Asynchronous Algorand: Reaching Agreement with Near Linear Communication and Constant Expected Time

ITTAI ABRAHAM, Intel Labs, Israel
ELI CHOUATT, Hebrew University, Israel
IVAN DAMGÅRD, Aarhus University, Denmark
YOSSI GILAD, Hebrew University, Israel
GILAD STERN, Tel Aviv University, Israel
SOPHIA YAKOUBOV, Aarhus University, Denmark

The celebrated Algorand protocol solves validated byzantine agreement in a scalable manner in the synchronous setting. In this paper, we study the feasibility of similar solutions in the asynchronous setting. Our main result is an asynchronous validated byzantine agreement protocol that we call *Asynchronous Algorand*. As with Algorand, it terminates in an expected constant number of rounds, and honest parties send an expected $O(n \text{ polylog } n)$ bits, where n is the number of parties. The protocol is resilient to a fully-asynchronous weakly-adaptive adversary that can corrupt a near-optimal number of parties ($< (1/3 - \epsilon)n$) and requires just a VRF setup and secure erasures.

A key innovation in Asynchronous Algorand is a rather simple but surprisingly effective method to do *committee-based role assignment* for asynchronous verifiable secret sharing in the YOSO (You Only Speak Once) model. This method achieves near-optimal resilience and near-linear communication complexity while relying solely on a verifiable random function (VRF) setup and secure erasures.

1 INTRODUCTION

In this paper, we study the complexity of asynchronous byzantine agreement and ask the following question:

> *Under what conditions can asynchronous byzantine agreement obtain near-optimal resilience, near-linear communication, and constant expected time?*

In the synchronous setting, the breakthrough result of Algorand [16, 28] obtains byzantine agreement with near-optimal resilience, near-linear communication, and constant expected time under the following conditions:

(1) Assuming a Verifiable Random Function (VRF) setup: this is the hallmark of Algorand, the ability to sub-sample committees in a fair manner. This requires assuming more than just a PKI setup because allowing the adversary to choose its public and private keys may allow it to compromise security.
(2) Weak adaptive adversary: the adversary can ask to corrupt a party at any time, and the party becomes corrupt immediately after it completes sending all its outgoing messages or after it listens for any incoming messages. The adversary cannot claw-back already sent messages.
(3) Secure Erasure: Honest parties can securely erase data, such that any corruption after the erasure cannot recover the erased data.

The main result in this paper is a new protocol, *Asynchronous Algorand*, that solves byzantine agreement with near-optimal resilience, near-linear communication, and constant expected time. Asynchronous Algorand does this under the same three conditions above, even when extended to the asynchronous communication model.

Authors' addresses: Ittai Abraham, Intel Labs, Tel Aviv, Israel; Eli Chouatt, Hebrew University, Jerusalem, Israel; Ivan Damgård, Aarhus University, Aarhus, Denmark; Yossi Gilad, Hebrew University, Jerusalem, Israel; Gilad Stern, Tel Aviv University, Tel Aviv, Israel; Sophia Yakoubov, Aarhus University, Aarhus, Denmark.

Theorem 1.1. *Given $0 < \epsilon < 1$, a computational security parameter λ, and a statistical security parameter κ, Asynchronous Algorand solves validated byzantine agreement in an asynchronous setting. Other than an error event that happens with probability $< 2^{-\lambda} + 2^{-\kappa}$, the protocol terminates in an expected constant number of rounds, and honest parties send an expected $O(n \text{ poly}(\log n, \lambda, \kappa, 1/\epsilon))$ bits. The protocol is resilient to a fully-asynchronous weak-adaptive adversary that can corrupt $f < (1/3 + \epsilon)n$ parties, assuming idealized PKE, Signature, VRF and PVSS schemes, and assuming secure erasures.*

On fully-asynchronous weak-adaptive adversaries: in this model, the adversary can *adaptively* delay the delivery of any message by any finite amount of time and can *adaptively* ask to corrupt any party at any time. The adversary can make these decisions based on all the current information they have. Once the adversary asks to corrupt a party, it becomes under the control of the adversary once it completes sending all the messages in its outgoing message queue. Put differently, corruption happens when the party accesses its incoming message queue.

On the requirement that the adversary is weak-adaptive: Obtaining near linear and constant expected time is trivial against a static adversary and impossible against a strongly adaptive adversary. The former is folklore, and the latter follows from the lower bound of [2] that extends the classic deterministic lower bound of Dolev and Reischuck [24] (also see [18]). Moreover, we are not aware of any near linear and constant expected time protocol against a non-weak adaptive adversary that cannot claw back already sent messages.

On the cryptographic assumptions: Our protocol uses a standard setup for verifiable random functions, a public-key encryption scheme, a signature scheme, and a publicly verifiable secret sharing scheme, which we abstract as idealized objects with no errors. For a discussion of these objects and of the error introduced by instantiations of the schemes see Section 3.3. For adaptive security, we use a key and message non-committing encryption (KM-NCE, [30]).

On erasures: To protect against a weak-adaptive adversary corrupting a party and using its secrets we rely on secure erasure. As detailed in Section 3.2, each time a party wants to send a message, it first signs it along with a new public key, then erases its current signature key, and only then sends the message to all parties.

Our contributions.

(1) This is the first near-linear, constant expected round asynchronous byzantine agreement that obtains near-optimal corruption threshold while assuming just a VRF setup. All previous near-optimal corruption threshold works required very complex MPC setup protocols that are quadratic (or more) in cost.
(2) A key innovation that enables our near-optimal corruption threshold with just a VRF setup is a new way to assign secrets to anonymous members. Roughly speaking, instead of each nominator choosing **one recipient** to hold its secret (and with constant probability, this recipient may be a malicious party - hence this approach has a non-optimal corruption threshold), each nominator chooses a random **committee of recipients** (using unverifiable randomness) and secret shares the secret among them. This reduces the error to statistically small and thus enables a near-optimal corruption threshold. This is critical for obtaining a near-optimal corruption threshold in our asynchronous verifiable secret-sharing protocol (see Section 6). Previous work either obtained far from optimal corruption thresholds or required very strong MPC-based setup assumptions that cannot be implemented in near-linear time.
(3) Conceptually, we show how to transform the $O(n^3)$-cost asynchronous byzantine agreement framework of [3] that is resilient to a strong adaptive adversary (that can even claw-back),

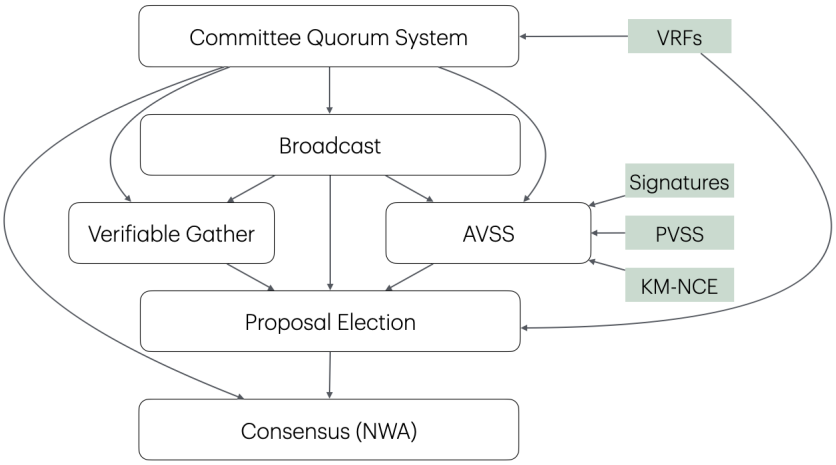

Fig. 1. A map of our constructions.

to a $O(n \text{ polylog } n)$-cost asynchronous byzantine agreement that is resilient to a weak-adaptive adversary. This transformation requires multiple new ideas and careful, non-trivial adaptations and modification of the framework to the committee-based setting. We believe the insights from this transformation will help obtain similar transformations for asynchronous MPC protocols.

In addition, we believe we are the first to formally define the weak-adaptive adversary in asynchrony, which is a very natural analog of the weak-adaptive adversary in synchrony. This weak-adaptive adversary in synchrony is the core adversarial definition of all Algorand and YOSO papers.

Overview of techniques. Figure 1 gives an overview of the building blocks, the cryptographic primitives they require (outlined in Section 3), and their relationships.

The first step in our work is to define a quorum system based on VRF sampling (see Section 4). We then show how each committee member can reliably broadcast a message (see Section 5). Using this committee broadcast primitive, we head to the main technical challenge, which is implementing an asynchronous verifiable secret sharing (AVSS) protocol (see Section 6).

A core challenge in verifiable secret sharing protocols against a weakly adaptive adversary is hiding the members holding the secret shares. This is often obtained via a receiver-anonymous communication channel abstraction [26]. Roughly speaking, we implement a *virtual* asynchronous receiver-anonymous communication channel by having each member of a nominating committee: (1) broadcast a public key, and (2) secret share the corresponding private key to an anonymous committee sampled via unverified randomness by the nominator, who jointly comprise the virtual anonymous party. If the nominator was corrupt, we get no guarantees about the anonymous committee she chose (meaning that we likely get a corrupt virtual party), but if the nominator was honest, she will almost certainly have chosen a mostly honest anonymous committee to hold shares of the secret key, which translates to an honest virtual party. We call this part the setup phase of the AVSS protocol (see Algorithm 5).

Once AVSS setup completes, in the AVSS share protocol, each party can wait to hear a quorum of public keys and use them to send a publicly verifiable secret sharing (PVSS) transcript (see Section 3) that uses these pubic keys (see Algorithm 6). Finally, during the AVSS reconstruction

phase, the parties that hold the shares of the virtual anonymous parties' secret keys send them, enabling the reconstruction of all of the PVSS's (see Algorithm 7).

Once we have AVSS, we proceed in a manner that is conceptually similar to [3] in that we use a gather protocol (see Section 7) to collect AVSS shares that completed. Roughly speaking, the gather protocol guarantees that there is a large common core that is a subset of the output of all parties. We require the gather to be binding (so the adversary must fix the core before knowing the hidden ranks) and verifiable (so that in the good event, malicious parties cannot verifiably choose a different prosper). We note that [23, 31] also implement a gather protocol in this setting but obtain a weaker binding gather property that is not sufficient for our needs of using it to implement a weak coin.

We then show how we can implement a proposal election protocol. Recall that the goal of this protocol is that with constant positive probability, in each view, a good event occurs where all parties choose the same valid input. Conditioned on this good event, the consensus protocol will terminate in a constant number of rounds.

At a high level, the proposal election protocol has three phases: first, parties use the committee AVSS share protocol (and committee broadcast) to propose a value and commit to a secret rank, then parties run a committee gather protocol. Once the gather protocol ends, parties start reconstructing the shared secret ranks and then choose the proposal sent by the party with the highest rank in their gather set (see Section 8).

Finally, we provide an asynchronous Byzantine agreement protocol that uses the proposal election protocol above. We use a new variation of No-Waitin'-Hotstuff (NWH) that we call No-Waitin'-Algorand (NWA) that is suited to the speak-once setting: each committee just listens to the previous one (see Section 9).

2 RELATED WORK

The related work can be divided into work in the synchronous model and the asynchronous model.

2.1 Synchronous Model

In the synchronous model, the Algorand protocol of Chen and Micali [16] introduces the use of VRFs for efficient consensus in the presence of a weak-adaptive adversary. Their protocol is designed so that only a few parties need to speak at a time. The adversary is unable to predict who will be speaking (since this is determined by the evaluation of a VRF with the potential speaker's secret key), and so is unable to halt the system by corrupting the speaker ahead of time.

Since no one can predict who a speaker will be, it becomes hard for such a system to maintain a secret state: A speaker must erase her secret state before sending her message since once she speaks, the adversary will learn her identity and might corrupt her and learn her secrets. For the secret state to be maintained, our speaker must send it to the next set of speakers (in some form). However, in order to send a private message, one typically needs to know the recipient.

To solve this problem, Benhamouda et al. [7] extend the work of Chen and Micali by introducing *nomination*: unpredictable speakers can *nominate* other speakers by assigning them an anonymous public key and sending them the corresponding secret key. The anonymous public key can now be used to communicate private messages without knowing the recipient. Since the work of Benhamouda et al., the storage of – and computation on – secrets in such a speak-once system has been an active area of research. YOSO (You Only Speak Once) protocols can be split into two largely orthogonal parts: Building so-called receiver-anonymous communication channels (RACCs), and building storage and computation on top of these RACCs.

Building RACCs There have been three approaches to building RACCs.

(1) The first is via the nomination approach of Benhamouda et al. The upside of this approach is that it assumes only a VRF setup. The downside of this approach is that it requires significantly more than half of the overall participants to be honest.
(2) The second is via running a PIR protocol in MPC [27]. The upside of this approach is that it supports a near-optimal corruption threshold, where just over half of the overall participants are honest. The downside is that a small subset of participants has to do work linear in the number of participants, and that an initial trusted setup, in the form of a first few "seed" RACCs, must be assumed.
(3) The last approach is via "encryption to the future" [13]. The upside of this approach is that it supports a near-optimal corruption threshold; the downside is that it either uses very expensive tools (such as witness encryption), or results in ciphertexts of size linear in the number of *possible* recipients.

Computing on top of RACCs Benhamouda et al. describe how to maintain a secret state: *committees* of nominated parties receive shares of a secret, and can re-share those shares to subsequent committees. Gentry et al. [26] build on top of this idea, showing how such committees can run a secure mutliparty computation (MPC). In concurrent and independent work, Choudhuri et al. [17] introduce the Fluid MPC model, the protocols from which naturally follow YOSO design. Subsequent work improves on the initial YOSO MPC protocols, in terms of number of rounds [32], setup [10, 32], and other efficiency metrics.

2.2 Asynchronous Model

In 2020, Cohen et al. [20, 21] obtained the first near-linear communication and constant expected time validated asynchornous byzantine agreement. However, they assume a much less powerful adaptive adversary called a delayed-adaptive adversary. Under this adversary, "honest party messages must be scheduled by the adversary regardless of their content, namely their VRF random values." This model limits the adversary to use asynchrony in essentially a non-adaptive manner. So proposers can simply attach a random VRF value, interpreted as their rank, and with constant probability, most honest parties see the same honest proposal as having the highest VRF rank.

In comparison, in our model, the adversary is fully adaptive in its network delays. For example, the simple approach above fails because the adversary can adaptively delay a party's message once it learns that it has a winning VRF value. Moreover, the protocol of [20, 21] requires at most $\approx n/4.5$ malicious parties, while our protocol can handle near optimal $(1/3 - \epsilon)n$ malicious parties.

Also in 2020, Blum et al. [8] obtained near-linear communication and constant expected time validated asynchronous byzantine agreement. Moreover, in this setting, they obtain the first near-optimal resilience of up to $(1/3 - \epsilon)n$ malicious parties. However, they require a very powerful setup assumption that can only be instantiated via a non-scalable MPC protocol. Most critically, they assume a blackbox use of a strong common coin (note that [25] prove that strong coins must have an infinite execution). All known implementations of strong coin need some sort of byzantine agreement. Implementing these common coins requires a powerful MPC that seems to require more than quadratic message complexity. In comparison, our work requires just a plain VRF setup, and uses just a public key encryption scheme and a PVSS scheme, and obtains near-liner communication without any oracle assumptions.

Kamp and Nielsen [31] obtain near-optimal resilience, near-linear communication and constant expected time, assuming a powerful setup assumption, including a threshold signature scheme with unique signatures, and FHE for some of its schemes. In addition, like [8], their work requires blackbox use of a strong common coin. Again, in comparison, our work requires just a plain VRF setup, and is self-contained (no other setup or oracle assumptions).

Damgård et al. [23] obtain near-optimal resilience and get full asynchronous MPC with a near-linear cost per gate. They use the consensus protocol of [31], so they need a strong common coin setup (they then use these initial coins to bootstrap more coins). Again, this strong common coin setup assumption seems to require an expensive MPC setup protocol. In comparison, our work focuses just on consensus and obtains it using a plain VRF setup, and in particular, does not depend on any MPC setup assumptions.

On assuming secure erasures: Our protocol requires honest parties to be able to *erase* data in such a way that if the party is later corrupted, the erased data cannot be recovered. In synchrony, the binary agreement protocol in [2] removes the need for assuming secure erasures. Removing the need for erasures for validated (non-binary) agreement in synchrony and, in general, for asynchrony remains an open question.

On near-optimal resilience: It is conjectured that the optimal resilience $n = 3f+1$ for asynchronous Byzantine agreement cannot be obtained with a near-linear and constant expected time protocol. For partial progress in the synchronous model, see Rambaud's result [34, Theorem 5].

3 PRELIMINARIES, MODEL DEFINITIONS

3.1 Adversary and network model

The network consists of n parties, with at most f parties being controlled by the adversary. Every pair of parties is connected by a secure and authenticated channel, meaning that only the sender and recipient can read the contents of messages, and parties know from whom they received each message. The network is asynchronous, meaning that every sent message eventually arrives at its destination, but there is no bound on how long messages can be delayed before arriving.

We consider a computationally bounded Byzantine adversary that can cause the parties it controls to arbitrarily deviate from the protocol. The adversary is adaptive, but weak in the sense that it can only corrupt a party at certain times: after it completes a multicast or after it listens to receive a message. Observe that this implies that when a party that is nonfaulty starts a sub-protocol, and the sub-protocol does not listen to new messages before sending a multicast, then it is guaranteed to send its multicast before being corrupted. Moreover, since there is no clawback, these messages will eventually be delivered to all recipients. Note that this is the asynchronous analogue of the synchronous Algorand (and YOSO) weak-adaptive adversary.

We stress that the adversary is fully adaptive in its decision about network delays and when it asks to corrupt someone. The only weakness is that corruption cannot take place until a party starts listening to receive messages. In other words, corruption cannot take place while parties are performing local computations (usually modeled as taking no time) or as they are sending messages.

3.2 Sending Messages with Forward Security

We assume that every message any party sends is signed, using digital signatures (Section 3.3.3). In our setting, we want to be able to prevent the weak-adaptive adversary from observing who speaks, corrupting that party, and — without clawing back the message they sent — sending a new, competing message, and delivering the competing message first.[1] We need *forward security*, meaning that when an adversary corrupts a party, they cannot produce signatures that look like they came from before the time of corruption. In order to obtain forward security, we utilize *key ratcheting*. When a party sends a message, they also do the following:

(1) generate a new signature key pair;

[1]Note that this is not an issue if every channel is assumed to be FIFO (first-in-first-out), where messages sent first are guaranteed to be delivered first.

(2) use the old signing key to sign the new verification key;
(3) erase the old signing key;
(4) include the new verification key and signature with the message they are sending.

This makes the attack described above impossible, because the adversary will no longer have access to the old key, tied to the party's verifiable right to send a given message.

Note that some messages also explicitly include signatures. These are additional signatures independent of the procedure described above, but they are signed with the most up-to-date key for any party. Technically, in order to be able to verify these signatures, parties need to make sure they use the same public verification keys. Seeing as an adversary can send different ratcheted keys to different parties, this could cause an issue with verification. In order to avoid this, parties can send chains of signed public keys, to prove that they originated from the sender. The total round complexity is constant in expectation, and thus these chains are also of constant expected length. More generally, one could use any forward-secure signature scheme [5].

3.3 Cryptographic primitives

This work uses four cryptographic primitives: public key encryption, signature schemes, verifiable random functions, and publicly verifiable secret sharing (PVSS). We consider these primitives as perfectly secure against our adversary. They are used a polynomial number of times (in n) in expectation in total. This means that the error probability introduced by the use of cryptographic primitives can be bounded by $\mathrm{poly}(n) \cdot \mathrm{negl}(\lambda)$, where λ is a cryptographic security parameter. This modeling follows Cachin et al. [11, 12] (also see [4]). All cryptographic primitives are of size $O(\lambda)$ bits, except for PVSS transcripts whose size is described below.

3.3.1 Public Key Setup. The schemes described below require public key pairs $(\mathsf{pk}, \mathsf{sk})$. Traditionally, each scheme is described as having its own KeyGen algorithm that can be run on the security parameter 1^λ to generate such a pair. A public key setup generates a public key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ for each party i. Every party i learns all public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$, and its own secret key sk_i. In the following descriptions, we omit the KeyGen algorithm from every scheme and combine them into a single KeyGen algorithm used to generate keys for all schemes, which is used in a public key setup.

3.3.2 Public Key Encryption. A Public Key Encryption (PKE) scheme comprises two algorithms: Enc and Dec. Using pk_i, parties can encrypt the message m by running $\mathsf{Enc}(\mathsf{pk}_i, m)$ to produce a ciphertext c. Using sk_i, i can decrypt a ciphertext c by running $\mathsf{Dec}(\mathsf{sk}_i, c)$ to produce a message m. A PKE scheme typically has two properties:

- Correctness: For every public key pair $\mathsf{sk}_i, \mathsf{pk}_i$ and message m, $\mathsf{Dec}(\mathsf{sk}_i, \mathsf{Enc}(\mathsf{pk}_i, m)) = m$.
- CCA-security: Without the secret key, a ciphertext reveals no information about the encrypted message, even given a decryption oracle (unless that oracle is called precisely on the ciphertext in question).

