Asynchronous Consensus without Trusted Setup or Public-Key Cryptography

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

Sisi Duan
Tsinghua University
Beijing, China
duansisi@mail.tsinghua.edu.cn

Shengqi Liu
Southern University of Science and Technology
Shenzhen, China
shengqi@illinois.edu

Atsuki Momose
University of Illinois at Urbana-Champaign
Urbana, USA
atsuki.momose@gmail.com

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

Victor Shoup
Offchain Labs
New York City, USA
victor@shoup.net

Abstract

Byzantine consensus is a fundamental building block in distributed cryptographic problems. Despite decades of research, most existing asynchronous consensus protocols require a strong trusted setup and expensive public-key cryptography. In this paper, we study asynchronous Byzantine consensus protocols that do not rely on a trusted setup and do not use public-key cryptography such as digital signatures. We give an Asynchronous Common Subset (ACS) protocol whose security is only based on cryptographic hash functions modeled as a random oracle. Our protocol has $O(\kappa n^3)$ total communication and runs in expected $O(1)$ rounds. The fact that we use only cryptographic hash functions also means that our protocol is post-quantum secure. The minimal use of cryptography and the small number of rounds make our protocol practical. We implement our protocol and evaluate it in a geo-distributed setting with up to 128 machines. Our experimental evaluation shows that our protocol is more efficient than the only other setup-free consensus protocol that has been implemented to date. En route to our asynchronous consensus protocols, we also introduce new primitives called asynchronous secret key sharing and cover gather, which may be of independent interest.

CCS Concepts

• **Security and privacy → Distributed systems security**.

Keywords

Consensus; Asynchrony; Asynchronous Common Subset

ACM Reference Format:
Sourav Das, Sisi Duan, Shengqi Liu, Atsuki Momose, Ling Ren, and Victor Shoup. 2024. Asynchronous Consensus without Trusted Setup or Public-Key Cryptography. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670327

1 Introduction

Byzantine consensus is a fundamental building block in distributed computing and cryptography. This paper studies the Byzantine consensus problem in asynchrony, and we are particularly interested in protocols that do not require a trusted setup and do not use public-key cryptography such as digital signatures.

Over the years, there have been many asynchronous Byzantine consensus protocols developed in the literature [5, 8, 14, 16, 23, 24, 27, 32, 36–39, 46]. The vast majority of them require trusted setups and public-key cryptography. This is partly due to the well-known FLP impossibility, which says randomization is necessary for asynchronous consensus. To circumvent FLP, most asynchronous consensus protocols, especially ones that target practical performance [24, 39, 40, 46], rely on *strong common coins*, i.e., random values that are agreed upon by all honest parties. However, strong common coins are not easy to obtain. The most common approach is to rely on unique threshold signatures [11, 14], which require trusted setups and public-key cryptography.

Starting from Canetti and Rabin [16], a line of work builds common coins in the setup-free setting and uses it to build consensus protocols. The protocol of Canetti and Rabin had a very high communication cost, and there was little improvement for a long time. The study of setup-free asynchronous consensus became active again recently, often in conjunction with the asynchronous distributed key generation (ADKG) problem [3, 4, 18, 20, 28, 35, 47]. This is a natural development because the problem of generating common coins is a lot like the problem of generating a common key. A setup-free consensus protocol usually consists of two logical parts: *common coin generation* and *consensus*. The common coin generation part provides either weak common coins (honest parties output the same coin with a certain probability) or strong common

Table 1: Comparison of setup-free asynchronous consensus protocol. [2] has 1/4 fault tolerance and all other protocols have the optimal 1/3 fault tolerance. In the table, DDH is Decisional Diffie-Hellman, SXDH is symmetric external Diffie–Hellman (a pairing assumption), and RO is random oracle.

Protocol	Expected Communication	Expected Rounds	Assumptions	Implemented?
Canetti-Rabin [16]	$O(n^7)$	$O(1)$	None	✗
Kokoris et al. [35]	$O(\kappa n^4)$	$O(n)$	DDH, RO	✗
Abraham et al. [4]	$O(\kappa n^3)$	$O(1)$	SXDH, RO	✗
Gao et al. [28]	$O(\kappa n^3)$	$O(1)$	SXDH, RO	✗
Das et al. [18, 20]	$O(\kappa n^3)$	$O(\log n)$	DDH, RO	✓
Abraham et al. [2]	$O(n^5)$	$O(1)$	None	✗
This work	$O(\kappa n^3)$	$O(1)$	RO[†]	✓

[†] In §9, we briefly discuss, how using ideas from [44] and additional techniques, we can prove our protocol secure without the random oracle model.

coins (the probability is 1). The coins are then used in the consensus part, e.g., for leader election [3, 4] or asynchronous binary agreement [18, 20], to circumvent FLP.

Table 1 lists all the recent setup-free asynchronous consensus protocols. Kokoris et al. [35] gives a protocol with $O(\kappa n^4)$ total communication and $O(n)$ expected rounds, where n is the number of parties and κ is a computational security parameter. Das et al. [18, 20] give a protocol with $O(\kappa n^3)$ expected communication and $\Omega(\log n)$ expected rounds (due to the use of n parallel instances of asynchronous binary agreement). Abraham et al. [4] and Gao et al.[28] give protocols with $O(\kappa n^3)$ expected communication and $\Omega(1)$ expected rounds. The above protocols all require public-key cryptography. Another very recent work [2] gives a setup-free and public-key-cryptography-free asynchronous consensus protocol with $O(n^5)$ communication and $O(1)$ expected rounds.

With the current state of affairs, an interesting open question is: *Can we obtain a practical consensus protocol without trusted setup and without public-key cryptography?*

Our result. In this work, we answer the question in the affirmative. We give an Asynchronous Common Subset (ACS) protocol (a variant of asynchronous Byzantine consensus) without using any private setups or public-key cryptography. The only cryptographic primitive we use is a cryptographic hash function. Our protocol has $O(\kappa n^3)$ expected communication and runs in expected $O(1)$ rounds. The fact that we use only cryptographic hash functions also means that our protocol is post-quantum secure.

To obtain a practical protocol without setup or public-key cryptography, we use several techniques as described below, each of which might be of independent interest.

Index consensus. In the conventional definition of consensus, each party inputs a message, and the output is an agreed-upon message or set of messages. We slightly tweak the problem definitions to better fit our design approach. In our approach, instead of agreeing on a set of messages, we often need to agree on a party (or set of parties) who correctly carried out some actions. The interface of the conventional message-based consensus does not fully match a use case like this. We thus define alternative index-based versions of ACS and Validated Asynchronous Byzantine Agreement (VABA) where the output is an agreed-upon party (or set of parties), represented by the party ID, who correctly carried out some prior actions.

Weak leader election from index cover gather and lightweight asynchronous secret key sharing (ASKS). To instantiate the weak leader election oracle without public-key cryptography, we propose a lightweight asynchronous secret key sharing protocol from hash functions only. We also extend the gather primitive [4] into a new primitive called index *cover* gather (ICG), which has an attractive property that prevents the adversary from arbitrarily manipulating the weak leader-election process.

Practical efficiency, implementation, evaluation. The minimal use of cryptography and the small number of rounds make our protocol practical. To demonstrate the practicality of our protocol, we implement it in Python and use Rust for cryptographic operations. Our experimental evaluation shows that our protocol is more efficient than the ACS protocol of [18], the only other setup-free consensus protocol that has been implemented to date.

Paper organization. The rest of the paper is organized as follows. In §2, we present the system model, preliminaries, formal definitions of our index consensus primitives, and a brief overview of our core ideas. In §3, we present details of our ASKS and prove its security. We formally define index cover gather and describe a perfectly secure construction of index cover gather protocol in §4. We then use our ASKS and the index cover gather, along with other standard primitives, to build an index VABA protocol in §5. We then use the index VABA to build index ACS in §6. We present our implementation and evaluation results in §7, discuss the related work in §8, and conclude in §9.

2 Model and Problem Definition

Notations. For any integer a, we use $[a]$ to denote the ordered set $\{1, 2, \ldots, a\}$. For any set S, we use $s \leftarrow\$ S$ to indicate that s is sampled uniformly randomly from S. We use $|S|$ to denote the size of set S. Throughout the paper, we will use "$\leftarrow$" for probabilistic assignment and ":=" for deterministic assignment. We use κ to denote the security parameter. A machine is *Probabilistic Polynomial Time* (PPT) if it is a probabilistic algorithm that runs in poly(κ) time. We also use negl(κ) to denote functions negligible in κ. Throughout this paper, we will use $\mathbb{F}$ to denote a large finite field, i.e., we have that $1/|\mathbb{F}|$ is negl(κ).

2.1 System Model

We consider a network of n parties $\{1, 2, \ldots, n\}$ where every pair of parties are connected via a pairwise private and authenticated channel. We consider the presence of a *static* adversary $\mathcal{A}$ that can corrupt up to t out of the $n \geq 3t + 1$ parties. Let $\mathcal{H} \subseteq [n]$ be the set of honest parties, and $C := [n] \setminus \mathcal{H}$ be the set of corrupt parties. We assume the network is asynchronous, i.e., $\mathcal{A}$ can arbitrarily delay any message but must eventually deliver all messages sent between honest parties.

2.2 Problem Definition

To describe index ACS more precisely, we introduce the notion of parties validating each other.

Party validation. For any $i, j \in [n]$, we say "party i *has validated* party j", if party i thinks, based on the messages it has seen so far, that party j has done some action correctly in the protocol. The precise notion of "correct actions" will depend on the specific protocol. We also say that party j has been *locally* validated if it has been validated by some honest party, and that party j has been *globally* validated if it has been validated by all honest parties.

Completeness of party validation. Throughout the paper, we will primarily work with a "validate" notion that satisfies the *completeness* property. The completeness property ensures that if a party is locally validated, then it will eventually be globally validated. Stating differently, it ensures that if an honest party i has validated a party $j \in [n]$, then eventually, party j will be validated by all honest parties.

Index ACS. We can now define the *Index Asynchronous Common Subset* problem.

Definition 2.1 (Index Asynchronous Common Subset). An index asynchronous common subset (ACS) is a protocol among n parties $\{1, 2, \ldots, n\}$, where each party i inputs a set Valid_i of parties it has validated, and outputs a subset $X_i \subseteq [n]$ of parties with $|X_i| \geq n - t$. Note that the input Valid_i may grow over time. An index ACS protocol must satisfy the following properties.

- *Agreement.* If two honest parties i and j output X_i and X_j, respectively, then $X_i = X_j$.
- *Validity.* If an honest party i outputs X_i, then every $j \in X_i$ has been locally validated.
- *Termination.* If the input party validation satisfies completeness and at least $n - t$ parties have been locally validated at some point during the protocol, then every honest party eventually outputs.

It is easy to build a message-based ACS protocol (i.e., the conventional ACS) from an index ACS protocol – see Section 6.2. Simply have each party send its input message via a reliable broadcast (RBC) and then use index ACS to agree on a common subset of RBCs that have finished. The output of the (message) ACS is the union of outputs from that common subset of RBC instances.

