Asynchronous Agreement on a Core Set in Constant Expected Time and More Efficient Asynchronous VSS and MPC

Ittai Abraham[*] Gilad Ashsarov[†] Arpita Patra[‡] Gilad Stern[§]

February 16, 2024

Abstract

A major challenge of any *asynchronous* MPC protocol is the need to reach an agreement on the set of private inputs to be used as input for the MPC functionality. Ben-Or, Canetti and Goldreich [STOC 93] call this problem *Agreement on a Core Set* (ACS) and solve it by running n parallel instances of asynchronous binary Byzantine agreements. To the best of our knowledge, all results in the perfect security setting used this same paradigm for solving ACS. This leads to a fundamental barrier of expected $\Omega(\log n)$ rounds for any asynchronous MPC protocol (even for constant depth circuits).

We provide a new solution for Agreement on a Core Set that runs in expected $\mathcal{O}(1)$ rounds. Our perfectly secure variant is optimally resilient ($t < n/4$) and requires just $\mathcal{O}(n^4 \log n)$ expected communication complexity. We show a similar result with statistical security for $t < n/3$. Our ACS is based on a new notion of *Asynchronously Validated Asynchronous Byzantine Agreement* (AVABA) and new information-theoretic analogs to techniques used in the authenticated model.

In addition to the above, we also construct a new perfectly secure packed asynchronous verifiable secret sharing (AVSS) protocol with communication complexity of $\mathcal{O}(nX + n^3 \log n)$ for sharing X secrets (of size $\mathcal{O}(\log n)$ bits each). The best prior required $\mathcal{O}(nX + n^4 \log n)$ for X secrets. AVSS is an important building block for our ACS, and for asynchronous MPC. We improve both communication complexity and round complexity in asynchronous MPC when plugging our new ACS and new AVSS.

[*]Intel Labs, `Ittai.abraham@intel.com`

[†]Bar-Ilan University, `Gilad.Asharov@biu.ac.il`

[‡]Indian Institute of Science, `arpita@iisc.ac.in`

[§]Tell Aviv University, `gilad.stern@mail.huji.ac.il`

Contents

1 Introduction

One of the core challenges for MPC protocols in the *asynchronous* setting is that they must reach *agreement* on which private inputs to use as input for the circuit. Ben-Or, Canetti and Goldreich (BCG) [10] call this problem Agreement on a Core Set (ACS). In this paper, we consider protocols with *optimal resilience in the asynchronous model against computationally unbounded adversaries.* From the lower bound of [4, 10, 12], *perfect security* for MPC implies that the number of corruptions in this setting is at most $t < n/4$, so optimal resilience is when $n = 4t+1$. This is in contrast to the optimal resilience of $n = 3t + 1$ in the synchronous perfect security setting and the asynchronous statistical security setting. The seminal result of [10, 15] is the first work to obtain perfect security with optimal resilience in the asynchronous model.

Before proceeding, we refine the problem definition. Our main motivating application for ACS is in asynchronous secure computation. Each party shares its input at the beginning of the protocol using asynchronous verifiable secret sharing (AVSS). When a dealer is honest, then all honest parties will eventually receive valid shares. If the dealer is corrupted and one honest party terminates the AVSS successfully, then all honest parties will eventually also receive valid shares. The parties then wish to agree on a common core set of $n-t$ parties whose AVSS has been successfully completed or will eventually terminate. Clearly, each honest party can just wait for $n-t$ AVSS instances of the honest parties to be completed. However, if more than $n-t$ AVSS instances have been completed, the honest parties cannot be sure that all honest parties agree on the same set, with which instances to proceed, and reaching an agreement is crucial for the sequel of the secure protocol. Using an ACS protocol, parties agree on some set of $n-t$ parties ("core") whose AVSS has been terminated or will eventually be terminated for all parties. The difficulty is that due to asynchrony, some of the inputs of honest parties (which instances terminated) might arrive dynamically, and the corrupted parties might input identities of instances that would never terminate.

In terms of round complexity, the best one can hope for is reaching agreement in constant expectation [23]. However, to the best of our knowledge, all results in the asynchronous information-theoretic setting run $\mathcal{O}(n)$ parallel asynchronous binary Byzantine agreement instances to agree on a core set. Composing n parallel agreement protocols, where each protocol runs in constant expected time, means that the expectation of the maximum is $\Omega(\log n)$. So for over 30 years, the best expected round complexity for asynchronous MPC has $\Omega(\log n)$ overhead (even for constant depth circuits)[1]. A natural question remained open:

> *Is there an asynchronous MPC with* **constant** *expected running time overhead? Or is there an inherent* $\Omega(\log n)$ *lower bound for ACS due to asynchrony?*

1.1 Our Contributions

Our **main** contributions are (1) a novel protocol for agreement on a core set in constant expected time via a new multi-valued agreement protocol with an *asynchronous validity predicate*; (2) Efficiency improvements in the communication complexity of asynchronous verifiable secret sharing. Our new ACS and AVSS together significantly improve the communication complexity and round complexity of asynchronous MPC.

Asynchronously Validated Asynchronous Byzantine Agreement (AVABA). We achieve ACS via AVABA. Our AVABA protocol is perfectly secure and resilient to $t < n/4$ corruptions.

[1]Some papers have claimed that other papers obtained better results, see Section 1.2.

For inputs of size $\mathcal{O}(n \log n)$ bits, it runs in $\mathcal{O}(1)$ expected time and requires $\mathcal{O}(n^4 \log n)$ expected communication complexity. Parties are guaranteed to reach an agreement on an input of one of the parties, and the value is guaranteed to pass an asynchronous validity predicate. In the MPC setting, this predicate checks that the input contains $n - t$ parties who verifiably completed the input-sharing phase. To the best of our knowledge, the most efficient agreement protocols [9, 22] with constant expected rounds and $t < n/4$ currently require $\mathcal{O}(n^6 \log n)$ bits to be sent in expectation. Furthermore, these protocols are binary agreement protocols. Our protocol improves the efficiency of those protocols and allows for multi-valued agreement.

Theorem 1.1 (Asynchronously Validated Asynchronous Byzantine Agreement (informal))**.** *There exists a perfectly secure protocol for asynchronous Byzantine agreement with an asynchronous validity predicate (AVABA) that is resilient to $t < n/4$ Byzantine corruptions. Each party has a valid input of size $\mathcal{O}(n \log n)$ bits. The protocol runs in constant expected time and $\mathcal{O}(n^4 \log n)$ expected communication complexity.*

Agreement on a Core Set (ACS). Using this AVABA protocol and an asynchronous validity predicate checking which parties shared their inputs, we implement a perfectly secure, $t < n/4$ resilient constant expected time protocol for Agreement on a Core Set (ACS) with an expected $\mathcal{O}(n^4 \log n)$ communication complexity. To the best of our knowledge, this is the first time ACS is solved via a multi-valued agreement in the information-theoretic setting without using any binary agreement building blocks. See Section 1.2 for more details.

Corollary 1.2. *There exists a perfectly secure protocol for asynchronous agreement on a core set (ACS) with an asynchronous validity predicate resilient to $t < n/4$ Byzantine corruptions. The protocol runs in constant expected time and $\mathcal{O}(n^4 \log n)$ expected communication complexity.*

As we elaborate in the related work section, the communication cost of ACS from [15] is $\mathcal{O}(n^7 \log n)$.

Extensions for $t < n/3$ corruptions and statistical security. More generally, our AVABA protocol requires $\mathcal{O}(n)$ secrets (each of size $\mathcal{O}(\log n)$ bits) to be shared per party per round and can be generalized to a protocol resilient to $t < n/3$ corruptions. Our ACS protocol uses packed AVSS to generate randomness.

As proven in [4, 12], when $n < 4t$, any AVSS protocol must have some non-zero probability of non-termination. The work of [16] constructs such an AVSS protocol with an adjustable security parameter ϵ, allowing the protocol to fail or not terminate with ϵ probability. It is possible to use such an AVSS protocol in our construction, resulting in an AVABA protocol with a similar probability of non-termination, as described in the following:

Theorem 1.3 (General Asynchronously Validated Asynchronous Byzantine Agreement (informal))**.** *Let $c \in [3, 4]$. Given a $n > ct$ resilient protocol for asynchronous verifiable secret sharing that has $S(n, \epsilon)$ communication complexity, $\epsilon \geq 0$ error, and $1 - \epsilon$ probability of termination, there exists an agreement protocol that is resilient to t corruptions as long as $n > ct$. Moreover, the protocol is $\tilde{\mathcal{O}}(\epsilon)$ secure (and in particular for $\epsilon = 0$ is perfectly-secure). With probability $1 - \tilde{\mathcal{O}}(\epsilon)$, the protocol runs in constant expected time (and in particular for $\epsilon = 0$ is almost-surely terminating) and has $\mathcal{O}(n^3 \log n + n^2 S(n, \epsilon))$ expected communication complexity.*

In the above theorem, setting $c = 3$, we get the first statistical ACS protocol for any $n > 3t$ parties that terminates in constant expected time, conditioned upon the success of the protocol.

Corollary 1.4. *There exists a statistically secure protocol for asynchronous agreement on a core set with an asynchronous validity predicate resilient to $t < n/3$ Byzantine corruptions. Conditioned upon the protocol succeeding with probability $1 - \epsilon$, the protocol runs in constant expected time.*

Asynchronous verifiable secret sharing. Our second main contribution is a new asynchronous verifiable secret sharing (AVSS):

Theorem 1.5. *There exists a perfectly secure protocol for asynchronous verifiable secret sharing resilient to $t < n/4$ Byzantine corruptions. For sharing X secrets (of $\mathcal{O}(\log n)$ bits each), the total communication complexity is $\mathcal{O}(nX + n^3 \log n)$.*

This means that we get $\mathcal{O}(n)$ overhead for sharing $\Omega(n^2)$ secrets. Prior to our work, the AVSS protocol of [10] achieves total communication complexity of $\mathcal{O}(n^4 \log n)$ for sharing *one* secret, and the one by [28, 18] which obtains total communication complexity of $\mathcal{O}(nX + n^4 \log n)$ bits for X secrets. That is, for obtaining $\mathcal{O}(n)$ overhead the dealer has to share $\Omega(n^3)$ secrets. In our ACS protocol, the dealer has to share just $\mathcal{O}(n)$ secrets, in which case our AVSS requires $\mathcal{O}(n^3 \log n)$ as opposed to $\mathcal{O}(n^4 \log n)$ by [28, 18], improving the communication by a factor of $\mathcal{O}(n)$.

Conclusion: MPC. When plugging our new AVSS and new ACS in the recent asynchronous MPC protocol of [3], we obtain the following corollary:

Corollary 1.6. *For a circuit with C multiplication gates and depth D, there exists a perfectly secure, optimally-resilient asynchronous MPC protocol with $\mathcal{O}((Cn + Dn^2 + n^4) \log n)$ communication complexity and $\mathcal{O}(D)$ expected run-time.*

Without our work, when combining the protocol of [3] with the ACS protocol of [15], together with the AVSS protocol of [28, 18], the cost of the entire MPC is $\mathcal{O}((Cn + Dn^2 + n^7) \log n)$ and $\mathcal{O}(D + \log n)$ expected-time. For a more detailed sketch of the MPC construction, see Section 9.

Synchronous vs. asynchronous MPC. We conclude this section by noting that the currently most efficient perfect *synchronous* MPC [2] achieves the exact complexity as ours: $\mathcal{O}((Cn + Dn^2 + n^4) \log n)$ with $\mathcal{O}(D)$ expected rounds.[2] This poses a fundamental question regarding the relationship between asynchronous and synchronous computation:

> *Is there a generic way to transform synchronous MPC protocol with optimal resilience to an asynchronous MPC protocol with optimal resilience with the same communication and round complexities?*

Our work shows that the (current) state of the art is the same in both models, suggesting that there might be an interesting connection. We remark that the synchronous and asynchronous protocols are inherently different, have different structures, and are based on different techniques. Yet, since the end result achieves the same complexities, perhaps there is a more generic, cleaner way to move from one model to another while preserving the same complexities.

1.2 Related Work

Agreement on a core set via n parallel binary agreements. In the asynchronous setting, an MPC protocol cannot wait for input from all parties. One important task of any MPC protocol in

[2]A somewhat incomparable result [25] removes the $\mathcal{O}(Dn^2)$ in the communication complexity: $\mathcal{O}((Cn + n^5) \log n)$ communication with $\mathcal{O}(D + n)$ expected rounds.

the asynchronous setting is to reach *agreement* on the set of parties whose private input is used as the input for the MPC circuit. To solve this, Ben-Or, Canetti and Goldreich [10] suggest a protocol called Agreement on a Core Set (ACS). To the best of our knowledge, all previous asynchronous MPC protocols (in the perfect and statistical security setting) use the same ACS protocol suggested by [10]. This ACS is based on running n parallel binary agreements. Roughly speaking, parties enter input 1 to the ith binary agreement when they see that party i has completed secret sharing its input and enter all remaining instances with the value 0 once they see at least $n-t$ agreements terminate with a decision of 1.

On the positive side, this elegant solution requires just simple binary agreement as a building block. On the negative side, each binary agreement instance has an independent constant probability of terminating in each round, so in isolation, each instance has a constant expectation. However, the expectation of the maximum of n such independent instances is $\Omega(\log n)$. Therefore this approach of running separate binary instances seems to have a natural barrier for obtaining $\mathcal{O}(1)$ expected round complexity. Lastly, the best known binary agreement protocols [9, 22] in this setting require the expected total of $\mathcal{O}(n^7 \log n)$ for the ACS.

On Ben-Or and El-Yaniv's work. The work of [10] claims that a result of Ben-Or and El-Yaniv solves ACS in constant expected rounds. Here we explain why this is not the case. The work by Ben-Or and El-Yaniv [11] (published 10 years after [10] cites them) deals with executing n concurrent instances of Byzantine Agreement. The first part of [11] is for the synchronous model and we believe it can be used to agree on a common subset in synchrony. The second part claims that these techniques can be extended to solve some variant of multi-sender agreement in constant expected rounds in the asynchronous model.

The situation for the asynchronous model is different. First, we note that [11] explicitly do **not** mention that they can solve ACS in the asynchronous model. Indeed, the stated results of [11] and the techniques of [11] do **not** provide a way to solve ACS (as needed for asynchronous MPC) in constant expected rounds. They only solve an easier problem in which the input of each party exists at the beginning of the protocol (unlike ACS, where due to asynchrony some of the inputs may arrive dynamically over time).

The work of Ben-Or, Kelmer and Rabin [12]. In [12]'s ACS protocol, parties first invoke the BA instances with input 1 for parties who are deemed valid according to a validity condition (in the case of MPC, dealers whose VSS instances have been completed). Parties input 0 to the remaining instances only after seeing $n-t$ instances with output 1. Trying to naively apply the techniques of [11] does not work because they require starting **all** BA instances at the same time and **synchronizing** them using Select (the Select protocol in round r waits for all $n\log(n)$ BA instances to reach round $r+1$) . It is possible that a less naive approach may work, synchronizing some of the BA instances using Select, and then initiating the rest. This seems to require a much more subtle approach since parties are required to wait for the agreed output for each party to be 1 before proceeding (while dealing with $\log(n)$ BA instances per party), and possibly using Select several times.

A possible alternative approach is having each party set all inputs to the BA instances at once, after seeing that at least $n-t$ of those inputs are 1. Using this approach, it is possible that no party has the unanimous support of all honest parties, meaning that each party has at least one honest party input 0 to its BA instance. In this case, parties can output 0 in all instances and thus output an empty set as the agreed core. Even protocols that strengthen the validity conditions are likely to fail because it is possible that most BA instances have many 0 and 1 inputs, resulting in small cores (for example, of size $t+1$ as opposed to $n-t$). We believe that obtaining a less naive protocol could potentially be an interesting follow-up work.

Recent and concurrent work. Two recent and concurrent works deal with tasks related to information-theoretic agreement on a core set with constant round complexity. Duan *et al.* [21] claims to have an information-theoretic construction of an ACS in constant expected rounds. However, their construction uses cryptographic hash functions and a threshold PRF, instantiated by a threshold signature scheme. So while their work solves ACS with constant round complexity it is not perfectly secure or statistically secure (it does not obtain $\mathcal{O}(1)$ expected rounds against an unbounded adversary). Moreover, our core consensus protocol obtains the same efficiency while requiring significantly weaker primitives. Our core consensus protocol obtains the same asymptotic $\mathcal{O}(n^3 \log n)$ bit complexity but requires just a weak leader election (which can be implemented information theoretically via AVSS, as we show). On the other hand, [21] requires a strong leader election, which, to the best of our knowledge, requires a DKG and a computationally bounded adversary for the required complexity.

Cohen *et al.* [19] construct a constant expected round protocol for a linear number of binary Byzantine agreement protocols (where all inputs arrive at the beginning of the protocol). The first version of [19] offhandedly remarks that this primitive can be used to construct a constant round ACS protocol. However, as stated in our discussion of Ben-Or and El-Yaniv's work, these protocols require parties to know their inputs to all instances of binary agreement at the same time. The currently known reductions from binary agreement to ACS in the *asynchronus* setting rely on this not being the case. On a later version, [19] fixed this issue, but their ACS protocol requires at least $\Omega(n^5 \log n)$ and is statistically secure.

2 Technical Overview

We provide a high level overview of our main techniques.