We use a stronger flavor of PKE, called KM-NCE (key-message non-committing encryption) [30]. KM-NCE additionally enables generating fake ciphertexts without any public key or message in such a way that those can later be explained as an encryption of any message to any key. Fake ciphertexts are required to look like real ones, which, in addition to implying CCA-security, also implies that real ciphertexts are anonymous (they do not reveal the public key to which they were encrypted). This ability to explain ciphertexts on the fly is perfectly suited for use against an adaptive adversary, since the adversary observing a ciphertext message wouldn't know who is the intended recipient they should corrupt.

Informally, KM-NCE additionally offers anonymity:[2]

- Anonymity under receiver-selective opening (ANON-RSO): If a ciphertext was computed using a key pk_i, only party i can detect this and decrypt the message, and other parties do not learn to whom the message was encrypted.

Canetti et al. [14] give an efficient ElGamal-based KM-NCE scheme in the programmable random oracle model.

3.3.3 Signature Schemes. . A Signature scheme comprises two algorithms: Sign and Sign_verify. Using sk_i, i can sign the message m by running $\mathsf{Sign}(\mathsf{sk}_i, m)$ to produce a signature σ. Using pk_i, parties can check the correctness of a signature σ on a message m by running $\mathsf{Sign\_verify}(\mathsf{pk}_i, m, \sigma)$, outputting 1 if the signature is correct and 0 otherwise. The Signature scheme has two properties:

- Correctness: For every public key pair $\mathsf{sk}_i, \mathsf{pk}_i$ and message m, $\mathsf{Sign\_verify}(\mathsf{pk}_i, m, \mathsf{Sign}(\mathsf{sk}_i, m)) = 1$.
- Unforgeability: The adversary cannot produce a signature σ on a message m such that $\mathsf{Sign\_verify}(\mathsf{pk}_i, m, \sigma) = 1$ if it doesn't know sk_i, unless it previously received exactly such a signature.

3.3.4 Verifiable Random Functions. A Verifiable Random Function scheme (VRF) [33] consists of two deterministic algorithms: VRF_evaluate and VRF_verify. A VRF is used to generate verifiable randomness, allowing parties to generate seemingly random values and prove that they did so correctly. Party i can run $\mathsf{VRF\_evaluate}(\mathsf{sk}_i, val)$ on a values val to generate a pair r, π. Parties can then run $\mathsf{VRF\_verify}(\mathsf{pk}_i, r, \pi)$ to check that r is indeed the correct output of the VRF, outputting 1 if it is and 0 otherwise. The VRF scheme has three properties:

- Correctness: For every honestly generated public key pair $\mathsf{sk}_i, \mathsf{pk}_i$ and value val, $\mathsf{VRF\_verify}(\mathsf{pk}_i, val, r, \pi) = 1$ for $r, \pi = \mathsf{VRF\_evaluate}(\mathsf{sk}_i, val)$.
- Uniqueness: If $r, \pi = \mathsf{VRF\_evaluate}(\mathsf{sk}_i, val)$ for some value val, no adversary can generate an r', π' such that $r \neq r'$ and $\mathsf{VRF\_verify}(\mathsf{pk}_i, r', \pi') = 1$.
- Pseudorandomness: From the point of view of an adversary not holding sk_i, the value r output from $\mathsf{VRF\_verify}(\mathsf{sk}_i, val)$ looks uniform and independent of all other values.

3.3.5 Publicly Verifiable Secret Sharing. A Publicly Verifiable Secret Sharing (PVSS) scheme [15, 35, 36] is a secret sharing scheme that enables everyone — not just shareholders — to verify that some secret has been shared successfully. In our setting, we need the PVSS to be *non-interactive*, allowing a dealer to include all information required for sharing and reconstructing the secret in a single transcript. Each transcript has an associated *threshold* d. We will assume d is a globally known constant and will not include it in the description of the algorithms. A non-interactive PVSS consists of five algorithms: PVSS, PVSS_verify, PVSS_dec, PVSS_share_verify and PVSS_rec. Parties use the described algorithms as follows:

- Every party can generate a transcript trans for sharing a secret s to a set of at least $d + 1$ public keys keys by running $\mathsf{PVSS}(\mathsf{keys}, s)$. The transcript trans is of size $poly(\lambda \cdot |\mathsf{keys}|)$.
- Parties can run $\mathsf{PVSS\_verify}(\mathsf{keys}, \mathsf{trans})$ to verify that a transcript is well-formed with respect to keys, outputting 1 if it is well-formed, and 0 otherwise.
- Using a secret key sk_i such that $\mathsf{pk}_i \in \mathsf{keys}$, party i can get a share of the PVSS by calling $\mathsf{PVSS\_dec}(\mathsf{keys}, \mathsf{trans}, \mathsf{sk}_i)$ and outputting share.

[2]In the work of Huang et al. [30], ANON-RSO and CCA-security are defined in a single security game. We express them independently for clarity of exposition.

- Shares can also be verified with respect to keys and to a specific public key pk_i by running PVSS_share_verify(keys, trans, pk_i, share), which outputs 1 if share is the share associated with pk_i and 0 otherwise.
- Finally, after receiving $d+1$ verifying shares, parties can reconstruct the secret s by calling PVSS_rec(keys, trans, $\{(\mathsf{pk}_i, \mathsf{share}_i)\}$).

A PVSS scheme has the following properties:

- Verification Correctness: If trans = PVSS(keys, s) for some set of at least $d+1$ keys, then PVSS_verify(keys, trans) = 1. In addition, if PVSS_verify(keys, trans) = 1 and share = PVSS_dec(keys, trans, sk_i) for some sk_i such that $\mathsf{pk}_i \in$ keys, then PVSS_share_verify(keys, trans, pk_i, share) = 1.
- Binding: If PVSS_verify(keys, trans) = 1, then there is a unique $s \neq \perp$ that can be output from PVSS_rec(keys, trans, shares) with any possible set of verifying shares.
- Reconstructibility: If PVSS_verify(keys, trans) = 1 and shares = $\{(\mathsf{pk}_i, \mathsf{share}_i)\}$ is a set of at least $d+1$ pairs such that for every $(\mathsf{pk}_i, \mathsf{share}_i) \in$ shares, PVSS_share_verify(keys, trans, pk_i, share_i) = 1 and $\mathsf{pk}_i \in$ keys, then PVSS_rec(keys, trans, shares) outputs $s \neq \perp$.
- Hiding: If trans = PVSS(keys, s) such that keys contains at least $d+1$ public keys pk_i and the adversary knows at most d of the secret keys sk_i such that $\mathsf{pk}_i \in keys$, then the adversary can learn nothing about s from trans.

4 SCALABLE QUORUM SYSTEM

At a high level, in this section, we build a quorum system based on VRF sampling. Just as in Algorand, to create committees of about $O(\log n)$ members, a party is a member of a committee if its associated VRF value is roughly $< O(\frac{\log n}{n})$. The main property we want to obtain is that the number of nonfaulty parties is always more than $(2/3)$ of the total number of parties in the committee, except for some error probability of roughly $2^{-O(\kappa)}$.

Note that since the adversary is adaptive, we must be careful when defining nonfaulty committee members (see how we define H_{tag}). Looking ahead, this essentially allows them to send one message via multicast.

In addition to defining a VRF committee (Definition 4.1), we also define a very simple way to sample a committee, where a single nominator essentially chooses the committee members via (unverified) random sampling (Definition 4.2). Looking ahead, we will use this second sampling as an essential step in the setup phase of the AVSS.

After defining the quorum systems, we prove three properties that will be useful throughout the next sections. These are the natural properties one would expect from a quorum system: that every two quorums have at least one nonfualty in the intersection and that any quorum has a majority of nonfualty parties.

On error events. Our goal is to show that the total error from using the quorum system for at most n^4 different committees is at most $2^{-\kappa}$. Note that the choice of n^4 committees is arbitrary and is just a conservative overestimate of the fact that we use no more than $O(n^2 \text{ polylog } n)$ committees. Once we obtain these bounds, we will assume in all later parts of the paper that whenever calling the quorum systems, no errors take place. The total error of at most $2^{-\kappa}$ is then incorporated in the final theorem (where we also add the computational error). Similarly to how the error stemming from the cryptography is bounded, roughly bounding the total number of times these systems are used by n^4 means that from the union bound, the total probability of any failure in the quorum system is bounded by $2^{-\kappa}$. Note that the error here is both in terms of correctness (safety violation) and in terms of liveness (may not terminate).

4.1 Definition

In our protocols, we require parties to be able to independently check whether they are committee members with a given tag and to prove their membership to each other. This process should guarantee that the committee is not too large and that a large portion of it is nonfaulty. Since this sampling is random, parties can be "unlucky" and sample a committee that does not have these properties. As such, a quorum committee system is parameterized by a statistical security parameter κ, and its properties should hold with probability $1 - 2^{-\kappa}$ or greater, even if we use $O(n^4)$ different tags (in fact our protocol uses much less).

We formalize a committee quorum system as follows:

Definition 4.1. A committee quorum system scheme with parameters h, m consists of two algorithms (get_mem, verify_mem). Parties call the get_mem algorithm with their secret key sk and a tag tag and output a boolean flag b indicating whether they are members and a proof π. Parties can verify the output of the get_mem algorithm by calling the verify_mem algorithm with a public key pk, a boolean flag b, a proof π and a tag tag and output either 1, indicating that b is the correct output from get_mem or 0 otherwise.

For any tag tag, define C_{tag} to be the set of parties that are members:

$$C_{tag} = \{i \mid \mathsf{get\_mem}(\mathsf{sk}_i, \mathrm{tag}) = 1, \pi \; for \; some \; \pi\}$$

and define H_{tag} to be the set of members that were nonfaulty when they checked their membership:

$$H_{tag} = \{i \mid i \in C_{tag} \text{ and } i \text{ is nonfaulty when first calling get\_mem with } tag\}$$

For any $\kappa > 1$, the pair (get_mem, verify_mem) is said to be a committee quorum system with parameters h, m and κ, c if for any tag tag, the following properties hold with probability at least $1 - 2^{-\kappa c \log n}$.

- **Liveness.** $|H_{tag}| \geq h$.
- **Safety.** $|C_{tag}| \leq m$ and $h > (2/3)m$.
- **Unpredictability.** An adversary that has access to all public keys and does not know sk_i learns nothing about $\mathsf{get\_mem}(\mathsf{sk}_i, tag)$ unless it receives it a nonfaulty party.
- **Binding.** If $\mathsf{get\_mem}(\mathsf{sk}_i, tag) = b, \pi$ for some b, π, then the adversary cannot produce a pair b', π' such that $\mathsf{verify\_mem}(\mathsf{pk}_i, b', \pi', tag) = 1$ and $b \neq b'$.

In addition to parties independently checking if they are members in a committee for a given tag, we require parties to be able to sample a committee to send messages to. This sampling process needs to have the same liveness and safety as a committee quorum system.

Note that this committee is sampled via unverified randomness. So looking ahead, we can only prove properties on such a committee if the sampler is nonfaulty at the time is samples.

Formally, we define a committee sampling algorithm as follows:

Definition 4.2. A committee sampling algorithm sample_committee can be called with no input, and output a set $S \subseteq [n]$. Define $C = S$ to be the set of all committee members and define H to be the set of members of C that were nonfaulty when running sample_committee.

For any $\kappa > 1$, sample_committee is said to be a committee sampling algorithm with parameters h, m and κ, c if the following properties hold with probability at least $1 - 2^{-\kappa c \log n}$.

- **Liveness.** $|H| \geq h$.
- **Safety.** $|C| \leq m$ and $h > \lceil (2/3)m \rceil$ (this also means that $(3/2)h > m$).

Setting the parameter c. Since in this paper we will use much fewer than n^4 tags, we can conservatively set $c = 4$ to guarantee a total error of at most $2^{-\kappa}$ by union bound.

Properties. The following properties follow assuming a committee quorum system or a committee sampling algorithm (with the sets C_{tag}, H_{tag} corresponding to the sets C, H) with parameters h, m, conditioned on no error:

- **Robust quorum intersection (RQI).** Any two sets of h members of C_{tag} must have at least one member H_{tag} in their intersection (because $2h > 2\lceil(2/3)m\rceil \geq (4/3)m \geq m + (m - h)$ and there are at most $m - h$ faulty parties).
- **Quorum intersection (QI).** Any set of h members and set of $\lceil h/2\rceil$ members must have at least one member in their intersection (because $h + \lceil h/2\rceil \geq (3/2)h > m$).
- **Honest majority (HM).** Any set of h members of C_{tag} contains more than $\lceil h/2\rceil$ members in H_{tag} (because at most $m - h < \lceil h/2\rceil$ parties are faulty, and thus the remaining $\lceil h/2\rceil$ must be nonfaulty).

4.2 Construction

In the following construction, we assume the VRF outputs a value $r \in [MAX]$ for some known value MAX (typically, we expect $MAX = 2^{O(\lambda)}$). In both the committee quorum system and committee sampling algorithm, parties are simply uniformly and independently included in the committee with probability $p = \frac{120\epsilon^{-2}\kappa \log n}{n}$. The exact probability is chosen in order to set the statistical error probability at κ even for n^4 tags and is derived in the security analysis (see the proof of Theorem 4.3 for the coarse choice of constants - this can probably be tightened).

Algorithm 1 get_mem(sk, tag)

$r, \pi \leftarrow$ VRF_evaluate(sk, tag)
$b \leftarrow 0$
if $r \leq \frac{120\epsilon^{-2}\kappa \log n}{n} \cdot MAX$ **then**
    $b \leftarrow 1$
output $(b, (r, \pi))$

Algorithm 2 verify_mem($\mathsf{pk}_j, b, (r, \pi), tag$)

if $b = 1$ and VRF_verify($\mathsf{pk}_j, tag, r, \pi$) $= 1$ **then**
    if $r \leq \frac{120\epsilon^{-2}\kappa \log n}{n} \cdot MAX$ **then**
        ouptut 1
output 0

Algorithm 3 sample_committee()

$S \leftarrow \emptyset$
for all $j \in [n]$ **do**
    flip a coin b_j with probability $\frac{120\epsilon^{-2}\kappa \log n}{n}$ of being 1
    if $b_j = 1$ **then**
        $S \leftarrow S \cup \{j\}$
output S

4.3 Security Analysis

Theorem 4.3. *The pair* (get_mem, verify_mem) *is a committee quorum system scheme.*

Proof. The unpredictability and binding of the scheme directly follow from the pseudorandomness and uniqueness of the VRF scheme. In the rest of the proof we analyze the sizes of H_{tag} and C_{tag} to prove the liveness and safety properties.

Assume $f < (1/3-\epsilon)n$ with $0 < \epsilon < 1$, and thus there are more than $(2/3+\epsilon)n$ nonfaulty parties. Each party is included in the committee if its output r from the VRF is at most $\frac{120\epsilon^{-2}\kappa \log n}{n} \cdot MAX$, and since r is uniformly distributed in $[MAX]$, this event occurs with probability $\frac{120\epsilon^{-2}\kappa \log n}{n}$. Letting S be the random variable describing the size of the committee $E[S] = c'\epsilon^{-2}\kappa \log n$ with $c' = 120$. Since there are $(2/3+\epsilon)n$ nonfaulty parties in the network, the expected number of nonfaulty parties in the committee is $E[H] = (2/3+\epsilon)E[S]$.

We bound the maximal committee size to be $m = (1+\epsilon/2)E[S]$. For a large enough constant $c' \geq 20c$:

$$P[S \geq m = (1+\epsilon/2)E[S]] \leq e^{-\frac{\epsilon^2 E[S]}{10}} = e^{-\frac{c'\kappa \log n}{10}} \leq 2^{-\kappa c \log n}$$

Nest, we bound the minimal number of nonfaulty parties in a committee to be $h > \lceil (2/3) \rceil m$. On the one hand, this means that $h > (2/3)m = (2/3)(1+\epsilon/2)E[s]$. On the other hand, in order to apply measure concentration arguments on the number of faulty parties, we would like to express h as a function of the expected number of honest parties and bound the probability that it is small: $h = (1-x)E[H] = (1-x)(2/3+\epsilon)E[S]$. Comparing the two quantities, we get:

$$\begin{aligned}
(1-x)(2/3+\epsilon)E[S] &> (2/3)(1+\epsilon/2)E[S] \\
1-x &> (2/3)(1+\epsilon/2)/(2/3+\epsilon) \\
1-x &> (2/3)(1+\epsilon/2)/((2+3\epsilon)/3) \\
x &< 1-(2/(2+3\epsilon))(1+\epsilon/2) \\
x &< 1-(2+\epsilon)/(2+3\epsilon) = 2\epsilon/(2+3\epsilon)
\end{aligned}$$

This means that we can set x to be smaller than $2\epsilon/(2+3\epsilon)$, e.g. $x = 2\epsilon/5$. For a large enough constant $c' \geq 30c$:

$$\begin{aligned}
P[H < (2/3)(1+\epsilon/2)E[S]] &\leq P[H < (1-2\epsilon/5)(2/3+\epsilon)E[S]] \\
&= P[H < (1-2\epsilon/5)E[H]] \\
&\leq e^{-\frac{4\epsilon^2 E[H]}{50}} = e^{-\frac{4(2/3+\epsilon)c'\kappa \log n}{50}} \leq 2^{-\kappa c \log n}
\end{aligned}$$

Concretely, for $c = 4$ we set $c' = 120$ and sample with expectation $E[S] = c'\epsilon^{-2}\kappa \log n$. This allows us to set $m = (1+\epsilon/2)E[S]$ and $h = (1-x)(2/3+\epsilon)E[S] > (2/3)m$ and assuming we use $< n^4$ tags obtain a total error that is less than $2^{-\kappa}$. □

Theorem 4.4. sample_committee *is a committee sampling algorithm, with an error of at most* $2^{\kappa c \log}$ *per tag.*

Proof. The proof follows an identical probability analysis to that of Theorem 4.3. □

5 SCALABLE BROADCAST

At a high level, a committee broadcast protocol allows each committee member to run a reliable broadcast protocol. The protocol asynchronously outputs pairs (j, m), which essentially indicate that party j is a committee member who broadcasted the message m. Note that this is a slightly

non-standard definition of broadcast, as usually there is one globally-known sender, and every party simply outputs the message it sends. In our use case, parties cannot know in advance which parties may play the role of broadcasters, as they are supposed to be anonymous until they speak once. As such, we consider all broadcast instances simultaneously and allow parties to output pairs (j, m) whenever they see that one of the broadcasts succeeded.

The definition below is called "committee broadcast protocol", and we show that using our scalable quorum system we can obtain a near-linear and constant time implementation of this definition. We call such a protocol a "scalable broadcast protocol".

Looking ahead, in one use, parties will wait for at least h such pairs to be output and use that set of indices as their input to a committee gather protocol.

5.1 Definition

Definition 5.1. In a Committee Broadcast protocol, every party i has a message m_i and a tag tag as input, and parties may output pairs (j, m'_j) such that $j \in [n]$. Parties also have oracle access to (get_mem, verify_mem). A Committee Broadcast protocol has the following properties:

- **Validity.** If j is nonfaulty at the time it calls the protocol and a nonfaulty party outputs (j, m'_j), then $m'_j = m_j$.
- **Correctness.** If two nonfaulty parties output (j, m) and (j, m'), then $m = m'$.
- **Verifiability.** If a nonfaulty party outputs (j, m'_j) from the Broadcast protocol with tag, then $\mathsf{get\_mem}(\mathsf{sk}_j, tag) = 1, \pi$ for some π.
- **Liveness.**
  - **Liveness Validity.** If j is nonfaulty at the time it calls the protocol and $\mathsf{get\_mem}(\mathsf{sk}_j, tag) = 1, \pi$ for some π, then all nonfaulty parties eventually output (j, m_j).
  - **Totality.** If some nonfaulty party outputs (j, m'_j) for some $j \in [n]$, then all nonfaulty parties eventually do so as well.

5.2 Construction

The protocol in Algorithm 4 is a slight variation on the standard broadcast protocol by Bracha [9]. Each party starts by checking if it is a member of the committee allowed to broadcast their message. Every committee member then sends the message it wants to broadcast to all parties in a “val” message. Adapting the protocol to a committee-based protocol, parties check whether they are allowed to participate in each round of the broadcaster’s protocol. Any party who is a member of the sender’s echo committee sends an “echo” message after hearing the “val” message. Similarly, every party sends a “vote” message if they are members of the vote committee and they either see a quorum of “echo” messages, or enough “vote” messages to know that at least one nonfaulty sender sent such a message.