Index VABA. As a stepping stone to our index ACS, we will define and build an index version of validated asynchronous Byzantine agreement (VABA). The only difference between VABA and ACS is that VABA outputs a single element instead of a set.

Definition 2.2 (Index Validated Asynchronous Byzantine Agreement). An index validated asynchronous Byzantine agreement (VABA) is a protocol among n parties $\{1, 2, \ldots, n\}$, where each party i inputs a set Valid_i of parties it has validated, and outputs a party's index $x_i \in [n]$. Note that the input Valid_i may grow over time. An index VABA protocol must satisfy the following properties.

- *Agreement.* If two honest parties i and j output x_i and x_j, respectively, then $x_i = x_j$.
- *Validity.* If an honest party i outputs x_i, then party x_i has been locally validated.
- *Termination.* Same as in index ACS.

2.3 Preliminaries

Index gather. Our index VABA protocol crucially relies on a *gather* primitive [16]. To fit our index interface, we define its index version. Index gather is similar to index ACS but has a weaker agreement property, which we refer to as *binding core*. Unlike the agreement property in Definition 2.1, the binding core property only guarantees the existence of a *core* set X of size at least $n - t$ that will be a subset of every honest party's (eventual) output.

Definition 2.3 (Index Gather). An index gather is a protocol among n parties $\{1, 2, \ldots, n\}$, where each party i inputs a set Valid_i of parties it has validated, and outputs a subset $X_i \subseteq [n]$ of parties with $|X_i| \geq n - t$. Note that the input Valid_i grows over time. An index gather protocol must satisfy the following *binding core* property, besides the same *validity* and *termination* properties of index ACS.

- *Termination.* Same as in index ACS.
- *Validty.* Same as in index ACS.
- *Binding core.* At the first time some honest party i outputs X_i, there exists a core set X, dependent only on the joint views of the honest parties at that time, with $|X| \geq n - t$ such that for all honest party j, $X \subseteq X_j$.

In the protocols we will present in this paper, such a core set X can be extracted efficiently from the joint views of the honest parties at the first time some honest party outputs.

Reliable broadcast. We will use the standard asynchronous reliable broadcast (RBC) [12, 15, 19] in a black-box manner.

Definition 2.4 (Reliable Broadcast). A *Reliable Broadcast* (RBC) is a protocol that allows a designated party D, referred to as the sender, to broadcast a message to a set of n parties $\{1, 2, \ldots, n\}$. We use the convention that $D \in [n]$. A RBC protocol must satisfy the following properties.

- *Agreement.* If two honest parties i and j output m_i and m_j, respectively, then $m_i = m_j$.
- *Totality.* If an honest party outputs a message, then every honest party i eventually outputs a message.
- *Validity.* If the sender is honest and broadcasts m, then every honest party i eventually outputs $m_i = m$.

Reliable agreement. We will also use a primitive we call *reliable agreement*. Intuitively, it is the agreement version of RBC where every party has an input.

Definition 2.5 (Reliable Agreement). A *reliable agreement (RA)* is a protocol among n parties $\{1, 2, \ldots, n\}$ where each party has an input message and possibly outputs a message. A reliable agreement protocol needs to satisfy the following properties.

- *Agreement.* Same as in RBC.
- *Totality.* Same as in RBC.
- *Validity.* If all honest parties input m, then eventually all honest parties output m.
- *Integrity.* If an honest party outputs m, then at least $n - 2t$ honest parties input m.

Bracha's RBC [12] can be easily modified into a reliable agreement protocol. We describe the protocol in Algorithm 1. It is not hard to see this protocol satisfies agreement, validity, and totality, and the proofs are identical to those of Bracha's RBC. For integrity,

Algorithm 1 Reliable agreement protocol for party i

```
1: upon receiving input $m_i$
2:     send $\langle \mathsf{ECHO}, m_i \rangle$ to all
3: upon receiving $\langle \mathsf{ECHO}, m \rangle$ from $n - t$ parties
4:     send $\langle \mathsf{READY}, m \rangle$ to all
5: upon receiving $\langle \mathsf{READY}, m \rangle$ from $t + 1$ parties
6:     Send $\langle \mathsf{READY}, m \rangle$ to all
7: upon receiving $\langle \mathsf{READY}, m \rangle$ from $n - t$ parties
8:     output $m$ and terminate
```

simply observe that for any honest party to send ready for m, there must be $n - t$ echoes for m, out of which $n - 2t$ must come from honest parties that input m. Algorithm 1 has $O(n^2L)$ communication complexity for L-bit messages, same as Bracha's RBC.

We want to note that, similar to RBC, a reliable agreement protocol has a weaker *termination* property than standard (Byzantine) consensus protocols [14, 39]. In particular, standard consensus protocols guarantee termination once every honest party provides some input to the protocol. In contrast, a reliable agreement protocol only guarantees termination if honest parties provide *matching* inputs. In return, reliable agreement can be instantiated deterministically in asynchrony and much more efficiently.

2.4 Technical Overview

A setup-free consensus protocol usually consists of two logical parts: *common coin generation* and *consensus*. The common coin generation part provides either weak common coins (honest parties output the same coin with a certain probability) or strong common coins (the probability is 1). The coins are then used in the consensus part, e.g., for leader election [2–4, 16] or asynchronous binary agreement [18, 20], to circumvent the FLP impossibility [25].

We follow a weak-coin-based framework that dates back to Canetti-Rabin [16] and has been adopted in many subsequent works [2–4]. The framework proceeds in iterations, where each iteration has two phases: *weak leader election* and *agreement*. The leader election is weak in the sense that honest parties might disagree on the leader with a (typically constant) probability. At a very high level, the weak-coin-based framework ensures that if parties agree on the leader, the entire protocol will succeed. Otherwise, parties enter the next iteration, and the process repeats.

For the agreement phase, we design a new index gather protocol based on the *weak core set* primitive in [28]. Our new index gather protocol is simpler and more efficient by constant factors than existing ones [2, 4]. But the more important difference between our work and prior works lies in the weak leader election phase, so we will focus on that part in this overview.

Weak leader election. The standard approach to weak leader election phase is to rely on a *gather* protocol. For notational consistency, we will describe it using an index gather protocol (see definition 2.3).

In each iteration, each party is assigned a random rank. These ranks are hidden at first. Parties run an index gather protocol to locally output a set of parties who performed some actions correctly. Recall from definition 2.3 that each party outputs a set of indices X_i that is a superset of some binding core set X. After the index gather protocol, the ranks are revealed, and each party i picks the party $\ell_i \in X_i$ with the highest rank as the leader. Intuitively, if the party with the highest rank among all parties happens to be in the core set, then all honest parties will output the same leader.

Given this framework, the natural question is how to derive ranks for parties such that: (1) the ranks remain hidden until some honest party outputs in the gather protocol; and (2) the ranks cannot be manipulated by the an adversary.

Next, we will describe a simple but *insecure* approach to derive ranks to illustrate the basic ideas in prior works [2–4, 16]. We will then explain why this approach is insecure, how prior works addressed it, and how we address it.

Insecure rank derivation. Each party i starts by sharing a uniformly random secret s_i using a verifiable secret sharing (VSS) scheme. Next, each party i selects a subset $P_i \subseteq [n]$ of $t + 1$ or more parties who shared their secrets correctly, and reliably broadcasts P_i to all parties. The rank r_i of party i is defined as:

$$r_i := \sum_{j \in P_i} s_i \tag{1}$$

Intuitively, since each party's rank depends on the secrets of $t + 1$ or more parties, the ranks remain hidden until parties reconstruct the secrets. A party starts reconstructing the secrets only after it outputs from the index gather protocol.

The issue with this approach is that an adversary can manipulate the ranks of some parties. Let party i be the first honest party to output from the index gather protocol. After that, the ranks of parties in X_i are fixed and cannot be manipulated by the adversary. However, the adversary can manipulate the ranks of parties outside X_i, i.e., in $[n] \setminus X_i$. This is because once party i reveals its shares of secrets shared by other honest parties, the adversary learns those secrets. The adversary can then manipulate the ranks of corrupt parties (outside X_i) to make them higher than the ranks of parties in X_i. More concretely, for a malicious party $\hat{i}$, the adversary can pick the set $P_{\hat{i}}$ to be $\hat{i}$ plus those honest parties whose secrets it now knows, and pick $s_{\hat{i}}$ accordingly to yield a high $r_{\hat{i}}$. The adversary next manipulates the network delay to make sure this high-rank corrupt party is included in some honest party's gather output, i.e., $\hat{i} \in X_j$ for some honest party $j \neq i$. This will make honest party j pick a different leader from party i, so the weak leader election always fails. We note that this attack does not violate any property of the gather protocol.

Prior works addressed this issue in a few ways. For example, [4] uses public key cryptography, specifically a pairing-based threshold verifiable random function, to derive ranks without revealing the secrets. [2] requires each party to share n secrets in every iteration so it can use independent secrets to derive ranks of different parties. This pushes the communication cost to $\Omega(\kappa n^4)$ per iteration.

Our approach. With this state of affairs, we would like to design a *setup-free* weak leader election protocol *without public-key cryptography* where each party shares *a single secret*.

The abovementioned attack works because the adversary is able to manipulate the ranks of parties in $[n] \setminus X$ to cause disagreement. A natural fix is to prevent rank manipulation completely. Unfortunately, we do not know how to do that efficiently. Instead, we do the following. We introduce a new *binding cover* requirement to the gather primitive. The binding cover property guarantees that when the first honest party outputs in the gather protocol, there exists a set $Y \subseteq [n]$ of locally validated parties, such that every honest party's eventual output is a subset of Y. We refer to Y as the *cover* set and refer to such a protocol as a *Index Cover Gather* (ICG) protocol.

We can ensure that ranks of all parties in the cover set Y are fixed by the first time an honest party starts reconstructing the secrets. This is because by that time, every party $j \in Y$ has reliably broadcast P_j such that every party $k \in P_j$ has shared its random secret s_k using VSS. It is not hard to see that these two properties combined fix the ranks of all parties in Y by the time the first honest party outputs in the ICG protocol. Note that an adversary might still be able to manipulate the ranks of parties in $[n] \setminus Y$. However, this is inconsequential because parties in $[n] \setminus Y$ will not be in any honest party's ICG output, and thus have no impact on the leader election outcome.

We design an ICG protocol that uses an index gather protocol in a modular way. Our ICG protocol incurs a communication cost of $O(n^3)$, and requires three additional rounds on top of an index gather protocol.

Hash-based secret sharing. Another step where prior works rely on public key cryptography is the verifiable secret sharing (VSS). For example, [4] needs a VSS scheme that is homomorphic and provides completeness. All known VSS schemes with these strong properties use public-key cryptography.

Since our rank derivation protocol first reconstructs the secrets shared by parties and then sum them up to derive ranks, we no longer require homomorphism or completeness. These relaxations allow us to design a weaker primitive called *Asynchronous Secret Key Sharing* (ASKS) protocol that is concretely efficient and uses only hash functions.

3 Asynchronous Secret Key Sharing

In this section, we define and present a simple construction of Asynchronous Secret Key Sharing (ASKS).