The model. Before we start, let us first introduce the model. We assume asynchronous communication, which means that the adversary can arbitrarily delay messages sent between honest parties, while it cannot necessarily see their content. Messages between honest parties can be delayed but must be delivered eventually. Honest parties, therefore, cannot distinguish between the case where a message (from a corrupted party to an honest party) has never been sent or whether a message (from an honest party to an honest party) is delayed. Thus, protocols must make sure that parties do terminate and parties cannot wait to receive messages from all parties. Parties can wait to receive messages from more than $n-t$ parties only when they are certain that not all messages from honest parties have been received.

Packed Asynchronous Verifiable Secret Sharing. Our packed AVSS protocol uses ideas from the packed AVSS of [18, 28] but reduces the overall communication from $\mathcal{O}(nX + n^4 \log n)$ to $\mathcal{O}(nX + n^3 \log n)$ for sharing X secrets. Looking ahead, in the ACS protocol, the dealer has to share $\mathcal{O}(n)$ secrets, and so our protocol reduces the cost from $\mathcal{O}(n^4 \log n)$ to $\mathcal{O}(n^3 \log n)$.

The base line is the original protocol of [10] that provides $\mathcal{O}(n^4 \log n)$ overhead for sharing a single secret. The protocol requires $\mathcal{O}(n^2 \log n)$ communication over the point-to-point channel in addition to $\mathcal{O}(n^2)$ (short) messages being broadcasted, resulting in an overall of $\mathcal{O}(n^4 \log n)$ when implementing the broadcast over point-to-point. This protocol relies on standard techniques for VSS where the dealer distributes shares on a bivariate polynomial of degree at most t in X and Y. The protocol of [18, 28] improves [10] using the following two aspects:

- **Packing:** The dealer can distribute a polynomial of degree-$2t$ in X and degree at most t in Y. This enables embedding $t + 1$ (*i.e.*, $\mathcal{O}(n)$) secrets in the polynomial, instead of just one;
- **Batching:** The main bottleneck in the protocol is the broadcast messages as there is some asymmetry between the communication over point-to-point ($\mathcal{O}(n^2 \log n)$) and those that are a result of implementing the broadcast ($\mathcal{O}(n^4 \log n)$). By batching $\mathcal{O}(n^2)$ instances simultaneously, i.e., running $\mathcal{O}(n^2)$ instances in parallel but using the same broadcasted messages across all instances, we balance between the two costs.

Overall, the above allows sharing $\mathcal{O}(n^3)$ secrets with the cost of $\mathcal{O}(n^4 \log n)$, and in general, $\mathcal{O}(nx + n^4 \log n)$ for X secrets. We improve this by optimizing further:

- **No broadcast:** We completely eliminate all broadcast messages in the protocol.

Specifically, we completely replace the broadcast with a multi-cast. This introduces some difficulty and makes verifying the correctness of the sharing more challenging. However, the fact that we do not have to pay the extra $\mathcal{O}(n^2)$ for any broadcasted message (and instead pay an overhead of $\mathcal{O}(n)$ overhead for multi-cast) reduces the basic communication complexity to $\mathcal{O}(n^3)$. We refer the reader to Section 4 for a more detailed discussion.

Agreement on a Core Set. We recall the problem statement: In a secure computation protocol, parties first secret-share their inputs using an asynchronous verifiable secret-sharing scheme (AVSS). When the dealer of the AVSS is honest, all parties will eventually terminate. If the dealer is corrupted and one honest party accepts its share, then all honest parties will eventually accept its share. The goal of the ACS is to reach an agreement on a core set of $n - t$ dealers that distributed valid shares, while the parties might receive confirmation for some dealers as the ACS proceeds, and while corrupted parties try to deceive the honest parties and might input identities of dealers whose sharing protocol never terminates.

We take a conceptually new approach to solving ACS in constant expected round complexity, which takes advantage of recent advances in agreement protocols in the authenticated setting (assuming a PKI setup). Roughly speaking, we provide a new protocol that can be viewed as an information-theoretic analog of these advances.

Instead of separating ACS into n parallel binary agreement, we run a single multi-valued instance, where the input of each party is a set of parties that completed their input sharing. The main challenge is that corrupt parties can suggest incorrect inputs. In the authenticated case (computational settings), this is overcome using a Validated Asynchronous Byzantine Agreement which uses an authenticated external validity function. Here we introduce a new information-theoretic *asynchronous validity predicate* (AVP). When a party P_j calls $\mathsf{validate}_j(i)$, then the predicate might eventually terminate with the output 1, or never terminate. If $\mathsf{validate}_j(i)$ terminates with an output 1 for some honest party P_j, then $\mathsf{validate}_k(i)$ will eventually terminate for every honest party P_k. Moreover, if P_i is honest, then $\mathsf{validate}_j(i)$ always terminates. Looking ahead, this predicate comes to model which instances of AVSS terminate in the MPC protocol.

Given this asynchronous validity predicate, we provide a new type of multi-valued agreement in the perfect security setting, which we call *asynchronously validated asynchronous Byzantine agreement* (AVABA).

Implementing an AVABA protocol. The construction of an AVABA protocol follows the ideas and construction of the No Waitin' HotStuff (NWH) protocol of [5]. Seeing as the NWH protocol is designed in the authenticated setting and uses a signature scheme, this work adapts these ideas to the information-theoretic setting, removing the need for cryptography. Roughly speaking, our AVABA protocol has two parts, each with its separate challenges.

Verifiable leader election. The first challenge is constructing a verifiable leader election protocol. In ordinary leader election, the goal is that all parties would agree on the identity of an honest leader with some constant probability. In our setting, the chosen leader might be validated by some (asynchronous) external validity predicate. Our protocol is inspired by the synchronous leader election protocol of [26], its efficiency improvements [1], and the asynchronous authenticated proposal election protocol in the computational setting of [5].

The main idea of leader election is to assign to each party a random value c_i, and then pick as the leader the party with the maximal random value. Each party cannot assign a random value to itself, as corrupted parties will not choose their values uniformly at random. Instead, each party P_j contributes a value $c_{j \to i}$ to each P_i, and we define (for now) $c_i = \sum_{j=1}^{n} c_{j \to i}$. We call each $c_{j \to i}$ as the contribution of P_j to P_i. We cannot just let parties contribute random values, as the corrupted parties will wait to see the values that the honest parties contributed and then pick their own values so that a corrupted leader will be elected. Instead, the parties first "commit" to the values and later "reveal" them. The commitment is performed using verifiable secret sharing.

We borrow ideas from [1, 5] and instead of having $\mathcal{O}(n^2)$ AVSS instances (i.e., $c_{i \to j}$ for every i, j), we use $\mathcal{O}(n)$ "packed" AVSS in which each dealer can share $\mathcal{O}(n)$ secrets at once. As mentioned, we improve the cost of packed AVSS by a factor of $\mathcal{O}(n)$, leading to a more efficient VLE.

Verifiable party gather. Since the model is asynchronous, the above protocol suffers from a similar problem to our starting point: how can the parties know and agree on which AVSS instances terminated successfully and can be considered as contributions? Parties do not know whether to wait until a particular AVSS instance terminates, as it might never terminate; on the other hand, agreeing on which AVSS instances were terminated is exactly the ACS problem!

Luckily, we do not have to reach an agreement fully. We avoid strong agreement using two tools. First, we let each party P_i choose a set of $t+1$ dealers that have successfully shared secrets. The value c_i of P_i is defined to be the sum of their secrets. Since it is a sum of $t+1$ parties, it must include at least one honest dealer, which means that c_i is uniformly distributed. Parties then broadcast their choice of dealers, and wait to receive at least $n-t$ such broadcasts.

However, if some broadcasts are delayed, we again run into a similar problem to ACS: different parties might not consider the same set of parties as potential leaders, and as such parties might not agree on the chosen leader. The parties have to agree on which broadcasts to consider. We now employ our second tool to "roughly agree" on which broadcasts were received: the verifiable party gather protocol. Verifiable party gather is a relaxation in which the parties might output distinct sets, say $C_1, \ldots, C_n$, but with the following two guarantees: (1) All parties in all sets have been validated by at least one honest party (which means that eventually, they will all be validated); (2) The different sets are distinct, but are all supersets of some large "core".

Since all of the c_i values are uniform, each party has the same probability of having the maximal value. If we are lucky and the party with the maximal random value is an honest party in the shared core, all parties will see its c_i value and elect it as a leader. Luckily, since the core is large, there are many honest parties in the core, and this event happens with a large probability. The core is of size $n-t$ in our case, and thus it contains $n-2t$ honest parties, which yields a success probability of $\frac{n-2t}{n} \geq \frac{1}{2}$.

Our verifiable party gather protocol is inspired by [5]. Unlike [5], which relies on signatures and authentication, we implement an information theoretic version of gather whose inputs comply with

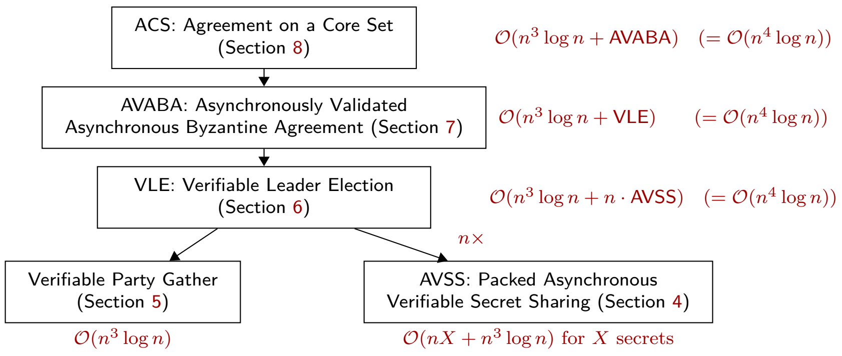
ACS: Agreement on a Core Set (Section 8)
$\mathcal{O}(n^3 \log n + \text{AVABA})$ $(= \mathcal{O}(n^4 \log n))$
AVABA: Asynchronously Validated Asynchronous Byzantine Agreement (Section 7)
$\mathcal{O}(n^3 \log n + \text{VLE})$ $(= \mathcal{O}(n^4 \log n))$
VLE: Verifiable Leader Election (Section 6)
$\mathcal{O}(n^3 \log n + n \cdot \text{AVSS})$ $(= \mathcal{O}(n^4 \log n))$
$n\times$
Verifiable Party Gather (Section 5)
$\mathcal{O}(n^3 \log n)$
AVSS: Packed Asynchronous Verifiable Secret Sharing (Section 4)
$\mathcal{O}(nX + n^3 \log n)$ for X secrets

Figure 1: The structure of our ACS protocol; In red: The communication complexity of each primitive.

an asynchronous validity predicate. Moreover, our gather protocol is unique in that it outputs a set of parties, while their values are inferred via an asynchronous validity predicate.

Verifiable leader election $\implies$ AVABA. We now describe how to use verifiable leader election to implement AVABA. Here the main challenge is working with asynchronously validated inputs and maintaining both safety and liveness over the views. For safety, we use a common approach in authenticated protocols [17, 29] of using lock certificates and adapt them to the information-theoretic setting inspired by the approach of [7]. The protocol of [7] modifies the cryptographic protocol of [29] to the information-theoretic setting in partial synchrony. Here we show how to obtain liveness under fully asynchronous network conditions. For liveness in asynchrony, there are two major challenges. The first is guaranteeing that all honest parties will reach agreement on the leader's proposal if a unique honest leader is elected. For this, we use the key certificate approach of [6, 29] and adapt it to the information-theoretic setting. The second, more challenging problem is guaranteeing that honest parties eventually proceed to a new view if the current view does not lead to agreement. As in [5], we observe that there are two triggers to changing views: the first is when two different parties have different leaders (equivocation event) and the second is when the leader sends a proposal whose key is lower than a lock held by some party (blame event). In [5] these two events can be verified cryptographically, so any honest party that observes this event simply forwards it to all parties. In our setting, we adapt these two events into asynchronously validated predicates. Roughly speaking, when an equivocation or blame message is sent, parties record it and wait for it to be asynchronously validated.

Organization. Figure 1 shows the structure of our ACS protocol, the efficiency of each primitive, and the organization of the rest of the paper. Due to a lack of space, some primitives are presented in the appendices. We also give a sketch of asynchronous MPC construction of [3] in Section 9.

3 Definitions and Preliminaries

3.1 Network and Threat Model

This work deals with protocols for n parties with point-to-point communication channels. The network is assumed to be asynchronous, which means that there is no bound on message delay, but all messages must arrive in finite time. The protocols below are designed to be secure against a computationally unbounded malicious (Byzantine) adversary. The AVSS protocol is secure when the adversary controls $t < \frac{n}{4}$ parties, whereas the other protocols are secure even if the adversary controls $t < \frac{n}{3}$ parties (assuming an AVSS protocol). Furthermore, the adversary is adaptive in the sense that it can choose to corrupt a party at any time given that it hasn't already corrupted t parties.

3.2 Asynchronous Validity Predicates

An asynchronous validity predicate is an asynchronous counterpart to an external validity function. An external validity function receives an input and returns 1 or 0, corresponding to a Boolean *true* or *false*. Similarly, in an asynchronous validity predicate, party i holds a predicate $\mathsf{validate}_i$, and the value of $\mathsf{validate}_i(v)$ can depend on i's internal state. Unlike an external validity function, $\mathsf{validate}_i$ could potentially not terminate. Hence a validity predicate is an asynchronous version of an external validity function checking the party's internal state, and only returning a value when specific conditions hold. Similar notions of stateful predicates have also been discussed in other works requiring a dynamic property to be checked before outputting a value, such as the ones in [12, 20, 24].

For such a predicate to be useful, we want two properties to hold. First, decisions are final – parties do not change their minds about the validity of a given value. This allows parties to confidently output a value without the possibility of it becoming invalid later (or thinking that a value is invalid and then changing their mind later). Secondly, honest parties' opinions will eventually be consistent: if one honest party outputs an opinion on the validity of a given value, eventually all honest parties will have the same opinion. This means that, eventually, opinions about validity will be universal and thus available to all parties in the system. Formally, these properties are captured in the following definition:

Definition 3.1 (Asynchronous Validity Predicate)**.** *Let $\mathcal{V}$ be a set of possible inputs and let every party i have a predicate $\mathsf{validate}_i : \mathcal{V} \to \{0, 1\}$. We say that $\mathsf{validate}$ is a validity predicate if the following properties hold:*

- ***Finality.*** *If $\mathsf{validate}_i(v)$ terminates with the output $b \in \{0, 1\}$ for some honest i, then any call to $\mathsf{validate}_i(v)$ terminates with the output b.*
- ***Consistency.*** *Let i be an honest party and v be some value such that $\mathsf{validate}_i(v)$ terminates with the output $b \in \{0, 1\}$. Eventually for every honest j, $\mathsf{validate}_j(v)$ terminates as well with the output b.*

For brevity, when we say that some property holds for the predicate $\mathsf{validate}$, we mean that it holds for $\mathsf{validate}_i$ for every i. As a shorthand, we say that $\mathsf{validate}_i(v) = b$ at a given time for an honest i if $\mathsf{validate}_i(v)$ had already terminated at that time and output b or if it would immediately terminate with the output b upon being called at that time.

In the above definition, note the similarity of these properties to those of a reliable broadcast protocol (described below), which guarantees that if an honest party outputs a value, every honest party outputs the same value. The properties of this predicate can be generalized to stateful functions that can output one of many values, as opposed to just 0 or 1.

3.3 Reliable Broadcast

We assume the existence of a Reliable Broadcast protocol [13]. A *Reliable Broadcast* protocol is an asynchronous protocol with a designated *sender*. The sender has some *input value M* from some known domain $\mathcal{M}$ and each party may *output* a value in $\mathcal{M}$. A Reliable Broadcast protocol has the following properties assuming all honest parties participate in the protocol:

- **Agreement.** If two honest parties output some value, then it's the same value.
- **Validity.** If the dealer is honest, then every honest party that completes the protocol outputs the dealer's input value, M.
- **Termination.** If the dealer is honest, then all honest parties complete the protocol and output a value. Furthermore, if some honest party completes the protocol, every honest party completes the protocol.

Broadcasting L bits requires $\mathcal{O}(n^2 L + n^2 \log n)$ over point-to-point channel using the protocol of [8].

4 Packed Asynchronous Verifiable Secret Sharing

4.1 Definition

A packed asynchronous verifiable secret sharing protocol (packed AVSS) over a finite field $\mathbb{F}$ consists of a pair of protocols $(\mathsf{Share}, \mathsf{Reconstruct})$ with a designated party acting as dealer. The dealer has inputs $s_0, \dots, s_m \in \mathbb{F}$ to the Share protocol, and the other parties have no input. On a later stage, each party can call $\mathsf{Reconstruct}(k)$ for each $k \in \{0, \dots, m\}$, which eventually returns some value. Honest parties only call the $\mathsf{Reconstruct}$ protocol after having completed the Share protocol. A packed AVSS protocol has the following properties:

- **Correctness.** Once the first honest party completes the Share protocol, there exist values $r_0, \dots, r_m$ such that:
  - if the dealer is honest then $\forall k \in \{0, \dots, m\}$ $r_k = s_k$; and
  - if some honest party completes $\mathsf{Reconstruct}(k)$ for some $k \in \{0, \dots, m\}$, then its output is r_k.
- **Termination.** If all honest parties participate in the Share protocol, then:
  - if the dealer is honest, all honest parties complete the Share protocol; and
  - if some honest party completes the Share protocol, then all honest parties complete the share protocol.