5.3 Security Analysis

Lemma 5.2. *No two nonfaulty parties will send two conflicting votes for the same k. That is, if nonfaulty parties send $\langle$“vote”, $m, \pi, k, tag\rangle$ and $\langle$“vote”, $m', \pi', k, tag\rangle$, then $m = m'$.*

Proof. Suppose by contradiction that the claim does not hold and let i and j be the first nonfaulty parties that send $\langle$“vote”, $m, \pi, k, tag\rangle$, and $\langle$“vote”, $m', \pi', k, tag\rangle$ respectively such that $m \neq m'$. Since they are the first parties that do so, when they sent their messages, they had only received $\langle$“vote”, $m'', \pi'', k, tag\rangle$ messages from Byzantine parties. This means that at that time they received at most $\lceil h/2 \rceil$ “vote” messages for k, as they first check that for every such received message $\mathsf{verify\_mem}(\mathsf{pk}_j, 1, \pi'', (k, \text{“vote”}, tag)) = 1$. Therefore, they sent their messages after receiving h “echo” messages for k. From the robust quorum intersection property of the quorum

Algorithm 4 Broadcast(m_i, tag)

// Round 1: disseminate message

1: val_member, $\pi_i \leftarrow$ get_mem(sk_i, tag)
2: **if** val_member = 1 **then**
3: sends $\langle$"val", $m_i, \pi_i, i, tag\rangle$ to all parties

// Round 2: echo message

4: **upon** receiving a message $\langle$"val", $m_j, \pi_j, j, tag\rangle$ from j, **do**
5: echo_member, $\pi_i \leftarrow$ get_mem$(\mathsf{sk}_i, (j,$ "echo", $tag))$
6: **if** echo_member = 1 and verify_mem$(\mathsf{pk}_j, 1, tag) = 1$ **then**
7: send $\langle$"echo", $m_j, \pi_i, j, tag\rangle$ to all parties

// Round 2: vote message after quorum of echoes or reliable subset of votes

8: **upon** receiving $\langle$"echo", $m_k, \pi_j, k, tag\rangle$ with the same m_k from h parties such that verify_mem$(\mathsf{pk}_j, 1, \pi_j, (k,$ "echo", $tag)) = 1$, **do**
9: vote_member, $\pi_i \leftarrow$ get_mem$(\mathsf{sk}_i, (k,$ "vote", $tag))$
10: **if** vote_member = 1 and i hasn't sent a "vote" message for k **then**
11: send $\langle$"vote", $m_k, \pi_i, k, tag\rangle$ to all parties
12: **upon** receiving $\langle$"vote", $m_k, \pi_j, k, tag\rangle$ with the same m_k from $\lceil h/2\rceil + 1$ parties such that verify_mem$(\mathsf{pk}_j, 1, \pi_j, (k,$ "vote", $tag)) = 1$, **do**
13: vote_member, $\pi_i \leftarrow$ get_mem$(\mathsf{sk}_i, (k,$ "vote", $tag))$
14: **if** vote_member = 1 and i hasn't sent a "vote" message for k **then**
15: send $\langle$"vote", $m_k, \pi_i, k, tag\rangle$ to all parties
16: **upon** receiving $\langle$"vote", $m_k, \pi_j, k, tag\rangle$ with the same m_k from h parties such that verify_mem$(\mathsf{pk}_j, 1, \pi_j, (k,$ "vote", $tag)) = 1$, **do**
17: **output** (k, m_k)

system, the h "echo" messages that i and j received have at least one nonfaulty sender in common, that only sends one such message to all parties. Therefore, $m = m'$, by contradiction. □

Theorem 5.3. *The* Broadcast *protocol is a Committee Broadcast protocol assuming a committee quorum system, conditioned on no errors.*

Proof. Each property is proven individually.

Validity. Assume that i is nonfaulty and that some nonfaulty party outputs (j, m). In that case, it received h $\langle$"vote", $m, \pi, k, tag\rangle$ messages with verifying proofs. As shown in the proof of Lemma 5.2, at the time the first nonfaulty party sends such a message, it received h $\langle$"echo", $m, \pi, k, tag\rangle$ messages with verifying proofs. Out of those, at least $\lceil h/2\rceil$ were sent by nonfaulty parties, that only send such a message after receiving a $\langle$"val", $m, \pi, k, tag\rangle$ message from k. A nonfaulty k only sends such a message with $m = m_k$, as required.

Correctness. Assume by way of contradiction that two nonfaulty parties output (j, m) and (j, m') with $m \neq m'$ for some $j \in [n]$. Before doing so, they each received h messages of the form $\langle$"vote", $m, \pi, j, tag\rangle$ and $\langle$"vote", $m', \pi', j', tag\rangle$ with verifying proofs. From the honest majority property of the quorum system, there is at least one nonfaulty party in each set of h messages, and thus $m = m'$ from Lemma 5.2.

Verifiability. Assume some nonfaulty party outputs (j, m). That nonfaulty party did so after receiving h "vote" messages for j. As shown in the proof of Correctness, the first party that sends such a "vote" message does so after receiving h "echo" message for j with verifying proofs. From the honest majority property of the quorum system, at least $\lceil h/2\rceil + 1$ of these messages were sent

by nonfaulty parties, that only send "echo" messages after receiving a $\langle$"val", $m, \pi, i, tag\rangle$ message from j with verify_mem$(\mathsf{pk}_j, 1, tag) = 1$, and thus get_mem$(\mathsf{sk}_j, tag) = 1, \pi$, as required.

Liveness. For the Liveness Validity property, let i be a nonfaulty party such that get_mem$(\mathsf{sk}_i, tag) = 1, \pi$ for some π. It starts the protocol by sending an "echo" message with m_i and π to all parties. All parties eventually receive the message and from the liveness of the quorum system, at least h nonfaulty parties j have get_mem$(\mathsf{sk}_j, (i, \text{"echo"}, tag)) = 1, \pi_j$ for some π_j and send "echo" messages for i. Similarly at least h nonfaulty parties j have get_mem$(\mathsf{sk}_j, (i, \text{"vote"}, tag)) = 1, \pi_j$ for some π_j and send "vote" messages for i after hearing the "echo" messages, if they haven't done so earlier. After receiving h such "vote" messages, every nonfaulty party eventually outputs (i, m_i).

For the Totality property, assume some nonfaulty party outputs (j, m) for some $j \in [n]$. That party outputs (j, m) after receiving h $\langle$"vote", $m, \pi, j, tag\rangle$ messages with verifying proofs. From the honest majority property of the quorum system, at least $\lceil h/2 \rceil + 1$ of these messages were sent by nonfaulty parties. Every nonfaulty party receives these messages, and from the liveness of the quorum system, at least h nonfaulty parties j have get_mem$(\mathsf{sk}_j, (i, \text{"vote"}, tag)) = 1$. These parties also send "vote" messages for i after receiving the $\lceil h/2 \rceil$ aforementioned messages. Note that as proven in Lemma 5.2 and in the proof of Correctness, if parties receive $\lceil h/2 \rceil$ "vote" messages for i, they receive the same value m, and thus all such "vote" messages sent by nonfaulty parties have the same value. Eventually, every nonfaulty party receives those messages and outputs (j, m). □

5.4 Efficiency Analysis

Lemma 5.4. *The* Broadcast *protocol has a total communication complexity of* $O((\lambda+\ell)nm^2)$*, where* ℓ *is the size of the input and* m *is the parameter defined in the quorum system. In addition, parties output a tuple* (j, r_j) *in* $O(1)$ *rounds if* j *is nonfaulty when calling the protocol and* get_mem$(\mathsf{sk}_j, tag) = 1$*, and if a nonfaulty party output* (j, r_j) *every nonfaulty party outputs that tuple* $O(1)$ *rounds following that.*

Proof. A nonfaulty party i sends a "val" message if get_mem$(\mathsf{sk}_i, tag) = 1, \pi$, and from the safety of the quorum system there are at most m such parties. In addition, every nonfaulty i may send an "echo" message for j if it receives "val" message from j and get_mem$(\mathsf{sk}_i, (j, \text{"echo"}, tag)) = 1, \pi$. From the safety of the quorum system, for each j with a verifying "val" message there are at most m parties such that get_mem$(\mathsf{sk}_i, (j, \text{"echo"}, tag)) = 1, \pi$, for a total of m^2 messages. As shown in the proof of the verifiability of the Broadcast protocol, get_mem$(\mathsf{sk}_j, tag) = 1, \pi$ also holds if some nonfaulty party sends a "vote" message for j. Following a similar calculation, nonfaulty parties send m^2 such messages. Each one of these messages contains a proof π of size λ, a message of size ℓ, an index of size $\log n \le \lambda$ and two constant-sized tags, for a total of $O((\lambda + \ell)nm^2)$.

If j is nonfaulty when calling the protocol and get_mem$(\mathsf{sk}_j, tag) = 1$, it starts the protocol by sending a "val" message. As shown in the proof of Termination, nonfaulty parties then send "echo" messages after one round and "val" messages a round after that. After receiving those "val" messages, in $O(1)$ rounds, parties output (j, m_j). In addition, as shown in the proof of Termination, if some nonfaulty party output (j, m_j), all nonfaulty parties receive $\lceil h/2 \rceil + 1$ "vote" messages for j in one round, they then receive h such messages after another round, and output (j, m_j) as well. □

Assuming $f < (1/3-\epsilon)n$ and instantiating Broadcast with the quorum system protocol described in Section 4, we get $m = O(\epsilon^{-2}\kappa \log n)$, and thus the communication complexity of the Broadcast protocol is $O((\lambda + \ell)n(\epsilon^{-2}\kappa \log n)^2)$.

6 SCALABLE ASYNCHRONOUS VERIFIABLE SECRET SHARING

At a high level, a committee AVSS protocol allows each member of a committee to run an AVSS protocol by sending a PVSS via the committee broadcast protocol of the previous section. In order

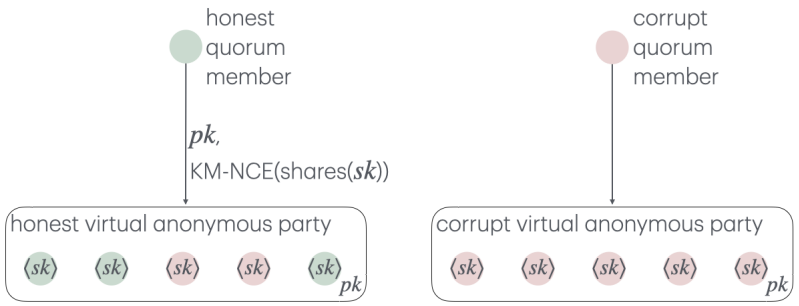

Fig. 2. A corrupt nominating committee may share its secret key to a corrupt committee, resulting in a corrupt virtual party; an honest nominating committee member is almost guaranteed to share its key to a mostly-honest committee, resulting in an honest virtual party.

to make sure the PVSS works well, parties first run a setup protocol. In this setup, each nominating committee member uses the committee broadcast protocol to publish a public key and to secret share its corresponding private key to a secret committee of its choice. This secret committee plays the role of a *virtual anonymous party*. A corrupt nominator may choose an entirely corrupt committee, resulting in a corrupt virtual anonymous party; an honest nominator is almost guaranteed to choose a most-honest committee. See Fig. 2 for an illustration. Once the virtual parties are established, anyone can send a PVSS to their public keys. Once parties decide to reconstruct the shared values, each committee provides shares of the virtual anonymous party's secret key, allowing parties to reconstruct all nonfaulty virtual parties' keys, and then the secret shared in any PVSS transcript.

Similarly to the broadcast protocol, the definition of the committee AVSS scheme is slightly non-standard in the sense that there is an anonymous committee of dealers that share their secrets simultaneously. Parties may output indices $j \in [n]$ whenever they see that the committee member j successfully shared its secret using the virtual anonymous parties' secret keys. Parties may then reconstruct *all secrets at the same time*, outputting pairs of the form (j, r_j), where j is a dealer and r_j is its reconstructed secret.

Again, note that the definition is called "committee AVSS protocol," and then we can obtain a near-linear and constant time implementation of this definition. We call such a protocol a "scalable AVSS protocol."

6.1 Definition

Definition 6.1. A Committee Asynchronous Verifiable Secret Sharing (AVSS) scheme consists of three protocols (Setup, Share, Rec). Parties call the Setup protocol with an input tag and complete the call with no output. Every party i has a secret s_i and a tag tag as input to the Share protocol and may output indices $j \in [n]$. Parties also have oracle access to (get_mem, verify_mem). Parties may call Rec with input tag and output pairs of the form (j, r_j) such that $j \in [n]$. Parties may only call the Share protocol after completing a call to the Setup protocol with the same tag. Parties share their state between calls to the three protocols with the same tag. The triplet (Setup, Share, Rec) is said to be a Committee AVSS scheme if the following properties hold:

- **Correctness.** Once the first nonfaulty party outputs $j \in [n]$ from Share with the tag tag, there exists a value r_j such that every nonfaulty party that outputs a tuple of the form (j, r) from Rec with the same tag does so with $r = r_j$. Furthermore, if j is nonfaulty at the time it calls the Share protocol, then $r_j = s_j$.

- **Verifiability.** If a nonfaulty party outputs j from Share or (j, r) from Rec with the tag tag, then get_mem$(\mathsf{sk}_j, tag) = 1, \pi$ for some π.
- **Liveness.**
  - **Setup liveness.** If all nonfaulty parties participate in the Setup protocol, then they all complete it.
  - **Share liveness.** If all nonfaulty parties participate in the Share protocol, then they all eventually output j for every party j that is nonfaulty at the time it calls the protocol such that get_mem$(\mathsf{sk}_j, tag) = 1, \pi$ for some π.
  - **Totality.** If some nonfaulty party outputs j from Share with the tag tag, every nonfaulty party eventually outputs j from Share with the same tag.
  - **Recover liveness.** If all nonfaulty parties call the Rec protocol with tag, then every nonfaulty party that output j from Share with the same tag eventually outputs (j, r_j) from Rec as well.
- **Secrecy.** Before the first nonfaulty party calls Rec the adversary's view is independent of s_j for every nonfaulty j.

6.2 Construction

The scheme consists of the Setup protocol, described in Algorithm 5, the Share protocol, described in Algorithm 6, and the Rec protocol, described in Algorithm 7. In the Setup protocol, parties generate a set of virtual anonymous parties with known public keys, as described above. Each member of the "key" committee does that by sampling a public key pair, and a committee to hold shares of the secret (forming the virtual anonymous party). The "key" committee member then computes a secret sharing of the secret key, encrypts each share to a member of his chosen committee, and broadcasts the public key along with the vector of encrypted shares. After seeing at least h such broadcasts, parties complete the setup.

In the Share protocol, parties check whether they are allowed to share their secret. Any party allowed to share its secret generates a PVSS transcript with the virtual committee members' public keys it saw during the Setup protocol and broadcasts it to all parties. After receiving such a broadcast, and seeing that it reports a set of enough ($\geq h$) public keys, parties consider this sharing successful and output the dealer's index. Note that having at least h virtual committee members guarantees that at least $\lceil h/2 \rceil + 1$ of them are nonfaulty, allowing parties to later reconstruct the shared secret.

Finally, in the Rec protocol any party holding a share of a virtual party's secret key publishes this key. After reconstructing this key for every nonfaulty virtual member, parties will be able to reconstruct any of the secrets shared in the Share protocol.

Technical notes. In line 9 of the Rec protocol, parties interpolate points to a polynomial using the Interpolate algorithm. The algorithm takes a set $P = \{(j, y_j)\}$ of size $d + 1$ such that $j \neq k$ for every $(j, y_j), (k, y_k) \in P$ and outputs the unique polynomial p of degree d or less such that $\forall (j, y_j) \in P\ p(j) = y_j$. Note that lines 2-6 of the Setup algorithm can be executed only by the parties that are members of the ("key", tag) committee.

6.3 Security Analysis

Theorem 6.2. *The triplet* (Setup, Share, Rec) *is a Committee AVSS scheme assuming a committee quorum system, conditioned on no errors.*

Proof. Each property is proven individually.

Correctness. Assume some nonfaulty party outputs $j \in [n]$ from Share with the tag tag. Before doing so, that party outputs $(j, I_j, \mathsf{trans}_j)$ from the Broadcast call with tag tag, saw that for every

Algorithm 5 Setup(tag)

key_transcripts$_i \leftarrow \emptyset$
// Create a public key pair, broadcast the public key and share the secret key to a random committee
$S \xleftarrow{\$}$ sample_committee(), (pk, sk) $\xleftarrow{\$}$ KeyGen(1^λ)
uniformly sample a polynomial $p(x) \in \mathbb{F}^{\lceil h/2 \rceil}[x]$ of degree $\lceil h/2 \rceil$ such that $p(0) = \text{sk}$
key_trans $\leftarrow \{\text{Enc}\left(\text{pk}_j, (p(j), \text{Sign}(\text{sk}_i, (j, p(j), tag))\right) | j \in S\}$
erase S, pk, sk, p and any randomness used in generating these values
call Broadcast((pk, key_trans), ("key", tag))
// Wait to receive enough public key pairs, and then complete the setup after having h
upon outputting $(j, \text{pk}'_j, \text{key\_trans}_j)$ from Broadcast with tag ("key", tag), **do**
  if $\left|\text{key\_trans}_j\right| \leq m$ **then**
    key_transcripts$_i \leftarrow$ key_transcripts$_i \cup \{(j, \text{pk}'_j, \text{key_trans}_j)\}$
    if $\left|\text{key\_transcripts}_i\right| = h$ **then**
      complete the protocol, but continue processing messages

Algorithm 6 Share(s_i, tag)

transcripts$_i \leftarrow \emptyset$
// Share s_i using the keys collected during the setup
trans $\xleftarrow{\$}$ PVSS($\{\text{pk}'_j | (j, \text{pk}'_j, \text{key\_trans}_j) \in \text{key\_transcripts}_i\}, s_i$)
$I \leftarrow \{j | (j, \text{pk}'_j, \text{key\_trans}_j) \in \text{key\_transcripts}_i\}$
erase s_i and any randomness used in generating trans, I
call Broadcast((I, trans), tag)
// After seeing that j shared a value with enough correct public keys, output j
upon outputting (j, I_j, trans_j) from Broadcast with tag tag, **do**
  upon $\forall k \in I_j \; \exists (k, \text{pk}'_k, \text{key\_trans}_k) \in \text{key\_transcripts}_i$, **do**
    if $\left|I_j\right| \geq h$ and PVSS_verify($\{\text{pk}'_k | k \in I_j\}, \text{trans}_j$) $= 1$ **then**
      transcripts$_i \leftarrow$ transcripts$_i \cup \{(j, I_j, \text{trans}_j)\}$
      output j from the protocol

$k \in I_j$ there exists a tuple $(k, \text{pk}', \text{key\_trans}_k)$ in its key_transcripts set, that $\left|I_j\right| \geq h$ and finally that PVSS_verify($\{\text{pk}'_k | k \in I_j\}, \text{trans}_j$) $= 1$. From the binding of the PVSS scheme, there exists a unique polynomial p of degree $\lceil h/2 \rceil$ or less such that any verifying share lies on the polynomial p, and reconstructing the transcript using any $\lceil h/2 \rceil + 1$ such shares results in $p(0)$. Let $r_j = p(0)$ for that polynomial. Every nonfaulty party that outputs a pair (j, r'_j) reconstructs the transcript trans'_j such that it output $(j, I'_j, \text{trans}'_j)$ from the Broadcast call with tag tag. From the Correctness of Broadcast, for every such output $(j, I'_j, \text{trans}'_j) = (j, I_j, \text{trans}_j)$.

Nonfaulty parties output (j, r'_j) after reconstructing the transcript trans_j using at least $\lceil h/2 \rceil + 1$ verifying shares and outputting r'_j. As mentioned above, this means that $r'_j = r_j$, as required. Furthermore, if j is nonfaulty, from the Validity of Broadcast every party outputs (j, trans_j) from the protocol with trans_j being the transcript j computed for sharing s_j. From the correctness of the scheme $s_j = p(0) = r_j$, as required.