3.1 ASKS Definition

An *Asynchronous Secret Key Sharing* (ASKS) scheme lets a designated party D, referred to as the dealer, share a uniformly random secret key $s \in \mathcal{S}$ to n parties. Here, $\mathcal{S}$ is the secret key space. An ASKS protocol has two phases: *Sharing* phase and *Reconstruction* phase.

In the sharing phase, D shares a secret key $s \leftarrow\$ \mathcal{S}$ to all parties. If D is honest, all honest parties will eventually terminate the sharing phase. Moreover, for an honest dealer, ASKS provides the secrecy guarantee that during the sharing phase, the secret key s remains indistinguishable from an uniformly random key to an adversary $\mathcal{A}$. Alternatively, if D is malicious, the sharing phase may or may not terminate, but if it does terminate for one honest party, it will eventually terminate for all honest parties. Moreover, for a malicious dealer, the secret key s will be fixed by the first time some honest party finishes the sharing phase.

After an honest party finishes the sharing phase, it may start the reconstruction phase. If all honest parties start the reconstruction phase, then all honest parties will output the secret key s from the sharing phase. If D is honest, $\mathcal{A}$ only learns s after the first honest party starts the reconstruction phase.

Remark. We want to note that the sharing phase of an ASKS scheme permits a situation where, in the case of a malicious dealer, some honest parties do not receive correct shares but instead output the special symbol $\perp$ as their shares.

Definition 3.1 (Asynchronous Secret Key Sharing). An asynchronous secret key sharing (ASKS) protocol consists of two phases: *Sharing* and *Reconstruction*. During the sharing phase, a dealer D shares a secret $s \leftarrow\$ \mathcal{S}$. During the reconstruction phase, parties interact to recover the secret. We say that an ASKS protocol is t-secure if the following properties hold with probability $1 - \mathsf{negl}(\kappa)$ against any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ that corrupts up to t parties:

- *Correctness.* (i) If one honest party outputs in the sharing phase, then all honest parties eventually output in the sharing phase. (ii) If all honest parties start the reconstruction phase, then eventually they all output in the reconstruction phase.
- *Validity.* If an honest dealer shares a secret s, then (i) every honest party eventually outputs in the sharing phase, and (ii) no honest party outputs $s' \neq s$ in the reconstruction phase.
- *Commitment.* At the first time some honest party outputs in the sharing phase, there is a secret $s \in \mathcal{S}$ that can be computed from the joint view of the honest parties, such that no honest party outputs $s' \neq s$ in the reconstruction phase.
- *Secrecy.* If the dealer is honest, then before any honest party starts the reconstruction phase, the secret key s is computationally indistinguishable from a uniformly random secret in $\mathcal{S}$.

Clarification on ASKS terminology. Our ASKS notion is very similar to the "asynchronous weak VSS" notion defined in [21] (except that our ASKS is defined only for random secrets). We avoid using the term "weak VSS" because there exist similar-sounding notions in the literature [42] that are very different.

3.2 ASKS Design

For our construction, we require that the secret space is large, i.e., $1/|\mathcal{S}|$ is $\mathsf{negl}(\kappa)$. Our ASKS protocol uses reliable broadcast (RBC) and reliable agreement (RA) in a black-box way. Let $\mathsf{H} : [0, n] \times \mathbb{F} \rightarrow \mathcal{S}$ be a cryptographic hash function. In this paper, we prove our ASKS construction secure assuming a random oracle. Using ideas from the recent work of Shoup and Smart [44], the reliance on the random oracle model can be removed. We give the construction in Algorithm 2 and describe it next.

Sharing phase. In the sharing phase, the dealer D chooses a random polynomial $p \in \mathbb{F}[x]$ of degree t. The dealer then shares the random secret $s := \mathsf{H}(0, p(0))$ as follows. For each party j, D computes its share $p(j)$ and a commitment $h_j = \mathsf{H}(j, p(j))$. We reiterate that the secret being shared is $s := \mathsf{H}(0, p(0))$.

Next, D broadcasts the commitment vector $\boldsymbol{h} := [h_1, h_2, \ldots, h_n]$ using an RBC protocol. D also sends $p(i)$ to each party i via the private channel. Simultaneously, all non-dealer parties participate in a reliable agreement instance RA. Pary i inputs 1 to RA if it receives consistent values from D via the RBC and via the private channel, i.e., if $\boldsymbol{h}[i] = \mathsf{H}(i, p(i))$.

Algorithm 2 ASKS protocol for party i

```
SHARING PHASE:
1:  if $i$ is the dealer then
2:      Let $p(\cdot)$ be a random degree-$t$ polynomial
3:      Let $h_j = \mathsf{H}(j, p(j))$ for each $j \in [n]$
4:      broadcast $\boldsymbol{h} = [h_1, h_2, \ldots, h_n]$ using a RBC
5:      send $\langle \mathsf{SHARE}, p(i) \rangle$ to party $i$
6:  start a reliable agreement protocol instance RA
7:  upon RBC outputs $\boldsymbol{h}$ and having received $\langle \mathsf{SHARE}, p(i) \rangle$
8:      if $\boldsymbol{h}[i] = \mathsf{H}(i, p(i))$ then
9:          input 1 to RA
10: upon RA outputs 1 and RBC outputs $\boldsymbol{h}$
11:     if received $\langle \mathsf{SHARE}, p(i) \rangle$ and $\boldsymbol{h}[i] = \mathsf{H}(i, p(i))$ then
12:         output $(\boldsymbol{h}, p(i))$ and terminate
13:     output $(\boldsymbol{h}, \perp)$ and terminate
```

```
RECONSTRUCTION PHASE:
    // every party i after finishing the sharing phase
14: Let $(\boldsymbol{h}, s_i)$ be its output from the Sharing phase.
15: if $s_i \neq \perp$ then
16:     send $\langle \mathsf{RECON}, s_i \rangle$ to all
17: Let $T = \{\}$ // the set of valid shares received so far
18: upon receiving $\langle \mathsf{RECON}, s_j \rangle$ from party $j$
19:     if $\boldsymbol{h}[j] = \mathsf{H}(j, s_j)$ then
20:         $T = T \cup \{s_j\}$
21:         if $|T| = t + 1$ then
22:             Let $p_i(x)$ be the polynomial defined by $T$
23:             if $\boldsymbol{h}[j] = \mathsf{H}(j, p_i(j))$ for all $j \in [n]$ then
24:                 output $\mathsf{H}(0, p_i(0))$ and terminate
25:             output $0 \in \mathcal{S}$ and terminate
```

All parties then wait for RA to output 1 (which may occur even if party i did not input 1 to RA) — if and when that happens, party i outputs either $(\boldsymbol{h}, p(i))$ or $(\boldsymbol{h}, \perp)$, depending on whether or not it received a valid share $p(i)$ from D.

Reconstruction phase. During the reconstruction phase, each party i who received a valid share $s_i \neq \perp$ during the sharing phase sends its share s_i to all. Upon receiving s_j from party j, party i accepts s_j as valid if $\boldsymbol{h}[j] = \mathsf{H}(j, s_j)$. Let T be the set of valid shares party i receives during the reconstruction phase. Upon receiving $t+1$ valid shares, party i uses them to interpolate a polynomial $p_i(x)$. Party i then checks whether $h_j = \mathsf{H}(j, p_i(j))$ for each $j \in [n]$. If the check passes, party i outputs $s := \mathsf{H}(0, p_i(0))$ as the reconstructed secret; otherwise, i outputs some default value in $\mathcal{S}$. (The default value could be a special "error" value that indicates the dealer was malicious.)

3.3 ASKS Analysis

The Validity of our ASKS scheme follows directly from the Validity of RBC and RA. Correctness is also relatively easy to prove.

Lemma 3.2 (Correctness). *If the hash function $\mathsf{H} : [0, n] \times \mathbb{F} \to \mathcal{S}$ is collision-resistant, then Algorithm 2 ensures correctness as per Definition 3.1.*

Proof. Part (i). An honest party terminates the sharing phase if and only if the RBC terminates and the RA outputs 1. The totality properties of the RBC and RA ensure that both the RBC and the RA will terminate at all honest parties. Furthermore, the agreement property of RA ensures that all honest parties will output 1 from RA, and hence terminate the Sharing phase.

Part (ii). An honest party starts the reconstruction phase only if terminates from the sharing phase. The sharing phase terminates only if RA outputs 1. By the Integrity property of RA, at least $n - 2t > t + 1$ honest parties input 1, indicating that they receive valid shares. Thus, in the reconstruction phase, every honest party will receive a set T of $t + 1$ valid shares and output. □

Next, we focus on the Commitment and Secrecy properties of our ASKS. We prove the Commitment property assuming the collision resistance of the hash function H. We prove the secrecy property by modeling H as a random oracle.

Lemma 3.3 (Commitment). *Assuming $\mathsf{H} : [0, n] \times \mathbb{F} \to \mathcal{S}$ is a collision-resistant hash function, Algorithm 2 ensures Commitment as per Definition 3.1.*

Proof. Consider the first point in time that some honest party terminates the sharing phase. Following the proof of Correctness part (ii), each honest party i will receive $t + 1$ valid shares and reconstruct some degree t polynomial $p_i(x)$.

The committed secret s can be extracted as follows. Let $T \subseteq [n]$ be an arbitrary set of $t+1$ honest parties who received consistent shares as per the above. Let $\tilde{p}(x)$ be the degree t polynomial interpolated from these $t + 1$ shares. Now, there are two possibilities.

First, if the vector $\boldsymbol{h}$ is consistent with the polynomial $\tilde{p}(x)$, i.e., $\boldsymbol{h}[i] = \mathsf{H}(i, \tilde{p}(i))$ for all $i \in [n]$, then let $s = \mathsf{H}(0, \tilde{p}(0))$. By the collision resistance of H, $p_i(x) = \tilde{p}(x)$ for every honest party i. Hence, every honest party will output s as the reconstructed secret.