  In addition, if all honest parties participate in $\mathsf{Reconstruct}(k)$ then they all complete the protocol.
- **Secrecy.** If the dealer is honest and no honest party called $\mathsf{Reconstruct}(k)$ for $k \in \{0, \dots, m\}$, the adversary's view is distributed independently of s_k.

4.2 Overview

In this section, we describe an information-theoretic packed AVSS protocol that requires $\mathcal{O}(nX + n^3 \log n)$ for sharing X secrets.

A quick overview of the AVSS protocol of [10]. We start with a quick overview of the protocol of the AVSS protocol of Ben-Or, Canetti and Goldreich [10]. As a first step, we present an inefficient version where the dealer runs in exponential time.

1. The dealer chooses a random bivariate polynomial $S(X,Y)$ of degree-t in both variables such that $S(0,0)=s$. It then gives to each party i the shares $f_i(X)=S(X,i)$, $g_i(Y)=S(i,Y)$.

2. After receiving their f_i and g_i polynomials, which we call their shares, and seeing that they are of the correct degrees, every party i forwards the values $f_i(j), g_i(j)$ to every j.

3. When a party i receives a forwarded pair of values $f_j(i), g_j(i)$ from some party j, it verifies that these values are consistent with the values it has received from the dealer. Namely, that $f_i(j) = g_j(i)$ $(= S(j,i))$ and $g_i(j) = f_j(i)$ $(= S(i,j))$. If this is the case, then i broadcasts $\langle$"ok"$, i, j\rangle$, signifying that the i agrees with j.

4. The dealer initiates a graph G with $V = [n]$ and $E = \emptyset$. Then, upon receiving broadcasted messages $\langle$"ok"$, i, j\rangle$ and $\langle$"ok"$, j, i\rangle$ it adds the edge (i,j) to E. It then looks for a clique $K \subseteq [n]$ in G. If found, it broadcasts $\langle$"clique"$, K\rangle$. Otherwise, it continues to listen to more "ok" messages, updates its graph, and repeats.

5. Each party also initiates a graph as the dealer and adds edges in a similar manner. Once the dealer broadcasts $\langle$"clique"$, K\rangle$, the party verifies that K is a clique in its respective graph. If so – it terminates. Otherwise, it continues to listen to more edges.

It is easy to see that if one honest party j terminates, all honest parties will eventually terminate. This is because j terminates only after the dealer has broadcasted a clique and j has verified the same clique in its respective graph. Since all the messages that j considers are broadcasted, each other honest party will eventually see the same clique in its graph. Moreover, if the dealer is honest, then the dealer will eventually see the clique of all honest parties in its graph, will broadcast it, and all honest parties will eventually verify it as well.

To show binding, assume that the dealer has broadcasted some clique and an honest party has verified that clique. Since the clique contains at least $3t+1$ parties, it contains at least $2t+1$ honest parties. Since each pair of honest parties has verified that their shares agree, they all lie on the same bivariate polynomial $S(x,y)$. Moreover, parties only consider members of the clique during reconstruction. This implies that in the reconstruction phase, we will have at least $2t+1$ correct shares and at most t errors, and therefore reconstruction is guaranteed.

The star algorithm. To make the dealer computationally efficient, Canetti [15] defines the FindStar algorithm that finds a large "star" [15] in a graph, which can be thought of as a relaxation of a clique. It receives an undirected graph $G = (V,E)$ as input and outputs a pair of sets $C, D \subseteq V$ such that $C \subseteq D$ and there exists an edge $(u,v) \in E$ for every $u \in C, v \in D$, and such that $|C| \geq n-2t$ $(= 2t+1)$ and $|D| \geq n-t$ $(= 3t+1)$. The algorithm might also output "no star was found". In addition, Canetti [15] showed a polynomial time algorithm such that, if there exists a clique of size $n-t$ then it finds such a STAR. The dealer then looks for a STAR in the graph instead of a clique, and once found, it broadcasts $\langle$"star"$, C, D\rangle$. Each party verifies that (C,D) is a STAR, and terminates if so.

This guarantees validity and binding: **Validity:** If the dealer is honest, then eventually, there will be a clique in the graph of size $n-t$. The STAR algorithm then outputs such (C,D), and all honest parties will eventually verify that (C,D) is a STAR. **Binding:** Once an honest party terminates (regardless of whether or not the dealer is honest), the set C contains at least $t+1$ honest parties, and all their shares must agree. Therefore, the set C defines a unique bivariate polynomial $S(X,Y)$, and the shares of all honest parties in C lie on that polynomial. Moreover, the set D contains at least $2t+1$ honest parties, and their shares agree with all parties in C and, therefore, must also agree with S. We obtain that there are at least $2t+1$ honest parties with valid shares, and therefore reconstruction is guaranteed even if t errors are introduced during the reconstruction phase.

Cost. When considering also the costs of broadcast "ok" messages, the above protocol requires $\mathcal{O}(n^4 \log n)$ bits transmitted over the point-to-point channels. This is because each message of size L being broadcasted requires $\mathcal{O}(n^2 L)$ bit sent over the point-to-point channels. Since each party i broadcasts ("ok", i, j) we have $\mathcal{O}(n^2)$ messages being broadcasted. This implies that overall, we have $\mathcal{O}(n^4 \log n)$ overhead for sharing a single secret.

Reducing the cost. To reduce the cost, the protocol of [28] utilizes the following two tricks:

- **Packing:** Instead of having a bivariate polynomial of degree t in both variables, the dealer can embed $t+1$ secrets in one bivariate polynomial. That is, the dealer holds secrets $s_0, \ldots, s_t$, and uniformly samples a bivariate polynomial $S(X,Y)$ of degree $2t$ in X and degree t in Y such that $S(-k, 0) = s_k$ for every $k \in \{0, \ldots, t\}$.

- **Batching:** Instead of having one instance of AVSS, we can run $\mathcal{O}(n^2)$ instances in parallel while re-using the broadcast messages across all instances. That is, each i broadcasts $\langle$"ok", $i, j\rangle$ only after it verified the shares of j across all $\mathcal{O}(n)$ instances.

Those two ideas together lead to a protocol in which parties send $\mathcal{O}(nX + n^4 \log n)$ bits point-to-point for sharing X secrets (of $\mathcal{O}(\log N)$ bits each). This gives $\mathcal{O}(n)$ overhead per secret, starting from $\Omega(n^3)$ secrets. However, if the dealer has to distribute only $\mathcal{O}(n)$ secrets (as in our ACS protocol), we get an overhead of $\mathcal{O}(n^3)$.

$\mathcal{O}(n)$-overhead for $o(n^3)$ secrets. To reduce the overhead when the dealer has to share $o(n^3)$ secrets, we further improve the protocol and add one more optimization to packing and batching:

- **No broadcast:** We completely eliminate any broadcast message in the protocol.

To achieve this property, first, consider trying to replace any broadcast message with multicast (the sender simply sends the message to all parties). Edges (i,j) between pairs of honest parties will appear in all graphs and will be consistent. On the other hand, edges between corrupted parties or between an honest party and a corrupted party might not be consistent in the different graphs. Moreover, the dealer might announce different STARs among the different parties.

To overcome this difficulty, we instruct each party i to look for its own STAR (C_i, D_i). Moreover, in addition to those two sets, we look for an extended star (see, e.g.,[28]) (C_i, D_i, E_i, F_i) which satisfies the following properties:

- C_i: a clique of size (at least) $n-2t$ (i.e., $2t+1$), as before.

- D_i: a set of size (at least) $n-t$ that agrees with all C_i (i.e., for all $d \in D_i$ and $c \in C_i$ there exists an edge $-$ (c,d)). This is again as before.

- F_i: a set of size $n-t$ of all vertices that have at least $n-2t$ edges to C_i.

- E_i: a set of size $n-t$ of all vertices that have at least $n-t$ edges to F_i.

Each party finds an extended star in its graph. Then, the challenge is that parties might have different graphs. Nevertheless, we claim that: (1) Validity: If the dealer is honest, then all honest parties will eventually find extended stars in their respective graphs; (2) Binding: If "enough" honest parties find extended stars - they all define the exact same bivariate polynomial although they do not have the same graphs:

Validity: It is also easy to see that if the dealer is honest, then all honest parties will eventually find such (C_i, D_i, E_i, F_i): The clique of all honest parties will eventually appear in the respective graphs of the honest parties. An extended star will then always be found.

Binding: First, we claim that for every honest party j, the honest parties in the sets that j has found, i.e., the honest parties in the sets (C_j, D_j, E_j, F_j) define a unique bivariate polynomial. Specifically:

1. The set C_j contains at least $t+1$ honest parties; take an arbitrary subset $C'_j \subseteq C_j$ of cardinality $t+1$; their f-shares (each is of degree-$2t$) define a unique bivariate polynomial $S_j(X,Y)$ of degree $(2t,t)$;
2. The set D_j contains at least $2t+1$ honest parties, and each such party agrees with all parties in C_j; As such, each such party must hold a g-share that lies on $S_j(X,Y)$. Since D_j contains at least $2t+1$ honest parties, it also implies that all the other honest parties (if exist) in $C_j \setminus C'_j$ hold f-shares that lie on S_j.
3. The set F_j contains at least $2t+1$ honest parties; each such honest party agrees with at least $n-2t$ (i.e., $\geq 2t+1$) parties in C_j, i.e., with at least $t+1$ honest parties in C_j. Thus, all the g-shares of parties in F_j lie on S_j.
4. The set E_j contains at least $2t+1$ honest parties; each such party agrees with at least $3t+1$ parties in F_j, i.e., with at least $2t+1$ honest parties. As such, all the f-shares of parties in E_j lie on S_j.

Moreover, we claim that for two honest parties j and k that might have distinct extended stars (C_j, D_j, E_j, F_j) and (C_k, D_k, E_k, F_k), the bivariate polynomial that each of them defines – S_j and S_k, respectively, must be the same. This is because E_j and E_k are both sets of size $3t+1$ and, therefore, must have an intersection of size at least $2t+1$, i.e., at least $t+1$ honest parties in their intersection. The f-shares of those parties uniquely define S_j and S_k, respectively, and it must hold that $S_k = S_j$.

The rest of the protocol. In the rest of the protocol, parties that do have shares help the other parties to reconstruct their values, and also hold shares on their polynomial. Since the shares of all honest parties that have an extended star must define the same bivariate polynomial, we get that, eventually, all parties would hold shares on that polynomial. Therefore, we get a *complete secret sharing* – all honest parties have shares at the end of the protocol. This makes the reconstruction phase almost trivial – parties just send their shares to one another, and use Reed Solomon decoding to eliminate errors.

4.3 The Protocol

Before describing the protocol, we first overview two algorithms that the protocol uses: FindStar and RobustInt.

FindStar. FindStar finds a large "star" [15] in a graph, which can be thought of as a weak version of a clique. It receives an undirected graph $G = (V, E)$ as input and outputs a pair of sets $C, D \subseteq V$

such that $C \subseteq D$ and there exists an edge $(u,v) \in E$ for every $u \in C, v \in D$. Additionally, if G contains a clique of size $n-t$, it is guaranteed that $|C| \geq n-2t$ and $|D| \geq n-t$. We also require that if there is a clique of size $n-t$ in the graph, then C will contain at least $n-2t$ vertices from the clique. This property was not originally formulated in [15] but is proven in [28]. In the original formulations of the algorithm, FindStar either returns C, D such that $C \subseteq D \subseteq [n]$ and $|C| \geq n-2t, |d| \geq n-t$ or outputs a failure message such as "star not found" if it fails to do so. For simplicity, we assume that the algorithm always outputs $C \subseteq D \subseteq [n]$. This can be implemented by simply outputting the sets $C = D = \emptyset$ instead of a failure message.

RobustInt. The RobustInt algorithm is a decoding algorithm for the Reed-Solomon encoding [27]. The algorithm receives three inputs S, d, e such that S is a set of tuples (j, y_j), $d \in \mathbb{N}$ is a degree, and $e \in \mathbb{N}$ is the number of allowed errors. It then outputs a polynomial p of degree d or less or the value $\bot$. If the algorithm outputs a polynomial p, then it is the unique polynomial of degree d or less such that for at most e tuples $(j, y_j) \in S$, $p(j) \neq y_j$. This means that RobustInt always outputs $\bot$ if $|S| < d+e+1$ because otherwise there is no unique polynomial for which the above holds. In addition, if for some $m \in \mathbb{N}$, $|S| \geq d+2m+1$ and for some polynomial p of degree d or less, there are at most m tuples $(j, y_j) \in S$ such that $p(j) \neq y_j$, then RobustInt outputs p. Intuitively this means that if S defines the polynomial p and it has m errors in it, it is enough to get $d+1$ correct points to define the polynomial, along with an additional correct point for each error.

Conceptually, the Share protocol described in Protocol 4.2 proceeds as follows:

Round 1: In the first round, the dealer uniformly samples a bivariate polynomial $S(X,Y)$ of degree $2t$ or less in X and degree t or less in Y such that $S(-k,0) = s_k$ for every $k \in \{0,\ldots,t\}$. It then defines the polynomials $f_i(X) = S(X,i), g_i(Y) = S(i,Y)$ and sends these polynomials to every party i.

Round 2: After receiving their f_i and g_i polynomials and seeing that they are of the correct degrees, every party i forwards the values $f_i(j), g_i(j)$ to every j.

Round 3: When party i receives a forwarded pair of values $f_j(i), g_j(i)$ from another party j, it checks that these values are consistent, i.e. $f_i(j) = g_j(i), g_i(j) = f_j(i)$. This should be the case because $f_i(j) = S(j,i) = g_j(i)$ and $g_i(j) = S(i,j) = f_j(i)$ should hold if the dealer is correct. If this is the case, i sends an "ok" message about j to all parties, signifying that their values are consistent.

Round 4: When parties receive "ok" messages from both i and j about each other, they consider those parties as being consistent with each other. They can then form a graph whose vertices are all the parties $[n]$, and any pair of parties has an edge if they are consistent with each other. Every party then attempts to find a star C, D in the graph using the FindStar algorithm. If such a star is found such that C is of size $n-2t$ and D is of size $n-t$, parties attempt to find additional sets E, F that expand this star. Concretely, F is defined to be the set of parties that have $n-2t$ neighbors in C, and E is defined to be the set of parties that have $n-t$ neighbors in F. Parties wait for both of these sets to be of size $n-t$ and then send them to each other. For simplicity, we will call the four sets C, D, E, F a star instead of just the pair C, D.

Round 5: After receiving a star from another party j, every party i attempts to use the star to interpolate a polynomial $g_{i,j}$. It does so by using the $f_j(i)$ values sent in round 2 but only considers the parties in E_j. It waits to receive enough values for the RobustInt algorithm to succeed when trying to find a polynomial of degree t with up to t errors. After this happens, i stores the resulting polynomial $g_{i,j}$. Once there are $t+1$ successful interpolations from $t+1$ different stars that all output the same polynomial, i stores that polynomial as q_i and sends the value $q_i(j)$ to every j.

Round 6: After receiving $q_j(i)$ from different parties, i attempts to interpolate these values to a polynomial p_i. Once $\mathsf{RobustInt}$ succeeds in interpolating these values to a polynomial of degree $2t$ with up to t errors, i stores that polynomial as p_i.
Round 7: In parallel to previous rounds, as a tool for termination, party i sends a "done" message to all parties after receiving $n-t$ stars, or after receiving $t+1$ "done" messages. This guarantees that the first party that sent such a message did so after seeing at least $n-2t$ stars from honest parties. Moreover, if some honest party receives $n-t$ "done" messages, every honest party will receive at least $t+1$ (in fact they will eventually receive $2t+1$), and forward that message as well.
Termination: After receiving $n-t$ "done" messages and having p_i and q_i polynomials, every party completes the protocol, using its p_i and q_i polynomial as shares for the reconstruction protocol.