Verifiability. Assume some nonfaulty party output either j from Share or (j, r_j) from Rec with tag tag. Before doing so, that party output (j, I_j, trans_j) from Broadcast with tag tag and added

Algorithm 7 Rec(tag)

1: $\text{keys}_i \leftarrow \emptyset, \forall j \in [n]\ \text{key\_shares}_i[j] \leftarrow \emptyset$
// Reconstruct secret keys
2: **upon** $(j, \text{pk}'_j, \text{key\_trans}_j) \in \text{key\_transcripts}_i$, **do**
// Send shares of secret keys if chosen as a share-holder
3: **if** $\exists \text{key\_enc} \in \text{key\_trans}_j$ *s.t.* $\text{Dec}(\text{sk}_i, \text{key\_enc}) \neq \perp$ **then**
4: send $\langle$"rec", $\text{Dec}(\text{sk}_i, \text{key\_enc}), j, tag\rangle$ to all parties
// Wait to receive enough shares of the secret keys, and then reconstruct them
5: **upon** receiving $\langle$"rec", $y_k, \sigma_k, j, tag\rangle$ from k, **do**
6: **if** $\text{Sign\_verify}(\text{pk}_j, (k, y_k, tag), \sigma_k) = 1$ **then**
7: $\text{key\_shares}_i[j] \leftarrow \text{key\_shares}_i[j] \cup \{(k, y_k)\}$
8: **if** $\left|\text{key\_shares}_i[j]\right| = \lceil h/2 \rceil + 1$ **then**
9: $p \leftarrow \text{Interpolate}(\text{key\_shares}_i[j])$
10: $\text{keys}_i \leftarrow \text{keys}_i \cup \{(j, \text{pk}'_j, p(0))\}$
// After seeing a transcript of a shared value and reconstructing its keys, reconstruct the secret
11: **upon** transcripts_i or keys_i being updated, **do**
12: **for all** $(j, I_j, \text{trans}_j) \in \text{transcripts}_i$ s.t. no tuple of the form (j, s_j) has been output **do**
// Attempt to use keys to get shares, check their correctness, and only consider correct shares
13: **for all** $k \in I_j$ s.t. $\exists (k, \text{pk}'_k, \text{sk}'_k) \in \text{keys}_i$ **do**
14: $\text{share}_k \leftarrow \text{PVSS\_dec}(\{\text{pk}'_k | k \in I_j\}, \text{trans}_j, \text{sk}'_k)$
15: **if** $\text{PVSS\_share\_verify}(\{\text{pk}'_k | k \in I_j\}, \text{trans}_j, \text{pk}'_k, \text{share}_k) = 0$ **then**
16: $\text{share}_k \leftarrow \perp$
17: $\text{shares} \leftarrow \{(\text{pk}'_k, \text{share}_k) | k \in I_j, (k, \text{pk}'_k, \text{sk}'_k) \in \text{keys}_i, \text{share}_k \neq \perp\}$
18: **if** $|\text{shares}| \geq \lceil h/2 \rceil + 1$ **then**
19: **output** $(j, \text{PVSS\_rec}(\{\text{pk}'_k | k \in I_j\}, \text{trans}_j, \text{shares}))$

that tuple to its transcripts set. From the verifiability of the Broadcast protocol, $\text{get\_mem}(\text{sk}_j, tag) = 1, \pi$ for some π, as required.

Liveness. For Setup liveness, assume all nonfaulty parties participate in the Share protocol with a given tag. Every nonfaulty j samples a set S, a public key pair $(\text{pk}'_j, \text{sk}'_j)$, a polynomial $p(x)$ of degree $\lceil h/2 \rceil$ such that $p(0) = \text{sk}$, generates a vector key_trans of encryptions of evaluations of p and calls Broadcast with the input $(\text{pk}'_j, \text{key\_trans}_j)$ and tag ("key", tag). From the liveness and validity of the Broadcast protocol, every nonfaulty party outputs $(j, \text{pk}'_j, \text{key\_trans}_j)$ from the protocol for every nonfaulty j such that $\text{get\_mem}(\text{sk}_j, (\text{"key"}, tag)) = 1, \pi$ for some π, and adds this tuple to its key_transcripts set. From the liveness of the quorum system, there are at least h such nonfaulty parties, and thus every nonfaulty party eventually has h values in its key_transcripts set and completes the Setup protocol.

For Share liveness, assume all nonfaulty parties participate in the Share protocol with a given tag after completing a call to Setup with the same tag. Every nonfaulty i starts the Share protocol by computing a PVSS transcript trans_i with the h received public keys pk'_j in the Setup protocol and a set I_i of the indices of parties who broadcasted these public keys. Following that, every nonfaulty i calls Broadcast with the input (I_i, trans_i) and tag tag. Similarly to above, from the liveness and validity of the Broadcast protocol every nonfaulty party outputs (i, I_i, trans_i) for every nonfaulty i such that $\text{get\_mem}(\text{sk}_i, tag) = 1, \pi$ for some π. From the termination of the Broadcast protocol, every nonfaulty party will eventually output the same $(j, \text{pk}'_j, \text{key\_trans}_j)$ values from

the Broadcast call with tag ("key", tag) as i did. This means that eventually, for every $j \in I_i$, every nonfaulty party will have the tuple $(j, \mathsf{pk}'_j, \mathsf{trans}_j)$ in its key_transcripts set, see that i did indeed choose a set I_i of size h and that PVSS_verify$(\{\mathsf{pk}'_j | j \in I_i\}, \mathsf{trans}_i) = 1$, and output i.

For Totality, assume some nonfaulty party output j from the protocol. This means that it output $(j, I_j, \mathsf{trans}_j)$ from the Broadcast call with tag tag, saw that $\forall k \in I_j$ there exists a tuple $(k, \mathsf{pk}'_k, \mathsf{key\_trans}_k)$ in its key_transcripts set, that $|I_j| \geq h$ and that PVSS_verify$(\{\mathsf{pk}'_k | k \in I_j\}, \mathsf{trans}_j) = 1$. That party adds the tuple $(k, \mathsf{pk}'_k, \mathsf{key\_trans}_k)$ to its key_transcripts set after outputting it from the call to Broadcast with tag ("key", tag). From the liveness and correctness of the Broadcast protocol, every nonfaulty party outputs the same $(j, I_j, \mathsf{trans}_j)$ and $(k, \mathsf{pk}'_k, \mathsf{key\_trans}_k)$ tuples from the respective calls to Broadcast and updates their own key_transcripts sets. After that, every nonfaulty party sees that the same conditions hold and output j from the Share protocol.

For Recover liveness, assume all nonfaulty parties call the Rec protocol with the tag tag and that some nonfaulty i output j from the Share protocol with the same tag. Since i output j from the Share protocol,it added a transcript $(j, I_j, \mathsf{trans}_j)$ to $\mathsf{transcripts}_i$ after checking that for every $k \in I_j$ there exists a tuple $(k, \mathsf{pk}'_k, \mathsf{key\_trans}_k) \in \mathsf{key\_transcripts}_i$, that $|I_j| \geq h$ and that PVSS_verify$(\{\mathsf{pk}'_k | k \in I_j\}, \mathsf{trans}_j) = 1$. A tuple $(k, \mathsf{pk}'_k, \mathsf{key\_trans}_k)$ is only added to $\mathsf{key\_trans}_i$ after outputting it from the Broadcast protocol with tag ("key", tag) in the Setup protocol. From the termination and validity of the Broadcast protocol, every nonfaulty party eventually outputs the same tuple and adds it to its own key_transcripts set. In addition, from the verifiability of the protocol, get_mem$(\mathsf{sk}_k, tag) = 1, \pi$ for some π for every one of the tuples above. From the honest majority property of the quorum system, and since $|I_j| \geq h$, there are more than $\lceil h/2 \rceil$ indices of nonfaulty parties in I_j. During the Setup protocol, every such nonfaulty party $k \in I_j$ sampled a public key pair (pk, sk), a set S of size at most m, and a polynomial p of degree $\lceil h/2 \rceil$ such that $p(0) = \mathsf{sk}$. Party k then computed key_trans as a set of encryptions of $p(l), \mathsf{Sign}(\mathsf{sk}_k, (l, p(l), tag)$ for every $l \in S$ and input pk, key_trans to the Broadcast protocol with tag ("key", tag). From the validity of the protocol, $(\mathsf{pk}'_k, \mathsf{key\_trans}_k) =$ (pk, key_trans). Every nonfaulty $l \in S$ will receive the message, be able to decrypt $p(l), \sigma_l$ from one of the cyphertexts in $\mathsf{key\_trans}_k$ and send $\langle$"rec", $p(l), \sigma_l, k, tag\rangle$ to all parties. Note that the encryptions also hide the recipients of the message, and thus the adversary cannot choose to adaptively corrupt parties in S, meaning that from the liveness of the quorum system, there are at least h parties in S and they remain nonfaulty until they send their "rec" message. Party i receives these messages, sees that the signature on $(l, p(l), tag)$ verifies, and adds the tuple $(l, p(l))$ to $\mathsf{key\_shares}_i[k]$. In addition, the adversary cannot forge a verifying signature σ for any (l', y'_l, tag) that wasn't generated by the nonfaulty party k, and thus i only adds pairs of the form $(l, p(l))$ to the $\mathsf{key\_shares}_i[k]$. After receiving $\lceil h/2 \rceil + 1$ such values, i interpolates the shares to the polynomial p and adds $(k, \mathsf{pk}'_k, \mathsf{sk}'_k)$ to keys_i. From the binding and correctness of the PVSS scheme, for every such key, i computes a verifying share from trans_j using every nonfaulty key sk'_k, and adds them to the set shares. Following that, there are at least $\lceil h/2 \rceil + 1$ such shares in shares, and thus i performs some local computation and outputs a tuple (j, r_j).

Secrecy. Assume no nonfaulty party called the Rec protocol with tag tag. Throughout the Share protocol, a nonfaulty i may sample a set S, generate a public key pair pk, sk, sample a polynomial p, encrypt evaluations of p to parties in S and broadcast the encryptions. In addition, after outputting h tuples of the form $(j, \mathsf{pk}_j, \mathsf{key\_trans}_j)$ from the Broadcast call with tag ("key", tag), i may generate a PVSS transcript trans_i sharing its secret s_i by using h of these public keys. It then inputs that transcript to Broadcast, along with the set I_i, which includes the indices of parties whose published public keys were used. From the verifiability of the Broadcast protocol, for every $(j, \mathsf{pk}_j, \mathsf{key\_trans}_j)$ that i output, get_mem$(\mathsf{sk}_j, tag) = 1, \pi$ for some π. From the honest majority of the quorum system, less than $\lceil h/2 \rceil$ of these parties are faulty, and thus the adversary only chose less than $\lceil h/2 \rceil$ of

the public keys used for generating trans_i, and potentially knows the associated secret keys. Every other pk_j was sampled by a nonfaulty j. As discussed above, that nonfaulty j uniformly sampled a set $S \subseteq [n]$ of size m and a polynomial p of degree $\lceil h/2 \rceil$ or less such that $p(0) = \text{sk}_j$, and broadcasted a vector of encryptions to the parties in S.

From the honest majority property of the committee sampling algorithm, less than $\lceil h/2 \rceil$ of the parties in S are faulty. In addition, the encryptions hide who the messages were encrypted to, so the adversary cannot adaptively choose parties to corrupt to learn their shares of sk_j. Overall this means that the adversary receives at most $\lceil h/2 \rceil$ evaluations of the polynomial p sampled by any nonfaulty party in I_i. All these evaluations are at points other than 0, and p is uniformly sampled from all polynomials of degree $\lceil h/2 \rceil$ or less such that $p(0) = \text{sk}_j$. Since the adversary less than $\lceil h/2 \rceil$ such evaluations, its view is independent of $p(0) = \text{sk}_j$, and thus it learns nothing about sk_j. In total, this means that the adversary knows less than $\lceil h/2 \rceil$ secret keys corresponding to the public keys used for generating trans_i, and only learns the other public keys used, with the rest of its view being independent of the other secret keys. From the secrecy of the PVSS scheme this means that the adversary learns nothing about s_i, as required. □

6.4 Efficiency Analysis

Lemma 6.3. *The* Setup *and* Share *protocols have a total communication complexity of* $O(\lambda nm^3)$*, and the* Rec *protocol has a total communication complexity of* $O(\lambda nm^2)$*, where* m *is the parameter described in the quorum system.*

Nonfaulty parties complete the Setup *protocol in* $O(1)$ *rounds and output* j *from the* Share *protocol in* $O(1)$ *rounds if* j *is nonfaulty when calling the protocol and* $\text{get\_mem}(\text{sk}_j, tag) = 1$*. Furthermore, if a nonfaulty party outputs* j *from* Share*, every nonfaulty party outputs* j *from* Share *in* $O(1)$ *rounds, and if all parties call* Rec*, every nonfaulty party outputs* (j, r_j) *in* $O(1)$ *rounds for every* j *that they output from* Share*.*

Proof. In the Setup and Share protocols, parties call the Broadcast protocol with either $O(m)$ encrypted values and a public key for a total of $O(\lambda m)$ bits, or a PVSS transcript of size $O(\lambda m)$. In total, this costs $O((\lambda m)nm^2) = O(\lambda nm^3)$. In the Rec protocol, every party included in a key transcript sends a message of size $O(\lambda)$ to all parties. There are only m parties who successfully broadcast their key transcript, and every party checks that the transcripts include at most m encrypted values. This means that in total, parties send $O(\lambda nm^2)$ bits.

In the beginning of the Setup protocol, every nonfaulty party calls the Broadcast protocol. Every party outputs such a transcript for every nonfaulty party in the ("key", tag) committee in $O(1)$ time. There are at least h such parties, and thus every party completes the protocol in $O(1)$ time. Similarly, every nonfaulty party that calls the Share protocol start by calling the Broadcast protocol. Parties output a tuple for each such nonfaulty j in the tag committee in $O(1)$ time and output j. In addition, if some nonfaulty party outputs j from the Share protocol, it did so after outputs a tuple (j, m_j) from the Broadcast call. Every nonfaulty party outputs the same value in $O(1)$ rounds and outputs j as well. Finally, if all nonfaulty parties call Rec, then all nonfaulty parties holding shares of the secret keys send them to all parties. Parties receive these messages in $O(1)$ time, at which time they can reconstruct every secret for which they previously output j from Share.

□

Assuming $f < (1/3 - \epsilon)n$ and instantiating Broadcast with the quorum system protocol described in Section 4, we get $m = O(\epsilon^{-2}\kappa \log n)$, and thus the communication complexity of the Setup and Share protocols is $O(\lambda n(\epsilon^{-2}\kappa \log n)^3)$, and the complexity of the Rec protocol is $O(\lambda n(\epsilon^{-2}\kappa \log n)^2)$. This entire analysis assumes the secrets are of size $O(\lambda)$ that can be shared PVSS transcripts of size

$O(\lambda m)$. Secrets of size $O(\lambda \ell)$ will need to be shared in transcripts of size $O(\lambda \ell m)$, adding a factor of ℓ to the complexity of the Share protocol.

7 SCALABLE VERIFIABLE PARTY GATHER

At a high level, a committee-verifiable party gather protocol allows parties to run a two-round gather protocol, where each round uses the committee broadcast protocol above. At the end of this protocol, each party outputs a set of parties that completed their AVSS, and we are guaranteed that there is a common core that all parties have in their outputs. Moreover, the common core is fixed once the first nonfaulty completes its call to the gather protocol, and the output of gather is verifiable (other parties can be convinced this output must contain the core).

Looking forward, the reason we need the gather to be verifiable, is that in the case of a good event, where the highest rank happens to be for a proposal from a nonfualty party that belongs to the core, we want to make it impossible for the adversary to claim that its output of the gather does not contain this party.

7.1 Definition

A Verifiable Party Gather [1] scheme consists of two protocols Gather and Gather_Verify. Every party i runs the Gather protocol with an input $S_i \subseteq [n]$ and a tag tag, and may output a set $X_i \subseteq [n]$. In addition, parties have access to an asynchronous validity predicate valid. As defined in [1], a call to the asynchronous validity predicate on a value x may eventually terminate with the output 1, eventually terminate with the output 0, or never terminate. Such a predicate is only said to be an asynchronous validity predicate if the following condition holds: if some nonfaulty party outputs $b \in \{0, 1\}$ from $\mathsf{valid}(x)$, then every party that calls $\mathsf{valid}(x)$ also eventually outputs b, and never outputs any other value. We say that $\mathsf{valid}(x) = 1$ for some party i if i would either output 1 upon calling $\mathsf{valid}(x)$, or has already output 1 from such a call. We will only consider asynchronous validity predicates valid for which $\mathsf{valid}(x) = 1$ for at most m values x. Finally, parties may call the Gather_Verify protocol on a set $X \subseteq [n]$ and a tag tag and either eventually output 1, eventually output 0, or never terminate.

The pair (Gather, Gather_Verify) is said to be a Committee Verifiable Gather scheme if the following properties hold assuming all nonfaulty parties start the Gather protocol with the same tag and with inputs S_i such that $|S_i| \geq h$ and $\forall j \in S_i\ \mathsf{valid}(j) = 1$:

- **Binding Core:** Once the first nonfaulty party outputs a value from the Gather protocol there exists a core set $X^* \subseteq [n]$ such that $|X^*| \geq h$ and if a nonfaulty party outputs X_i, then $X^* \subseteq X_i$.
- **Termination of Output:** All nonfaulty parties eventually output a gather-set.
- **Completeness:** For any two nonfaulty parties i, j, if j outputs X_j from Gather with tag tag, and i calls $\mathsf{Gather\_Verify}(X_j, tag)$, i eventually completes the call with the output 1.
- **Agreement on Verification:** For any two nonfaulty parties i, j, and any set of indices $X \subseteq [n]$, if i outputs the value $b \in \{0, 1\}$ from a call to $\mathsf{Gather\_Verify}(X, tag)$ and j also calls $\mathsf{Gather\_Verify}(X, tag)$, then it also eventually outputs b.
- **Includes Core:** If a nonfaulty i completes a call to $\mathsf{Gather\_Verify}(X, tag)$ with the output 1, then $X^* \subseteq X$ (with X^* being the binding-core defined in the Binding Core property).
- **External Validity:** If a nonfaulty i completes a call to $\mathsf{Gather\_Verify}(X, tag)$ with the output 1, then $\forall x \in X\ \mathsf{valid}(x) = 1$ at that time for i.

7.2 Construction

The scheme consists of the Gather protocol, described in Algorithm 8 and the Gather_Verify protocol, described in Algorithm 9. Parties start the Gather by broadcasting their inputs S_i using the committee broadcast protocol. Whenever a set S_i is received, its contents are added to the eventual output from the protocol. After receiving h of these broadcasts (i.e. potentially from every nonfaulty party in the first broadcast committee), parties broadcast the indices of parties whose broadcasts they heard, stored in T_i sets. This also indirectly informs other parties of the contents of the broadcasts they heard, as they will eventually output the same values. After receiving h broadcasts of T_j sets and waiting to hear all of the broadcasts they indicate, parties complete the protocol, outputting the values they collected. In Lemma 7.1, we will show that any quorum of parties who sent T_i sets must have at least $\lceil h/2 \rceil + 1$ parties who included the same index i^* in their set. Parties wait to hear from a quorum before outputting values, and thus they must hear from at least one of these $\lceil h/2 \rceil + 1$ parties. This also means that when verifying the output X in the Gather_Verify protocol, parties can simply wait to hear that it includes the T_i broadcasts of a quorum of the senders in order to know that it includes the core.

Algorithm 8 Gather(S_i, tag)

$R_i \leftarrow \emptyset, T_i \leftarrow \emptyset, U_i \leftarrow \emptyset$
// Round 1: Proposal of inputs
call Broadcast$(S_i, (\text{“propose”}, tag))$
// Round 2: Aggregation of inputs to larger sets
upon outputting (j, S_j) from Broadcast with tag (“propose”, tag) such that $|S_j| \geq h$, **do**
  upon valid(k) outputting 1 for every $k \in S_j$, **do**
    $R_i \leftarrow R_i \cup S_i, \quad T_i \leftarrow T_i \cup \{j\}$
    if $|T_i| = h$ **then**
      call Broadcast$(T_i, (\text{“aggregate”}, tag))$
// Round 3: Save T sets for verification, output R after enough were received
upon outputting (j, T_j) from Broadcast with tag (“aggregate”, tag) such that $|T_j| \geq h$, **do**
  upon $T_j \subseteq T_i$, **do**
    $U_i \leftarrow U_i \cup \left\{\left(j, \bigcup_{k \in T_j} S_k\right)\right\}$
    if $|U_i| = h$ **then**
      output R_i but continue updating internal sets and sending messages

Algorithm 9 Gather_Verify(X, tag)

upon $|\{j \mid \exists (j, V_j) \in U_i, V_j \subseteq X\}| \geq h$ and valid(j) outputting 1 for every $j \in X$, **do**
  output 1 and **terminate**

7.3 Security Analysis

As discussed in Section 7.1, in all of the following proofs we will assume that there are at most m indices $j \in [n]$ such that valid(j) outputs 1 for any nonfaulty party. We will show that this condition holds when using the scheme in the rest of the paper.