Second, if $\boldsymbol{h}$ is inconsistent with $\tilde{p}(x)$, i.e., there exists an index $k \in [n]$ such that $\boldsymbol{h}[j] \neq \mathsf{H}(k, \tilde{p}(k))$. Let s be the default secret. Clearly, an honest party i with $p_i(x) = \tilde{p}(x)$ will output the default secret. It remains to show that an honest party j with $p_j(x) \neq \tilde{p}(x)$ will also output the default secret. Note that $p_j(x) \neq \tilde{p}(x)$ implies that there exists some $k \in T$ such that $p_j(k) \neq \tilde{p}(k)$. For this k, by the collision resistance property of H, $\boldsymbol{h}[k] = \mathsf{H}(k, \tilde{p}(k)) \neq \mathsf{H}(k, p_j(k))$. This implies that party j will also output the default secret. □

Lemma 3.4 (Secrecy). *If we model the hash function $\mathsf{H} : [0, n] \times \mathbb{F} \to \mathcal{S}$ as a random oracle, then Algorithm 2 ensures secrecy as per Definition 3.1.*

Proof. Let $\mathcal{A}$ be the PPT adversary. Let C be the set of corrupt parties, and $\mathcal{H} := [n] \setminus C$ be the set of honest parties. $\mathcal{A}$ learns the t evaluation points on the random degree t polynomial $p(x)$ for $x \in C$ and the $n - t$ random oracle outputs $\mathsf{H}(i, p(i))$ for each $i \in \mathcal{H}$. The perfect secrecy of the Shamir secret sharing implies that $\mathcal{A}$ learns no information about $p(0)$ from the t shares. When we model H as a random oracle, the random oracle outputs $\mathsf{H}(i, p(i))$ for all $i \in \mathcal{H}$ do not reveal any information about $p(0)$ unless $\mathcal{A}$

Algorithm 3 Index gather protocol for party i

Input: Valid_i, the set of parties that p_i has validated so far *// Valid_i is a growing set and satisfies completeness (see §2.2)*

```
1:  upon |Valid_i| = n − t
2:      Let S_i := Valid_i
3:      send ⟨INFORM, S_i⟩ to all
4:  upon S_j ⊆ Valid_i becomes true for ⟨INFORM, S_j⟩ received from party j
5:      send ⟨ACK⟩ to party j // if not done already
6:  upon receiving ACK from n − t distinct nodes
7:      Let T_i := Valid_i
8:      send ⟨PREPARE, T_i⟩ to all
9:  Let C_i := {} // set of indices of nodes from whom party i received a valid PREPARE message
10: upon T_j ⊆ Valid_i becomes true for ⟨PREPARE, T_j⟩ received from party j
11:     C_i := C_i ∪ {j}
12:     if |C_i| = n − t then
13:         output X_i := ⋃_{j∈C_i} T_j and terminate
```

queries the random oracle on some $(i, p(i))$. Since $\mathcal{A}$ makes only polynomially many random oracle queries, and all $p(i)$'s for $i \in \mathcal{H}$ remain random (though fully dependent on one another) given the t shares, the probability that $\mathcal{A}$ queries the random oracle on some $(i, p(i))$ is negligible. Hence, from $\mathcal{A}$'s view, $s := \mathsf{H}(0, p(0))$ and $\hat{s} \leftarrow\$ \mathcal{S}$ are indistinguishable. ◻

Performance analysis. Clearly, the message complexity of our ASKS protocol is $O(n^2)$. Assuming hash outputs and field elements are κ-bit long, the message the dealer broadcasts is $O(\kappa n)$ bits in size. Hence, using the RBC protocol of [19], the communication cost of the sharing phase of our ASKS protocol is $O(\kappa n^2)$ bits. During the reconstruction phase, each party sends $O(\kappa)$-bit messages to all, so the total communication cost is also $O(\kappa n^2)$.

4 Index Cover Gather

In this section, we will first give an index gather protocol. The protocol we use is embedded in the multivalued validated Byzantine agreement protocol of the FIN ACS [24, Section 5.3]. We distill their gather protocol and adapt it to an index version. Next, we will describe how we can transform the index gather protocol to have an additional important property called binding cover that we will define later.

4.1 Index Gather

The index gather protocol is given in Algorithm 3. We will use the notation of parties inputting validating of each other described in §2.2.

Design. Let Valid_i denote the set of parties that party i has validated. Note that Valid_i grows over time. Each party i waits until it has validated $n-t$ parties and sends this set of $n-t$ parties to all in a message $\langle \mathsf{INFORM}, S_i \rangle$. Party i, upon receiving $\langle \mathsf{INFORM}, S_j \rangle$ from party j, waits until $S_j \subseteq \mathsf{Valid}_i$ becomes true (note that Valid_i grows). In other words, party i waits until it has validated all the parties that j claims to have validated. When this happens, party i responds back to party j with an ACK message.

A party i then waits to receive $n-t$ ACK messages from other parties. Party i then sends to all its Valid_i set at that time in a message $\langle \mathsf{PREPARE}, T_i \rangle$. Finally, each party i waits for $n-t$ distinct PREPARE messages $\langle \mathsf{PREPARE}, T_j \rangle$ whose $T_j \subseteq \mathsf{Valid}_i$. Note again that party i may need to give its own input set Valid_i time to grow for this to finally happen. Let C_i be the set of parties from which i has received such a T_j. Party i then outputs its gather set X_i as:

$$X_i = \bigcup_{j \in C_i} T_j \qquad (2)$$

Security analysis. Note that in Algorithm 3, each party i outputs $X_i \subseteq \mathsf{Valid}_i$, so our index gather protocol ensures the validity property.

Next, we prove, given that the input validation mechanism satisfies the *completeness* property, then the index gather protocol in Algorithm 3 satisfies termination.

Lemma 4.1 (Termination). *If party validation in the inputs to Algorithm 3 protocol satisfies completeness and at least $n-t$ parties have been locally validated at some point during the protocol, then every honest party i eventually outputs X_i.*

Proof. The completeness property of party validation in the inputs ensures that the $n-t$ locally validated parties will eventually become globally validated. Hence, eventually, $|\mathsf{Valid}_i| \geq n-t$ for all $i \in \mathcal{H}$. This implies that all honest parties will send a INFORM message.

The completeness property of party validation in the inputs also ensures that for every $i, j \in \mathcal{H}$, eventually $S_j \subseteq \mathsf{Valid}_i$ and $T_j \subseteq \mathsf{Valid}_i$. This implies that each honest party i will receive at least $n-t$ ACK messages, send PREPARE message to all, receive $n-t$ PREPARE messages, and will have a set C_i with $|C_i| \geq n-t$. Hence, each honest party will output X_i and terminate. ◻

Next, we prove Algorithm 3 satisfies the binding core property.

Lemma 4.2 (Binding core). *In Algorithm 3, let i be the first honest party to output X_i, then S_i is the binding core, i.e., $X = S_i$ and for every honest party j, $X \subseteq X_j$.*

Proof. Any honest party will output in the gather protocol only upon receiving $n-t$ PREPARE message, i.e., only after $n-2t$ honest parties have sent their PREPARE message.

Consider the first point in time that some honest party, say i, sends its PREPARE message. At this time, party i has received ACK messages from $n-t$ parties, and at least $n-2t$ of those are honest. Let H be these $n-2t$ honest parties.

Each party $j \in H$ sends its PREPARE message after party i sends PREPARE, which is in turn after party j sends ACK to party i. Valid_j is a superset of S_i when party j sends ACK to party i, and T_j takes on the value of Valid_j at a later time. Thus, $S_i \subseteq T_j, \forall j \in H$.

For any honest party k, by quorum intersection, it must be the case that $H \cap C_k \neq \emptyset$. This implies that $S_i \subseteq X_k$. S_i can hence serve as the required binding core set. ◻

Performance analysis. Clearly, Algorithm 3 has $O(n^2)$ message complexity and $O(n^3)$ communication complexity.

Algorithm 4 Index cover gather protocol for party i

```
Input: Valid_i, the set of parties that p_i has validated so far // Note that Valid_i is a growing set.
1:  start n parallel reliable agreement instances, one for each j ∈ [n]. Let RA_j for be the j-th reliable agreement instance.
2:  Let IGValid_i := {} // input of the index gather protocol
3:  start index gather with IGValid_i as input
4:  Let withdraw_i := False
5:  upon validating a new party j, i.e., when j is added to Valid_i
6:      if withdraw_i = False then
7:          input 1 to RA_j
8:  upon RA_j outputs 1
9:      IGValid_i := IGValid_i ∪ {j}
10:     if |IGValid_i|= n − t then
11:         withdraw_i := True
12:         send ⟨WITHDRAW⟩ to all
13: upon receiving ⟨WITHDRAW⟩ from n − t parties
14:     wait until the index gather protocol outputs X_i
15:     output X_i and terminate
```

4.2 Index Cover Gather

Definition. We will now strengthen the index gather protocol above to what we call an index *cover* gather (ICG) protocol. An index cover gather protocol is an index gather protocol that additionally satisfies the *binding cover* property below.

Definition 4.3 (Binding cover). At the first time some honest party i outputs, there exists a locally validated set of parties Y, dependent only on the joint views of the honest parties at that time, such that $X_j \subseteq Y$ for all honest party $j \in [n]$. We call Y the *cover* set.

It may be helpful to contrast the binding cover property with the binding core property. A *core* set is a *subset* of every honest party's eventual output. A *cover* set is a *superset* of every honest party's eventual output. The term *binding* means that these two sets can be determined by the first time some honest party outputs.

We will next present a construction for ICG using an index gather protocol and reliable agreement. We give the protocol in Algorithm 4 and describe it next.

Design. Let set Valid_i be the input of party i to our ICG protocol. Similar to index gather, Valid_i is the growing set of parties that party i has validated so far.

Parties participate in n parallel reliable agreement instances, one for each $j \in [n]$. Let RA_j be the j-th reliable agreement instance. Each party i also maintains a boolean variable $\mathsf{withdraw}_i$, initially set to False (but will eventually be set to True). Intuitively, the variable $\mathsf{withdraw}_i$ indicates whether or not party i will continue to input 1 to remaining reliable agreement instances. Parties also start an index gather protocol. $\mathsf{IGValid}_i$ will be party i's input to the index gather protocol, which is also a growing set of party indices.

Whenever party i has validated j, i.e., j gets added to Valid_i, party i inputs 1 to RA_j, unless $\mathsf{withdraw}_i$ has been set to True (in which case we say party i has *withdrawn*). Next, whenever party i outputs 1 from some RA_j, it adds the index j to the set $\mathsf{IGValid}_i$. This means party i has validated party j for the purpose of index gather (in Algorithm 3). Note that RA satisfies agreement and totality. Therefore, $\mathsf{IGValid}_i$ satisfies our completeness requirement as inputs to index gather.

When the size of $\mathsf{IGValid}_i$ reaches $n-t$, party i sets $\mathsf{withdraw}_i :=$ True, thereby withdrawing from inputting 1 to any remaining RA instance. Party i also sends a ⟨WITHDRAW⟩ message to all. We note that parties continue to participate (i.e., send READY messages whenever needed in Algorithm 1) in all RA instances even after withdrawing.

Let X_i be the output of the index gather protocol. Party i then waits to receive ⟨WITHDRAW⟩ from $n-t$ distinct parties, and outputs X_i as its index cover gather (ICG) output.

Security analysis. The validity and binding core properties of our ICG protocol follow from the validity and binding properties of the index gather protocol we use. We now prove that our ICG protocol terminates if $n-t$ parties are globally validated during any time of the protocol. (Note that the two conditions in the termination property imply that $n-t$ parties are globally validated.)

Lemma 4.4 (Termination). *If $n-t$ parties are globally validated, then Algorithm 4 terminates.*

Proof. We first show that if at least $n-t$ parties are globally validated in the inputs to ICG, then some honest party will eventually withdraw. For the sake of contradiction, suppose no honest party ever withdraws. Then, all honest parties will input 1 to all RA instances of these $n-t$ globally validated parties. Then, by the totality property of RA, all these $n-t$ RA instances will output 1. As a result, all honest parties will withdraw, which is a contradiction.