Protocol 4.1: $\mathsf{Deal}(s_0,\dots,s_t)$

```
1: uniformly sample a bivariate polynomial $S(X,Y)$ of degree $2t$ in $X$ and $t$ in $Y$ such that
   $\forall k \in \{0,\dots,t\}\ S(-k,0)=s_k$
2: for all $i \in [n]$ do
3:     $f_i(X)=S(X,i), g_i(Y)=S(i,Y)$
4:     send $\langle$"polynomials"$, f_i, g_i\rangle$ to every $i \in [n]$
```

Protocol 4.2: $\mathsf{Share}_i()$

```
1: $f_i \leftarrow \bot, g_i \leftarrow \bot, p_i \leftarrow \bot, q_i \leftarrow \bot$
2: $\mathsf{points}_{g,i} \leftarrow \emptyset, \mathsf{edges}_i \leftarrow \emptyset, \mathsf{stars}_i \leftarrow \emptyset, \mathsf{interpolated}_i \leftarrow \emptyset, \mathsf{points}_{p,i} \leftarrow \emptyset$
3: $C_i \leftarrow \emptyset, D_i \leftarrow \emptyset, E_i \leftarrow \emptyset, F_i \leftarrow \emptyset$
4: if $i$ is the dealer with secrets $s_0,\dots,s_t$ then
5:     call $\mathsf{Deal}(s_0,\dots,s_t)$
6: upon receiving a $\langle$"polynomials"$, f'_i, g'_i\rangle$ message from the dealer, do
7:     if $\deg(f'_i) \le 2t, \deg(g'_i) \le t$ then
8:         $f_i \leftarrow f'_i, g_i \leftarrow g'_i$
9:         send $\langle$"values"$, f_i(j), g_i(j)\rangle$ to every $j \in [n]$
10: upon receiving a $\langle$"values"$, f_j(i), g_j(i)\rangle$ message from party $j$, do
11:     $\mathsf{points}_{g,i} \leftarrow \mathsf{points}_{g,i} \cup \{(j, f_j(i))\}$
12:     upon $f_i \ne \bot, g_i \ne \bot$, do
13:         if $f_i(j)=g_j(i), g_i(j)=f_j(i)$ then
14:             send $\langle$"ok"$, j\rangle$ to all parties
15: upon receiving both $\langle$"ok"$, k\rangle$ from $j$ and $\langle$"ok"$, j\rangle$ from $k$, do
16:     $\mathsf{edges}_i \leftarrow \mathsf{edges}_i \cup \{(j,k)\}$
17:     if $|E_i| < n-t$ or $|F_i| < n-t$ then
18:         $C_i, D_i \leftarrow \mathsf{FindStar}([n], \mathsf{edges}_i)$
19:         if $|C_i| \ge n-2t, |D_i| \ge n-t$ then
20:             $F_i \leftarrow \{j \in [n]\ s.t.\ |\{k \in C_i | (j,k) \in \mathsf{edges}_i\}| \ge n-2t\}$
21:             $E_i \leftarrow \{j \in [n]\ s.t.\ |\{k \in F_i | (j,k) \in \mathsf{edges}_i\}| \ge n-t\}$
22:             if $|E_i| \ge n-t, |F_i| \ge n-t$ then
23:                 send $\langle$"star"$, C_i, D_i, E_i, F_i\rangle$ to all parties
24: upon receiving a $\langle$"star"$, C_j, D_j, E_j, F_j\rangle$ message from party $j$, do
25:     $\mathsf{stars}_i \leftarrow \mathsf{stars}_i \cup \{(j, C_j, D_j, E_j, F_j)\}$
```

```
26: upon $\mathsf{stars}_i$ or $\mathsf{points}_{g,i}$ being updated and $q_i = \bot$, do
27:     for all $(j, C_j, D_j, E_j, F_j) \in \mathsf{stars}_i$ do
28:         if there is no tuple $(j, g_{i,j}) \in \mathsf{interpolated}_i$ then
29:             $g_{i,j} \leftarrow \mathsf{RobustInt}(\{(k, y_k) \in \mathsf{points}_{g,i} | k \in E_j\}, t, t)$
30:             if $g_{i,j} \neq \bot$ then
31:                 $\mathsf{interpolated}_i \leftarrow \mathsf{interpolated}_i \cup \{(j, g_{i,j})\}$
32: upon updating $\mathsf{interpolated}_i$, do
33:     if there exists a polynomial $q'_i$ such that $|\{j|(j, q'_i) \in \mathsf{interpolated}_i\}| \geq t+1$ then
34:         $q_i \leftarrow q'_i$
35:         send $\langle$"col", $q_i(j)\rangle$ to every $j \in [n]$
36: upon receiving a $\langle$"col", $q_j(i)\rangle$ message from $j$, do
37:     $\mathsf{points}_{p,i} \leftarrow \mathsf{points}_{p,i} \cup \{(j, q_j(i))\}$
38:     $p'_i \leftarrow \mathsf{RobustInt}(\mathsf{points}_{p,i}, 2t, t)$
39:     if $p'_i \neq \bot$ then
40:         $p_i \leftarrow p'_i$
41: upon $|\mathsf{stars}_i| = n - t$ or receiving $\langle$"done"$\rangle$ messages from $t+1$ parties, do
42:     send $\langle$"done"$\rangle$ to all parties
43: upon receiving $\langle$"done"$\rangle$ from $n - t$ parties and $p_i \neq \bot, q_i \neq \bot$, do
44:     terminate
```

Reconstruction. The reconstruction protocol is straightforward. When reconstructing the k'th secret, every party sends $p_i(-k)$ to all other parties. Parties then try to interpolate the received values to a polynomial $q_{-k}(Y)$ of degree t in Y with up to t errors. Once they succeed in the interpolation using the RobustInt algorithm, they output $q_{-k}(0)$ as the k'th secret.

Protocol 4.3: $\mathsf{Reconstruct}_i(k)$

```
1: $\mathsf{points}_i \leftarrow \emptyset$
2: send $\langle$"rec", $k, p_i(-k)\rangle$
3: upon receiving a $\langle$"rec", $k, y_j\rangle$ message from $j$, do
4:     $\mathsf{points}_i \leftarrow \mathsf{points}_i \cup \{(j, y_j)\}$
5:     $q_{-k} \leftarrow \mathsf{RobustInt}(\mathsf{points}_i, t, t)$
6:     if $q_{-k} \neq \bot$ then
7:         output $q_{-k}(0)$ and terminate
```

4.4 Security Analysis

In order to prove the correctness of the protocol we show that each star collected by an honest party defines a unique bivariate polynomial G of degree $2t$ in X and t in Y. We then show that every two honest parties' stars define the same polynomial and that parties' stored f_i and g_i polynomials are consistent with this polynomial.

Lemma 4.4. *Assume some honest party i sent a $\langle$"star"$, C_i, D_i, E_i, F_i\rangle$ message. Then every honest party $j \in C_i \cup D_i \cup E_i \cup F_i$ has $f_j \neq \bot, g_j \neq \bot$. In addition, there exists a unique polynomial G of* *degree $2t$ in X and t in Y such that for every honest $j \in C_i \cup E_i$ $f_j(X) = G(X, j)$ and for every honest $j \in D_i \cup F_i$ $g_j(X) = G(j, Y)$.*

Proof. Since i sent the "star" message, it computed C_i, D_i using FindStar and found that C_i is at least of size $n - 2t$ and D_i, E_i, F_i are at least of size $n - t$. By the definition of the FindStar algorithm, for every $j \in C_i, k \in D_i$, there exists an edge $(j, k) \in \mathsf{edges}_i$. Party i only adds such an edge to edges_i after receiving an $\langle$"ok"$, j\rangle$ message from k and an $\langle$"ok"$, k\rangle$ message from j. Note that if j and k are honest, they only send such messages after having non-$\perp$ f_j, f_k and g_j, g_k polynomials of degrees $2t$ and t respectively, sending each other "values" messages and seeing that $f_j(k) = g_k(j), g_j(k) = f_k(j)$. Since $|C_i| \geq n - 2t, |D_i| \geq n - t$, C_i has indices of at least $t + 1$ honest parties and D_i has indices of at least $2t + 1$ honest parties. In other words, for every honest $j \in C_i$ and $k \in D_i$, f_j is a polynomial of degree $2t$, g_k is a polynomial of degree t and $f_j(k) = g_k(j)$. In such a setting, there exists a unique bivariate polynomial $G(X, Y)$ of degree $2t$ in X and t in Y such that for every such j, k, $f_j(X) = G(X, j)$ and $g_k(Y) = G(k, Y)$.

Now, observe an honest party $j \in F_i$. For similar reasons, its f_j, g_j fields contain polynomials of degrees $2t$ and t respectively. Similarly, for every honest $k \in C_i$, $g_j(k) = f_k(j) = G(j, k)$. There are $t + 1$ such honest parties $k \in C_i$, so $g_j(Y)$ and $G(j, Y)$ are two polynomials of degree t or less that agree on at least $t + 1$ points, and thus $g_j(Y) = G(j, Y)$. Following very similar reasons, for every $j \in E_i$, there are at least $2t + 1$ honest parties $k \in F_i$ for which $f_j(k) = g_k(j) = G(k, j)$ and thus $f_j(X) = G(X, j)$. □

Lemma 4.5. *Assume two honest parties i, j sent "star" messages and let G_i, G_j be the polynomials defined for them in Lemma 4.4. It is the case that $G_i = G_j$.*

Proof. As in Lemma 4.4, the E_i, E_j, F_i, F_j sets sent by i, j are of size $n - t$. Since $E_i \cup E_j \subseteq [n]$, $|E_i \cap E_j| \geq 2t + 1$, and at least $2t + 1 - t = t + 1$ of the shared indices those of honest parties. For each such honest $k \in E_i \cap E_j$, $G_i(X, k) = f_k(X) = G_j(X, k)$. Therefore, for honest $k \in E_i \cap E_j$ and for every possible v, $G_i(v, k) = G_j(v, k)$. Both G_i and G_j have degree t in Y, and for every value v, $G_i(v, Y)$ agrees with $G_j(v, Y)$ at $t + 1$ points. Therefore, for every value v, $G_i(v, Y) = G_j(Y)$ and thus the polynomials are equal. □

Lemma 4.6. *Assume some honest party i has $p_i \neq \perp$ or $q_i \neq \perp$. Then some honest party j sent a $\langle$"star"$, C_j, D_j, E_j, F_j\rangle$, and i has $p_i(X) = G(X, i)$ or $q_i(Y) = G(i, Y)$ respectively for the G defined in Lemma 4.4.*

Proof. First, assume i had $q_i \neq \perp$. It updates q_i after adding at least $t + 1$ tuples to $\mathsf{interpolated}_i$, which it does after receiving a $\langle$"star"$, C_j, D_j, E_j, F_j\rangle$ from at least $t + 1$ different parties j and successfully interpolating a polynomial for each one. At least one of the parties is honest. Let that party be j and the sent values be C_j, D_j, E_j, F_j, and let G be the polynomial defined for j in Lemma 4.4.

Party i updates q_i after successfully interpolating $g_{i,k}$ polynomials for different parties k, adding tuples of the form $(k, g_{i,k})$ to $\mathsf{interpolated}_i$, and seeing that for $t{+}1$ of those tuples $g_{i,k} = q_i$. It does so after receiving a message $\langle$"star"$k, C_k, D_k, E_k, F_k\rangle$ for each such k and adding a (k, C_k, D_k, E_k, F_k) tuple to stars_i. Following that, it interpolates $g_{i,k}$ by calling $\mathsf{RobustInt}(S, t, t)$ for $S = \{(l, y_l) \in \mathsf{points}_{g,i} | l \in E_k\}$. From the definition of RobustInt, $g_{i,k}$ is of degree t or less. In addition, RobustInt only returns a non-$\perp$ value if it receives as input a set with at least $t + t + 1$ tuples, and returns the unique polynomial $g_{i,k}$ such that $g_{i,k}(l) = y_l$ for all but t tuples $(l, y_l) \in S$. Party i adds tuples

(l, y_l) to $\mathsf{points}_{g,i}$ after receiving a $\langle$"values"$, v_l, y_l\rangle$ message from l, and honest parties send such a message with $y_l = g_l(i)$. From Lemma 4.5, for every such honest l, $g_l(Y) = G(l, Y)$ and thus $y_l = G(l, i)$. Therefore, for every $(l, y_l) \in S$ such that l is honest, $y_l = G(l, y_l)$. In other words, $g_{i,k}(Y) = G(Y, i)$ is a polynomial of degree at most t such that $g_{i,k}(l) = y_l$ for all but t tuples $(l, y_l) \in S$, and thus it must also be the unique such polynomial output by $\mathsf{RobustInt}$. Since this is the case for every honest party k, $\mathsf{interpolated}_i$ contains tuples of the form $(k, G(Y, k))$ for every honest k, and thus there are at most t tuples $(k, g_{i,k}(Y))$ for which $g_{i,k}(Y) \neq G(Y, k)$. Finally, this means that when i updated q_i it did so to the polynomial $q_i(Y) = G(Y, i)$.

Now assume that $p_i \neq \perp$. Party i updates p_i after successfully interpolating it using $\mathsf{RobustInt}(\mathsf{points}_{p,i}, 2t, t)$. It only adds tuples of the form (k, y_k) to $\mathsf{points}_{p,i}$ after receiving a $\langle$"col"$, y_k\rangle$ message from party k, and honest parties only send such a message with $y_k = q_k(i) = G(k, i)$. As in the above argument, $\mathsf{RobustInt}$ only outputs p_i after $\mathsf{points}_{p,i}$ contains at least $2t + t + 1$ tuples. Out of those, at least $2t + 1$ are tuples of the form $(k, G(k, i))$ added after receiving messages from honest parties. This means that the polynomial $G(X, i)$ is a polynomial of degree $2t$ or less such that at most t tuples in $\mathsf{points}_{p,i}$ disagree with it. Since $p_i(X)$ is the unique polynomial for which this holds, $p_i(X) = G(X, i)$. □

The following lemmas are used to prove that the protocol maintains secrecy. This is done by first showing that the adversary's view, in conjunction with any set of secrets, defines a unique polynomial sampled by the adversary, and then showing that there is an identical number of possible polynomials for each set of secrets. This means that any given view has the same probability of having been produced under any set of secrets.

Lemma 4.7. *Let $f_1, \ldots, f_t$ be univariate polynomials of degree $2t$ or less and $g_{-t+1}, \ldots, g_t$ be polynomials of degree t or less such that for every $i \in \{1, \ldots, t\}$ and $j \in \{-t+1, \ldots, t\}$ $f_i(j) = g_j(i)$. Then for every $s_t \in \mathbb{F}$, there exists a unique bivariate polynomial $S(X, Y)$ of degree $2t$ in X and degree t in Y such that $S(-t, 0) = s_t$ and $\forall i \in \{1, \ldots, t\}, j \in \{-t+1, \ldots, t\}$, $S(X, i) = f_i(X)$, and $S(j, Y) = g_j(Y)$.*

Proof. First define $f_0(X)$ to be the unique polynomial of degree $2t$ such that $\forall j \in \{-t+1, \ldots, t\}$ $f_0(j) = g_j(0)$ and $f_0(-t) = s_f$. By Lagrange interpolation, there is exactly one such polynomial. Let $a_{i,j}$ be the coefficient of X^j in the polynomial $f_i(X)$. Now let the matrix A be the $(t+1)\times(2t+1)$ matrix for which $A_{i,j} = a_{i,j}$. In addition, define V_1 to be the $(t+1) \times (t+1)$ Vandermonde matrix such that for every $i, j \in \{0, \ldots, t\}$: $(V_1)_{i,j} = i^j$. Finally, define V_2 to be the $(2t + 1) \times (2t + 1)$ Vandermonde matrix such that for every $i, j \in \{0, \ldots, 2t\}$:

$$(V_2)_{i,j} = \begin{cases} i^j & i \leq t \\ (t-i)^j & i > t \end{cases}$$

In both of these cases, 0^0 is defined to be 1. Let $B = (V_1)^{-1}A$ and $S(X, Y)$ be the bivariate polynomial that has $b_{i,j}$ as the coefficient of the term X^iY^j. Now observe $(V_1BV_2^T)_{i,j}$ for any

$i \in \{-t+1, \dots, t\}, j \in \{0, \dots t\}$:

$$\begin{aligned}(V_1 B V_2^T)_{i,j} &= \sum_{l=0}^{2t} (V_1 B)_{i,l} \cdot (V_2^T)_{l,j} \\ &= \sum_{l=0}^{2t} (\sum_{m=0}^{t} (V_1)_{i,m} \cdot F_{m,l})(V_2^T)_{l,j} \\ &= \sum_{l=0}^{2t} \sum_{m=0}^{t} B_{m,l} \cdot (V_1)_{i,m} \cdot (V_2^T)_{l,j}\end{aligned}$$

For every $j \leq t$, evaluating this expression results in:

$$\begin{aligned}(V_1 B V_2^T)_{i,j} &= \sum_{l=0}^{2t} \sum_{m=0}^{t} B_{m,l} \cdot (V_1)_{i,m} \cdot (V_2^T)_{l,j} \\ &= \sum_{l=0}^{2t} \sum_{m=0}^{t} B_{m,l} \cdot i^m \cdot j^l = S(j,i)\end{aligned}$$

On the other hand, for every $j > t$:

$$\begin{aligned}(V_1 B V_2^T)_{i,j} &= \sum_{l=0}^{2t} \sum_{m=0}^{t} B_{m,l} \cdot (V_1)_{i,m} \cdot (V_2^T)_{l,j} \\ &= \sum_{l=0}^{2t} \sum_{m=0}^{t} B_{m,l} \cdot i^m \cdot (t-j)^l = S(t-j,i)\end{aligned}$$

Computing these values differently:

$$(V_1 B V_2^T)_{i,j} = (A V_2^T)_{i,j} = \sum_{l=0}^{2t} A_{i,l} \cdot (V_2^T)_{l,j}$$

Again, if $j \leq t$:

$$\begin{aligned}(V_1 B V_2^T)_{i,j} &= \sum_{l=0}^{2t} A_{i,l} \cdot (V_2^T)_{l,j} \\ &= \sum_{l=0}^{2t} a_{i,l} \cdot j^l = f_i(j)\end{aligned}$$

And if $j > t$:

$$\begin{aligned}(V_1 B V_2^T)_{i,j} &= \sum_{l=0}^{2t} A_{i,l} \cdot (V_2^T)_{l,j} \\ &= \sum_{l=0}^{2t} a_{i,l} \cdot (t-j)^l = f_i(t-j)\end{aligned}$$