Lemma 7.1. *Assume some nonfaulty party completed the protocol. There exists some i^* such that this nonfaulty party output (j, T_j) from the* Broadcast *protocol with tag (“aggregate”, tag) with $i^* \in T_j$ for at least $\lceil h/2 \rceil + 1$ parties $j \in [n]$.*

Proof. Assume some nonfaulty party i completed the protocol. Before completing the protocol, it found that $U_i \geq h$, and thus it had output h pairs of the form (j, T_j) such that $|T_j| \geq h$ and $T_j \subseteq T_i$ from the Broadcast protocol with tag ("aggregate", tag). Let I be the set of parties who sent those broadcasts. Now assume by way of contradiction that for every index $k \in [n]$, there is no subset of $J \subseteq I$ of size $\lceil h/2 \rceil + 1$ such that for every $\forall j \in J$ $k \in T_j$. Note that for every $j \in I$, $T_j \subseteq T_i$, and i only adds an index k to T_i after outputting (k, S_k) from the Broadcast protocol with tag ("propose", tag). From the Verifiability of the Broadcast protocol, for every such k, get_mem(sk_k, ("propose", tag)) $= 1, \pi$ for some π. From the safety of the protocol, there are at most m such indices, and thus there exists a set $Y \subseteq [n]$ such that $|Y| \leq m$ and $\forall j \in I$ $T_j \subseteq Y$. Since there are a total of m possible values, the total number of elements in all sets is no greater than $m \cdot \lceil h/2 \rceil$. On the other hand, there are h such sets, each containing h elements or more, resulting in at least h^2 elements overall. Combining these two observations:

$$h^2 \leq m \cdot \lceil h/2 \rceil$$

From AM-GM inequality:

$$\lceil h/2 \rceil \cdot m \leq (\frac{\lceil h/2 \rceil + m}{2})^2$$

From the safety property of definition 4.1, $m < (3/2)h$, so $\lceil h/2 \rceil + m < 2h$. Therefore,

$$h^2 \leq \lceil h/2 \rceil \cdot m < (\frac{2h}{2})^2 = h^2$$

Which never holds for $h \geq 1$, reaching a contradiction. □

Lemma 7.2. *Let R_i (resp. T_i or U_i) be the set held by a nonfaulty i at some point in time in the protocol. For every nonfaulty $j \in [n]$, $R_i \subseteq R_j$ (resp. $T_i \subseteq T_j$ or $U_i \subseteq U_j$) eventually holds.*

Proof. Observe the sets R_i and T_i at some time in the protocol. Party i updates R_i and T_i after outputting (j, S_j) from Broadcast with the tag ("propose", tag) and seeing that $|S_k| \geq h$ and that valid(l) outputs 1 for every $l \in S_k$. After seeing that this holds, i adds all of S_k to R_i and k to T_i. From the Totality and Correctness of the broadcast protocol, every nonfaulty party eventually outputs the same pair from Broadcast and eventually sees that valid(l) outputs 1 as well for every $l \in S_k$. At that time, j also adds all of S_k to R_j and k to T_j, meaning that $R_i \subseteq R_j$ and $T_i \subseteq T_i$.

Similarly, observe the set U_i at some time in the protocol. Party i adds $(k, \bigcup_{l \in T_k} S_l)$ to U_i after outputting the pair (k, T_k) from the Broadcast call with tag ("aggregate", tag) and seeing that $|T_k| \geq h$ and that $T_k \subseteq T_i$. From the Totality and Correctness of the broadcast protocol, every nonfaulty party j eventually outputs the same pair from Broadcast and the same S_l sets as described above. At that time, j also adds $(k, \bigcup_{l \in T_k} S_l)$ to U_j, meaning that $U_i \subseteq U_j$. □

Theorem 7.3. *The pair* (Gather, Gather_Verify) *is a Committee Verifiable Gather scheme assuming a committee quorum system, conditioned on no errors.*

Binding Core. Assume the first nonfaulty party that completes the Gather protocol is p^*. From Lemma 7.1, there exists some i^* such that p^* output at least h pairs of the form (k, T_k) with $i^* \in T_k$ from the Broadcast protocol with tag ("aggregate", tag). Party p^* only adds a tuple (k, V_k) to U_{p^*} after outputting (k, T_k) from its call to Broadcast with tag ("aggregate", tag). Therefore for some $(k, V_k) \in U_{p^*}$, $i^* \in T_k$. Note that $T_k \subseteq T_{p^*}$, so $i^* \in T_{p^*}$. Before adding i^* to T_{p^*}, p^* output a pair (i^*, S_{i^*}) from the Broadcast protocol with tag ("propose", tag). Let the binding core X^* be the set S_{i^*}. Clearly $|X^*| \geq h$ because $|S_{i^*}| \geq h$. The fact that X^* is a subset of every nonfaulty party's output from the protocol is a direct corollary of the Completeness and Include Core properties of the Gather protocol, proven below.

Termination of Output. Assume every nonfaulty party i calls the Gather protocol with an input $S_i \subseteq [n]$ such that $|S_i| \geq h$ and $\forall x \in S_i$ valid$(x) = 1$ for i. Every nonfaulty i starts by calling Broadcast$(S_i, (\text{“propose”}, tag))$. From the liveness of the quorum system, there are at least h nonfaulty parties i for which get_mem$(\text{sk}_i, (\text{“propose”}, tag)) = 1$, and from the Validity and termination of the protocol, every nonfaulty party eventually outputs (i, S_i) for every such i. Every nonfaulty party will then see that $|S_i| \geq h$ for every such S_i, and since valid is an asynchronous validity predicate, eventually also see that valid(x) outputs 1 for every $x \in S_i$. At that time, every nonfaulty j updates R_j and adds an index to T_j. After adding h such indices, every nonfaulty j calls Broadcast$(T_i, (\text{“aggregate”}, tag))$. Similarly to above, there are at least h nonfaulty parties j for which get_mem$(\text{sk}_j, (\text{“propose”}, tag)) = 1$, and from the Validity and termination of the protocol, every nonfaulty party eventually outputs (j, T_j) with $\left|T_j\right| \geq h$ for every such j. From Lemma 7.2, eventually $T_j \subseteq T_k$ for every nonfaulty k. This means that every nonfaulty k eventually adds a tuple to U_i for each one of these h nonfaulty parties j, sees that $|U_i| = h$, and outputs R_i.

Completeness. Assume some nonfaulty party i completes the Gather protocol and outputs a set X_i. The set X_i is the set R_i at the time i completed the protocol. Party i outputs X_i after outputting h pairs (j, T_j) from the Broadcast protocol with tag (“aggregate”, tag), seeing that $\left|T_j\right| \geq h$ and $T_j \subseteq T_i$, and adding a tuple (j, V_j) to U_i. From Lemma 7.2, every nonfaulty k eventually has $(j, V_j) \in U_k$ for every $(j, V_j) \in U_i$. This means that it is enough to show that for every one of these h tuples, $V_j \subseteq X_i$. Let (j, V_j) be one such tuple, and let T_j be the set received in the broadcast described above. As shown above, $T_j \subseteq T_i$ at the time i outputs X_i. Let k be some index in $T_j \subseteq T_i$. Party i only adds k to T_i after outputting (k, S_k) from the Broadcast protocol with tag (“propose”, tag), and at that time it also updates R_i to be $R_i \cup S_k$. This means that for every $k \in T_j$, $S_k \subseteq R_i$ at the time i outputs $X_i = R_i$. Since $V = \bigcup_{k \in T_j} S_k$, $V_j \subseteq X_i = R_i$ for every $(j, V_j) \in U_i$ at that time, completing the proof.

Agreement on Verification. Assume that some nonfaulty party i outputs $b \in \{0, 1\}$ from a call to Gather_Verify(X, tag), and that all the nonfaulty parties participate in the Gather protocol. Note that nonfaulty parties only output 1 from the protocol and thus $b = 1$. At the time i completed the protocol, it saw that $|\{k | \exists (k, V_k) \in U_i, V_k \subseteq I\}| \geq h$ and that valid(k) output 1 for every $k \in X$. From Lemma 7.2, for every nonfaulty j, eventually $U_i \subseteq U_j$ and thus $\left|\left\{k | \exists (k, V_k) \in U_j, V_k \subseteq X\right\}\right| \geq h$ also eventually holds. In addition, since valid is an asynchronous validity predicate, valid(k) eventually output 1 for j for every $k \in X$. Therefore, if j calls Gather_Verify(X, tag), it eventually outputs 1 as well.

Include Core. Assume that a nonfaulty party i outputs 1 from its call to Gather_Verify(X, tag) for some $X \subseteq [n]$. Party i found that $|\{k | \exists (k, V_k) \in U_j, V_k \subseteq X\}| \geq h$. As discussed above, party i only adds (j, V_j) to U_i after outputting (j, T_j) from the Broadcast call with tag (“aggregate”, tag). Let i^* be defined as it is in Lemma 7.1 and in the Binding Core property. The party p^* defined in the Binding Core property output $\lceil h/2 \rceil + 1$ tuples of the form (j, T_j) with $i^* \in T_j$ from the Broadcast call with tag (“aggregate”, tag). In addition, there are h tuples such that $(j, V_j) \in U_i$ and $V_j \subseteq X$. Note that for any such T_j or V_j, some nonfaulty party output a tuple of the form (j, T_j) from the Broadcast call with tag (“aggregate”, tag) and thus from the Verifiability of the Broadcast protocol, get_mem$(\text{sk}_j, (\text{“aggregate”}, tag)) = 1, \pi$ for some π. Therefore, from the quorum intersection property of the quorum system, for at least one of those tuples (j, V_j), $i^* \in T_j$. By definition, $V_j = \bigcup_{k \in T_j} S_k$, and thus $X^* = S_{i^*} \subseteq V_j \subseteq X$, as required.

External Validity. Assume that for some nonfaulty i, Gather_Verify(X, tag) terminates with the output 1. Before doing so, i checks that valid(j) output 1 for every $j \in X$, as required.

7.4 Efficiency Analysis

Lemma 7.4. *The* Gather *protocol has a total communication complexity of* $O((\lambda + m \log n)nm^2)$ *and the* Gather_Verify *protocol requires no communication, where* m *is the parameter described in the quorum system. and a round complexity of* $O(1)$*. Assuming that if a nonfaulty party outputs* 1 *from* valid(x)*, every nonfaulty party does so in* $O(1)$ *time, the* Gather *protocol has a round complexity of* $O(1)$*, and if a nonfaulty party outputs* X *from* Gather *or* 1 *from* Gather_Verify *run on* X*, every nonfaulty party does so after* $O(1)$ *rounds.*

Proof. In the Gather protocol, parties call Broadcast twice with inputs of $O(m)$ indices, each of size $O(\log n)$. In total, this requires $O((\lambda + m \log n)nm^2)$ bits to be sent. Parties only perform local computation in the Gather_Verify protocol.

Parties output (j, S_j) for every nonfaulty j in the ("propose", tag) committee. After outputting these tuples for the h nonfaulty parties in the committee, every party broadcasts its T_i set. Similarly, parties receive h such sets in $O(1)$ time from the nonfaulty parties in the ("aggregate", tag) committee. They also receive all the S_i values received by other nonfaulty parties in $O(1)$ time and see that valid$(x) = 1$ for every $x \in S_i$ in $O(1)$ time, after which they accept the received T_i sets and complete the protocol in a total of $O(1)$ rounds. If a nonfaulty i output X from Gather or 1 from Gather_Verify on X, it had h tuples of the form $(j, V_j) \in U_i$, that were added after outputting (j, T_j) from Broadcast, and saw that valid$(x) = 1$ for every $x \in X$. Following similar arguments, every nonfaulty party outputs the same values in $O(1)$ rounds and outputs 1 from Gather_Verify. □

Assuming $f < (1/3 - \epsilon)n$ and instantiating Broadcast with the quorum system protocol described in Section 4, we get $m = O(\epsilon^{-2}\kappa \log n)$, and thus the communication complexity of the Gather protocol is $O(\lambda n(\epsilon^{-2}\kappa \log n)^2 + n \log n(\epsilon^{-2}\kappa \log n)^3)$.

8 SCALABLE PROPOSAL ELECTION

At a high level, a proposal election scheme allows parties to propose an input so that, with constant positive probability, all parties will output the same input. To do that, parties use the committee broadcast to publish their inputs and the committee AVSS to commit to an unpredictable rank from their VRF. Parties then run the committee gather protocol. Since the adversary doesn't know the rank of honest parties, there is a constant probability that the adversary chooses a binding core that includes a nonfaulty proposer (who has the highest rank). Once a party completes the committee gather protocol, it starts the protocol for the committee AVSS recover. By starting the recover after gather, we guarantee that the binding core choices are independent of the nonfaulty party ranks. Finally, each party waits to recover the ranks of all parties in its gather set, and outputs the proposal sent by the party with the highest rank.

Essentially, this protocol adapts the PE protocol of [3] to the setting with a committee-based quorum system using our committee AVSS and committee broadcast.

8.1 Definition

Definition 8.1. A Proposal Election scheme consists of two protocols PE and PE_Verify. Every party i calls PE with an input x_i and a tag tag, and may eventually output a value y_i and a proof π_i. Parties may call the PE_Verify protocol with an input x and a proof π, and either eventually output 1, eventually output 0 or never output anything from the protocol. Parties also have access to an external validity function valid that outputs either 1 or 0 on possible outputs, with the assumption that valid$(\perp) = 0$. The protocol is modeled as having a write-once register x^* that can hold a possible output from the protocol. A Proposal Election scheme has the following properties assuming all nonfaulty parties call PE with externally valid inputs and a tag tag:

- α-**Binding.** For any adversary strategy, with probability α or greater, by the time the first nonfaulty party completes its call to PE, x^* is set to an input of a party that was nonfaulty at the time it called the PE protocol.
- **Termination of Output.** All nonfaulty parties eventually output a pair (y, π).
- **Completeness.** For any two nonfaulty parties i, j, the output (y_j, π_j) of party j from PE will eventually be verified by party i, i.e. i eventually outputs 1 from PE_Verify(y_j, π_j, tag).
- **Agreement on Verification.** For any two nonfaulty parties i, j, and any value x and proof π, if i outputs $b \in \{0, 1\}$ from PE_Verify(x, π, tag) and j calls PE_Verify(x, π, tag), then it eventually outputs b as well.
- **Binding Verification.** If $x^* \neq \perp$ and i outputs 1 from PE_Verify(x, π, tag), then $x = x^*$.
- **External Validity.** If a nonfaulty i outputs 1 from PE_Verify(x, π, tag), then valid$(x) = 1$.

8.2 Construction

The scheme consists of the PE protocol, described in Algorithm 10, and the PE_Verify protocol, described in Algorithm 12. Parties start the protocol by generating a random rank using a VRF, along with proof that the rank was correctly computed. Parties *simultaneously* broadcast their proposals and share their ranks and proofs of correct computation. After receiving many (at least h) such broadcasts and seeing their respective secret shares complete, parties call the Gather protocol to output sets of parties who completed the first phase with a large intersection. After completing the Gather, parties reconstruct each party's rank, and choose the proposal of the party with the highest valid rank. In the good event, that the maximal rank among all parties in the tag committee belongs to a nonfaulty party, all parties output that party's proposal. Considering the size of the committee and the number of nonfaulty parties in the core, this event happens with probability $> \frac{1}{3}$.

Parties can prove that they computed the correct proposal by providing the gather-set they used. To verify the output, parties simply check that the gather set is correct (and thus includes the core) and that the proposal with the maximal associated rank was chosen. In the good event described above, every verifying output must be the same proposal provided by a nonfaulty party, as every verifying gather set includes the core, and thus that proposal's rank will be seen as maximal.

Technical notes. In line 5 we call Share and Broadcast "at the same time" and with the same tag, tag. Formally we mean that we run just one committee broadcast with tag tag that *concatenates* the content of the committee broadcast of the share protocol (see line 5 in algorithm 6) and the committee broadcast in line 5 to one. This is done to ensure that members of the tag committee speak only once, and thus, the weak-adaptive adversary cannot corrupt the parties after they send one of the messages but not the other. This can be resolved in other, slightly less natural, ways. For example, parties can include their proposal in the shared values, even though the proposal is actually public information. However, this would be less efficient for large proposals. Another option is defining a single "share-and-broadcast" primitive, allowing parties to send both public and private information, having both properties of a secret sharing scheme for the private information and the properties of a broadcast scheme for the public information.

8.3 Security Analysis

In Section 7.3, we only showed the security of the Gather protocol under the assumption that there are at most m indices $j \in [n]$ for which parties might output 1 from the validity predicate Gather_validity. In the following lemma, we show that this assumption does indeed hold.

Lemma 8.2. *For any tag tag, there are at most m indices $j \in [n]$ for which any nonfaulty party ever outputs* 1 *from* Gather_validity(j, tag).

Algorithm 10 PE(prop_i, tag)

$\text{proposals}_i \leftarrow \emptyset, \text{ranks}_i \leftarrow \emptyset, X_i \leftarrow \emptyset$
// Set up the AVSS scheme for this session
call Setup(tag)
// Generate a rank, share it, and broadcast your proposal simultaneously, *i.e. add* prop_i *to the broadcasted information in* Share
upon Setup(tag) terminating, **do**
  $r_i, \pi_i \leftarrow \text{VRF\_evaluate}(\text{sk}_i, (\text{“rank”}, tag))$
  call Share($(r_i, \pi_i), tag$) and Broadcast(prop_i, tag) *at the same time*
// Wait for h completed shares and valid broadcasts and then call Gather
upon outputting j from Share and (j, prop_j) from Broadcast with the tag tag, **do**
  if $\text{valid}(\text{prop}_i) = 1$ **then**
    $\text{proposals}_i \leftarrow \text{proposals}_i \cup \{(j, \text{prop}_j)\}$
    if $\left|\text{proposals}_i\right| = h$ **then**
      call Gather($\{k | (k, \text{prop}_k) \in \text{proposals}_i\}, tag$) with predicate Gather_validity
// Start reconstructing all ranks after completing Gather *and store them*
upon outputting X from Gather with the tag tag, **do**
  $X_i \leftarrow X$
  call Rec(tag)
upon outputting (j, r_j, π_j) from Rec(tag), **do**
  $\text{ranks}_i \leftarrow \text{ranks}_i \cup \{(j, r_j, \pi_j)\}$ ▷ If the output is not of this form, add it to ranks_i, but consider the proof as invalid below
// After reconstructing all ranks for X_i, output the proposal with the maximal valid rank
upon $X_i \neq \emptyset$ and having $(j, \text{prop}_j) \in \text{proposals}_i, (j, r_j, \pi_j) \in \text{ranks}_i$ for every $j \in X_i$, **do**
  $\ell \leftarrow \text{argmax}\{r_k | k \in X_i \; s.t. \; \text{VRF\_verify}(\text{pk}_k, (\text{“rank”}, tag), r_k, \pi_k) = 1\}$
  output prop_ℓ and X_i, but continue updating state

Algorithm 11 Gather_validity(j)

upon $\exists (j, \text{prop}_j) \in \text{proposals}_i$, **do**
  output 1 and **terminate**

Algorithm 12 PE_Verify(prop, π, tag)

upon Gather_Verify(π, tag) outputting 1, **do**
  upon having $(j, \text{prop}_j) \in \text{proposals}_i, (j, r_j, \pi_j) \in \text{ranks}_i$ for every $j \in \pi$, **do**
    $\ell \leftarrow \text{argmax}\{r_k | k \in \pi \; s.t. \; \text{VRF\_verify}(\text{pk}_k, (\text{“rank”}, tag), r_k, \pi_k) = 1\}$
    if $\text{prop} = \text{prop}_\ell$ **then**
      output 1 and **terminate**

Proof. A nonfaulty party i only outputs 1 from Gather_validity(j, tag) if it has a tuple (j, prop_j) in proposals_i. It only adds such a tuple after outputting it from the Broadcast protocol with tag tag. Nonfaulty parties only output such tuples if $\text{get\_mem}(\text{sk}_j, tag) = 1, \pi$, and from the safety of the quorum system, there are at most m such values. □

Theorem 8.3. *The pair* (PE, PE_Verify) *are a Proposal Election scheme with* $\alpha = \frac{1}{3}$.

Proof. Each property is proven individually.

α-Binding. Observe the time the first nonfaulty party i completes the PE protocol. At that time, it had $X_i \neq \emptyset$, meaning that it output X_i from the Gather protocol. From the external validity and completeness of the Gather protocol, for every $j \in X_i$, Gather_validity$(j) = 1$ at that time, and thus i output j from Share and (j, prop_j) from Broadcast and added (j, prop_j) to X_i. From the verifiability of the broadcast protocol get_mem$(\text{sk}_j, tag) = 1$ for for every such j, and from the safety of the quorum system, there are at most $\lceil h/2 \rceil$ faulty parties for which this holds. Note that at the time of calling the Share and Broadcast protocols, no nonfaulty party sent a message using get_mem with this tag, and thus the adversary cannot adaptively attack parties increasing this number. Let $X^* \subseteq [n]$ be the core defined in the binding core property of the Gather protocol. For every party $j \in [n]$, let $r_j, \pi_j = \text{VRF\_evaluate}(\text{sk}_i, (\text{“rank”}, tag))$. If $\ell^* = \text{argmax}\{r_j | j \in [n] \; s.t. \; \text{get\_mem}(\text{sk}_j, tag) = 1, \pi\}$ is the index of a party that was nonfaulty when calling Broadcast and $\ell^* \in X^*$, define $x^* = \text{prop}_{\ell^*}$. Otherwise, define $x^* = \perp$.

Note that $|X^*| \geq h$, and at most $\lceil h/2 \rceil$ of the outputs (j, prop_j) from Broadcast are of parties that were faulty at the time they called Broadcast. Since some nonfaulty party output (j, prop_j) for every $j \in X^*$, at least $h - \lceil h/2 \rceil$ of the indices $j \in X^*$ are of parties that were nonfaulty when calling PE. Furthermore, from the unpredictability of the VRF, every party j such that get_mem$(\text{sk}_j, tag) = 1, \pi$ has the same probability of having the maximal value r_j, and since there are at most m such parties, that probability is at least $\frac{1}{m}$. Since the outputs of the VRF are unpredictable, and no party reconstructed the shared r_j, π_j value, the adversary does not know the values r_j of nonfaulty parties at the time the first nonfaulty party completes the Gather protocol, and thus the indices of nonfaulty parties included of X^* are independent of the r_j values. This means that the probability that ℓ^* is the index of a party that was nonfaulty when calling Broadcast (and thus also when calling PE) and that $\ell^* \in X^*$ is at least $\frac{h - \lceil h/2 \rceil}{m} \geq \frac{1}{3}$ from the safety of the quorum system.