Now, a party withdraws only after $n-t$ RA instances output 1. By the totality property of RA, all honest parties will eventually withdraw. Let i be some honest party that withdraws. By the totality property of RA, all parties in $\mathsf{IGValid}_i$ will eventually be globally validated in the inputs to index gather, and these inputs to index gather have the completeness property. Thus, the index gather protocol will eventually terminate. Therefore, every honest party will eventually receive $n-t$ ⟨WITHDRAW⟩ messages as well as its output from the index gather protocol, and will output from the ICG protocol. □

We now prove that our ICG protocol in Algorithm 4 satisfies the binding cover property. Recall from Definition 4.3 that we need to prove that at the first time some honest party outputs, a cover set of validated parties Y exists such that every honest party's eventual output is a subset of Y.

Lemma 4.5 (Binding Cover). *Algorithm 4 satisfies binding cover.*

Proof. Consider the first point in time that an honest party produces an output. At this point in time, at least $n-2t$ honest parties must have already withdrawn. Thus, if any party j has not been locally validated yet, at most t honest parties will input 1 to RA_j. By the integrity property of RA, this is insufficient for RA_j to output 1. This implies that j will never appear in $\mathsf{IGValid}_i$ for any honest party i. Then, by the validity property of index gather, j can never appear in the output set of any honest party. Therefore, we can define the cover set Y to be:

$$Y := \bigcup_{i \in \mathcal{H}} \mathsf{IGValid}_i. \quad (3)$$

where we use IGValid of honest parties at the time when the first honest party outputs from the index gather protocol. □

Performance analysis. Our ICG protocol runs n parallel binary reliable agreement and a single index gather. The communication cost of our protocol is thus $O(n^3)$.

5 Index Validated Asynchronous Byzantine Agreement

In this section, we will present our index validated asynchronous byzantine agreement (VABA) protocol. Our index VABA protocol uses the following primitives in a black-box manner: (i) RBC , (ii) ICG, (iii) ASKS, and (iv) RA.

Party ranks. Looking ahead, our index VABA protocol has a step where we assign a random rank to each party. Parties then take further actions based on the ordering of ranks for (subsets of) parties. Intuitively, we require these ranks to satisfy: (i) ranks of parties are independent of each other and uniformly random from a sufficiently large space; and (ii) rank of any party is not revealed to the adversary until the VABA protocol reaches a certain stage.

For ease of presentation, we will first present in §5.1 our index VABA protocol, assuming a trusted oracle that assigns ranks that satisfy the above mentioned properties. We want to note that with such a trusted rank oracle, we only need to use an index gather protocol (without the binding cover property). Later in §5.3, we will present the complete protocol, where we derive ranks of parties carefully using ASKS and replace the index gather protocol with an index cover gather (ICG) protocol.

5.1 Index VABA with a Rank Oracle

We give our index VABA protocol with a rank oracle in Algorithm 7. The input of the i-th party to the index VABA protocol is the set Valid_i of parties it has validated so far. Again, the set Valid_i grows over time, and we require the party validation in inputs to satisfy the completeness property (see §2.2).

Our index VABA protocol proceeds in *views* $v = 0, 1, \ldots$, where each view consists of the following steps. To aid understanding, we will break down the algorithm into parts.

Prevote (lines 1-4). In each view v, each party i will "prevote" for one party. Let $\mathsf{pre}_{i,v} \in [n]$ denote the index of the party that i prevotes for in view v. In view v, party i will only prevote for a party in its Valid_i set (at the time i chooses $\mathsf{pre}_{i,v}$). Naturally, in view 0, party i chooses the first party added to its Valid_i set. In any view $v \geq 1$, a prevote must be accompanied by a "justification", denoted $\mathsf{justify}_{i,v}$. In view 0, justification is not needed and $\mathsf{justify}_{i,0} := \emptyset$.

Then, party i broadcasts $(\mathsf{pre}_{i,v}, \mathsf{justify}_{i,v})$ using an RBC. Hereon, when clear from the context, we will drop the subscript v. Thus, the above message is written as $(\mathsf{pre}_i, \mathsf{justify}_i)$ for readability.

Running the index gather protocol (lines 5-12). Each party i then prepares an input $\mathsf{IGValid}_i$, initially empty, for the index gather protocol they will run in view v. We reiterate that we only require an index gather (without the binding cover property) with the help of the rank oracle.

Let $(\mathsf{pre}_j, \mathsf{justify}_j)$ be the message (if any) that party j broadcasts using RBC. Party i adds j to $\mathsf{IGValid}_i$ only upon:

(1) pre_j is in Valid_i.

(2) for views $v \geq 1$, $\mathsf{justify}_j \subseteq M_{i,v-1}$ (for $M_{i,v-1}$ we define later), $|\mathsf{justify}_j| \geq n - t$, and pre_j is a most frequent vote in $\mathsf{justify}_j$ (i.e., appears at least as often as any other).

Parties then run an instance of the index gather protocol where party i uses $\mathsf{IGValid}_i$ as input.

Computing ranks (line 13). Each party i then waits for the index gather protocol to finish. Let X_i be its output from the index gather protocol. Party i then computes the rank of each party $j \in X_i$ by querying the rank oracle. The rank oracle reveals the ranks of parties only *after* at least one honest party outputs in the index gather protocol.

Algorithm 5 Index VABA protocol for party i with a rank oracle

```
Input: $\mathsf{Valid}_i$ the set of parties, party $i$ has validated so far // $\mathsf{Valid}_i$ is a growing set and has the completeness property
// The protocol proceeds in views $v = 0, 1, 2, \ldots$, where in $v$:
1:  if $v = 0$ then
2:      Let $\mathsf{pre}_i$ be the first element added to $\mathsf{Valid}_i$
3:      Let $\mathsf{justify}_i := \emptyset$ // first justification is empty
4:  broadcast $(\mathsf{pre}_i, \mathsf{justify}_i)$ using an RBC denoted $\mathsf{RBC}_{\mathsf{pre},i}$
    // Running an index gather instance
5:  Let $\mathsf{IGValid}_i := \{\}$ // input to the index gather protocol
6:  start the index gather protocol with $\mathsf{IGValid}_i$ as input
    // validating party j's prevote
7:  upon $\mathsf{pre}_j \in \mathsf{Valid}_i \wedge \mathsf{justify}_j \subseteq M_{i,v-1}$ becomes true where
    $(\mathsf{pre}_j, \mathsf{justify}_j)$ is the output of $\mathsf{RBC}_{\mathsf{pre},j}$
8:      if $v \geq 1$ then
9:          assert $|\mathsf{justify}_j| \geq n - t$
10:         assert $\mathsf{pre}_j$ is a most frequent vote in $\mathsf{justify}_j$
11:     $\mathsf{IGValid}_i := \mathsf{IGValid}_i \cup \{j\}$ // if all assertions are true
12: wait until the index gather protocol outputs $X_i$
    // Getting the ranks from an oracle
13: For each party $j \in X_i$, let $\mathsf{rank}_{j,v}$ be its rank in view $v$
    // Trying to agree on the index with maximum rank
14: Let $\ell \in X_i$ be the index with maximum rank among $X_i$
15: Let $\mathsf{vote}_i := \mathsf{pre}_\ell$ where $(\mathsf{pre}_\ell, \cdot)$ is the output of $\mathsf{RBC}_{\mathsf{pre},\ell}$
16: broadcast $\mathsf{vote}_i$ using an RBC denoted $\mathsf{RBC}_{\mathsf{vote},i}$
17: Let $M_{i,v} := \{\}$ // set of finished vote RBC and their outputs
18: upon $\mathsf{vote}_j \in \mathsf{IGValid}_i$ becomes true where $\mathsf{vote}_j$ is the output
    of $\mathsf{RBC}_{\mathsf{vote},j}$
19:     $M_{i,v} := M_{i,v} \cup \{(j, \mathsf{vote}_j)\}$
20:     if $|M_{i,v}| = n - t$ then
21:         $\mathsf{justify}_i := M_{i,v}$ // to be used in the next view
22:         Set $\mathsf{pre}_i$ to a most frequent vote in $\mathsf{justify}_i$
// One reliable agreement across all views, as a termination gadget
23: start a reliable agreement instance RA
24: upon $M_{i,v}$ (for any $v$) contains $n - t$ matching $(\cdot, k^*)$
25:     input $k^*$ to RA // input only once
26:     participate in view $v + 1$ but not $v + 2$ or later views
27: upon RA outputs $k^*$
28:     output $k^*$ and terminate
```

Trying to agree on the maximum rank (lines 14-22). Let $\ell \in X_i$ be the party with the maximum rank among all parties in X_i. Party i will adopt the prevote of ℓ as its own vote and broadcast its vote using an RBC. Party i also maintains a set $M_{i,v}$, initially empty, consisting of tuples of vote RBC instances and their outputs (j, vote_j) upon vote_j becomes locally validated by the index gather, i.e., $\mathsf{vote}_j \in \mathsf{IGValid}_i$ becomes true. More precisely, for the j-th RBC output vote_j, party i adds party j to $M_{i,v}$ only upon $\mathsf{vote}_j \in \mathsf{IGValid}_i$.

When the set $M_{i,v}$ reaches size $n-t$, party i computes its prevote for the next view as follows. Party i sets $\mathsf{justify}_i := M_{i,v}$ and pre_i to a most frequent vote in $\mathsf{justify}_i$.

Final output (lines 23-28). If it ever happens that $M_{i,v}$ contains $n-t$ matching tuples $(\cdot, \mathsf{vote})$ for some vote, party i inputs vote to an reliable agreement instance RA. It is important to note that there is only a single reliable agreement instance RA across all views and parties participate in RA even if they have not input anything to RA. It is also important that, after inputting to RA, a party will participate in one more view, but not in any view after that. Finally, when RA outputs k^*, a party outputs k^* as the index VABA output and terminates.

5.2 Index VABA Analysis

We now prove that Algorithm 5 is a secure index VABA protocol given a trusted rank oracle.

Lemma 5.1 (Validity). *Algorithm 5 satisfies validity.*

Proof. The integrity property of the RA ensures that the output of RA is input by at least $n-2t$ honest parties. An honest party i inputs vote_i to RA only when vote_i appears in $M_{i,v}$ for some view v. Now, node i adds vote_i to $M_{i,v}$ only upon $\mathsf{vote}_i \in \mathsf{IGValid}_i$, and hence only upon $\mathsf{vote}_i \in \mathsf{Valid}_i$. □

Proof of agreement. The agreement property of our index VABA in Algorithm 5 is straightforward from the agreement property of the termination gadget RA. However, we still need to prove that honest parties provide the same input to RA because RA only guarantees termination in the case of matching inputs. (In other words, the termination gadget RA converts a potential violation of agreement to a violation of termination.) This is a proof of agreement in essence, and we will refer to it as such.

Lemma 5.2. *If in view v, $n-t$ vote RBC instances output the same value k^*, then in view $v+1$, every honest party will input k^* to RA, if it has not input to RA in previous views.*

Proof. If $n-t$ vote RBC instances output the same value k^*, every other honest party will receive at least $n-2t$ of these RBC outputs of k^* among their $n-t$ RBC outputs. Thus, k^* will be the most frequent vote in $M_{j,v}$ for every honest party j, and party j will set its $\mathsf{pre}_{j,v+1}$ to k^* in view $v+1$.

