting code, specifically the Reed Solomon code [49]. They use the observation by McEliece and Sarwate [41] that sharing of a secret using a degree ℓ polynomial among n nodes is equivalent to encoding the message $(x, a_1, a_2, \cdots, a_t)$ using a $[n, \ell + 1, n - \ell]$ Reed Solomon code [49].

Let C be a $[n, k, d]$ linear error correcting code over $\mathbb{Z}_q$ of length n and minimum distance d. Also, let $C^\perp$ be the dual code of C i.e., $C^\perp$ consists vectors $\boldsymbol{y}^\perp \in \mathbb{Z}_q^n$ such that for all $\boldsymbol{x} \in C$, $\langle \boldsymbol{x}, \boldsymbol{y}^\perp \rangle = 0$. Here, $\langle \cdot, \cdot \rangle$ is the inner product operation. Scrape's PVSS.Verify uses the following basic fact (Lemma E.1) of linear error correcting code. We refer readers to [21, Lemma 1] for its proof.

Lemma E.1. *If $\boldsymbol{x} \in \mathbb{Z}_q^n \setminus C$, and $\boldsymbol{y}^\perp$ is chosen uniformly at random from $C^\perp$, then the probability that $\langle \boldsymbol{x}, y^\perp \rangle = 1$ is exactly $1/q$.*

The PVSS scheme of Scrape provides the IND1-Secrecy property stated in Theorem E.3. Intuitively, for any (n, ℓ) PVSS scheme, IND1-secrecy ensures that prior to the reconstruction phase, the public information together with the secret keys sk_i of any set of at most ℓ players gives no information about the secret. Formally this is stated

Let s be the secret a node (the dealer) with public-private key pair (sk, pk) wants to share with set of nodes with public keys $\{pk_j\}_j$ for $j = 1, 2, \dots, n$. Let g_0, g_1 be two randomly chosen generators of group $\mathbb{G}$.

PVSS.Share$(s, g_0, g_1, n, \ell, \{pk\}_{j,j=1,2,\dots,n}) \rightarrow (\boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi})$:

(1) Sample uniform random $a_i \in \mathbb{Z}$ for $k = 1, 2, \dots, \ell$ and let
$$p(x) = s + a_1 x + \ldots + a_\ell x^\ell;$$
(2) Let $v_j := g_0^{p(j)}$; and $c_j := pk_j^{p(j)}$, for $j = 1, \dots, n$.
(3) Let $\pi_j := \mathsf{dleq.Prove}(p(j), g_0, v_j, pk_j, c_j)$
(4) Output $\boldsymbol{v} = \{v_1, v_2, \dots, v_n\}$; $\boldsymbol{c} = \{c_1, c_2, \dots, c_n\}$, and $\boldsymbol{\pi} = \{\pi_1, \pi_2, \dots, \pi_n\}$.

PVSS.Verify$(g_0, g_1, n, \ell, \{pk\}_{j,j=1,2,\dots,n}, \boldsymbol{v}, \boldsymbol{c}, \boldsymbol{\pi}) \rightarrow 0/1$:

(1) Sample a random code word $\boldsymbol{y}^\perp \in C^\perp$ and check whether
$$\prod_{k=1}^{n} v_k^{x_k^\perp} = 1_{\mathbb{G}} \tag{8}$$
where $1_{\mathbb{G}}$ is the identity element of $\mathbb{G}$.
(2) Check whether $\mathsf{dleq.Verify}(\pi_j, g_0, v_j, pk_j, c_j) = 1$ for all j.
(3) Output 1 if both checks pass, output 0 otherwise.

Let T be the set of valid tuples of the form $(\tilde{s}_i, \tilde{\pi}_i)$ where $\tilde{\pi}_i = \mathsf{dleq.Prove}(sk_i, g_1, pk_i, \tilde{s}_i, c_i)$ where $|T| = t + 1$, then

PVSS.Recon$(\{\tilde{s}_i\}_{i \in T}) \rightarrow g_1^s$:

(1) Output
$$\prod_{i \in T} (\tilde{s}_i)^{\mu_i} = \prod_{i \in T} g_1^{\mu_i \cdot p(i)} = g_1^{p(0)} \tag{9}$$
where $\mu_i = \prod_{j \neq i} \frac{j}{j-i}$ for $i \in T$ are Lagrange coefficients.

Figure 5: Scrape's PVSS scheme.

as in the following indistinguishability based definition adapted from [34, 50]:

Definition E.2. (IND1-Secrecy) A (n, ℓ) PVSS is said to be IND1-secret if for any probabilistic polynomial time adversary $\mathcal{A}$ corrupting at most ℓ parties, $\mathcal{A}$ has negligible advantage in the following game played against an challenger.

(1) The challenger runs the Setup phase of the PVSS as the dealer and sends all public information to $\mathcal{A}$. Moreover, it creates secret and public keys for all honest nodes, and sends the corresponding public keys to $\mathcal{A}$.
(2) $\mathcal{A}$ creates secret keys for the corrupted nodes and sends the corresponding public keys to the challenger.
(3) The challenger chooses values s_0 and s_1 at random in the space of secrets. Furthermore it chooses $b \leftarrow \{0, 1\}$ uniformly at random. It runs the phase of the protocol with s_0 as secret. It sends $\mathcal{A}$ all public information generated in that phase, together with s_0.

The advantage of $\mathcal{A}$ is defined as $|\Pr[b = b'] - 1/2|$.

Theorem E.3. (IND1-Secrecy [21, Theorem 1]) *Under the decisional Diffie-Hellman assumption, the PVSS protocol in [21] is IND1-secret against a static probabilistic polynomial time adversary that can collude with up to ℓ nodes.*