Termination of Output. Assume all nonfaulty parties call PE with a tag tag and externally valid inputs. Every nonfaulty party starts by calling Setup(tag), and from the Liveness property of the AVSS scheme, eventually completes the call. Every nonfaulty i then computes a rank and proof $r_i, \pi = \text{VRF\_evaluate}(\text{sk}_i, (\text{“rank”}, tag))$, and calls Share$((r_i, \pi), tag)$ and Broadcast(prop_i, tag). From the Liveness of the protocols, every nonfaulty party outputs j from Share and (j, prop'_j) for every party j that was nonfaulty when calling the protocol such that get_mem$(\text{sk}_j, tag) = 1, \pi$. From liveness of the quorum system, there are at least h such parties. Every such nonfaulty j inputs prop_j to the Broadcast protocol and from the Validity of the Broadcast protocol, every nonfaulty party outputs $\text{prop}'_j = \text{prop}_j$. After outputting these tuples, every nonfaulty i sees that valid$(\text{prop}_j) = 1$ for every such nonfaulty j, and add (j, prop_j) to proposals$_i$. After adding h such tuples to proposals$_i$, every nonfaulty i calls the Gather protocol with the set of indices $S_i = \{k | (k, \text{prop}_k) \in \text{proposals}_i\}$. Note that at that time there clearly is a tuple $(k, \text{prop}_k) \in \text{proposals}_i$ for every $k \in S_i$ and thus party Gather_validity$(k) = 1$ for i, as required. From the Termination of Output of the Gather protocol, every nonfaulty i eventually outputs a set X from Gather, stores it in X_i and calls Rec(tag). From the Liveness of the Rec protocol, every nonfaulty i outputs a tuple (j, r_j, π_j) for every j that it output from Share(tag) and adds it too ranks$_i$. In addition, from the External Validity and completeness properties of the Gather protocol, Gather_validity$(j, tag) = 1$ for every $j \in X_i$ at that time and thus there exists a tuple $(j, \text{prop}_j) \in \text{proposals}_i$ for every such j. Since this holds, i performs a local computation and outputs a value from PE, as required.

Completeness. Assume some nonfaulty party i output prop, π from the PE protocol with tag tag and that a nonfaulty j calls Gather_Verify(prop, π, tag). At that time, i $X_i \neq \perp$ and had a tuple $(j, \text{prop}_j) \in \text{proposals}_i$ and $(j, r_j, \pi_j) \in \text{ranks}_i$ for every $j \in X_i$. It then computed $\ell =$

$\text{argmax}\{r_k | k \in \pi \; s.t. \text{VRF\_verify}(\text{pk}_k, (\text{“rank”}, tag), r_k, \pi_k) = 1\}$, and output $\text{prop} = \text{prop}_\ell$ and $\pi = X_i$. Note that party j only calls PE_Verify after completing the PE protocol, and it thus had $X_i \neq \emptyset$ after outputting it from Gather, and called Rec and Broadcast with the tag tag. From the Completeness property of the Gather protocol, j eventually outputs 1 from $\text{Gather\_Verify}(\pi, tag)$. From the Liveness and Correctness properties of the Broadcast and AVSS scheme, j eventually adds the same tuples to its proposals_j and ranks_j sets. Following that, j performs the same local computation and outputs 1.

Agreement on Verification. Assume some nonfaulty party i called $\text{PE\_Verify}(x, \pi, tag)$ and output $b \in \{0, 1\}$ and that some nonfaulty j called $\text{PE\_Verify}(x, \pi, tag)$. First, note that nonfaulty parties only output 1 from PE_Verify. Before outputting 1 from PE_Verify, i had output 1 from $\text{Gather\_Verify}(\pi, tag)$, had tuples $(k, \text{prop}_k) \in \text{proposals}_i$ and $(k, r_k, \pi_k) \in \text{ranks}_i$ for every $k \in \pi$ and saw that $\text{prop} = \text{prop}_\ell$ for $\ell = \text{argmax}\{r_k | k \in \pi \; s.t. \text{VRF\_verify}(\text{pk}_k, (\text{“rank”}, tag), r_k, \pi_k) = 1\}$. Every tuple (k, prop_k) is added to proposals_i after outputting if from the Broadcast protocol with the tag tag and every tuple (k, r_k, π_k) is added to ranks_i after outputting it from the Rec protocol the tag tag. Similarly to the proof of completeness, party j calls PE_Verify after completing the PE protocol, and it thus completed its call to Gather, and called Rec and Broadcast with the tag tag. From the Agreement on Verification property of the Gather protocol, j eventually outputs 1 from $\text{Gather\_Verify}(\pi, tag)$ as well. From the Liveness and Correctness properties of the Broadcast and AVSS scheme, j eventually adds the same tuples to its proposals_j and ranks_j sets. Following that, j performs the same local computation and outputs 1.

Binding Verification. Assume some nonfaulty party completed the PE protocol and define x^* as it is defined in the α-Binding property. If $x^* = \perp$, the claim trivially holds. Otherwise, x^* was defined to be prop_{ℓ^*} for a party ℓ^* that was nonfaulty when calling Share and Rec, such that ℓ^* is in the binding core X^* of the Gather protocol and $\ell^* = \text{argmax}\{r_j | j \in [n] \; s.t. \; \text{get\_mem}(\text{sk}_j, tag) = 1, \pi\}$ where $r_j, \pi_j = \text{VRF\_evaluate}(\text{sk}_j, (\text{“rank”}, tag))$ for every $j \in [n]$. If a nonfaulty i that outputs 1 from $\text{PE\_Verify}(x, \pi, tag)$, it first saw that $\text{Gather\_Verify}(\pi, tag)$ output 1, and had a tuple $(j, \text{prop}_j) \in \text{proposals}_i$ and a tuple $(j, r_j, \pi_j) \in \text{ranks}_i$ for every $j \in \pi$. Following that, it computed $\ell = \text{argmax}\{r_k | k \in \pi \; s.t. \; \text{VRF\_verify}(\text{pk}_k, (\text{“rank”}, tag), r_k, \pi_k) = 1\}$, saw that $x = \text{prop}_\ell$ and output 1. From the Includes Core property of the Gather protocol, $\ell^* \in X^* \subseteq \pi$. Since ℓ^* was nonfaulty when calling Share and Broadcast, from the Correctness of the AVSS scheme and from the Validity of the Broadcast protocol, i output $(\ell^*, \text{prop}_{\ell^*})$ from Broadcast and $(\ell^*, r_{\ell^*}, \pi_{\ell^*})$ from Rec. From the correctness of the VRF scheme, $\text{VRF\_verify}(\text{pk}_{\ell^*}, (\text{“rank”}, tag), r_{\ell^*}, \pi_{\ell^*})$, and thus ℓ^* was considered as one of the possible indices ℓ. In addition, from the verifiability of the Broadcast protocol, for every $(j, \text{prop}_j) \in \text{proposals}_i$, $\text{get\_mem}(\text{sk}_j, tag) = 1$ since i output this tuple from Broadcast. Only tuples $(j, r_j, \pi_j) \in \text{ranks}_i$ such that $\text{VRF\_verify}(\text{pk}_j, (\text{“rank”}, tag), r_j, \pi_j) = 1$ were considered when computing $\ell = \text{argmax}\{r_k | k \in \pi \; s.t. \; \text{VRF\_verify}(\text{pk}_k, (\text{“rank”}, tag), r_k, \pi_k) = 1\}$, and from the binding of the VRF scheme, $r_j, \pi = \text{VRF\_evaluate}(\text{sk}_i, (\text{“rank”}, tag))$ for every such tuple. Seeing as ℓ^* is defined as having the maximal r_{ℓ^*}, and as the VRF verification succeeds for ℓ^*, $\ell^* = \ell$. Therefore, $x = \text{prop}_\ell = \text{prop}_{\ell^*}$, as required.

External Validity. Assume some nonfaulty i outputs 1 from $\text{PE\_Verify}(\text{prop}, \pi, tag)$. Before doing so, it saw that it had $(j, \text{prop}_j) \in \text{proposals}_i$ for every $j \in \pi$ and output prop_ℓ for some $\ell \in \pi$. Nonfaulty parties only add a tuple (j, prop_j) to proposals after outputting that tuple from Broadcast and seeing that $\text{valid}(\text{prop}_j) = 1$, and thus $\text{valid}(\text{prop}) = \text{valid}(\text{prop}_\ell) = 1$. □

8.4 Efficiency Analysis

Lemma 8.4. *The* PE *protocol has a total communication complexity of* $O(\lambda n m^3)$ *and the* PE_Verify *protocol requires no communication, where* m *is the parameter described in the quorum system and a*

round complexity of $O(1)$. *In addition, if a nonfaulty party outputs* x *from* PE *or* 1 *from* PE_Verify *run on* x *with proof* π, *every nonfaulty party does so after* $O(1)$ *rounds.*

Proof. In the PE protocol, parties call the Setup, Share and Rec protocols for a total communication complexity of $O(\lambda nm^3)$. In addition, parties call the Gather protocol with a complexity of $O((\lambda + m \log n)nm^2)$. Since $\lambda > \log n$, the total complexity is $O(\lambda nm^3)$. The PE_Verify consists of a call to Gather_Verify and local computation, both of which require no communication.

Parties complete the call to Setup in $O(1)$ rounds, and output j and (j, r_j) from Share and Broadcast respectively for every nonfaulty j in the *tag* committee in $O(1)$ rounds. There are h such parties, and thus parties call the Gather protocol in $O(1)$ rounds. Parties call the Gather protocol with the asynchronous validity predicate Gather_validity. If some nonfaulty party output 1 when calling Gather_validity(j), then it had output j from Share and (j, prop) from Broadcast. Every other nonfaulty party outputs the same values in $O(1)$ time, after which Gather_validity(j) also outputs 1. In other words, the conditions in the efficiency analysis of Gather hold, and thus all parties complete the Gather in $O(1)$ time. Following that, nonfaulty parties call Rec and output a tuple (j, r_j, π_j) in $O(1)$ time for every j that was previously output from Share. This holds for every $j \in X$ from the external validity of the Gather protocol, and thus parties complete the protocol in $O(1)$ time.

Now assume some nonfaulty party either output x from PE or 1 from PE_Verify run on x with proof π. In that case, that party output 1 from Gather_Verify when run on π, and had tuples (j, prop_j) and (j, r_j, π_j) in its proposals and ranks sets for every $j \in \pi$ after outputting them from Broadcast and Rec. Every nonfaulty party outputs 1 from Gather_Verify when run on π in $O(1)$ time. In addition, every nonfaulty party outputs the same values (j, prop_j) and (j, r_j, π_j) from Broadcast and Rec in $O(1)$ time, after which it also outputs 1. □

Assuming $f < (1/3 - \epsilon)n$ and instantiating Broadcast with the quorum system protocol described in Section 4, we get $m = O(\epsilon^{-2}\kappa \log n)$, and thus the communication complexity of the PE protocol is $O(\lambda n(\epsilon^{-2}\kappa \log n)^3)$.

9 NO-WAITIN' ALGORAND

Here we describe No-Waitin' Algorand (NWA), which is a validated asynchronous Byzantine agreement protocol that uses the proposal election protocol as its liveness mechanism.

At a high level, No-Waitin' Algorand (NWA) is a view-based protocol. Each view consists of 4 rounds. In the first round, parties call the PE protocol to choose a proposal. Roughly speaking, the protocol then has two paths: a good path and an error path. In the good path, parties send "echo" messages in round 1, then "key" messages in round 2, then "lock" messages in round 3, and finally "commit" messages in round 4.

Progression in the good path from round i to round $i + 1$ typically requires h round i message that are not error messages (see Algorithms 19-21). If there is any error message, then we switch to the error path and propagate this error messages through rounds 1 to 4.

Safety mechanism: Parties maintain the highest lock they saw for safety (Algorithm 13, line 2). The proposal that is output from the PE contains a key certificate. The core safety check is done in round 1 where parties check that the view of the proposed key is at least as large as the view of their lock (Algorithm 15, line 8). To maintain safety, parties update their lock if a valid one with a higher view is received (Algorithm 16, line 27).

Liveness mechanism inside a view: To maintain liveness in a view, nonfaulty parties in round i always send a message (either a good message or an error message) and parties in round $i + 1$ wait for h round i messages.

Liveness mechanism between views: To maintain liveness between views, parties update their key if a valid one with higher view is received (Algorithm 16, line 7).

Constant expected number of views: If there no commit message is sent in views $< k$, the protocol has a constant probability to reach the commit round in view k with no errors. This is because inside the proposal election, with constant probability, the proposal of a nonfaulty party is unanimously elected. In that case all parties output this proposal from PE, and no other value verifies, meaning that no correct "equivocate" message can be sent. Since this proposer is nonfaulty, it chooses a value with the highest key certificate it saw. This will guarantee that the core safety check will succeed for all nonfaulty parties, and thus no correct "blame" can be sent. Once parties reach the end of the view with no detected errors, they will commit and terminate.

On lack of explicit view change: Note that all parties update their highest key and lock certificates in rounds 3 and 4 respectively. So parties that complete a view must have essentially "read from a quorum". Essentially, this means that parties essentially exchange the information traditionally sent during a view change during the view itself.

The structure of Algorithm 15-16: in each round is logically separated into two parts. The first part is done by all parties and does not include sending any messages, but does include any updates to locks or keys. The second part is done by a dedicated committee and includes sending a message. The fact that every party updates its locks and keys, regardless of whether they were committee members in this view, guarantees that they will act correctly if they are chosen as committee members in any subsequent view.

9.1 Definition

Definition 9.1. In a Validated Asynchronous Byzantine Agreement protocol, every party i has an input x_i, and may output a value y_i. Parties also have oracle access to an external validity function valid that receives a value x as input and either outputs 1, indicating that it is externally valid, or 0, indicating that it is not. A protocol is said to be a Validated Asynchronous Byzantine Agreement protocol if it has the following properties:

- **Agreement**: All nonfaulty parties that complete the protocol output the same value.
- **Validity**: If a nonfaulty party outputs a value, it is externally valid.
- α – **Quality**: With probability at least α, all parties output the input x_i of the same party i that was nonfaulty when it started the protocol.
- **Termination**: All nonfaulty parties almost-surely (i.e. with probability 1) complete the protocol and output a value.

9.2 Construction

The NWA protocol is a view-based protocol, where each view is an attempt by all parties to pick a leader and reach agreement. Parties start each view by calling PE, inputting their proposed values, and each outputting the value of a random leader. Parties then proceed in four rounds. If they see no errors, the rounds proceed as follows:

- In the first round, parties send "echo" messages, echoing the value they received from PE, along with proofs that they are correct outputs.
- In the second round, parties send "key" messages, containing proof that they received quorum of "echo" messages with the same value. This guarantees non-equivocation of the "key" messages by quorum intersection.
- In the third round, parties send "lock" messages, containing proof that they received a quorum of "key" messages with the same value. This guarantees that if some party sent a "lock" message, it received "key" messages from a quorum of "key" committee members.

Any party waits to hear from a quorum of "key" committee members will hear from at least one nonfaulty party in the intersection.
Crucially, when parties hear such "key" messages, they update their keys for future use. Intuitively, these keys will be used to "open any lock" set by nonfaulty parties, to be explained in the next bullet.

- In the fourth round, parties send "commit" messages, containing proof that they received a quorum of "lock" messages. This guarantees that if some party sent a "commit" message, it received "lock" messages from a quorum of "lock" committee members. Any party waits to hear from a quorum of "lock" committee members will hear from at least one nonfaulty party in the intersection.
  Similarly to above, when parties hear such "key" messages, they update their keys for future use. A party with a lock generated in a given view will not be willing to echo any value from an older view. Therefore, before committing to a value, parties guarantee that everybody will be able to update their locks, forcing the adversary to values from this view (or later). This means that seeing a correct "commit" messages allows parties to output the associated value without worrying that other parties might output different values. However, setting such locks might lead to liveness issues: parties might refuse to listen to an honest leader. As described in the previous bullet, before setting a lock, parties make sure that every party will be able to form a key, which means that they will propose a message from this view or later, along with a proof.

In order to guarantee termination, parties also run a termination gadget, in which members of a "termination" committee send every "commit" message they receive, either directly as a round four message, or indirectly from other "termination" committee members. This boosting process guarantees that parties can terminate after hearing $\lceil h/2 \rceil + 1$ such messages, because then every committee member will hear this many messages and also forward them.

Parties may detect two types of error in any view: old keys or conflicting outputs from the PE. If parties see they received an old key, they send a "blame" message, including the key and the lock it had at the time. If parties see two conflicting outputs from the PE, they send an "equivocate" message, including the two conflicting values. Both of these messages indicate that the view failed: either by allowing a faulty leader to suggest a value (no nonfaulty leader would suggest an old key) or by the PE failing (if it succeeds, it would have only one verifying value). If parties see such a message, they propagate it throughout the rounds, replacing the "correct" round messages, i.e. the "echo", "key", "lock", or "commit" messages. Parties always wait to hear a quorum of messages in each round before proceeding, even if they see errors, making sure that any information sent by a quorum in the "no error" path of the protocol. Finally, in the end of the view, parties check if they saw errors in round 4, and proceed to the next view if they do.

Algorithm 13 NWA(val_i)

```
1: echo_view_i ← 0,   echo_val_i ←⊥,   π_i^echo ← ∅
2: key_view_i ← 0,    key_val_i ← val_i,   π_i^key ← ∅
3: lock_view_i ← 0,   lock_val_i ←⊥,   π_i^lock ← ∅
4: view_i ← 1
5: continually run Termination_Gadget()
6: while party i does not terminate do
7:     delay any message with view > view_i
8:     call process_view(view_i)
```

Algorithm 14 Termination_Gadget()

```
1: term_count_i ← 0
2: upon receiving the first ⟨"round 4", "commit", m, view⟩ from j, do
3:     if verify_mem(pk_j, 1, π_j, ("commit", view)) = 1 then
4:         if message_correct(⊥, ("commit", m), view, ⊥) = 1 then
5:             val_member, π_i ← get_mem(sk_i, "termination")
6:             if val_member = 1 and i didn't send a "termination" message yet then
7:                 sends ⟨"termination", m, view, π_i⟩ to all parties
8: upon receiving the first⟨"termination", m, view⟩ from j, do
9:     if verify_mem(pk_j, 1, π_j, ("termination")) = 1 then
10:        if message_correct(⊥, ("commit", m), view, ⊥) = 1 then
11:            val_member, π_i ← get_mem(sk_i, "termination")
12:            if val_member = 1 and i didn't send a "termination" message yet then
13:                sends ⟨"termination", m, view, π_i⟩ to all parties
14:            term_count_i ← term_count + 1
15:            if term_count_i = ⌈h/2⌉ + 1 then
16:                output commit_val and terminate
```

The echo_correct checks that an echo message is valid relative to the PE validity.

The key_correct checks that a key certificate is valid by checking it has h valid echo messages with the same value from valid members.

The lock_correct checks that a lock certificate is valid by checking it has h valid key certificates with the same value from valid members.

The commit_correct checks that a commit certificate is valid by checking it has h valid lock certificates with the same value from valid members.

10 SECURITY ANALYSIS

This lemma says that if there is a commit certificate in view k then and key certificate from this view and onwards must be with the same value.

Lemma 10.1. *If there exits a view k, value v, proof π for which* commit_correct$(k, p, \pi) = 1$ *then there cannot be a key with value $v' \neq v$, view $k' \geq k$, and proof π' such that* key_correct$(\mathsf{pk}', k', v', \pi', \sigma') = 1$ *for some* pk', σ'.

Proof. In this proof we will use the shorthand terms key certificate, lock certificate, commit certificate to indicate the existence of a proof of the key, lock, commit that returns 1 on the key_correct, lock_correct, commit_correct, respectively.

We prove a stronger statement by induction on k': if the conditions of the lemma hold, then any nonfaulty member that ends view $\geq k$ will have a lock certificate for value v and view $\geq k$.

For $k' = k$: Suppose by contradiction that there exist two keys with valid proofs with different values in the same view k. From the quorum system robust quorum intersection property, at least one nonfaulty party's signature was used two π^{key} values, meaning that a nonfaulty party has sent two echo messages with different values in the same view. This is a contradiction.

For the second part, observe that from the quorum system honest majority property, there are at least $\lceil h/2 \rceil + 1$ nonfaulty members that send a lock certificate for this value in view k. So from the quorum intersection of the quorum system, all nonfaulty parties wait for h round 3 messages, will see at least one lock certificate and will update their lock. This concludes the base case.