In view $v+1$, every honest party i will check $|\mathsf{justify}_{j,v+1}| \geq n-t$, $\mathsf{justify}_{j,v+1} \subseteq M_{i,v}$, and $\mathsf{pre}_{j,v+1}$ is a most frequent vote in $\mathsf{justify}_j$. This ensures that a malicious party will be added to IGValid only if $\mathsf{pre}_{j,v+1} = k^*$. This implies that in view $v+1$, regardless of the ranks of the parties, an honest party i will only populate $M_{i,v+1}$ with tuples $(\cdot, k^*)$ and hence will input k^* to RA, if it has not input to RA in previous views. □

Lemma 5.3 (Agreement). *Honest parties do not provide distinct inputs to* RA *in Algorithm 5.*

Proof. Consider the smallest view v in which some honest party inputs to RA. Suppose this party inputs k^* to RA only upon receiving $n-t$ matching RBC outputs of k^*. By Lemma 5.2, every honest party will input k^* to RA in view $v+1$, unless it has input to RA in previous views. Since v is the smallest view in which some honest party inputs to RA, we only need to consider view v. For another honest party to input k' to RA in view v, it must receive $n-t$ matching RBC output of k' in view v. By a standard quorum intersection, $k' = k^*$ □

Recall that we require ranks to be independent, uniformly random from a large space, and hidden until the protocol reaches a certain stage. Concretely, we assume each rank is an independent and random κ-bit integer where κ is a security parameter, and ranks in view v remain hidden until some honest party outputs in the index gather protocol of view v.

Next, we will prove the following simple but crucial Lemma that says our index VABA protocol terminates in each view with a probability of at least $2/3$.

Lemma 5.4. *With at least $2/3$ probability, an index in the core set X has the unique maximum rank.*

Proof. The probability that all ranks are distinct, i.e., there is no collision, is at least $1 - n^2 2^{-\kappa}$. Conditioned on no collision, each index has a $1/n$ probability of being the maximum rank. The binding core property ensures that X is fixed when the first honest party outputs from the index gather, at which point all ranks remain hidden from the adversary. Thus, X is independent of the ranks, and the probability that the maximum rank belongs to some index in X is $|X|/n \geq (n-t)/n$. Therefore, the probability that an index in X has the unique maximum rank is at least $(1 - n^2 2^{-\kappa}) \cdot \frac{n-t}{n}$, which is greater than $2/3$ for $n \geq 3t+1$ and sufficiently large κ. □

Lemma 5.5 (Termination). *Algorithm 7 ensures termination.*

Proof. From Lemma 5.4, in each view v, a party $\ell \in X$ has the unique maximum party with probability at least $2/3$. When this happens, ℓ also has the unique maximum rank among each honest party i's index gather output X_i (note that $X_i \supseteq X$).

In this case, every honest party i will set its $\mathsf{vote}_{i,v}$ to be $\mathsf{pre}_{\ell,v}$. There will be $n-t$ RBCs that output $\mathsf{pre}_{\ell,v}$. Then, by Lemma 5.2, all honest parties will input to RA in view $v+1$, if not in smaller views. By Lemma 3.4, honest parties can only input the same value to RA, so RA terminates in view $v+1$.

Furthermore, an honest party participates in only one more view after inputting to RA. Therefore, every honest party participates in expected constant number of views before terminating. □

5.3 Index VABA without a Rank Oracle

In this section, we will describe how we can instantiate the rank oracle for each view using n parallel ASKS instances and an index cover gather (ICG) protocol.

We give the protocol in Algorithm 6 and highlight the changes from Algorithm 5 in gray. We describe these changes next. The first

Algorithm 6 Index VABA protocol for party i without a rank oracle

```
Input: Valid_i the set of parties, party i has validated so far // Valid_i is a growing set and has the completeness property
The protocol proceeds in views v = 0, 1, ..., where in each view v:
1:  if $v = 0$ then
2:      Let $\mathsf{pre}_i$ be the first element added to $\mathsf{Valid}_i$
3:      Let $\mathsf{justify}_i := \emptyset$ // first justification is empty
4:  start $n$ ASKS instances and act as the dealer in the $i$-th ASKS
5:  Let $\mathsf{Shared}_i$ be the (growing) set of ASKS instances whose sharing phase finished at $i$
6:  upon $|\mathsf{Shared}_i| = t + 1$ for the first time
7:      Let $P_i := \mathsf{Shared}_i$ // to propose this set to all
8:      broadcast $(\mathsf{pre}_i, P_i, \mathsf{justify}_i)$ using $\mathsf{RBC}_{\mathsf{pre},i}$
    // Running an index cover gather instance
9:  Let $\mathsf{IGValid}_i := \{\}$ // input to the index gather protocol
10: start the index cover gather protocol with $\mathsf{IGValid}_i$ as input
    // validating party j's prevote
11: upon $\mathsf{pre}_j \in \mathsf{Valid}_i \wedge \mathsf{justify}_j \subseteq M_{i,v-1} \wedge P_j \subseteq \mathsf{Valid}_i$ becomes true where $(\mathsf{pre}_j, P_j, \mathsf{justify}_j)$ is the output of $\mathsf{RBC}_{\mathsf{pre},j}$
12:     assert $|P_j| \geq t + 1$
13:     if $v \geq 1$ then
14:         assert $|\mathsf{justify}_j| \geq n - t$
15:         assert $\mathsf{pre}_j$ is a most frequent vote in $\mathsf{justify}_j$
16:     $\mathsf{IGValid}_i := \mathsf{IGValid}_i \cup \{j\}$ // if all assertions are true
17: wait until the index cover gather protocol outputs $X_i$
    // Computing the rank of parties
18: start reconstruction of all ASKS instances in $\mathsf{Shared}_i$. If a new index gets added to $\mathsf{Shared}_i$, reconstruct that as well.
19: wait until reconstruction of all secrets in $\bigcup_{j \in X_i} P_j$ finishes. Let $s_k$ be the reconstructed secret of $k$-th ASKS.
20: Compute for each party $j \in X_i$, $\mathsf{rank}_{j,v} = \sum_{k \in P_j} \mathsf{H}_{\mathsf{Rank}}(j, s_k)$
// The rest of the protocol is the same as lines 14 to 28 in Algorithm 5
```

change is to replace the index gather protocol in Algorithm 5 with an index cover gather (ICG) protocol.

ASKS sharing (lines 4-8). Before running the index (cover) gather protocol in each view, parties start n ASKS sharing instances, with party i as the dealer for the i-th instance. Each party i as the dealer shares a uniformly random secret $s_i \in \mathcal{S}$. Each party i then maintains a set Shared_i of ASKS instances whose sharing phases finished at party i. Note that Shared_i grows over time. Let P_i be the first set of $t + 1$ ASKS instances that terminate at party i. Party i adds P_i to its prevote RBC.

Running the index cover gather protocol (steps 9-17). As the voting RBC of party j now additionally includes P_j, two extra conditions need to become satisfied before we add j to the input of ICG: P_j contains $t + 1$ parties that i has validated (i.e., $P_j \subseteq \mathsf{Valid}_i$).

ASKS reconstruction and computing ranks (lines 18-20). Each party i then waits for the ICG protocol to finish. Party i initiates the reconstruction phase of all the ASKS instances in Shared_i (and will continue to do so for other ASKS instances as the set Shared_i continues to grow). Let X_i be the output of party i from the ICG protocol. Let T_i be the union of ASKS instances proposed by all parties in X_i, i.e.,

$$T_i := \bigcup_{j \in X_i} P_j \tag{4}$$

Each party i waits to reconstruct the secrets for all indices in T_i. Then, for each $j \in X_i$, the rank of party j is computed as:

$$\mathsf{rank}_j \leftarrow \sum_{k \in P_j} \mathsf{H}_{\mathrm{Rank}}(j, s_k) \tag{5}$$

Here, $\mathcal{R}$ denote the finite space of ranks with $1/|\mathcal{R}|$ being negligible in κ, and $\mathsf{H}_{\mathrm{Rank}} : [n] \times \mathcal{S} \to \mathcal{R}$ is a hash function modeled as an random oracle.

The rest of the protocol is identical to the index VABA protocol with a rank oracle in §5.3.

Analysis of the rank assignment protocol. At the time the first honest party starts reconstructing a secret, let Y be the binding cover set of the index cover gather (ICG) protocol. We now prove that before any honest party starts the reconstruction, ranks of all parties in Y are fixed, and are computationally indistinguishable from uniformly random ranks.

Lemma 5.6. *In each view of Algorithm 6, before any honest party starts the reconstruction, ranks of all parties in Y are fixed, and are computationally indistinguishable from uniformly random ranks.*

Proof. Recall the rank computation in Equation (5). Party i has committed to the set P_i via RBC before any honest party starts reconstructing the secrets. This implies the rank of each party in Y is fixed before any party starts reconstruction.

Now, for each $i \in Y$, P_i contains some honest party since $|P_i| \geq t + 1$. Let j be an honest party in P_i. Then, by the secrecy property of the ASKS scheme, for each $i \in Y$ the secret key s_j used in computing the rank of party i is computationally indistinguishable from uniform random, and hence so is the output $\mathsf{H}_{\mathrm{Rank}}(i, s_j)$. □

Analysis of the index VABA protocol. Observe that the proofs of validity and agreement in Section 5.2 did not depend on the ranks. For termination, Lemma 5.4 can be easily adapted to show that, with probability at least 2/3, an index $\ell \in X$ has the unique maximum rank among the cover set Y. When this happens, ℓ also has the unique maximum rank among each honest party i's index cover gather (ICG) output X_i, because $X \subseteq X_i \subseteq Y$. This is sufficient for the proof of Lemma 5.5.

6 Asynchronous Common Subset and its Applications

In this section, we will first present our index asynchronous common subset (ACS) protocol, and then describe how we can build a standard ACS using an index ACS and RBC in an black-box manner. Lastly, we also briefly describe other applications of index ACS such as hash-based common coin and asynchronous DKG.

6.1 Index ACS

Our index ACS design is based on the standard known technique of using a VABA protocol to construct an ACS protocol [2, 4, 20, 24]. We summarize our protocol in Algorithm 7, and describe it next.