In other words, for every $i \in \{0, \dots, t\}, j \in \{-t, \dots, t\}$, $S(j, i) = f_i(j)$. Since both are univariate polynomials of degree $2t$ or less, $S(X, i) = f_i(X)$. Similarly, for every $i, j \in \{1, \dots, t\}$, $S(j, i) = f_i(j) = g_j(i)$, and thus $S(j, Y) = g_j(Y)$. Note that $S(-t, 0) = f_0(-t)$ and by definition $f_0(-t) = s_t$. Finally, note that every bivariate polynomial S' which fulfills the requirements of the lemma must also have $(V_1 S' V_2^T)_{i,j} = (AV_2^T)_{i,j}$, where S' is the coefficient matrix for S'. Since V_1 is invertible, there is only one such matrix $S' = (V_1)^{-1}A = S$, and thus S is unique. □

Lemma 4.8. *For every $s_0, \dots, s_t \in \mathbb{F}$ there exist exactly $|\mathbb{F}|^{2t(t+1)}$ bivariate polynomials $S(X, Y)$ of degrees no greater than $2t$ in X and t in Y such that $\forall k \in \{0, \dots, t\}$ $S(-k, 0) = s_k$.*

Proof. A bivariate polynomial $S(X, Y)$ of degree no greater than $2t$ in X and t in Y is a function of the form:

$$S(X, Y) = \sum_{k=0}^{2t} \sum_{l=0}^{t} a_{k,l} X^k Y^l \ .$$

Since each of these polynomials is uniquely defined by the coefficients $a_{k,l}$, it is enough to count the number of distinct possible combinations of coefficients. We start by observing that we can rewrite $S(X, Y)$ as:

$$S(X, Y) = \sum_{k=0}^{2t} a_{k,0} X^k + \sum_{k=0}^{2t} \sum_{l=1}^{t} a_{k,l} X^k Y^l \ .$$

The total number of possible combinations of coefficients in the expression $\sum_{k=0}^{2t} \sum_{l=1}^{t} a_{k,l} X^k Y^l$ is $|\mathbb{F}|^{(2t+1)t}$, because there are $(2t + 1)t$ coefficients, and each one can be chosen independently from the field $\mathbb{F}$. To finish the proof, we only need to count the number of possible combinations of coefficients in the polynomial $S(X, 0) = \sum_{k=0}^{2t} a_{k,0} X^k$ under the condition that $S(-k, 0) = s_k$ for every $k \in \{0, \dots, t\}$. Since $S(X, 0)$ is a polynomial of degree at most $2t$, it is uniquely defined by $2t + 1$ points (x, y) such that $S(x, 0) = y$. Since we require that $S(-k, 0) = s_k$ for $k \in \{0, \dots, t\}$ there are $2t + 1 - (t + 1) = t$ points left that can be chosen freely to uniquely define $S(X, 0)$. Thus there are $|\mathbb{F}|^t$ possible combinations of coefficients $a_{0,0}, \dots, a_{0,2t}$ satisfying the required conditions. Overall, the total number of possible polynomials is exactly:

$$|\mathbb{F}|^{(2t+1)t} \cdot |\mathbb{F}|^t = |\mathbb{F}|^{(2t+2)t} = |\mathbb{F}|^{2t(t+1)} \ .$$

□

Using the above lemmas, we prove the following theorem. The efficiency of the protocol is analyzed in Appendix A.2.

Theorem 4.9. *The pair* (Share, Reconstruct) *is a packed AVSS protocol resilient to $t < \frac{n}{4}$ Byzantine parties.*

Proof. We will prove each property individually.

Correctness. Observe the time the first honest party completes the Share protocol. By that time it has updated p and q to non-$\perp$ values. From Lemma 4.6, some party i sent a "star" message. Let i be that party, and C_i, D_i, E_i, F_i be the values sent. In addition, let G be the polynomial defined in Lemma 4.4 for party i, and define $r_k = G(-k, 0)$ for every $k \in \{0, \dots, t\}$.

For the first part of the correctness property, we will show that if the dealer is honest, $r_k = s_k$ for every $k \in \{0, \dots, t\}$. The dealer starts by uniformly sampling a polynomial $S(X, Y)$ of degree

$2t$ in X and t in Y such that $\forall k \in \{0, \dots, t\}$ $S(-k, 0) = s_k$. It then sends every party j a $\langle$"polynomials", $f_j, g_j\rangle$ message with $f_j(X) = S(X, j)$ and $g_j(Y) = S(j, Y)$. Upon receiving this message, every honest party saves these polynomials in its f_j, g_j fields. The polynomial $G(X, Y)$ defined in Lemma 4.4 is the unique bivariate polynomial of degree $2t$ in X and t in Y such that for every honest $k \in C_i, l \in D_i$, $f_k(l) = g_l(k), g_k(l) = f_l(k)$. Since S is such a polynomial, $S = G$, and thus also $s_k = S(-k, 0) = G(-k, 0) = r_k$ for every $k \in \{0, \dots, t\}$.

Now assume that some honest party i outputs r_k from $\mathsf{Reconstruct}_i(k)$ for some $k \in [n]$. It does so after successfully interpolating the polynomial q_{-k} by computing $\mathsf{RobustInt}(\mathsf{points}_i, t, t)$. Party i adds tuples (j, y_j) to points_i after receiving a $\langle$"rec", $k, y_j\rangle$ message from j, and honest parties send such messages with $y_j = p_j(-k)$. From Lemma 4.6, $p_j(X) = G(X, j)$ for every such honest j, and thus points_i contains at most t tuples (j, y_j) such that $y_j \neq G(-k, j)$. As in the above arguments, $G(-k, Y)$ is a polynomial of degree t or less such that for all but t tuples $(j, y_j) \in \mathsf{points}_i$ $y_j = G(-k, j)$ and $q_{-k}(Y)$ is the unique polynomial for which this holds, and thus $q_{-k}(Y) = G(-k, Y)$. Party i then outputs $q_{-k}(0) = G(-k, 0) = r_k$, as required.

Termination. For the first part of the property, assume the dealer is honest. In that case, it starts by sampling a bivariate polynomial $S(X, Y)$ of degree $2t$ in X and t in Y and sends every party i the polynomials $f_i(X) = S(X, i), g_i(Y) = S(i, Y)$. Every party i receives these polynomials, sees that they are of the correct degree, and sends every party j a $\langle$"values", $f_i(j), g_(j)\rangle$ message. Upon receiving that message, j sees that $f_i(j) = S(j, i) = g_j(i), g_i(j) = S(i, j) = f_j(i)$, adds $(i, f_i(j))$ to points, and sends an $\langle$"ok", $i\rangle$ message to all parties. This means that every two honest parties j, k send "ok" messages about each other. Therefore, every honest party i eventually adds an edge (j, k) to its edges_i set for every honest parties j, k. After party i does so, the graph $([n], \mathsf{edges}_i)$ has a clique of size $n - t$ consisting of all honest parties, and thus eventually i computes C_i, D_i using $\mathsf{FindStar}$ such that $|C_i| \geq n - 2t$ and $|D_i| \geq n - t$. In addition, C_i contains at least $n - 2t$ parties from that clique of honest parties. Following that, computing S_i, i sees that every honest party has an edge to every honest party in C_i and thus has at least $n - t$ neighbors in C_i. Therefore, every honest party is added to S_i. Similarly, every honest party has $n - t$ honest neighbors in S_i and is thus also added to E_i. Following that, S_i, E_i will both be of size $n - t$, and thus i will send a "star" message to all parties if it hasn't done so earlier. Every honest party i will eventually receive a $\langle$"star", $C_j, D_j, E_j, F_j\rangle$ message from every honest j and send a "done" message to all parties if it hasn't done so before. In addition, as stated above, i eventually has a tuple $(k, f_k(i)) \in \mathsf{points}_{g,i}$ for every honest $k \in E_j$. After that, it will successfully interpolate these points to $g_{i,j}(Y) = S(i, Y)$ for similar reasons as the ones above and add the tuple $(j, S(i, Y))$ to $\mathsf{interpolated}_i$. After doing so for $t + 1$ honest parties, i will have $t + 1$ tuples of that form in $\mathsf{interpolated}_i$, update q_i to $S(i, Y)$ and send $\langle$"col", $q_i(j)\rangle$ to every j. Finally, every honest party i eventually receives such a $\langle$"col", $q_j(i)\rangle$ message from every honest j with $q_j(i) = S(j, i)$ and adds a tuple $(j, S(j, i))$ to $\mathsf{points}_{p,i}$. It then successfully interpolates its $\mathsf{points}_{p,i}$ set to the polynomial $p_i(X) = S(X, i)$. At that point, it received "done" messages from $n - t$ parties and has $p_i \neq \bot, q_i \neq \bot$, and completes the Share protocol.

For the second part of the property, assume that some honest party completes the Share protocol. It does so after having its p and q polynomials not equal $\bot$ and receiving $n - t$ "done" messages, with at least one of those being sent by an honest party. The first honest party that sent such a message may have received "done" messages from at most t Byzantine parties at that time, so it sent the "done" message as a result of having received $n - t$ "star" messages and adding the received values to its stars set. Out of these messages, at least $n - 2t$ are sent by honest parties, so every honest party receives those messages as well and adds a tuple (j, C_j, D_j, E_j, F_j) to its stars set. From Lemmas 4.4 and 4.5 all of those messages define the same bivariate polynomial $G(X, Y)$ of degree $2t$ in X and t in Y. Following the same arguments as the ones for the previous property, every honest i party eventually interpolates the polynomial $g_{i,j}(Y) = G(i, Y)$ for every $(j, C_j, D_j, E_j, F_j) \in \mathsf{stars}_i$ such that j is honest and adds $(j, G(i, Y))$ to $\mathsf{interpolated}_i$. In addition, $\mathsf{interpolated}_i$ contains at most t tuples $(j, g_{i,j}(Y))$ such that $g_{i,j}(Y) \neq G(i, Y)$. Therefore, after adding a tuple to $\mathsf{interpolated}_i$ for the $n - 2t$ honest parties from which it received "star" messages, i sees that there are $t+1$ tuples tuples of the form $(j, G(i, Y))$ in $\mathsf{interpolated}_i$ and updates $q_i(Y)$ to $G(i, Y)$. Now following the exact same argument as above, every honest party eventually updates $p_i(X)$ to $G(X, i)$. In addition, as stated above, the honest party that completed the protocol received $n - t$ "done" messages, and thus at least $n - 2t \geq t + 1$ of those messages were sent by honest parties. Every honest party receives those messages as well and sends a "done" message as well. This means that every honest i eventually has $p_i \neq \bot, q_i \neq \bot$ and receives "done" messages from at least $n - t$ parties, and thus it completes the Share protocol as well.

For the final part of the property, assume that all honest parties completed the Share protocol and called $\mathsf{Reconstruct}(k)$. Before completing the Share protocol, they each have $p_i \neq \bot$ and $q_i \neq \bot$. From Lemma 4.6, there exists a bivariate polynomial $G(X, Y)$ of degree $2t$ in Y and t in X such that for every honest i, $p_i(X) = G(X, i)$ and $q_i(Y) = G(i, Y)$. Every honest party j starts $\mathsf{Reconstruct}_j(k)$ by sending a $\langle \text{"rec"}, k, p_i(-k)\rangle$ with $p_j(-k) = G(-k, j)$. Every honest i receives those messages and adds a tuple (j, y_j) to points_i for each. After receiving messages from $2t + 1$ honest parties, points_i contains $2t+1$ tuples (j, y_j) such that $y_j = G(-k, j)$ and at most t additional tuples for which this doesn't hold. The polynomial $G(-k, Y)$ is the unique polynomial of degree t or less for which this is the case, so $\mathsf{RobustInt}$ outputs $q_{-k}(Y) = G(-k, Y)$. At that points i outputs $q_{-k}(0)$ and terminates.

Secrecy. Assume the dealer is honest and observe the adversary's view before some honest party calls $\mathsf{Reconstruct}(k)$ for some $k \in \{0, \dots, t\}$. By that time, the adversary could gain information of f_i and g_i for every Byzantine i directly from the dealer. Furthermore, every honest j sends every Byzantine i the points $f_j(i)$ and $g_j(i)$. However, $f_j(i) = S(j, i) = g_i(j)$ and $g_j(i) = S(i, j) = f_i(j)$, so these points do not reveal any information that f_i and g_i haven't already revealed. In addition, the honest parties might also send "ok", "star", and "done" messages with information that is independent of S. Furthermore, they also send "col" messages, and as shown in the proof of correctness these messages also contain the values $g_j(i) = f_i(j)$. Finally, the honest parties might have already participated in $\mathsf{Reconstruct}(l)$ for every $l \in \{0, \dots, t\} \setminus \{k\}$. During these calls, every honest j sends the point $f_j(-l) = S(j, -l)$. Define the polynomial $g_{-l}(X) = S(X, -l)$. In that case, the honest parties send points on the polynomial g_{-l}, and thus the adversary could also gain enough information to learn the polynomials g_{-l} for every $l \in \{0, \dots, t\} \setminus \{k\}$.

Let the set of Byzantine parties be B and $I = \{-0, \dots, -t\} \setminus \{-k\}$. In that case, for every $i \in B$ and $j \in B \cup I$, $f_i(j) = g_j(i)$. Using the exact same arguments as the ones in Lemma 4.7, for every possible $s_k \in \mathbb{F}$, there exists a unique polynomial S which is consistent with all of these polynomials and for which $S(-k, 0) = s_k$. From Lemma 4.8, for every set of secrets $s_0, \dots, s_t$, the dealer samples a polynomial S uniformly from a set with $p = |\mathbb{F}|^{2t(t+1)}$ possibilities, and thus the probability of the adversary having a particular view is exactly $\frac{1}{p}$ regardless of the final secret s_k. Note that this discussion only dealt with the most information the adversary can gain throughout the protocol. The fact that there exists a unique polynomial given this view and that the dealer samples the polynomial S uniformly implies that in that case the adversary's view is sampled uniformly and independently from the final secret s_k. A uniform distribution on the most information the adversary could learn also induces a uniform distribution on any lesser amount of information the adversary might gain, meaning that the adversary's view is always distributed uniformly and independently of the final secret s_k. □

4.5 Additional Reconstruction Protocols

We are interested in protocols that also allow for reconstructing sums of secrets. That is, packed AVSS schemes with an additional $\mathsf{Sum-Reconstruct}$ protocol that takes an input $\mathsf{dealers} \subseteq [n]$ and an input $k \in \mathbb{N}$ and outputs a value. Parties call $\mathsf{Sum-Reconstruct}(k, \mathsf{dealers})$ only after having completed the Share protocol with i as dealer for every $i \in \mathsf{dealers}$. The $\mathsf{Sum-Reconstruct}$ protocol has the following two properties:

- **Correctness.** If an honest party completes $\mathsf{Sum-Reconstruct}(k, \mathsf{dealers})$ for some $k \in [m]$, then its output is $\sum_{i \in \mathsf{dealers}} r_{i,k}$, with $r_{i,k}$ being the value r_k defined for the dealer i in the Correctness property above.
- **Termination.** If all honest parties call $\mathsf{Sum-Reconstruct}(k, \mathsf{dealers})$, then they all complete the protocol.

It is possible to trivially construct a packed AVSS protocol from an AVSS protocol (i.e. a protocol with $m = 0$) by simply sharing each value independently. However, some protocols share several values more efficiently than sharing them each one independently. In addition, it is possible to reconstruct the sum $\sum_{i \in \mathsf{dealers}} r_{i,k}$ by simply reconstructing each of the $r_{i,k}$ values individually and then summing the outputs. However, some protocols allow for more efficient reconstruction of sums as well.

Two additional reconstruction protocols are provided for efficient reconstruction of all secrets and of sums of secrets in Protocol 4.10 and Protocol 4.11. The below protocols have a total word complexity of $\mathcal{O}(n^2)$ for reconstructing either a sum or all secrets of a single dealer. As shown above, the sharing protocol has a word complexity of $\mathcal{O}(n^3)$. This means that when reconstructing a sum of $\mathcal{O}(n)$ secrets, as done above, simply reconstructing each secret individually has a cost of $\mathcal{O}(n^3)$ words, which is efficient enough to not change the asymptotic behavior of the protocol. Similarly, reconstructing each secret individually for batch reconstruction also doesn't incur additional costs. These protocols are still provided for completeness and can be useful when parties need to reconstruct sums without revealing each element in the sum individually, or when they need to reconstruct a full degree $2t$ polynomial with the embedded secrets. In the description of $\mathsf{Sum-Reconstruct}$, $p_{i,j}$ is defined to be the field p_i in the Share invocation with j as dealer.

Protocol 4.10: $\mathsf{Sum-Reconstruct}_i(k, \mathsf{dealers})$

```
1: $\mathsf{points}_i \leftarrow \emptyset$
2: send $\langle \text{“rec”}, k, \sum_{j \in \mathsf{dealers}} p_{i,j}(-k) \rangle$
3: **upon** receiving a $\langle \text{“rec”}, k, y_j \rangle$ message from $j$, **do**
4:     $\mathsf{points}_i \leftarrow \mathsf{points}_i \cup \{(j, y_j)\}$
5:     $q_{-k} \leftarrow \mathsf{RobustInt}(\mathsf{points}_i, t, t)$
6:     **if** $q_{-k} \neq \bot$ **then**
7:         **output** $q_{-k}(0)$ and **terminate**
```

Protocol 4.11: $\mathsf{Batch-Reconstruct}_i()$

```
1: $\mathsf{points}_i \leftarrow \emptyset$
2: send $\langle$"rec", $k, q_i(0)\rangle$
3: upon receiving a $\langle$"rec", $k, y_j\rangle$ message from $j$, do
4:     $\mathsf{points}_i \leftarrow \mathsf{points}_i \cup \{(j, y_j)\}$
5:     $p_0 \leftarrow \mathsf{RobustInt}(\mathsf{points}_i, 2t, t)$
6:     if $p_0 \neq \bot$ then
7:         output $p_0(0), \ldots, p_0(-k)$ and terminate
```

In $\mathsf{Sum-Reconstruct}(k)$, parties output $\sum_{j \in \mathsf{dealers}} r_{k,j}$, where $r_{k,j}$ is the value r_k defined in the invocation of Share with j as dealer. In $\mathsf{Batch-Reconstruct}$, parties output the entire set of secrets $r_0, \ldots, r_t$. The proof that this is the case follows the same reasoning as the one of the correctness property of the original protocol, and is thus omitted. In addition, from counting the above messages, one can see that the total word complexity is $\mathcal{O}(n^2)$ since parties only send a single message with a field element.