Algorithm 15 process_view($view$) //part 1

1: $round_i \leftarrow 1$
2: delay every message with $round > round_i$
3: r1_msgs $\leftarrow \emptyset$, r2_msgs $\leftarrow \emptyset$, r3_msgs $\leftarrow \emptyset$, r4_msgs
// round 1: echo round
4: **call** PE((key_view$_i$, key_val$_i$, π_i^{key}), $view$)
5: **upon** outputting $(k, kv, \pi^{key}), \pi^{\text{PE}}$ from PE with tag $view$, **do**
6: is_r1_mem, $\pi_i \leftarrow$ get_mem(sk$_i$, ("round 1", $view$))
7: **if** is_r1_mem = 1 **then** ▷ send round 1 messages if in committee
8: **if** $view > k \geq$ lock_view$_i$ **then**
9: $\sigma \leftarrow$ Sign(sk$_i$, (("echo", $view$), kv))
10: send ⟨"round 1", "echo", $(k, kv, \pi^{key}), \pi^{\text{PE}}, \sigma, \pi_i, view$⟩ to all parties
11: **else** send ⟨"round 1", "blame", $(k, kv, \pi), \pi^{\text{PE}}$, (lock_view$_i$, lock_val$_i$, π_i^{lock}), $\pi_i, view$⟩
12: $round_i \leftarrow round_i + 1$
// round 2: key round
13: **upon** receiving the first ⟨"round 1", $m, \pi_j, view$⟩ message from j, **do**
14: **if** verify_mem(pk$_j$, 1, π_j, ("round 2", $view$)) = 1 **then**
15: **if** message_correct(pk$_j$, m, ("round 1", $view$)) = 1 **then**
16: **if** m = ("echo", $k, kv, \pi^{key}, \pi^{\text{PE}}\sigma$) **then**
17: **if** echo_view$_i < view$ **then**
18: echo_view$_i \leftarrow view$, echo_val$_i \leftarrow kv, \pi_i^{\text{echo}} \leftarrow (k, kv, \pi^{key}, \pi^{\text{PE}})$
19: **if** echo_val$_i \neq kv$ **then**
20: r1_msgs$_i \leftarrow$ r1_msgs$_i \cup$ {("error", ("equivocate", $((k, kv, \pi^{key}, \pi^{\text{PE}}), \pi_i^{\text{echo}}))$)}
21: **else** r1_msgs$_i \leftarrow$ r1_msgs$_i \cup \{(j, \sigma, \pi_j)\}$
22: **else**
23: r1_msgs$_i \leftarrow$ r1_msgs$_i \cup \{(j, \sigma, \pi_j)\}$
24: **upon** $|$r1_msgs$_i|$ = h, **do** ▷ process round 1 messages, send round 2 messages if in committee
25: is_r2_mem, $\pi_i \leftarrow$ get_mem(sk$_i$, ("round 2", $view$))
26: **if** is_r2_mem = 1 **then**
27: **if** $\exists$("error", m) $\in$ r2_msgs$_i$ **then**
28: send ⟨"round 2", $m, \pi_i, view$⟩ to all parties
29: **else**
30: $\sigma \leftarrow$ Sign(pk$_i$, ("key", $view$), echo_val$_i$))
31: send ⟨"round 2", "key", (echo_view$_i$, echo_val$_i$, r1_msgs), $\sigma, \pi_i, view$⟩ to all parties
32: $round_i \leftarrow round_i + 1$

Suppose this holds for all k'' such that $k \leq k'' < k'$, and let's prove for view k'. From the second part of the induction hypothesis, all nonfaulty members of the round 1 committee of view k' will have a lock for value v with a view of at least k. Hence any proposal with a key certificate with a view less than k will be rejected via blame messages by all nonfaulty parties (and will not obtain the h signatures needed for a key certificate). The only proposal that can be accepted must have a key certificate that is at least of view k. So by the first part of the induction hypothesis, this key certificate must have a value of v, and if this view generates a lock certificate, it must also have the value v as well (because lock certificate values must come from key certificate values). This concludes the proof. □

Algorithm 16 process_view($view$) // part 2

// round 3: lock round
upon receiving the first $\langle$"round 2", $m, \pi_j, view\rangle$ message from j, **do**
  if verify_mem($\mathsf{pk}_j, 1, \pi_j$, ("round 2", $view$)) = 1 **then**
    if message_correct(pk_j, m, ("round 2", $view$)) = 1 **then**
      if m = ("key", k, kv, π, σ) **then**
        $\mathsf{r2\_msgs} \leftarrow \mathsf{r2\_msgs} \cup \{(j, \sigma, \pi_j)\}$
        if $\mathsf{key\_view}_i < k$ **then**
          $\mathsf{key\_view}_i \leftarrow k$, $\mathsf{key\_val}_i \leftarrow kv$, $\pi_i^{\text{key}} \leftarrow \pi$
      else
        $\mathsf{r2\_msgs}_i \leftarrow \mathsf{r2\_msgs}_i \cup \{$("error", m)$\}$
upon $|\mathsf{r2\_msgs}_i| = h$, **do** ▷ process round 2 messages, send round 3 messages if in committee
  $\mathsf{is\_r3\_mem}, \pi_i \leftarrow \mathsf{get\_mem}(\mathsf{sk}_i$, ("round 3", $view$))
  if is_r3_mem = 1 **then**
    if $\exists$("error", m) $\in \mathsf{r2\_msgs}_i$ **then**
      send $\langle$"round 3", $m, \pi_i, view\rangle$ to all parties
    else
      $\sigma \leftarrow \mathsf{Sign}(\mathsf{pk}_i$, ("lock", $view$), lock_val))
      send $\langle$"round 3", "lock", $(\mathsf{key\_view}_i, \mathsf{key\_val}_i, \mathsf{r2\_msgs}_i), \sigma, \pi_i, view\rangle$ to all parties
  $round_i \leftarrow round_i + 1$
// round 4: commit round
upon receiving the first $\langle$"round 3", $m, \pi_j, view\rangle$ message from j, **do**
  if verify_mem($\mathsf{pk}_j, 1, \pi_j$, ("round 3", $view$)) = 1 **then**
    if message_correct(pk_j, m, ("round 3", $view$)) = 1 **then**
      if m = ("lock", l, lv, π, σ) **then**
        $\mathsf{r3\_msgs}_i \leftarrow \mathsf{r3\_msgs}_i \cup \{(j, \sigma, \pi_j)\}$
        if $\mathsf{lock\_view}_i < l$ **then**
          $\mathsf{lock\_view}_i \leftarrow l$, $\mathsf{lock\_val}_i \leftarrow lv$, $\pi_i^{\text{lock}} \leftarrow \pi$
      else
        $\mathsf{r3\_msgs} \leftarrow \mathsf{r3\_msgs}_i \cup \{$("error", m)$\}$
upon $|\mathsf{r3\_msgs}| = h$, **do** ▷ process round 3 messages, send round 4 messages if in committee
  $\mathsf{is\_r3\_mem}, \pi_i \leftarrow \mathsf{get\_mem}(\mathsf{sk}_i$, ("round 3", $view$))
  if is_r3_mem = 1 **then**
    if $\exists$("error", m, π, j) $\in$ "round 3" **then**
      send $\langle$"round 4", $m, \pi_i, view\rangle$ to all parties
      $view_i \leftarrow view_i + 1, round_i \leftarrow 1$
    else
      $\sigma \leftarrow \mathsf{Sign}(\mathsf{pk}_i$, ("commit", $view$), commit_val))
      send $\langle$"round 4", "commit", (lock_view, lock_val, r3_msgs), $\sigma, \pi_i, view\rangle$ to all parties
upon receiving the first $\langle$"round 4", $error\_tag, m, \pi_j, view\rangle$ message from j, **do**
  if verify_mem($\mathsf{pk}_j, 1, \pi_j$, ("round 4", $view$)) = 1 **then**
    if message_correct(pk_j, ($error\_tag, m$), ("round 4", $view$)) = 1 **then**
      $view_i \leftarrow view_i + 1$

Algorithm 17 message_correct(pk, $m, view$)

if $m =$ ("echo", pe_output, σ) **then**
  upon echo_correct(pk, $pe\_output, \sigma, view$)) outputting 1, **do** ▷ this check is interactive
    output 1
if $m =$ ("key", (key_view, key_val, π^{key}), σ) **then**
  output key_correct(pk, key_view, key_val, π^{key}, σ)
if $m =$ ("lock", (lock_view, lock_val, π^{lock}), σ) **then**
  output lock_correct(pk, lock_view, lock_val, $\pi^{\text{lock}}, \sigma$)
if $m =$ ("commit", (commit_view, commit_val, π^{commit}), σ) **then**
  output commit_correct(pk, commit_view, commit_val, $\pi^{\text{commit}}, \sigma$)
if $m =$ ("blame", pe_output, (lock_view, lock_val, π^{lock})) **then**
  upon blame_correct(pe_output, lock_view, lock_val, π^{lock}) outputting 1, **do** ▷ this check is interactive
    output 1
if $m =$ ("equivocate", pe_output_1, pe_output_2) **then**
  upon equivocate_correct($pe\_output_1, pe\_output_2, view$) outputting 1, **do** ▷ this check is interactive
    output 1
output 0

Algorithm 18 echo_correct(pk, $k, kv, \pi^{key}, \pi^{\text{PE}}\sigma, view$)

if Sign_verify(pk, (("echo", $view$), val), σ) = 1 **then**
  upon PE_Verify((k, kv, π^{key}), $\pi^{\text{PE}}, view$) outputting 1, **do**
    output 1
output 0

Algorithm 19 key_correct(pk, key_view, key_val, π^{key}, σ)

if Sign_verify(pk, (("key", key_view), key_val), σ) = 1 **then**
  if $|\pi^{\text{key}}| \geq h$ and $\forall(j, \sigma_j, \pi_j) \in \pi^{\text{key}}$ verify_mem(pk_j, 1, π_j, ("round 1", key_view)) = 1 $\wedge$ Sign_verify(pk_j, (("echo", key_view), key_val), σ) = 1 **then**
    output 1
output 0

Algorithm 20 lock_correct(pk, lock_view, lock_val, $\pi^{\text{lock}}, \sigma$)

if Sign_verify(pk, (("lock", lock_view), lock_val), σ) = 1 **then**
  if $|\pi^{\text{lock}}| \geq h$ and $\forall(j, \sigma_j, \pi_j) \in \pi^{\text{lock}}$ verify_mem(pk_k, 1, π_k, ("round 2", lock_view)) = 1 $\wedge$ Sign_verify(pk_j, (("key", lock_view), lock_val), σ_j) = 1 **then**
    output 1
output 0

Algorithm 21 commit_correct(commit_view, commit_val, π^{commit})

1: **if** $|\pi^{\text{commit}}| \geq h$ and $\forall (j, \sigma_j, \pi_j) \in \pi^{\text{commit}}$ verify_mem$(\text{pk}_j, 1, \pi_j, (\text{“round 3”}, view)) = 1 \quad \wedge$ Sign_verify$(\text{pk}_j, ((\text{“lock”}, \text{commit_view}), \text{commit_val}), \sigma_j) = 1$ **then**
2: **output** 1
3: **output** 0

Algorithm 22 blame_correct(pe_output, lock_view, lock_val, π^{lock})

1: **if** $view \leq \text{key\_view} \vee \text{key\_view} < \text{lock\_view}$ **then**
2: **if** $|\pi^{\text{lock}}| \geq h$ and $\forall (j, \sigma_j, \pi_j) \in \pi^{\text{lock}}$ verify_mem$(\text{pk}_k, 1, \pi_k, (\text{“round 2”}, \text{lock_view})) = 1 \wedge$ Sign_verify$(\text{pk}_j, ((\text{“key”}, \text{lock_view}), \text{lock_val}), \sigma_j) = 1$ **then**
3: **upon** PE_Verify$(pe_output, view)$ terminating with output 1, **do**
4: **output** 1
5: **output** 0

Algorithm 23 equivocate_correct($\text{key\_view}_1, \text{key\_val}_1, \pi_1^{\text{key}}, \pi_1^{\text{PE}}, \text{key\_view}_2, \text{key\_val}_2, \pi_2^{\text{key}}, \pi_2^{\text{PE}}$, $view$)

1: **if** $((\text{key\_view}_1, \text{key\_val}_1, \pi_1^{\text{key}}) \neq (\text{key\_view}_2, \text{key\_val}_2, \pi_2^{\text{key}}))$ **then**
2: **upon** outputting 1 from PE_Verify$((\text{key_view}_1, \text{key_val}_1, \pi_1^{\text{key}}), \pi_1, view)$ and PE_Verify$((\text{key_view}_2, \text{key_val}_2, \pi_2^{\text{key}}), \pi_2, view)$, **do**
3: **output** 1
4: **output** 0

The next lemma simple states that nonfaulty parties' messages are correct.

Lemma 10.2. *If a nonfaulty party i sends a message $\langle \text{“round”}, m, \pi, view \rangle$, then* message_correct$(\text{pk}_i, m, (\text{“round”}, view)) = 1$.

Proof. We will show that every message sent by a nonfaulty party i satisfies message_correct.

A nonfaulty party signs each message it sends using its secret key sk_i. By the correctness of the signature scheme, Sign_verify(pk_i, π, m) always returns true for messages signed by a nonfaulty party. If i sends an "echo" message, it does so using the output of PE. By the Completeness property of PE, the PE_Verify algorithm eventually terminates with output 1, and message_correct returns true for this message. If i sends a "key" message, it does so after receiving $h$correct "echo" messages from distinct parties in the "round 1" committee, both containing signatures on the same value. Therefore, message_correct returns true for the "key" message. The same argument applies to "lock" and "commit" messages, with respect to "key", "commit", and committees "round 2" and "round 3". If i sends a "blame" message in "round 1", it uses the output of PE along with its lock field. By the Completeness property of PE, PE_Verify eventually outputs 1, and i updates its key/lock fields according to valid messages. Therefore, message_correct returns true for the "blame" message. If i sends an "equivocate" message in "round 2", it does so based on two correct "echo" messages with conflicting values. By the Completeness property of PE, PE_Verify eventually terminates with output 1 for both messages, and thus message_correct returns true for the "equivocate" message. These messages are forwarded messages that i has already validated. The "termination" messages are also forwarded "commit" messages. Finally, if i forwards a message m from party j in round

("round", $view$), it only does so if message_correct(pk_j, m, ("round" $-1, view$)) $= 1$. Therefore, any forwarded message also satisfies message_correct.

□

In the next lemmas, we prove statements about the liveness of the protocol. We start by defining what it means for parties to "reach" or "be in" a view.

Definition 10.3. A nonfaulty party is said to reach view k if at any point its local field $view_i$ is updated to k. Similarly, a nonfaulty party is said to be in view k if its local field $view_i$ equals k at that time.

We then prove in that parties terminate if they reach the end of a view with no errors, or if some other party terminates. In addition, we prove that parties don't get stuck in a view, that is, they either proceed to the next view or terminate, and that if the "good event" of the PE protocol takes place, parties don't proceed to the next view. Combining these two statements, since parties don't proceed to the next view if the "good event" takes place, they must terminate instead.

Lemma 10.4. *If one nonfaulty party in the "commit" committee for $view$ sends a "commit" message, all nonfaulty parties eventually terminate. In addition, if some nonfaulty party completes the protocol, every nonfaulty party eventually does.*

Proof. Suppose a nonfaulty party in the "commit" committee for $view$ has sent a commit certificate in a "commit" message. From Lemma 10.2, the commit certificate is correct. Every nonfaulty party receives the "commit" message, and all the nonfaulty parties in the (global) "termination" committee will eventually send a "termination" message with that commit certificate to all the other parties. From the liveness of the quorum system, there are at least $h \geq \lceil h/2 \rceil + 1$ nonfaulty members in the "termination" committee, and thus every party terminates. Furthermore, no nonfaulty party in "termination" committee terminates before sending termination message, since they terminate after receiving several "termination" messages, but send a message after receiving at least one such message. In addition, if some nonfaulty party completes the protocol, it received $\lceil h/2 \rceil + 1$ "termination" messages from members of the "termination" committee. From the safety of the quorum system, there are fewer than $\lceil h/2 \rceil$ faulty senders in the committee, and thus at least one of these messages was sent by a nonfaulty sender. Every party receives this message, including all members of the "termination" committee, who then send a "termination" message if they haven't already done so. As before, at least $\lceil h/2 \rceil + 1$ members of the "termination" committee send such messages, meaning every party terminates. □

Lemma 10.5. *If in view $view$, the binding value x^*, as defined in the α − Biding property of the* PE*, is the input of some party that was nonfaulty when calling* PE*, then no nonfaulty party reaches view $view + 1$.*

Proof. Suppose the binding value is indeed the input of some party that acted in a nonfaulty manner. From the Binding Verification property and Completeness property of PE, every nonfaulty party outputs $(\mathsf{key\_view}_i, \mathsf{key\_val}_i, \pi_i^{\mathsf{key}})$ with a proof π^{PE} for the same nonfaulty party i, and $(\mathsf{key\_view}_i, \mathsf{key\_val}_i, \pi_i^{\mathsf{key}})$ is the only tuple such that for a nonfaulty party j outputs 1 from PE_Verify with the tag $view$.

We will start by proving that no nonfaulty party can receive a correct blame message. Suppose by way of contradiction that there is a party j with a correct lock certificate such that $\mathsf{lock\_view}_j > \mathsf{key\_view}_i$ and $\mathsf{lock\_val}_j \neq \mathsf{key\_val}_i$. Then at the lock round of $\mathsf{lock\_view}_j$, party j has received h correct keys from members of the "round 2" committee of view $\mathsf{lock\_view}_j$. From the robust quorum intersection property of the quorum system, all nonfaulty parties receive "key" messages

from at least $\lceil h/2 \rceil$ of the parties above. From the safety of the quorum system, at least one of these parties is nonfaulty. After receiving such a message, every nonfaulty party updates its key_view to lock_view_j. Since every nonfaulty party inputs its key fields PE, and key_view only increases, we can conclude that $\text{key\_view}_i \geq \text{lock\_view}_j$, by contradiction. Then, no party can send a correct blame message, concluding that no nonfaulty party can read a correct blame message. Furthermore, PE_Verify only outputs 1 for $(\text{key\_view}_i, \text{key\_val}_i, \pi_i^{\text{key}})$, so no party can send a correct "equivocate" message with two different verifying tuples. Since a nonfaulty party reaches view $view + 1$ only after reading at least one correct error message in round 4, we can conclude that no nonfaulty party reaches view $view + 1$. □

Lemma 10.6. *If all nonfaulty parties reach view and no nonfaulty terminates in view, then all nonfaulty parties reach view* + 1.

Proof. Assume all nonfaulty parties reach $view$, and observe the rounds of $view$. For round 1, all nonfaulty parties participate in the protocol and from termination property of the PE algorithm output some value. Suppose that all nonfaulty parties have reached round k (for k in $\{1, 2, 3\}$) and let's prove that all of them reach round $k + 1$. From the liveness of the quorum system, there are at least h nonfaulty parties in the round k committee who send "round k" messages. From Lemma 10.2, every nonfaulty party will eventually receive h correct messages from those nonfaulty committee members and continue to the next round. Similarly, since all parties reach round 4, all nonfaulty parties in the round 4 committee send a "round 4" message. Assume by way of contradiction that no nonfaulty party terminates during $view$, and that some nonfaulty party i did not proceed to the next $view$. All nonfaulty parties eventually receive the "round 4" messages sent by the nonfaulty committee members, including the nonfaulty i that stayed in $view$. Since it did not proceed, the messages sent were not error messages, but "commit" messages. All nonfaulty parties eventually hear these messages as well, including the nonfaulty members of the "termination" committee. These parties then send "termination" messages, and from the liveness of the quorum system, there are at least $h > \lceil h/2 \rceil$ such parties. After receiving these messages, i terminates, while still in view $view$, by contradiction. □

Theorem 10.7. *The* NWA *protocol is a Validated Asynchronous Byzantine Agreement protocol with* $\alpha = \frac{1}{3}$*, conditioned on no errors.*

Proof. Each property is proven individually.

Agreement. Assume, for the sake of contradiction, that two nonfaulty parties i and j decide on different values v and v', respectively, where $v \neq v'$. So there must be two commit certificates, with values v and v' and views k, k'. Assume without loss of generality that $k \leq k'$. If there is a commit certificate for v' with view k', then there must be a key certificate for v' with view k'. We can now apply lemma Lemma 10.1 to obtain a contradiction.

Validity. If some nonfaulty party i outputs a value v, it first receives a verifying "commit" message. This message contains at least h signatures provided in "lock" messages, with at least one of them provided by a nonfaulty party. That nonfaulty party saw at least h signatures provided in "key" messages before sending a "lock" message. Similarly, at least one of these parties was nonfaulty and sent the message after receiving at least h signatures provided in "echo" messages. Finally, one of these messages was sent by a nonfaulty party who first checked that PE_Verify output 1 on a tuple including that value v. From the external validity of PE, $\text{valid}(v) = 1$.