Algorithm 7 Index ACS protocol for party i

```
Input: Valid_i the set of parties, party i has validated so far // Valid_i is a growing set and has the completeness property
1: Let $\mathsf{VABAValid}_i := \{\}$ // to be used as the index VABA input
2: upon $|\mathsf{Valid}_i| = n - t$
3:     Let $I_i := \mathsf{Valid}_i$
4:     broadcast $I_i$ using an RBC instance $\mathsf{RBC}_i$
5: upon $I_j \subseteq \mathsf{Valid}_i$ becomes true where $I_j$ is the output of $\mathsf{RBC}_j$
6:     if $|I_j| \geq n - t$ then
7:         $\mathsf{VABAValid}_i := \mathsf{VABAValid}_i \cup \{j\}$
8: start Index VABA with input $\mathsf{VABAValid}_i$
9: upon Index VABA outputs $i^*$
10:    wait until $\mathsf{RBC}_{i^*}$ outputs $I_{i^*}$
11:    output $I_{i^*}$ and terminate
```

Algorithm 8 ACS protocol for party i

```
Input: Message $m_i$
1: broadcast $m_i$ using an RBC instance $\mathsf{RBC}_i$
2: Let $\mathsf{Valid}_i := \{\}$ // to be used as the index ACS input
3: upon $j$-th RBC finishes and outputs $m_j$
4:     Let $\mathsf{Valid}_i := \mathsf{Valid}_i \cup \{j\}$
5: start Index ACS with input $\mathsf{Valid}_i$
6: upon Index ACS outputs $X$
7:     wait until $\mathsf{RBC}_j$ outputs $m_j$ for each $j \in X$
8:     output $\{m_j\}_{j \in X}$ and terminate
```

At the start of the index ACS protocol, each party i, upon validating $n - t$ parties, proposes this set I_i of $n - t$ parties using an RBC instance RBC_i.

Party i also maintains a set $\mathsf{VABAValid}_i$, a set of parties whose RBC outputs become validated. More precisely, party i adds a party j to $\mathsf{VABAValid}_i$, only when its input Valid_i becomes a superset of the proposal I_j broadcasted by party j and $|I_j| \geq n - t$.

Parties then run an instance of the index VABA protocol where party i uses $\mathsf{VABAValid}_i$ as its input. Let i^* be the index VABA output. Parties then wait until the RBC instance RBC_{i^*} outputs I_{i^*}, and then outputs I_{i^*} as the index ACS output.

It is easy to prove this gives an index ACS protocol following a similar and standard proof as [2].

6.2 ACS

We summarize our ACS protocol in Algorithm 8, and describe it next. Each party i start the ACS protocol by broadcasting its input message m_i using an RBC instance RBC_i. Party i also maintains a set Valid_i, a set of parties whose RBC instances has finished at party i. More precisely, party i adds a party j to Valid_i, only upon outputting m_j from RBC_j. Parties then run an instance of the index ACS protocol where party i uses Valid_i as its input. Let X be the index ACS output. Parties then wait until the RBC instance RBC_j outputs m_j for all $j \in X$. Then, each party outputs $\{m_j\}_{j \in X}$ as the ACS output.

It is easy to prove this gives an ACS protocol assuming secure index ACS and RBC protocols.

6.3 Application to Hash-based Common Coin

Our index ACS protocol can be used with a hash-based verifiable secret sharing scheme to design a hash-based asynchronous common coin or randomness beacon. The protocol has two phases: *preparation* and *reveal*.

During the preparation phase, each node acts as a dealer to share a uniformly random secret using a hash-based secret sharing protocol. Nodes then agree on a subset of at least $t + 1$ dealers who correctly shared their secrets, using an index ACS protocol. During the reveal phase, nodes reconstruct each of the $t + 1$ shared secrets. The coin/beacon output is then the sum of all reconstructed secrets.

Using the recent hash-based asynchronous complete secret sharing scheme from [44] along with the standard randomness extraction technique using hyper-invertible matrices [7, 18], we can generate $\Theta(n^2)$ common coins with amortized $O(\kappa n)$ communication in the failure-free case.

6.4 Application to Asynchronous DKG

We can also use our index ACS to improve the asynchronous DKG protocol of Das et al. [20] by simply replacing their index ACS protocol with our index ACS protocol. This improves both the asymptotic round complexity and concrete runtime. We want to note that although our index ACS protocol only relies on hash functions, the overall asynchronous DKG protocol of Das et al. [20] still requires public key cryptography.

7 Implementation and Evaluation

We implement our ACS protocol in `Python 3.7.13` on top of the open-source asynchronous DKG codebase of [18, 20]. Our implementation uses the same libraries as [18, 20], such as `asyncio` for concurrency, RBC implementation from [19] and finite field operations from [1, 29]. Our implementation is single-threaded at each party and is publicly available at https://github.com/shengqi647/acs.

7.1 Optimizations

In previous sections, we prioritized the modularity of the protocol and forgo several optimizations. For example, since RA and RBC have many similarities, RA can often piggyback on RBC. Similarly, we can also merge different RBC instances from different sub-protocols. Next, we will briefly describe the optimizations we implement and refer the reader to our open-source codebase for more details.

First, we merge the RBC and RA instances in our ASKS protocol (Algorithm 2) as follows. In the sharing phase, the dealer sends $\langle \mathsf{SHARE}, p(i) \rangle$ and $\boldsymbol{h}$ together to party i; each party i participates in the RBC of the message $\boldsymbol{h}$ only if $\langle \mathsf{SHARE}, p(i) \rangle$ also satisfies the condition $\boldsymbol{h}[i] = \mathsf{H}(i, p(i))$. With this optimization, the RA in ASKS comes for free. Second, we similarly merge the RA instances in the ICG protocol (line 1 of Algorithm 4) into the RBC instances of the index VABA protocol (line 8 of Algorithm 6). Third, we omit the RBC instances on line 4 of Algorithm 7. Instead, in our implementation, party i uses $\mathsf{RBC}_{\mathsf{pre},i}$ (line 8 of Algorithm 5) in view $v = 0$ to propose I_i. More precisely, in view $v = 0$ of the index VABA protocol, each party proposes $(\mathsf{pre}_i, P_i, I_i, \mathsf{justify}_i)$ using $\mathsf{RBC}_{\mathsf{pre},i}$. With this optimization, we get our index ACS protocol from our index VABA protocol without using any additional rounds of communication.

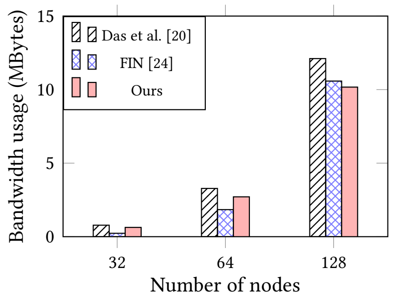

Figure 1: Bandwidth usage

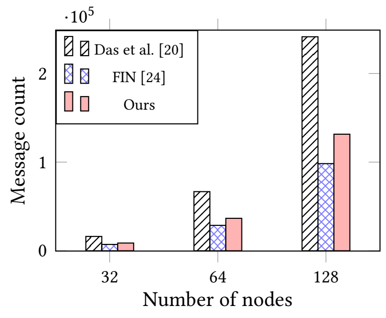

Figure 2: Message count

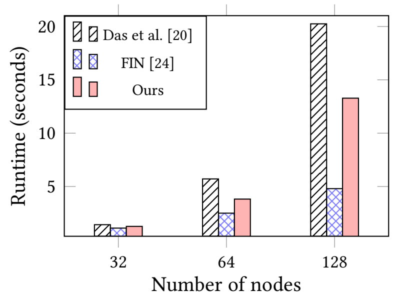

Figure 3: Runtime

7.2 Evaluation Setup

With our evaluation, we seek to illustrate that our *index ACS* is practical and achieves comparable or better performance than prior works while relying only on hash functions. Thus, we only evaluate the costs of index ACS for both our protocol and baselines. To facilitate the evaluation of index ACS evaluation, we use 1-bit messages as inputs to Algorithm 8. Note that, unlike standard ACS, whose performance crucially depends on the input size, the performance of index ACS is independent of the input size.

Experimental setup. We evaluate our implementation with a varying number of parties: 32, 64, and 128. We run all parties on Amazon Web Services (AWS) *m5.xlarge* virtual machines (VMs) with one party per VM. Each VM has 4 vCPUs, 16GB RAM, and runs Ubuntu 20.04. We place parties evenly across eight different AWS regions: Canada, Ireland, N. California, N. Virginia, Oregon, Ohio, Singapore, and Tokyo. We create an overlay network in which all parties are pair-wise connected, i.e., they form a complete graph. We run ten repetitions of our experiments for each choice of parameters, where in each experiment, we let the protocol run until all nodes locally output from the index ACS protocol.

Baselines. We distill the index ACS protocol from the Das et al. asynchronous DKG [20] as the main baseline. This is the only known implementation of ACS that does not rely on a private setup or idealized common coins. We also compare with FIN [24], a recent ACS protocol that is signature-free but assumes idealized common coins. We implement and evaluate an index ACS protocol based on FIN (called MVBA in the paper). The FIN implementation uses the discrete logarithm based threshold pseudorandom function from [14] to implement the common coin.

Evaluation metrics. We evaluate our protocol and the baselines as per the following three metrics: *bandwidth usage*, *message count*, and *runtime*. The bandwidth usage measures the amount of data in Bytes a party sends during the entire index ACS protocol. Similarly, the message count is the number of messages a party sends during the entire index ACS protocol. Finally, the runtime is measured from the start of the ACS protocol to the time a party outputs from the ACS protocol. We measure the bandwidth usage and runtime to illustrate the overall performance of our index ACS protocol. We also measure the message count, since [31] reports message count as a bottleneck in the running time of ACS protocol, and that aligns with what we observe in our experiments.

7.3 Evaluation Results

Bandwidth usage. We report the average bandwidth usage in Figure 1. Asymptotically, the bandwidth usage in all three protocols is $O(\kappa n^2)$. Thus, as expected, we see a quadratic increase in bandwidth usage with an increasing number of parties. Concretely, the bandwidth usage of [20] is slightly higher than ours due to its reliance on the Feldman VSS scheme, built using discrete logarithm assumption. The bandwidth usage of FIN [24] is better than ours for 32 and 64 parties and slightly worse than ours for 128 parties. We believe the better bandwidth usage of FIN is due to the fact that it does not require any VSS scheme and instead relies on an external common coin oracle for shared randomness.

Message count. We report the average message count in Figure 2. As expected, the message counts of all three protocols increase quadratically with the number of parties. The message count of [20] is approximately 1.8× higher than ours. This is because the index ACS protocol in [20] requires each party to participate in n parallel asynchronous binary agreement (ABA) instances. The message count of FIN protocol is approximately 75% of ours [24]. FIN has a lower message count as it does not require parties to participate in VSS schemes and it uses only one ABA instance in each iteration of the protocol.

Runtime. We report the average runtime in Figure 3. The runtime of our ACS protocol is approximately 55%-65% of that in [20]. This was expected as our protocol has slightly better bandwidth usage, and our message count is approximately only 45% than that of [20]. This also points to message count being the main bottleneck for runtime.

The runtime of FIN is lower than ours. For example, with 128 parties, the runtime of FIN is only about 36% of our runtime. This was unexpected as the bandwidth usage of FIN is comparable to ours, and its message count is 75% of ours. Part of our extra runtime is because we require one additional RBC and ASKS phase for coin generation. The runtime advantage of FINs also comes from its implementation in `Golang` and its better engineering. For example, with 128 parties, n parallel 1-bit RBC instances in the FIN implementation take approximately one second, whereas it takes approximately 2.7 seconds in our implementation.