5 Verifiable Party Gather

5.1 Definition

Verifiable Party Gather is a variation of the Verifiable Gather protocol of [5]. The first difference is that we only output a set of parties (no values) and the second difference is that the cryptographic external validity function is replaced by an information theoretic asynchronous validity predicate as defined in Section 3.2 that takes as input a party index. Intuitively, the goal of a *party gather* protocol is to have some common *core of parties* such that each honest party outputs a set of parties that is a super-set of this core. Intuitively, the goal of a *verifiable party gather* protocol is to make sure that the set of parties that are output by an honest party can be verified to be correct outputs of the protocol. Observe that different parties may output different super-sets of the core and there is no agreement on who is in the core.

Formally, a verifiable party gather protocol consists of a pair of protocols $(\mathsf{Gather}, \mathsf{Verify})$ and takes as input a validity predicate $\mathsf{validate} : [n] \to \{0, 1\}$. For Gather, each party i has a set of parties $S_i \subseteq [n]$ as input such that $\mathsf{validate}_i(x) = 1$ for every $x \in S_i$ at the time i calls the protocol. Each party may decide to *output* a set X_i. After outputting sets X_i, parties must continue to update their local state according to the Gather protocol in order for the verification protocol to continue working. The properties of Gather:

- **Binding Core.** Once the first honest party outputs a value from the Gather protocol there exists a core set X^* such that $|X^*| \geq n - t$, and, if an honest party j outputs the set X_j, then $X^* \subseteq X_j$.
- **Termination of Output.** All honest parties eventually output a set of indices.

The Verify protocol receives a set $X \subseteq [n]$ and can either terminate with the output 1 signifying that X was verified, terminate with the output 0 signifying that X wasn't verified, or not terminate at all. The verification protocol only allows the adversary to report sets of parties that contain the binding core X^*, or not pass verification. A party i can check any set X, which we denote by executing $\mathsf{Verify}_i(X)$. If the execution of $\mathsf{Verify}_i(X)$ terminates and outputs 1, we say that i has verified the set X. The termination properties of Verify:

- **Completeness.** For any two honest parties i, j, if j outputs X_j from Gather, then $\mathsf{Verify}_i(X_j)$ eventually terminates with the output 1.
- **Agreement on Verification.** For any two honest i, j, and any set X, if $\mathsf{Verify}_i(X)$ terminates with the output $b \in \{0, 1\}$, then $\mathsf{Verify}_j(X)$ eventually terminates with the same output.

The correctness properties of the Verify protocol:

- **Includes Core.** If $\mathsf{Verify}_i(X)$ terminates with the output 1 for some honest i, then $X^* \subseteq X$ (for the core X^* defined in the Binding Core property of Gather).
- **Validity.** If $\mathsf{Verify}_i(X)$ terminates with the output 1 for some honest i, then for each $x \in X$, $\mathsf{validate}_i(x) = 1$ at the time i output X.

Combining the Includes Core and Completeness properties we can see that all honest parties output sets that contain X^*.

As part of our verifiable leader election protocol, we require a "verifiable party gather". In the leader election protocol, we want to agree on a large set of parties that actively participated and elect a leader among them. However, exactly agreeing on the set is non-trivial and potentially expensive. Therefore we slightly relax our requirements: there exists some core C of size $n - t$ or greater such that the output of every honest party contains C. Furthermore, we would like parties to be able to prove that they "acted correctly" and included C in their output.

5.2 The Protocol

The protocol takes place in two rounds. In the beginning, all parties broadcast their inputs $S_i \subseteq [n]$, which are sets of parties. We also assume that all parties have access to an asynchronous validity predicate $\mathsf{validate}$ and that for every honest i, $\mathsf{validate}_i(x) = 1$ for every $x \in S_i$ at the time it calls the Gather protocol. After an honest P_i receives such a set S_j from party P_j, it waits to see that $\mathsf{validate}_i(x) = 1$ for every $x \in S_j$. When this condition holds, P_i records P_j as a party from whom it received a set and stores j in T_i. In addition, it adds all indices in S_i to its eventual output value, R_i. In the second round, P_i sends its set T_i once its size is at least $n - t$. When receiving a set T_j from party P_j, i waits until it sees that $T_j \subseteq T_i$. After seeing that this is the case for $n - t$ parties, i terminates and outputs R_i.

In order to be able to verify reported values, when i accepts a set T_j from j, it also stores all parties which j should have seen in S sets before sending T_j. In other words, it also stores $(j, \cup_{k \in T_j} S_k)$ in a set U_i used for verification. In the discussion below, we show that there exists some index i^* that is included in at least $t + 1$ of the T sets broadcasted by parties. Since every party waits to receive T sets from at least $n - t$ parties before terminating, it will see at least one with that index, and thus include S_{i^*} in its output. This is true for any honest party, so S_{i^*} can serve as a common core in the output of all honest parties. Similarly, when verifying a set X, i makes sure that it contains the values referenced by the T sets received from at least $n - t$ parties, and thus also includes S_{i^*} in it.

Protocol 5.1: $\mathsf{Gather}_i(S_i)$

```
1: $R_i \leftarrow \emptyset, T_i \leftarrow \emptyset, U_i \leftarrow \emptyset$
2: **broadcast** $\langle 1, S_i \rangle$
3: **upon** receiving $\langle 1, S_j \rangle$ from $j$ such that $|S_j| \geq n - t$, **do**
4:     **upon** $\mathsf{validate}_i(x)$ terminating with output 1 for every $x \in S_j$, **do**
5:         $R_i \leftarrow R_i \cup S_j$
6:         $T_i \leftarrow T_i \cup \{j\}$
7:         **if** $|T_i| = n - t$ **then**
8:             **broadcast** $\langle 2, T_i \rangle$                    ▷ $T$ sets reference $S$ sets
9: **upon** receiving $\langle 2, T_j \rangle$ from $j$ such that $|T_j| \geq n - t$, **do**
10:    **upon** $T_j \subseteq T_i$, **do**                    ▷ relevant $S$ sets and values are received
11:        $U_i \leftarrow U_i \cup \{(j, \bigcup_{k \in T_j} S_k)\}$          ▷ save all parties in the $S$ sets referenced by $T_j$
12:        **if** $|U_i| = n - t$ **then**
13:            **output** $R_i$, but continue updating internal sets and sending messages
```

Protocol 5.2: $\mathsf{GatherVerify}_i(X)$

```
1: **upon** $|\{j | \exists (j, V_j) \in U_i, V_j \subseteq X\}| \geq n - t$ and $\mathsf{validate}_i(x)$ terminating with the output 1 for
   every $x \in X$, **do**
2:     **output** 1 and **terminate**
```

5.3 Security Analysis

We start by proving that many parties send T sets with an index i^*, which will later be used for defining the common core of the protocol. We then show that parties eventually have consistent views, and conclude with proving that (Gather, $\mathsf{GatherVerify}$) is a Verifiable Party Gather protocol in Theorem 5.5. The efficiency of the protocol is analyzed in Appendix A.1.

Lemma 5.3. *Assume some honest party completed the protocol. There exists some i^* such that at least $t+1$ parties sent broadcasts of the form $\langle 2, T \rangle$ with $i^* \in T$.*

Proof. Assume some honest party completed the protocol. Before completing the protocol, it found that $|U_i| \geq n-t$, and thus it received $n-t$ broadcasts of the form $\langle 2, T_j \rangle$ such that $|T_j| \geq n-t$. Let I be the set of parties who sent those broadcasts. Now assume by way of contradiction that every index k appears in at most t of the broadcasted sets T_j such that $j \in I$. Since there are a total of n possible values, this means that the total number of elements in all sets is no greater than nt. On the other hand, there are $n-t$ such sets, each containing $n-t$ elements or more, resulting in at least $(n-t)^2$ elements overall. Combining these two observations:

$$
\begin{aligned}
(n-t)^2 &\leq nt \\
n^2 - 2nt + t^2 &\leq nt \\
n^2 - 3nt + t^2 &\leq 0
\end{aligned}
$$

However, by assumption $n > 3t$, and thus:

$$\begin{aligned} 0 &\geq n^2 - 3nt + t^2 \\ &= n^2 - n \cdot (3t) + t^2 \\ &> n^2 - n^2 + t^2 \\ &= t^2 \geq 0 \end{aligned}$$

reaching a contradiction. Therefore, there exists at least one value i^* such that for at least $t+1$ of the $\langle 2, T \rangle$ broadcasts sent, $i^* \in T$. $\square$

Lemma 5.4. *Let i, j be two honest parties. Observe the sets T_i, U_i at any time throughout the protocol. Eventually $T_i \subseteq T_j$ and $U_i \subseteq U_j$.*

Proof. Observe some $k \in T_i$. Party i added k to T_i after receiving a $\langle 1, S_k \rangle$ broadcast from k such that $|S_j| \geq n - t$ and seeing that $\mathsf{validate}_i(x) = 1$ for every $x \in S_k$. From the Agreement and Termination properties of the broadcast protocol, j eventually receives the same message, and from the Consistency property of the asynchronous validity predicate, it will eventually see that $\mathsf{validate}_i(x) = 1$ for every $x \in S_k$. At that point, j will add k to T_j as well. Similarly, observe some $(k, V_k) \in U_i$. Party i received a $\langle 2, T_k \rangle$ message such that $|T_k| \geq n - t$ and saw that $T_k \subseteq T_i$. Party j will also receive that same message and eventually see that $T_k \subseteq T_i \subseteq T_j$, at which point it will add a tuple (k, V'_k) to U_i. Note that both parties compute V_k and V'_k to be the union of the S_l sets such that $l \in T_k$. Since both of them receive the same broadcasts $\langle 1, S_l \rangle$, they both compute the same set, and thus j adds the same tuple $(k, V'_k) = (k, V_k)$ to its U_j set. $\square$

Theorem 5.5. *The pair $(\mathsf{Gather}, \mathsf{GatherVerify})$ is a verifiable reliable gather protocol resilient to $t < \frac{n}{3}$ Byzantine parties.*

Proof. Each property is proven separately.

Termination of Output. Assume that for every honest i and for every $x \in S_i$ $\mathsf{validate}_i(x) = 1$ at the time i calls Gather. Every honest party i starts the Gather protocol by broadcasting $\langle 1, S_i \rangle$. Every honest j receives that message and from the Consistency property of the asynchronous validity predicate eventually sees that $\mathsf{validate}_j(x) = 1$ as well for every $x \in S_i$. At that point j adds i to T_j. After adding such an index for every honest party, j sees that $|T_j| = n - t$ and broadcasts $\langle 2, T_j \rangle$. Similarly, every honest party k eventually receives that broadcast and sees that $|T_j| \geq n - t$. From Lemma 5.4, eventually $T_k \subseteq T_j$, at which point k adds a tuple (j, V_j) to U_k. After adding such a tuple for every honest j, every honest k sees that $|U_k| \geq n - t$, outputs R_k and terminates.

Completeness. Assume some honest party i completes the Gather protocol and outputs R_i. At the time i output R_i, it found that $|U_i| = n - t$. We will start by showing that at that time $\cup_{(j,V_j) \in U_i} V_j \subseteq R_i$. Before adding (j, V_j) to U_i, i received a $\langle 2, T_j \rangle$ broadcast from j and saw that $T_j \subseteq T_i$. It then added $(j, \cup_{k \in T_j} S_k)$ to U_i. Similarly, before adding $k \in T_j$ to T_i, i received a $\langle 1, S_k \rangle$ broadcast from k and updated R_i to $R_i \cup S_k$. In other words, for every $k \in T_j$, $S_k \subseteq R_i$ and since $V_j = \cup_{k \in T_j} S_k$, also $V_j \subseteq R_i$. Let j be some honest party that called $\mathsf{GatherVerify}_j(R_i)$. From Lemma 5.4, eventually $U_i \subseteq U_j$. At that time, for every $(k, V_k) \in U_i \subseteq U_j$, $V_k \subseteq R_i$. When i completes the Gather protocol, $|U_i| = n - t$ and thus j will eventually see that $|\{k | \exists (k, V_k) \in U_j, V_k \subseteq R_i\}| \geq n - t$. In addition, i only adds elements to R_i by updating R_i to $R_i \cup S_k$ after seeing that $\mathsf{validate}_i(x) = 1$ for every $x \in S_k$. From the Consistency property of

$\mathsf{validate}$, for every $x \in S_k$ eventually $\mathsf{validate}_j(x) = 1$ as well. Therefore, j will eventually see that all of the conditions of the $\mathsf{GatherVerify}$ protocol hold, output 1, and terminate.

Agreement on Verification. Assume that some honest i completes the $\mathsf{GatherVerify}_i(X)$ protocol on some set X and outputs $b \in \{0,1\}$. Honest parties never output 0 from the $\mathsf{GatherVerify}$ protocol, and thus i output 1. This means that it saw that $|\{k|\exists(k,V_k) \in U_i, V_k \subseteq X\}| \geq n-t$ and that for every $x \in S_k$, $\mathsf{validate}_i(x) = 1$. Let j be some honest party that called $\mathsf{GatherVerify}_j(X)$. From Lemma 5.4, eventually $U_j \subseteq U_i$ and thus eventually $|\{k|\exists(k,V_k) \in U_j, V_k \subseteq X\}| \geq n-t$. In addition, from the Consistency property of $\mathsf{validate}$, j will also eventually see that $\mathsf{validate}_j(x) = 1$ for every $x \in X$. After both of those conditions hold, j outputs 1 from the $\mathsf{GatherVerify}$ protocol and terminates.

Binding Core. Assume the first honest party that completes the Gather protocol is j, and observe the index i^* as defined in Lemma 5.3. Party j only adds a tuple (k, V_k) to U_j after receiving a $\langle 2, T_k\rangle$ message from party k. Before completing the protocol, j received $n-t$ such broadcasts and found that $T_k \subseteq T_j$. From Lemma 5.3, $t+1$ of the parties broadcast some message $\langle 2, T_k\rangle$ such that $i^* \in T_k$. Therefore, for at least one party k, $i^* \in T_k \subseteq T_j$. Before adding i^* to T_j, pj received a $\langle 1, S_{i^*}\rangle$ broadcast from party i^* such that $S_{i^*} \subseteq S_j$ and $|S_{i^*}| \geq n-t$. Let the binding-core X^* be S_{i^*}. Clearly $|X^*| \geq n-t$ because $|S_{i^*}| \geq n-t$. The fact that X^* is a subset of every honest party's output from the protocol is a direct corollary of the Completeness and Includes Core properties of the Gather protocol.

Include Core. Let i be some honest party and X be some set such that $\mathsf{GatherVerify}_i(X)$ terminates with the output 1. Party i found that $|\{k|\exists(k,V_k) \in U_j, V_k \subseteq X\}| \geq n-t$. As discussed above, party i only adds (j, V_j) to U_i after receiving a $\langle 2, T_j\rangle$ message from j. Let i^* be defined as it is in Lemma 5.3 and in the Binding Core property. Seeing as there are at least $t+1$ parties that sent broadcasts of the form $\langle 2, T\rangle$ with $i^* \in T$ and $n-t$ parties j such that $(j, V_j) \in U_i$ and $V_j \subseteq X$, for at least one of those parties $i^* \in T_j$. By definition, $V_j = \bigcup_{k \in T_j} S_k$, and thus $S_{i^*} \subseteq V_j \subseteq X$, as required.

Validity. Assume that for some honest i, $\mathsf{GatherVerify}_i(X)$ terminates with the output 1. Before doing so, i checks that $\mathsf{validate}_i(x) = 1$ for every $x \in X$. $\square$

6 Verifiable Leader Election

We now turn our attention to verifiable leader election. The protocol uses the AVSS protocol described and constructed in Section 4, and a Verifiable Party Gather protocol, described and constructed in Section 5.

6.1 Definition

A perfect leader election would allow all parties to output one common randomly elected party. *Verifiable Leader Election* (VLE) is an asynchronous protocol that tries to capture this spirit but obtains weaker properties. Intuitively, there is only a constant probability all parties elect the same honest party. As in the Verifiable Party Gather protocol, we also add a verification protocol. Crucially, in the good event mentioned above, the only value that passes verification is this commonly elected leader. In the remaining cases, the adversary can control the output and even cause different parties to have different outputs. However, even in these cases, all parties eventually output some verifying value.