α − Quality. With probability $\frac{1}{3}$, the binding event of the PE protocol takes place in the first view. In that case, no party proceeds to the second view from Lemma 10.6, and as shown in the termination property, all parties terminate in this view. In this view, only the input x^* provided by a nonfaulty party can be verified in PE_Verify, and thus all verifying "echo" messages must contain

x^*. This means that nonfaulty parties only send "echo" messages with that input. Every verifying "key" message must contain at least one nonfaulty party's signature on its value, and thus only "key" messages whose value is x^* will verify. Following similar logic, only "lock" and "commit" messages whose value is x^* will verify. Finally, before terminating, parties receive "termination" messages with a correct "commit" message, and output x^*. In other words, all nonfaulty parties output the input of a nonfaulty party with probability $\alpha = \frac{1}{3}$ or greater.

Termination. From Lemma 10.4, if a nonfaulty party terminates, eventually all nonfaulty parties do. Therefore, it is enough to show that at least one nonfaulty party completes the protocol. In each view, there is a probability of at least $\frac{1}{3}$ that the binding event of the PE takes place, in which case no nonfaulty party continues to the next view. Since there is an independent $\frac{1}{3}$ probability of the binding event taking place in each view, the probability that the event does not take place decreases exponentially with the number of views, approaching 0 as the number of views grows. In other words, the probability that the binding event takes place in some view is 1. Let $view$ be the minimal view for which this happens. If some party terminates before reaching $view$, we are done. Otherwise, using Lemma 10.6 $view - 1$ times, all parties reach $view$. Then, seeing as no nonfaulty party proceeds to $view + 1$, from Lemma 10.6, some nonfaulty party terminated, and thus we are also done. □

10.1 Efficiency Analysis

Lemma 10.8. *The* NWA *protocol has an expected communication complexity of* $O(\lambda nm^3)$ *and* $O(1)$ *round complexity, where* m *is the parameter described in the quorum system.*

Proof. As discussed in the termination property, the binding event of PE takes place in each view with an independent probability of $\frac{1}{3}$ or greater. This means that the expected number of views until this takes place is at most 3. Each view consists of a single call to PE and a constant number of rounds in which members of a single committee send a message to all parties. From the safety of the quorum system, there are at most m parties in each such committee. Each such message consists of $O(m)$ cryptographic elements or indices of size $O(\lambda)$. Therefore, parties send $O(\lambda nm^2)$ bits in total in these messages. In addition, the PE protocol has a communication complexity of $O(\lambda nm^3)$, meaning that the total complexity is $O(\lambda nm^3)$. Each view consists of a single call to PE and an additional constant number of rounds. The PE protocol requires a constant number of rounds, so in total the protocol requires a constant number of rounds. □

Assuming $f < (1/3 - \epsilon)n$ and instantiating Broadcast with the quorum system protocol described in Section 4, we get $m = O(\epsilon^{-2}\kappa \log n)$, and thus the communication complexity of the PE protocol is $O(\lambda n(\epsilon^{-2}\kappa \log n)^3)$.

REFERENCES

[1] Ittai Abraham, Gilad Asharov, Arpita Patra, and Gilad Stern. Perfectly secure asynchronous agreement on a core set in constant expected time. *IACR Cryptol. ePrint Arch.*, page 1130, 2023.

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

[3] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 363–373, New York, NY, USA, 2021. Association for Computing Machinery.

[4] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 337–346, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. *IACR Cryptol. ePrint Arch.*, page 16, 1999.

[6] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

[7] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part I*, volume 12550 of *Lecture Notes in Computer Science*, pages 260–290, Durham, NC, USA, November 16–19, 2020.

[8] Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. In Rafael Pass and Krzysztof Pietrzak, editors, *Theory of Cryptography*, pages 353–380, Cham, 2020. Springer International Publishing.

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

[10] Lennart Braun, Ivan Damgård, and Claudio Orlandi. Secure multiparty computation from threshold encryption based on class groups. In *Advances in Cryptology – CRYPTO 2023, Part I*, Lecture Notes in Computer Science, pages 613–645, Santa Barbara, CA, USA, August 2023.

[11] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 524–541, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[12] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 123–132, New York, NY, USA, 2000. Association for Computing Machinery.

[13] Matteo Campanelli, Bernardo David, Hamidreza Khoshakhlagh, Anders Konring, and Jesper Buus Nielsen. Encryption to the future - A paradigm for sending secret messages to future (anonymous) committees. In *Advances in Cryptology – ASIACRYPT 2022, Part III*, Lecture Notes in Computer Science, pages 151–180, December 7–11, 2022.

[14] Ran Canetti, Sebastian Kolby, Divya Ravi, Eduardo Soria-Vazquez, and Sophia Yakoubov. Taming adaptivity in YOSO protocols: The modular way. In *TCC 2023: 21st Theory of Cryptography Conference, Part II*, Lecture Notes in Computer Science, pages 33–62, November 2023.

[15] Ignacio Cascudo and Bernardo David. SCRAPE: scalable randomness attested by public entities. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *Applied Cryptography and Network Security - 15th International Conference, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings*, volume 10355 of *Lecture Notes in Computer Science*, pages 537–556. Springer, 2017.

[16] Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777(C):155–183, July 2019.

[17] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid MPC: Secure multiparty computation with dynamic participants. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 94–123, Virtual Event, August 16–20, 2021.

[18] Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, Anton Paramonov, and Manuel Vidigueira. All byzantine agreement problems are expensive. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, PODC '24, page 157–169, New York, NY, USA, 2024. Association for Computing Machinery.

[19] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th Annual ACM Symposium on Theory of Computing*, pages 364–369, Berkeley, CA, USA, May 28–30, 1986. ACM Press.

[20] Shir Cohen, Idit Keidar, and Alexander Spiegelman. Brief announcement: Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, page 175–177, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a COINcidence: Sub-Quadratic Asynchronous Byzantine Agreement WHP. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[22] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299, Innsbruck, Austria, May 6–10, 2001.

[23] Ivan Bjerre Damgård, Simon Holmgaard Kamp, Julian Loss, and Jesper Buus Nielsen. Asynchronous YOSO a la paillier. Cryptology ePrint Archive, Paper 2025/128, 2025.

[24] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, January 1985.

[25] Luciano Freitas, Petr Kuznetsov, and Andrei Tonkikh. Distributed randomness from approximate agreement, 2022.

[26] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. YOSO: You only speak once - secure MPC with stateless ephemeral roles. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 64–93, Virtual Event, August 16–20, 2021.

[27] Craig Gentry, Shai Halevi, Bernardo Magri, Jesper Buus Nielsen, and Sophia Yakoubov. Random-index PIR and applications. In Kobbi Nissim and Brent Waters, editors, *TCC 2021: 19th Theory of Cryptography Conference, Part III*, volume 13044 of *Lecture Notes in Computer Science*, pages 32–61, Raleigh, NC, USA, November 8–11, 2021.

[28] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Jens Groth and Rafail Ostrovsky. Cryptography in the multi-string model. *Journal of Cryptology*, 27(3):506–543, July 2014.

[30] Zhengan Huang, Junzuo Lai, Shuai Han, Lin Lyu, and Jian Weng. Anonymous public key encryption under corruptions. In *Advances in Cryptology – ASIACRYPT 2022, Part III*, Lecture Notes in Computer Science, pages 423–453, December 7–11, 2022.

[31] Simon Holmgaard Kamp and Jesper Buus Nielsen. Byzantine agreement decomposed: Honest majority asynchronous atomic broadcast from reliable broadcast. Cryptology ePrint Archive, Paper 2023/1738, 2023.

[32] Sebastian Kolby, Divya Ravi, and Sophia Yakoubov. Constant-round YOSO MPC without setup. 1(3):30, 2024.

[33] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science*, pages 120–130, New York, NY, USA, October 17–19, 1999. IEEE Computer Society Press.

[34] Matthieu Rambaud. Adaptively secure consensus with linear complexity and constant round under honest majority in the bare PKI model, and separation bounds from the idealized message-authentication model. Cryptology ePrint Archive, Paper 2023/1757, 2023.

[35] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 148–164. Springer, 1999.

[36] Markus Stadler. Publicly verifiable secret sharing. In Ueli M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 190–199. Springer, 1996.

A IMPLICATIONS FOR SYNCHRONOUS YOSO MPC

The techniques used in this paper have exciting implications for the synchronous setting with a weak adaptive adversary, often referred to as the YOSO setting. Specifically, the same VRF-based role assignment approach resulting in *a committee of committees* works in synchrony. This leads to the first end-to-end synchronous YOSO MPC (excluding constructions based on general-purpose witness-encryption) with sublinear communication assuming *only* a VRF setup with (near) optimal resilience.

In this appendix, we give a high-level overview of our synchronous YOSO MPC construction. Like in almost all YOSO MPC literature, we separate the problem of assigning roles to parties (in such a way that the role can receive secret messages while the party playing it remains anonymous), and using those roles to compute.

Informally, during our role assignment, each nominator N_i randomly chooses *a sub-committee* $H_{i,1}, \ldots, H_{i,n}$, so that we end up with a committee of sub-committees. When N_i is corrupt, we have to assume that all of $H_{i,1}, \ldots, H_{i,n}$ will be as well. When N_i is honest, since $H_{i,1}, \ldots, H_{i,n}$ are chosen randomly, we have the guarantee that a majority of them are honest.

While we do not have an honest majority among all of $\{H_{i,j}\}_{i,j\in[1,\ldots,n]}$, we do have an honest majority *among a majority of subcommittees*. This is enough to build an MPC with guaranteed output delivery: we nest a threshold linear secret sharing scheme in order to obtain a linear secret sharing scheme for our committee-of-committees access structure, and then run MPC on top of that scheme. (Note that, while a majority of a majority of sub-committees is not itself a majority, our result does not contradict the lower bound of Cleve [19] because it uses a Q_2 access structure: it is impossible to have two *disjoint* qualified sets under this access structure.)

Using similar techniques also leads to the first end-to-end asynchronous YOSO MPC with sublinear communication assuming only a VRF setup with (near) optimal resilience.

A.1 Role Assignment

Let n be a number large enough s.t. when n parties are sampled at random, at least half are honest with overwhelming probability. We want to assign roles to which secret messages can be sent, or, in other words, *form a secret-holding committee*. We can do this as follows:

(1) *n nominators* self-elect using a VRF-based Algorand-style cryptographic sortition. As in the work of Benhamouda *et. al* [7], the nominators cannot form a secret-holding committee directly, since someone who might want to send them a secret has no way to do so (without relying on e.g. witness encryption).
(2) Each nominator N_i does the following:
   (a) Chooses *n parties* $H_{i,1}, \ldots, H_{i,j}$ to serve on the secret-holding committee. (This is in contrast to the work of Benhamouda *et. al*, where each nominator chooses only a single party.)
   (b) For each $j \in [1, \ldots, n]$:
      (i) Samples a public key encryption key-pair $(\mathsf{ek}_{i,j}, \mathsf{dk}_{i,j})$.
      (ii) Encrypts $\mathsf{dk}_{i,j}$ to $H_{i,j}$ using key-message non-committing encryption (KMNCE), as done by Canetti *et. al* [14, Section 7]. (KMNCE is necessary for adaptive security. It automatically provides anonymity.) Let $c_{i,j}$ denote the resulting ciphertext.
   (c) Erases all of its secret state.
   (d) Broadcasts $\{(\mathsf{ek}_{i,j}, c_{i,j})\}_{j\in[1,\ldots,n]}$.

This realizes the role assignment functionality $\mathcal{F}_{\mathsf{RA}}$ from Canetti *et. al* [14]. Since the construction is exactly like theirs (except for the fact that each nominator chooses multiple holders), the proof follows as a direct extension of theirs.

$\mathcal{F}_{\text{RA}}$ only provides secure point-to-point communication to roles. In order to run MPC, we often need to verify the validity of private messages sent to roles, and their relationship to one another. (E.g., it could be important to verify that private messages sent to roles comprise a consistent sharing of a secret.) In order to accommodate this, our role assignment protocol can be extended in the natural way to realize the stronger functionality $\mathcal{F}_{\text{VeSPa}}$ (for verifiable state propagation) of Kolby *et. al* [32], which supports the verification of arbitrary statements about messages sent and received. This can be achieved in the classic way of including non-interactive zero-knowledge proofs (NIZKs) with the messages. (Note that it may seem that this requires a CRS; however, we could instead leverage multi-string NIZKs [29], which rely on a *set* of random strings most of which are generated correctly. A single self-electing committee could set these strings up for us.) For the rest of this section, we always use $\mathcal{F}_{\text{VeSPa}}$ to *verifiably* send private messages to roles.

A.2 NSS: Nested Secret Sharing

A key observation is that any *linear* secret sharing scheme, such as Shamir, can be nested. Indeed, this is exploited during degree reduction in BGW, where shares are re-shared.

A.2.1 Building Block: Linear Secret Sharing. Let $\mathsf{LSS} = (\mathsf{Share}, \mathsf{Rec})$ be a linear secret sharing scheme.

$\mathsf{LSS.Share}(x, n, t) \to (s_1, \ldots, s_n)$ produces n shares, and

$\mathsf{LSS.Rec}(\{s_i\}_{i \in R}) \to x'$ uses a qualified subset of those shares (of size at least t) to recover the secret by taking a linear function. Let $\lambda_{(R,i)}$ be the reconstruction coefficient for the i'th share when used with qualified reconstruction set R, so that

$$\sum_{i \in R} \lambda_{(R,i)} s_i = x.$$

(For e.g. additive sharing, the only valid R is $[1, \ldots, n]$, and $\lambda_{(R,i)} = 1$ for every i. For Shamir sharing, $\lambda_{(R,i)}$ are the Lagrange coefficents.)

A.2.2 Building Nested Secret Sharing. We can nest a linear secret sharing scheme LSS, resulting in a secret sharing scheme NSS for an access structure over a committee of n size-n sub-committees, where any set of at least t members of at least t sub-committees is qualified.

For simplicity of notation, we will write R to denote a reconstruction set for NSS (where R contains tuples of the form (i, j)). We will write R_i to denote $\{j\}_{(i,j) \in R}$, and R' to denote $\{i\}_{|R_i| \geq t}$. R is qualified if $|R'| \geq t$.

The reconstruction coefficients for NSS are

$$\lambda^{\mathsf{NSS}}_{R,(i,j)} = \begin{cases} \lambda^{\mathsf{LSS}}_{(R',i)} \lambda^{\mathsf{LSS}}_{(R_i,j)} & \text{if } i \in R', \\ 0 & \text{otherwise.} \end{cases}$$

$\mathsf{NSS.Share}(x, n, t)$**:**

(1) $(s_1, \ldots, s_n) \leftarrow \mathsf{LSS.Share}(x, n, t)$.
(2) For $i \in [1, \ldots, n]$, $(s_{i,1}, \ldots, s_{i,n}) \leftarrow \mathsf{LSS.Share}(s_i, n, t)$.
(3) Return $\{s_{i,j}\}_{i,j \in [1,\ldots,n]}$.

$\mathsf{NSS.Rec}(\{s_{i,j}\}_{(i,j) \in R})$**:** return $\sum_{(i,j) \in R} \lambda^{\mathsf{NSS}}_{R,(i,j)} s_{i,j}$.

Correctness holds since

$$\sum_{(i,j) \in R} \lambda^{\mathsf{NSS}}_{R,(i,j)} s_{i,j} = \sum_{i \in R'} \sum_{j \in R_i} \lambda^{\mathsf{LSS}}_{(R',i)} \lambda^{\mathsf{LSS}}_{(R_i,j)} s_{i,j} = \sum_{i \in R'} \lambda^{\mathsf{LSS}}_{(R',i)} \sum_{j \in R_i} \lambda^{\mathsf{LSS}}_{(R_i,j)} s_{i,j}.$$

By the correctness of the LSS, this equals

$$\sum_{i \in R'} \lambda^{\text{LSS}}_{(R',i)} s_i.$$

Applying correctness of the LSS once again, we get x.

Privacy holds for NSS since if a set R of shares is unqualified, then for more than $n - t$ indices i, privacy of s_i follows by the privacy of the LSS. Applying privacy of the LSS once again, we get privacy of the secret.

A.3 NSS to Committees of Sub-Committees

In order to verifiably NSS-share a secret x to a committee of sub-committees $C_1, \ldots, C_n$, the dealer runs $\{s_{i,j}\}_{i,j \in [1,\ldots,n]} \leftarrow \text{NSS.Share}(x)$, and calls $\mathcal{F}_{\text{VeSPa}}$ to privately and verifiably send each share $s_{i,j}$ to the jth member of C_i.

A.3.1 Passing a Sharing to the Next Committee. Now, say that a committee of sub-committees $C = (C_1, \ldots, C_n)$ holds an NSS-sharing of x, and wishes to pass this sharing to another committee of sub-committees $C' = (C'_1, \ldots, C'_n)$.

(1) Role $H_{i,j}$ on sub-committee C_i with share $s_{i,j}$ simply NSS-shares $s_{i,j}$ to C' as described in Section A.3.1. Let $s_{i,j,k,l}$ be the share sent by $H_{i,j}$ on sub-committee C to $H'_{k,l}$ on sub-committee C'. Let R be the set of indices (i, j) such that role $H_{i,j}$ successfully completed this step.
(2) Role $H'_{k,l}$ on sub-committee C'_k computes share $s'_{k,l} = \sum_{(i,j) \in R} \lambda_{(R,(i,j))} s_{i,j,k,l}$.

Correctness can be argued in much the same way as in Section A.2.2:

$$\sum_{(k,l) \in R'} \lambda_{(R',(k,l))} s'_{k,l} = \sum_{(k,l) \in R'} \lambda_{(R',(k,l))} \Big(\sum_{(i,j) \in R} \lambda_{(R,(i,j))} s_{i,j,k,l} \Big) = \sum_{(i,j) \in R} \lambda_{(R,(i,j))} \Big(\sum_{(k,l) \in R'} \lambda_{(R',(k,l))} s_{i,j,k,l} \Big)$$

By correctness of the NSS, this equals

$$\sum_{(i,j) \in R} \lambda_{(R,(i,j))} s_{i,j}.$$

Applying correctness of the NSS once again, we get x.

A.4 MPC

Now that we have a linear secret sharing scheme NSS for our committee-of-committees access structure, we can use NSS to run an MPC. Broadly speaking, there are two promising approaches to this: BGW [6] and CDN [22].

A.4.1 BGW. One natural thing to do is to use the linearity of NSS to run BGW-style computation [6]. Each input is NSS-shared to the current committee-of-committees. To add two values, the committee members perform addition locally. To multiply two values, several approaches may be used.

One might hope that committee members could multiply their shares locally as well. However, this results in the degrees of both layers of Shamir sharing doubling. Since only half of the outer Shamir shares belong to mostly-honest sub-committees, this is not something we will be able to recover from.

Instead, we follow the template of Kolby *et. al* [32] (which they call *YOSO-LHSS*). First, we use Beaver triples to turn more of the computation linear; then, we use a linearly homomorphic encryption scheme in order to turn the local linear computation into public computation on ciphertexts. Up until now, we assumed that within $\mathcal{F}_{\text{VeSPa}}$, each committee member proved, in

zero knowledge, that its outgoing messages were correctly computed as a function of its incoming messages and some local randomness. By substituting local computation with public computation on ciphertexts, we reduce this burden of proof.

To generate Beaver triples, we leverage additional committees (which can be self-elected and have an dishonest majority), as done by Gentry *et. al* [26]. Each Beaver triple consists of NSS-shared values a, b and c s.t. $c = ab$. The current committee-of-committees gets shares of a and b, and the next committee-of-committees gets shares of a and c. Then, to multiply NSS-shared values x and y, current committee members locally compute shares of $\epsilon = x + a$ and $\delta = y + b$ and broadcast those shares, allowing public reconstruction of ϵ and δ. At the same time, the current committee members pass on y to the next committee, as describe in Section A.3.1. We observe that $\epsilon y - \delta a + c = (x + a)y - (y + b)a + c = xy + ay - ay - ab + ab = xy$; so, the next committee can locally compute shares of $z = xy$ as a linear function of y, a and c.

With the use of Beaver triples, committee members only reshare, open, and perform linear operations on shares. By using linearly homomorphic encryption (e.g. an El-Gamal-style variant of Paillier) for communication to committee members, we enable the homomorphic evaluation of linear combination of shares, relaxing the need for committee members to prove that they performed such linear operations correctly.

A.4.2 CDN. An alternative approach is computing over *threshold* linearly homomorphic encryption, as introduced by Cramer, Damgård and Nielsen [22] and made YOSO by Gentry *et. al* [26]. In these constructions, a single public-secret key pair is associated with the system. Inputs to the computation are encrypted to the public key, and the corresponding secret key is secret-shared. (In our case, the secret key is NSS-shared among a committee-of-committees.) As in the previous approach, addition is done locally, and multiplication is done with the help of Beaver triples (generated by a few self-elected dishonest-majority committees).

The advantage of using the CDN approach is that values encrypted to the system public key are implicitly passed from one committee-of-committees to the next, as the decryption key is re-shared. While the BGW approach requires l re-sharings in order to keep l values "alive", the CDN approach requires only one.

An apparent dis-advantage of using CDN is that the secret key for a linearly homomorphic encryption scheme is an exponent-value in a group of unknown order, and thus needs to be secret shared *over the integers*. The YOSO CDN construction of Gentry *et. al* involved a growth in share size with every re-sharing; however, this was fixed in the recent work of Damgård *et. al* [23].