8 Related Work

ACS and asynchronous consensus using idealized common coins. Asynchronous common subset (ACS) is introduced by Ben-Or, Canetti, and Goldreich under a different name called *agreement on a core set* [9]. Ben-Or, Kelmer, and Rabin (BKR) [10] later presented a practical construction. Both works study the ACS primitive, assuming an idealized common coin oracle. In recent years, many practical ACS protocols have adopted the BKR paradigm and instantiate the common coin oracle with a distributed pseudorandom function (also known as unique threshold signatures) [23, 37, 46]. The ACS construction in the BKR paradigm uses n parallel asynchronous binary agreement instances, and incurs $O(n^3)$ message and communication complexity and takes expected $O(\log n)$ rounds to terminate. Very recently, the FIN protocol [24] reduces the round complexity to $O(1)$.

Besides ACS, a line of work studies signature-free asynchronous Byzantine agreement assuming idealized common coins, beginning with the seminal work by Rabin [41] and followed by [17, 39, 40].

Asynchronous consensus using weak common coin. An alternate approach to asynchronous consensus is to first build a weak common coin, meaning that honest parties may disagree on their values with non-negligible probability, and then use these weak coins to build an agreement protocol. This is also the approach we adopt in this paper. This approach first appears in the work of Canetti and Rabin [16], where they use this approach to design a setup-free asynchronous binary agreement (ABA) protocol. The ABA protocol of [16] is information-theoretically secure and has a very high $O(n^7)$ communication cost.

As we discuss in §2.4, recent works [2, 4] have adopted this approach to build ACS with improved efficiency compared to [16]. Abraham et al., in [4] adopts this approach and designs an ACS protocol with $O(\kappa n^3)$ communication costs, where it relies on public key cryptography, specifically a pairing-based threshold verifiable random function, for efficiency. Very recently, Abraham et al. [2] adopted this approach to build statistically secure ACS with $O(n^5)$ communication costs for $t < n/4$. For $t < n/3$, they rely on other strong primitives (statistically secure AVSS), and the cost is not fully specified, but it is safe to assume it will be worse than $O(n^5)$.

Asynchronous consensus with cryptography and trusted setup. Cachin, Kursawe, Petzold, and Shoup (CKPS) [13] present an alternate approach to designing efficient asynchronous consensus with cryptography. A recent work Dumbo-MVBA [36] improves upon CKPS and leads to an ACS protocol with $O(\kappa n^2)$ communication cost. This paradigm crucially relies on threshold signatures [43], threshold pseudorandom functions [14], and constant size polynomial commitments [34] both for efficiency and for generating strong common coins needed to bypass the FLP impossibility. It is worth mentioning that these works also assume idealized common coins.

Asynchronous distributed key generation. A distributed key generation (DKG) protocol lets a set of parties setup secret keys for threshold cryptography and agree on a public key. Thus, DKG is a special type of agreement protocol. Asynchronous DKG that does not rely on a trusted setup can also be viewed as a setup-free ACS protocol [4, 20, 35]. The state-of-the-art asynchronous DKG protocol incurs $O(\kappa n^3)$ communication cost and $O(1)$ round complexity [4]. All existing asynchronous DKG protocols and the ACS protocols distilled from them rely on public-key cryptography.

Asynchronous distributed randomness beacon. Very recently, Bandarupalli et al. [6] proposed a hash-based randomness beacon protocol called HashRand. The protocol can generate shared coins with amortized $O(\kappa n^2 \log n)$ communication cost per coin output after a long bootstrapping period that takes $\Omega(\kappa \log n)$ rounds and $\Omega(\kappa^2 n^3 \log n)$ communication.

The bootstrapping phase of HashRand uses the asynchronous common coin protocol from [26]. The common-coin protocol of [26] uses n parallel asynchronous VSS instances and n parallel deterministic approximate agreement instances to let parties derive shared randomness. It is possible to combine the common coin protocol of HashRand with the index ACS protocol from FIN [24] to design a hash-based index ACS protocol without relying on any trusted setup or public key cryptography. However, the resulting index ACS would require $\Omega(\kappa^2 n^3 \log n)$ communication and $\Omega(\kappa \log n)$ rounds.

Asynchronous verifiable secret sharing (AVSS). Existing AVSS constructions that do not rely on trusted setup and ensure completeness assume public key infrastructure (PKI) [30, 33, 45]. Our ASKS construction is closely related to the "secure message distribution" primitive by Shoup and Smart [44] and the weak AVSS scheme of [21, 22]. Our ASKS can be viewed as a secret sharing version of the secure message distribution primitive in [44]. In particular, if at least one honest party completes the sharing, a fixed secret can eventually be reconstructed. A notion of "asynchronous weak VSS" similar to our ASKS primitive has been introduced in [21, 22]. However, the security properties and corresponding analysis presented in [21, 22] are very informal.

9 Discussion and Conclusion

In this paper, we have presented trusted-setup-free and public-key-cryptography-free validated asynchronous byzantine agreement (VABA) and asynchronous common subset (ACS) protocols with $O(n^3)$ expected communication and $O(1)$ expected rounds, improving over previous works, which require either $O(\log n)$ rounds or $O(n^5)$ communication. Along the way, we also introduce new primitives of asynchronous secret key sharing and cover gather, which may be of independent interest. We implemented a prototype and evaluated it with up to 128 geographically distributed parties. Our experiments demonstrate better performance over the current best practical schemes.

Removing the random oracle assumption. Recall that our index ACS protocol relies on the random oracle (RO) assumption in two places: for the security of our ASKS primitive and in deriving ranks of parties.

We can remove the need for RO from our ASKS design and instead rely on the *linear hiding assumption* as described in [44]. We can remove the need for RO in rank derivation by using a pseudorandom function (PRF). In particular, the rank of party j will be the sum of PRF evaluations at j with the secrets $s_k \in P_j$ as the PRF keys. More precisely, let $F : \mathbb{F} \times \{0,1\}^* \to \mathbb{F}$ be a PRF. Then, the rank of party j will be:

$$\mathsf{rank}_j = \sum_{j \in P_j} F(s_k, j) \tag{6}$$

Intuitively, this ensures a party's rank remains random before the secrets shared by parties in P_j are revealed.

Acknowledgements

This work is funded in part by a Chainlink Labs Ph.D. fellowship, the National Science Foundation award #2240976, and the Beijing Natural Science Foundation award #M23015.

References

[1] [n. d.]. curve25519dalek: A pure-rust implementation of group operations on ristretto and curve25519, 2021. https://github.com/dalek-cryptography/curve25519-dalek.

[2] Ittai Abraham, Gilad Asharov, Arpita Patra, and Gilad Stern. 2023. Perfectly Secure Asynchronous Agreement on a Core Set in Constant Expected Time. *Cryptology ePrint Archive* (2023).

[3] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, and Gilad Stern. 2023. Bingo: Adaptivity and asynchrony in verifiable secret sharing and distributed key generation. In *Annual International Cryptology Conference*. Springer.

[4] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. 2021. Reaching consensus for asynchronous distributed key generation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. 363–373.

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

[6] Akhil Bandarupalli, Adithya Bhat, Saurabh Bagchi, Aniket Kate, and Michael Reiter. 2024. HashRand: Efficient Asynchronous Random Beacon without Threshold Cryptographic Setup. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security.*

[7] Zuzana Beerliová-Trubíniová and Martin Hirt. 2008. Perfectly-secure MPC with linear communication complexity. In *Theory of Cryptography Conference*. Springer.

[8] Michael Ben-Or. 1983. Another advantage of free choice (extended abstract) completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*. 27–30.

[9] Michael Ben-Or, Ran Canetti, and Oded Goldreich. 1993. Asynchronous secure computation. In *STOC*. ACM, 52–61.

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

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

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

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

[14] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2000. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. 123–132.

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

[16] 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.

[17] Tyler Crain. 2020. Two More Algorithms for Randomized Signature-Free Asynchronous Binary Byzantine Consensus with $t < n/3$ and $O(n^2)$ Messages and $O(1)$ Round Expected Termination. *arXiv preprint arXiv:2002.08765* (2020).

[18] Sourav Das, Zhuolun Xiang, Lefteris Kokoris-Kogias, and Ling Ren. 2023. Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling. In *32nd USENIX Security Symposium (USENIX Security 23)*.

[19] Sourav Das, Zhuolun Xiang, and Ling Ren. 2021. Asynchronous Data Dissemination and its Applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*.

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

[21] Shlomi Dolev, Bingyong Guo, Jianyu Niu, and Ziyu Wang. 2023. SodsBC: a post-quantum by design asynchronous blockchain framework. *IEEE Transactions on Dependable and Secure Computing* (2023).

[22] Shlomi Dolev and Ziyu Wang. 2020. Sodsbc: Stream of distributed secrets for quantum-safe blockchain. In *2020 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 247–256.

[23] 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.

[24] Sisi Duan, Xin Wang, and Haibin Zhang. 2023. Fin: Practical signature-free asynchronous common subset in constant time. In *ACM CCS*.

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

[26] Luciano Freitas, Petr Kuznetsov, and Andrei Tonkikh. 2022. Distributed randomness from approximate agreement. *arXiv preprint arXiv:2205.11878* (2022).

[27] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1187–1201.

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

[29] Jack Grigg and Sean Bowe. [n. d.]. zkcrypto/pairing. https://github.com/zkcrypto/pairing.

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

[31] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Speeding dumbo: Pushing asynchronous bft closer to practice. In *NDSS*.

[32] 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*.

[33] Aniket Kate, Easwar Vivek Mangipudi, Pratyay Mukherjee, Hamza Saleem, and Sri Aravinda Krishnan Thyagarajan. 2024. Non-interactive VSS using Class Groups and Application to DKG. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security.*

[34] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. 2010. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology-ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings 16*. Springer, 177–194.

[35] 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.

[36] 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.*

[37] 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.

[38] Henrique Moniz, Nuno Ferreria Neves, Miguel Correia, and Paulo Verissimo. 2008. RITAS: Services for randomized intrusion tolerance. *IEEE transactions on dependable and secure computing* 8, 1 (2008), 122–136.

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

[40] Achour Mostéfaoui and Michel Raynal. 2017. Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with t< n/3, O(n^2) messages, and constant time. *Acta Informatica* 54, 5 (2017), 501–520.

[41] Michael O Rabin. 1983. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. IEEE, 403–409.

[42] Tal Rabin and Michael Ben-Or. 1989. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*. 73–85.

[43] Victor Shoup. 2000. Practical threshold signatures. In *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19*. Springer, 207–220.

[44] Victor Shoup and Nigel P. Smart. 2024. Lightweight Asynchronous Verifiable Secret Sharing with Optimal Resilience. *Journal of Cryptology* (2024).

[45] Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. 2022. hbACSS: How to Robustly Share Many Secrets. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium.*

[46] Haibin Zhang and Sisi Duan. 2022. Pace: Fully parallelizable bft from reproposable byzantine agreement. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3151–3164.

[47] Haibin Zhang, Sisi Duan, Chao Liu, Boxin Zhao, Xuanji Meng, Shengli Liu, Yong Yu, Fangguo Zhang, and Liehuang Zhu. 2023. Practical asynchronous distributed key generation: improved efficiency, weaker assumption, and standard model. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 568–581.