We assume a validity predicate $\mathsf{validate} : [n] \to \{0,1\}$ that given any index $k \in [n]$ can check the validity of k. A Verifiable Leader Election protocol consists of a pair of protocols $(\mathsf{VLE}, \mathsf{Verify})$. When an honest i calls the VLE protocol, $\mathsf{validate}_i(i) = 1$ already holds. The output of the VLE protocol is a pair (ℓ, π) where $\ell \in [n]$ and π is a proof used in the Verify protocol. We model these protocols as having some ideal write-once state ℓ^*. We assume $\perp$ is not valid and let $\ell^* \in [n] \cup \{\perp\}$. Intuitively, if $\ell^* \neq \perp$ then the output of all parties will be ℓ^*, but when $\ell^* = \perp$ then the adversary can cause different parties to output different verifying values.

- α-**Binding**. For any adversary strategy, with probability α, ℓ^* is set to be the index of a party that behaved in an honest manner when it started the VLE protocol.

In addition, the VLE protocol has a natural termination property:

- **Termination of Output.** All honest parties eventually output a pair (ℓ, π).

A party i can check any pair of index and proof, (ℓ, π), which we denote by executing $\mathsf{Verify}_i(\ell, \pi)$. If $\mathsf{Verify}_i(\ell, \pi)$ terminates with the output 1, we say that i has verified ℓ. If the binding value ℓ^* is not $\perp$, then the only value for which the verify protocol can terminate and output 1 is ℓ^*. This limits the adversary to either reporting ℓ^* or remaining silent. The termination properties of Verify:

- **Completeness.** For any two honest parties i, j, the output (ℓ, π) of party j from VLE will eventually be verified by party i, i.e. $\mathsf{Verify}_i(\ell, \pi)$ eventually terminates with the output 1.
- **Agreement on Verification.** For any two honest parties i, j, and any index ℓ and proof π, if $\mathsf{Verify}_i(\ell, \pi)$ terminates with the output $b \in \{0,1\}$ $\mathsf{Verify}_j(\ell, \pi)$ eventually terminates with the same output.

Finally, the correctness properties of Verify:

- **Binding Verification.** If $\ell^* \neq \perp$ then for every honest party i, and every (ℓ, π), if $\mathsf{Verify}_i(\ell, \pi)$ terminates with the output 1, then $\ell = \ell^*$.
- **Validity.** If $\mathsf{Verify}_i(\ell, \pi)$ terminates with the output 1 for some honest i, then $\mathsf{validate}_i(\ell) = 1$ at the time $\mathsf{Verify}_i(\ell, \pi)$ terminated.

6.2 Overview

Informally, parties participate in a Verifiable Leader Election (VLE) protocol and elect some leader. The goal is that with a constant probability, all honest parties output an honest leader. With the remaining probability, parties might not agree on the leader's identity or elect a dishonest leader. In any one of the cases, every party's output must be an asynchronously validated leader according to an asynchronous validity predicate $\mathsf{validate}$. Every honest party starts the protocol believing it is a valid leader, i.e., with $\mathsf{validate}_i(i) = 1$. In addition, parties can verify each other's output with a verification protocol, $\mathsf{VLEVerify}$. If a single honest leader is elected, that should be the only party that can be verified. This means that if parties inform each other of their elected leaders, even corrupt parties won't be able to send any other party's index and convince the honest parties that it is a valid output from the protocol. For a formal definition of a VLE protocol, see Section 6.1.

Overview. Our construction uses techniques inspired by synchronous weak leader election [26], and cryptographic proposal election [5]. The protocol proceeds in 5 rounds described below:
Round 1: In the first round, every party shares n random values using a packed AVSS protocol, one for each party. Parties then participate in the packed AVSS instances with every party as a dealer.
Round 2: In the second round, after completing the Share protocol for $t+1$ dealers, party i broadcasts an "attach" message with the set $\mathsf{dealers}_i$ for which it completed the sharing protocol. After receiving such a message from party j with a set $\mathsf{dealers}_j$, i checks that it also completed the Share protocol for the dealers in $\mathsf{dealers}_j$ and waits until it considers j to be valid according to $\mathsf{validate}_i(j)$. That is, it checks that j is a valid leader that has committed to a random value, which is the sum of the j'th secrets shared by the dealers in $\mathsf{dealers}_j$.
Round 3: In the third round, i waits to see that $n-t$ parties are committed to their random values and then inputs the set of those parties, $\mathsf{attached}_i$, to the Gather protocol. It does so with an asynchronous validity predicate checking that each party in $\mathsf{attached}_i$ is a valid candidate committed to a random secret.
Round 4: After completing the Gather protocol, i outputs a set of parties that it considers to be viable candidates who can be chosen as leaders and output from the VLE protocol. In order to be able to choose a single leader, i broadcasts a "candidates" message with that set of candidates, asking parties to help reconstruct their attached random value.
Round 5: After receiving a "candidates" message from party j with a set $\mathsf{candidates}_j$, i checks that $\mathsf{candidates}_j$ is a valid output from the Gather protocol by calling the GatherVerify protocol. After the GatherVerify protocol returns 1 on the set $\mathsf{candidates}_j$, i starts reconstructing the sum of the k'th secrets shared by dealers in $\mathsf{dealers}_k$ for every $k \in \mathsf{candidates}_j$. In other words, it helps reconstruct the random value for each candidate. Note that parties only start reconstructing the secrets associated with a party k after seeing that k broadcasted its set of dealers. Since the set of dealers must be at least of size $t+1$, one of those secrets was shared by an honest dealer. This guarantees that the sum will be completely random and unknown to k, who has yet to see any k'th secret reconstructed.
Output: Finally, after reconstructing the random values associated with each of its own candidates, i outputs the candidate with the highest random value. As proof, it also outputs the set of candidates $\mathsf{candidates}_i$.

Intuitively, every party outputs a set of candidates from the Gather protocol who have already committed to their random value. If the party with the maximal random value happens to be an honest party ℓ^* in the binding core of the Gather protocol, then all honest parties will see that random value and pick ℓ^* as their output. Since the values are sampled uniformly in an unbiased manner, this means that every party has the same probability of having the maximal evaluation associated with it. When counting the number of honest parties in the common core, we find that the probability of the aforementioned event is at least $\frac{1}{3}$. This mechanism also allows to check whether a given proposal could have been the correct output from the VLE protocol.

To convince an honest party that a value is a correct output from the VLE protocol, parties can provide their output from the Gather protocol. Parties will then be able to check if that set of parties is a possible output from Gather (i.e., if it contains the core) and if the correct leader was elected based on that set. If the maximal random value is associated with a party in the core, then only sets containing that party will verify, which means that only the honest leader ℓ^* will verify.

6.3 The Protocol

Protocol 6.1: $\mathsf{VLE}_i()$

```
1: $\mathsf{dealers}_i \leftarrow \emptyset, \mathsf{attached}_i \leftarrow \emptyset, \mathsf{candidates}_i \leftarrow \emptyset, \mathsf{ranks}_i \leftarrow \emptyset$
2: $s_1, \ldots, s_n \xleftarrow{\$} \mathbb{F}$
3: share $s_1, \ldots, s_n$ using a packed AVSS protocol and participate in the PAVSS instances with every party as dealer
4: upon completing all Share calls with $j$ as dealer, do
5:     $\mathsf{dealers}_i \leftarrow \mathsf{dealers}_i \cup \{j\}$
6:     if $|\mathsf{dealers}_i| = t + 1$ then
7:         broadcast $\langle$"attach", $\mathsf{dealers}_i\rangle$
8: upon receiving an $\langle$"attach", $\mathsf{dealers}_j\rangle$ broadcast from $j$, do
9:     upon $\mathsf{dealers}_j \subseteq \mathsf{dealers}_i$, $|\mathsf{dealers}_j| \geq t + 1$ and $\mathsf{validate}_i(j)$ terminating with the output 1, do
10:        $\mathsf{attached}_i \leftarrow \mathsf{attached}_i \cup \{(j, \mathsf{dealers}_j)\}$
11:        if $|\mathsf{attached}_i| = n - t$ then
12:            call $\mathsf{Gather}_i(\{k | \exists(k, \mathsf{dealers}_k) \in \mathsf{attached}_i\})$ with $\mathsf{checkValidity}_i$ as validity predicate
13: upon $\mathsf{Gather}_i$ outputting the set $X_i$, do        ▷ continue updating state according to Gather
14:     $\mathsf{candidates}_i \leftarrow X_i$
15:     broadcast $\langle$"candidates", $\mathsf{candidates}_i\rangle$
16: upon receiving a $\langle$"candidates", $\mathsf{candidates}_j\rangle$ broadcast from $j$, do
17:     upon $\mathsf{GatherVerify}_i(\mathsf{candidates}_j)$ terminating with the output 1 and $\mathsf{Gather}_i$ terminating, do
18:         for all $k \in \mathsf{candidates}_j$ do
19:             call $\mathsf{Sum-Reconstruct}(k, \mathsf{dealers}_k)$ for $(k, \mathsf{dealers}_k) \in \mathsf{attached}_i$
20: upon $\mathsf{Sum-Reconstruct}(j, \mathsf{dealers}_j)$ terminating with the output $r_j$, do
21:     $\mathsf{ranks}_i \leftarrow \mathsf{ranks}_i \cup \{(j, r_j)\}$
22: upon $\mathsf{candidates}_i \neq \bot$ and $\forall j \in \mathsf{candidates}_i \; \exists(j, r_j) \in \mathsf{ranks}_i$, do
23:     $\ell \leftarrow argmax\{r_j | j \in \mathsf{candidates}_i, (j, r_j) \in \mathsf{ranks}_i\}$    ▷ $\ell$ is the party with the highest rank $r_\ell$
24:     $\pi_i \leftarrow \mathsf{candidates}_i$
25:     output $(\ell, \pi_i)$, but continue updating internal sets and sending messages
```

Protocol 6.2: $\mathsf{checkValidity}_i(k)$

```
1: upon there being a tuple of the form $(k, \mathsf{dealers}_k)$ in $\mathsf{attached}_i$, do
2:     output 1 and terminate
```

Protocol 6.3: $\mathsf{VLEVerify}_i(k, \pi)$

```
1: upon $\forall j \in \pi \; \exists(j, r_j) \in \mathsf{ranks}_i$, do
2:     upon $\mathsf{GatherVerify}_i(\pi)$ terminating with the output 1 and $\mathsf{Gather}_i$ terminating, do
3:         $\ell \leftarrow argmax\{r_j | j \in \pi, (j, r_j) \in \mathsf{ranks}_i\}$
4:         if $k = \ell$ then
5:             output 1 and terminate
```

6.4 Security Analysis

We start by proving that parties' views are eventually consistent and that $\mathsf{checkValidity}$, which is used as an asynchronous validity predicate for the Gather protocol, is indeed one. We then prove that $(\mathsf{VLE}, \mathsf{VLEVerify})$ are indeed a Verifiable Leader Election protocol in Theorem 6.6. The efficiency of the protocol is analyzed in Appendix A.3.

Lemma 6.4. *Let i and j be honest parties. Observe the sets $\mathsf{dealers}_i$,$\mathsf{attached}_i$, and ranks_i at any time throughout the protocol. Eventually $\mathsf{dealers}_i \subseteq \mathsf{dealers}_j$, $\mathsf{attached}_i \subseteq \mathsf{attached}_j$ and $\mathsf{ranks}_i \subseteq \mathsf{ranks}_j$.*

Proof. Let k be some index in $\mathsf{dealers}_i$. Party i adds k to $\mathsf{dealers}_i$ after completing all Share calls with k as dealer. From the Termination property of the AVSS scheme, j completes those calls as well and adds k to $\mathsf{dealers}_j$.

Let $(k, \mathsf{dealers}_k)$ be a tuple in $\mathsf{attached}_i$. Party i adds the tuple to $\mathsf{attached}_i$ after receiving an $\langle$"attach", $\mathsf{dealers}_k\rangle$ broadcast from k, seeing that $\mathsf{dealers}_k \subseteq \mathsf{dealers}_i$, that $|\mathsf{dealers}_k| \geq t+1$ and that $\mathsf{validate}_i(k) = 1$. Eventually, j receives the same broadcast and as shown above eventually sees that $\mathsf{dealers}_k \subseteq \mathsf{dealers}_i \subseteq \mathsf{dealers}_j$ and from the Consistency property of $\mathsf{validate}$, $\mathsf{validate}_j(k)$ eventually terminates with the output 1 as well. It then adds $(k, \mathsf{dealers}_k)$ to $\mathsf{attached}_j$.

Finally, let (k, r_k) be a tuple in ranks_i. Party i adds such a tuple to ranks_i for every $(k, \mathsf{dealers}_k) \in \mathsf{attached}_i$ after calling $\mathsf{Sum} - \mathsf{Reconstruct}(k, \mathsf{dealers}_k)$, and the protocol terminating with the output r_k. Before calling the protocol, i receives a $\langle$"candidates", $\mathsf{candidates}_l\rangle$ broadcast from some party l such that $k \in \mathsf{candidates}_l$ and that $\mathsf{GatherVerify}_l(\mathsf{candidates}_l)$ terminates with the output 1. In addition i completes its call to Gather_i, which it called after having $|\mathsf{attached}_i| = n-t$. As shown above, eventually $\mathsf{attached}_j \subseteq \mathsf{attached}_i$, so j will also see that $|\mathsf{attached}_j| = n-t$ at some point. It will then call Gather_j, and from the Termination property of the Gather protocol complete the call to Gather_j as well. In addition, j will receive the same "candidates" broadcast, and from the Agreement on Verification property of $\mathsf{GatherVerify}$, $\mathsf{GatherVerify}_j(\mathsf{candidates}_l)$ will terminate with the output 1, at which point j will call $\mathsf{Sum} - \mathsf{Reconstruct}(k, \mathsf{dealers}_k)$. Finally, from the correctness property of the AVSS protocol, the protocol will terminate with the output r_k and j will add (k, r_k) to ranks_j. □

Lemma 6.5. $\mathsf{checkValidity}$ *is an asynchronous validity predicate.*

Proof. We will show that the predicate has the Finality and Consistency properties.

Finality. Assume that $\mathsf{checkValidity}_i(k)$ terminated with the output b for some honest i. First note that $\mathsf{checkValidity}_i$ never outputs 0 so $b = 1$. In that case, i saw that there exists a tuple of the form $(k, \mathsf{dealers}_k)$ in $\mathsf{attached}_i$. Parties never remove tuples from their $\mathsf{attached}_i$ sets, so i will output 1 in any subsequent calls to $\mathsf{checkValidity}_i(k)$.

Consistency. Assume that $\mathsf{checkValidity}_i(k) = 1$ for some honest k. Since $\mathsf{checkValidity}_i(k) = 1$, i saw that there exists a tuple of the form $(k, \mathsf{dealers}_k)$ in $\mathsf{attached}_i$. From Lemma 6.4, eventually $(k, \mathsf{dealers}_k)$ will be added to $\mathsf{attached}_j$ as well, and $\mathsf{checkValidity}_j(k)$ will terminate with the output 1. □

Theorem 6.6. *The pair $(\mathsf{VLE}, \mathsf{VLEVerify})$ is a Verifiable Leader Election protocol resilient to $t < \frac{n}{4}$ Byzantine parties with $\alpha = \frac{1}{3}$.*

Proof. Each property is proven separately.

Termination of Output. If all honest parties participate in the VLE protocol, then they all sample n values and share them using packed AVSS. From the Termination property of AVSS, every honest party i will complete those calls, add every honest j to $\mathsf{dealers}_i$ and broadcast $\langle$"attach"$, \mathsf{dealers}_i\rangle$ when it $|\mathsf{dealers}_i| = t+1$. After an honest i receives an "attach" message from an honest j, it sees that $|\mathsf{dealers}_j| \geq t+1$. In addition, from Lemma 6.4, eventually $\mathsf{dealers}_j \subseteq \mathsf{dealers}_i$. By assumption, $\mathsf{validate}_j(j) = 1$ for every honest j at the time it starts the VLE protocol, so from the Consistency property of the predicate $\mathsf{validate}_i(j)$, will eventually output 1 for every honest i. When these conditions hold, i adds $(j, \mathsf{dealers}_j)$ to $\mathsf{attached}_i$. After adding such a tuple for every honest j, i sees that $|\mathsf{attached}_i| = n-t$ and it calls Gather_i with the input $S_i = k|\exists(k, \mathsf{dealers}_k) \in \mathsf{attached}_i$. Clearly, for every $k \in S_i$ there is a tuple of the form $(k, \mathsf{dealers}_k)$ in $\mathsf{attached}_i$, so $\mathsf{checkValidity}_i(k) = 1$. From Lemma 6.5, $\mathsf{checkValidity}$ is an asynchronous validity predicate and all honest parties eventually call the Gather protocol, so from the Termination protocol of Gather they all eventually complete the protocol. When an honest i completes the Gather protocol with an output X_i, it updates its $\mathsf{candidates}_i$ set to X_i and broadcasts $\langle$"candidates"$, \mathsf{candidates}_i\rangle$. Every honest j receives that broadcast and from the Completeness property, $\mathsf{Gather}_j(\mathsf{candidates}_i)$ eventually terminates with the output 1. At that point, j calls $\mathsf{Sum}-\mathsf{Reconstruct}(k, \mathsf{dealers}_k)$ for every $k \in \mathsf{candidates}_i$ with $(k, \mathsf{dealers}_k) \in \mathsf{attached}_j$. Note that honest parties add those tuples after receiving the same $\langle$"attach"$, \mathsf{dealers}_k\rangle$ broadcast, so they all call $\mathsf{Sum}-\mathsf{Reconstruct}$ with the same set of dealers. From the Termination property of the AVSS protocol, i completes the $\mathsf{Sum}-\mathsf{Reconstruct}_i(k, \mathsf{dealers}_k)$ call for every $k \in \mathsf{candidates}_i$ and adds a tuple (k, r_k) to ranks_i. Finally, after having $\mathsf{candidates} \neq \perp$ and there being a tuple $(k, r_k) \in \mathsf{ranks}_i$ for every $k \in \mathsf{candidates}_i$, i performs local computations, outputs some value, and terminates.

Completeness. Assume some honest party i outputs the index ℓ and proof π from VLE. Party i chooses ℓ to be the index ℓ such that $\ell = argmax\{r_j | j \in \mathsf{candidates}_i, (j, r_j) \in \mathsf{ranks}_i\}$ and sets π to $\mathsf{candidates}_i$, which was i's output from the Gather_i protocol. Observe some honest party j that calls $\mathsf{VLEVerify}_j(\ell, \pi)$. Note that i only completes the VLE protocol after seeing that for every $k \in \mathsf{candidates}_i$ there exists a tuple $(k, r_k) \in \mathsf{ranks}_i$. From Lemma 6.4, eventually $\mathsf{ranks}_j \subseteq \mathsf{ranks}_i$ and thus $\forall k \in \pi\ \exists(k, r_k) \in \mathsf{ranks}_j$. In addition, from the Completeness property of $\mathsf{GatherVerify}$, $\mathsf{GatherVerify}_j(\pi)$ eventually terminates with the output 1. As shown above, j eventually completes Gather_j and proceeds to compute ℓ. Both i and j compute ℓ to be $argmax\{r_k | k \in \mathsf{candidates}_i\}$, with r_k being the output from $\mathsf{Sum}-\mathsf{Reconstruct}(k, \mathsf{dealers}_k)$. From the Correctness property of AVSS, both i and j receive the same output r_k for every k, and thus they compute the same ℓ as the index with maximal r_k. Therefore, $\mathsf{VLEVerify}_j(\ell, \pi)$ outputs 1 and terminates.

α-**Binding.** From the Includes Core property of the Gather protocol, at the time the first honest party completes the Gather protocol, there exists a binding core X^* of at least $n-t$ indices in $[n]$ such that if $\mathsf{GatherVerify}_i(X)$ terminates with the output 1 for an honest party i, then $X^* \subseteq X$. Note that at least $n - 2t \geq t+1 > \frac{n}{3}$ of those indices are honest parties' indices. Let I be the set of all parties k for which at least one honest party j called $\mathsf{Sum}-\mathsf{Reconstruct}_j(k, \mathsf{dealers}_k)$. Before calling the protocol, j completes Gather_j, calls $\mathsf{GatherVerify}_j(\mathsf{candidates})$ and sees that it terminates with the output 1 for some set $\mathsf{candidates}$ which includes k. From the Validity property of the Gather protocol, there already existed a tuple $(k, \mathsf{dealers}_k) \in \mathsf{attached}_i$ at that time, and from the Binding Core property of the Gather protocol X^* is already defined at that time. Note that from the Correctness property of $\mathsf{Sum}-\mathsf{Reconstruct}$, r_k is the sum of the k'th secrets shared by the dealers in $\mathsf{dealers}_k$. For each $(k, \mathsf{dealers}_k) \in \mathsf{attached}_j$, $\mathsf{dealers}_k$ has at least $t+1$ indices, and thus at least one of the dealers was honest. That honest dealer shared a uniformly sampled value, and no honest party started reconstructing r_k or the uniformly sampled secret shared by the honest dealer before receiving an "attach" broadcast from k. Therefore, from the Secrecy property of the AVSS protocol, the value shared by the honest dealer is sampled uniformly and independently of the adversary's view at that time. This means that for every $k \in I$, r_k is sampled uniformly and independently from all other values and from the set X^*. Therefore, the probability of a given party $k \in I$ having the maximal rank r_k is $\frac{1}{|I|} \geq \frac{1}{n}$[3]. This means that the probability that there exists an honest party ℓ^* such that $\ell^* \in X^*$ and r_{ℓ^*} is the maximal rank among all r_k such that $k \in I$ is at least $\frac{n}{3} \cdot \frac{1}{n} = \frac{1}{3}$. If that is the case, define ℓ^* to be that party's index, otherwise, define it to be $\perp$.

Binding Verification. If ℓ^* as defined in the α-Binding property equals $\perp$, the property trivially holds. Assume that $\ell^* \neq \perp$ and that $\mathsf{VLEVerify}_i(\ell, \pi)$ terminates with the output 1 for some honest i. Before $\mathsf{VLEVerify}$ terminates, i checks that for every $k \in \pi$ there exists a tuple $(k, r_k) \in \mathsf{ranks}_i$. Afterwards, i calls $\mathsf{GatherVerify}_i(\pi)$, which eventually terminates with the output 1. From the Includes Core property of the Gather protocol, $X^* \subseteq X$ and thus $\ell^* \in X$. Now, note that i only adds a tuple (k, r_k) to ranks_i if it completes $\mathsf{Sum} - \mathsf{Reconstruct}(k, \mathsf{dealers}_k)$ with the output k. By definition, ℓ^* has the maximal rank r_{ℓ^*} and thus $\ell^* = argmax\{r_k | k \in \pi, (k, r_k) \in \mathsf{ranks}_i\}$. Party i eventually terminated, and thus it found that $\ell = \ell^*$, as required.

Agreement on Verification Let i, j be two honest parties and ℓ, π be two values such that $\mathsf{VLEVerify}_i(\ell, \pi)$ terminates with the output b. Honest parties only output 1 from the $\mathsf{VLEVerify}$ protocol so $b = 1$. Party i starts $\mathsf{VLEVerify}$ by waiting until $\forall k \in \pi$, there exists a tuple $(k, r_k) \in \mathsf{ranks}_i$. From Lemma 6.4, eventually $\mathsf{ranks}_i \subseteq \mathsf{ranks}_j$, so j will see that this condition holds. Following that, i sees that $\mathsf{GatherVerify}_i(\pi)$ terminates with the output 1 and that Gather_i terminates. From the Agreement on Verification and Termination properties of the Gather protocol, j sees that these conditions hold as well. At that time, $\mathsf{checkValidity}_i(k) = 1$ was true for every $k \in \pi$ and thus there was a tuple of the form $(k, \mathsf{dealers}_k) \in \mathsf{attached}_i$. The same holds for j, which received the same broadcast and added the same tuples to $\mathsf{attached}_j$. In addition, both i and j add the tuples (k, r_k) to their respective ranks sets after reconstructing r_k in the call to $\mathsf{Sum} - \mathsf{Reconstruct}(k, \mathsf{dealers}_k)$. From the Correctness property of the protocol, they both reconstruct the same value, so they have the same tuples for every $k \in \pi$. They then compute ℓ in the same way with respect to the same tuples (k, r_k). For that index, i saw that $k = \ell$, and thus j will see that the same condition holds, output 1, and terminate.

Validity. Observe some honest party i, and ℓ, π such that $\mathsf{VLEVerify}_i(\ell, \pi)$ terminates with the output 1. This means that i saw that $\mathsf{GatherVerify}_i(\pi)$ terminated with the output 1 and that $\ell = argmax\{r_j | j \in \pi, (j, r_j) \in \mathsf{ranks}_i\}$. In other words $\ell \in \pi$ so from the Validity property of the $\mathsf{GatherVerify}$ protocol, $\mathsf{checkValidity}_i(\ell) = 1$. This means that there exists a tuple of the form $(\ell, \mathsf{dealers})$ in $\mathsf{attached}_i$. Honest parties only add such a tuple after seeing that $\mathsf{validate}_i(\ell)$ terminated with the output 1, completing the proof. □

In the above theorem, the threshold $t < \frac{n}{4}$ stems from using AVSS protocols, for which this threshold is necessary in order to guarantee their termination [4, 12]. Similar results can be achieved by using AVSS protocols with an ϵ probability of failure or non-termination, resulting in probabilistic guarantees.

[3]We ignore a negligible probability of two parties having the same rank. This can be accounted for by sampling from a large enough $\mathbb{F}$ and noting that $t + 1 \geq \frac{n}{3} + \frac{1}{n}$

7 Asynchronously Validated Asynchronous Byzantine Agreement

7.1 Definition

In an Asynchronously Validated Asynchronous Byzantine Agreement protocol, there is some asynchronous validity predicate $\mathsf{validate}$ that every party has access to. In addition, the protocol has some success parameter $\alpha \in (0,1)$. Each honest party i starts with an input x_i such that $\mathsf{validate}_i(x_i) = 1$ at the time i calls the protocol. Every honest party outputs a value when completing the protocol. A Validated Asynchronous Byzantine Agreement protocol has the following properties:

- **Agreement.** All honest parties that complete the protocol output the same value.
- **Validity.** If an honest party i outputs a value y_i then $\mathsf{validate}_i(y_i) = 1$ at that time.
- **α-Quality.** With probability α, all parties output the input x_i of a party i that was honest when starting the protocol.
- **Termination.** All honest parties almost surely terminate, i.e., with probability 1.

7.2 Overview

This section deals with constructing our AVABA protocol, which is built upon ideas in [5] and [7] and adapts them to the asynchronous information-theoretic setting. For a full construction and proofs, see Sections 7.3 and 7.4. Using this protocol, constructing an ACS protocol is straightforward. For a construction and a proof of an ACS protocol, see Section 8.

The protocol of [5] heavily relies on cryptographic primitives (signatures) to obtain externally valid outputs. Here, we use the framework of asynchronous validity predicates to replace external validity with an information-theoretic counterpart. This requires redefining and adapting new information-theoretic variants of verifiable gather (party gather) and verifiable leader election. The protocol of [7] modifies the cryptographic protocol of [29] to the information-theoretic setting in partial synchrony. Here, we show how to extend this to full asynchronous network conditions, which requires a new information-theoretic view change protocol and consistency checks for sent values.

In the AVABA protocol, parties proceed in "views". In each view, parties propose values on which to agree and then try to choose an honest leader using the VLE protocol. Our VLE protocol has α-Quality for $\alpha = \frac{1}{3}$, so this event should take place with probability $\frac{1}{3}$ or greater. Once this happens, the AVABA protocol guarantees that all parties will terminate with the proposal suggested by that honest party.

AVABA uses the "Key-Lock-Commit" paradigm used in previous HotStuff protocols (VABA, IT-HS and NWH) to maintain safety and liveness. As explained in [5]:

- **Key:** Parties set a local key field that indicates that no other value was committed to in previous rounds. The keys help maintain liveness: if at any point some party sets a lock (to be explained later) in a view where no commitment takes place, then they will eventually see a key from that view (or a later view) that will convince them to participate in the current view.

  A key consists of two values: key, a view number, and $\mathsf{key\_val}$, the suggested value.

- **Lock:** Before committing to a value in a given view, parties will wait to hear that enough other parties have set a lock on the same value in that view. Before parties set a lock in a given view, they make sure that enough other parties have set a local key field that indicates that no other value was committed to in previous rounds. Parties that are locked on a value won't only be willing to participate in a later view in which another value was suggested unless a key from a later view is provided. This mechanism helps in guaranteeing the safety of decision values. If a commitment takes place, then there will be a large number of honest parties that are locked on that value. Those parties won't be willing to participate in views with different values, which will prevent any party from setting a key in a later view with a different value. This, in turn, will guarantee that no party will be able to provide erroneous proof that the locks can be opened.

  A lock consists of two values: lock, which is a view number, and lock_val, which is the value seen when setting the lock.

- **Commit:** If an honest party commits to a value, no other honest party ever commits to another value, using the locking mechanism. Before terminating, parties ensure that every party will hear many commit messages, which guarantees that they can also commit and terminate.

The parties proceed in 5 rounds in each view. The general idea is that parties will first confirm that they all agree on the leader elected in the VLE protocol, set a lock to the elected leader's proposal, and confirm that they are all locked, commit to the lock, and terminate. If they see that the VLE protocol failed at any point throughout the view, they move onto a new view and announce that they are doing so (with proof). In the NWH protocol, parties provided cryptographic proofs for their keys and locks in the form of signatures on "echo" and "key" messages, respectively. These signatures are inherently transferable since they can be sent to any party who can verify them independently. To allow the "transfer" of such proofs, parties broadcast their "echo" and "key" messages. This allows a party that formed a key or a lock to know that any other party will eventually hear the same "echo" and "key" messages and believe it could have formed that key or lock. Similar techniques are employed when providing "blame" and "echo" messages, which are used to inform parties of a failed VLE session. In more detail, the protocol proceeds as follows:
Round 1: The first round in each view begins with a viewChange protocol. In viewChange, parties choose their proposals and broadcast them. They send their current key to all other parties in a "suggest" message. Before accepting a key, parties ensure it could have been achieved in the relevant view by waiting to receive the broadcasted messages required to form a key ("echo" messages to be explained later). Upon accepting $n-t$ keys, parties choose the key and value from the most recent view and broadcast the chosen key and value in a "proposal" message. Following that, they call the VLE protocol to choose a leader for the current view, using $\mathsf{leaderCorrect}_i$ as an asynchronous validity predicate. This guarantees that any chosen leader has broadcasted a proposal.
Round 2: In the second round, parties check whether the VLE was successful or not. If it was successful, they continue in that view, but if unsuccessful, they inform each other and proceed to the following view.

- Upon electing a leader using the VLE protocol, if the leader's proposed value is correct, then echo that message to all other parties and include proof that this is the leader elected in the VLE protocol.
- If the leader's proposed value is incorrect, send a "blame" message and proof that this is the leader elected in the VLE protocol and that its proposed value is incorrect and proceed to the next view. In this context, an incorrect proposal means its key was not high enough to open the

receiving party's current lock. Every party can check that the purported lock could have been set in a later view by waiting to receive the same broadcasted "key" messages required to set a lock.

Upon receiving a correct "blame" message and proof, send the "blame" message to all parties and proceed to the following view.

- Upon receiving "echo" messages with two different values suggested by two different leaders who were independently elected in the VLE protocol, send an "equivocation" message containing the two values and the two proofs to all parties and proceed to the following view.

  Upon receiving an "equivocation" message with different values and correct proofs, forward that message and proceed to the following view.

Round 3: Parties proceed to this round if they have received many "echo" messages without seeing an error in the form of a "blame" or an "equivocation" message. This also means that no other value was committed to in an earlier view, meaning that a key can be formed. Upon receiving $n-t$ "echo" messages, update the key and key_val fields before sending a "key" message to all parties.
Round 4: Upon receiving $n-t$ "key" messages, update the lock and lock_val fields before sending a "lock" message to all parties. By doing this, every party ensures that at least $t+1$ honest parties set their keys to the current value before setting a lock. This guarantees that when choosing which value and key to input to the VLE protocol, all honest parties will hear of the current value and be capable of opening any older lock an honest party might have.
Round 5: Finally, upon receiving $n-t$ correct "lock" messages, parties send "commit" messages with the same value. Such a message is sent after receiving "lock" messages from $n-t$ parties, guaranteeing that $t+1$ parties have set their lock in the current view. These parties will not echo any message about any other value in subsequent views unless an adequate key is provided. Since forming a key requires a message from one of those parties, we can reason inductively that no correct key will be formed for a differing value in any subsequent view.
Output: In order to allow parties to terminate, a termination gadget is also run outside of any specific view. Similarly to Bracha broadcast [14], every party echoes a "commit" message if it sees $t+1$ such messages with the same value. Finally, parties terminate after seeing $n-t$ such messages.

7.3 The Protocol

Protocol 7.1: AVABA(x_i)

```
1: $\mathsf{key}_i \leftarrow 0, \mathsf{key\_val}_i \leftarrow x_i$
2: $\mathsf{lock}_i \leftarrow 0, \mathsf{lock\_val}_i \leftarrow \bot$
3: $\forall v \in \mathbb{N}\ \mathsf{proposals}_i \leftarrow \emptyset, \mathsf{echoes}_{i,v} \leftarrow \emptyset, \mathsf{keys}_{i,v} \leftarrow \emptyset, \mathsf{locks}_{i,v} \leftarrow \emptyset$
4: $\mathsf{view}_i \leftarrow 1$
5: continually run checkTermination()
6: while true do
7:     cur_view ← $\mathsf{view}_i$
8:     as long as cur_view = $\mathsf{view}_i$, run
9:         delay any message from any view $v$ such that $v > \mathsf{view}_i$
10:        call viewChange($\mathsf{view}_i$) and continually run the upon commands within it
11:        continually run processMessages($\mathsf{view}_i$) and processFaults($\mathsf{view}_i$)
12:        continue updating sets and participating in broadcasts from older views, but do not send
   news messages or broadcasts or update $\mathsf{key}_i, \mathsf{key\_val}_i, \mathsf{lock}_i, \mathsf{lock\_val}_i$ in previous views
```

Protocol 7.2: $\mathsf{processMessages}(\mathsf{view})$

```
1: upon $\mathsf{VLE}_{i,\mathsf{view}}$ outputting $\ell, \pi$, do    ▷ continue updating state according to $\mathsf{VLE}_{i,\mathsf{view}}$
2:     let $(k, v)$ be a tuple such that $(\ell, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$
3:     if $k \geq \mathsf{lock}_i$ then
4:         broadcast $\langle$“echo”$, k, v, \ell, \pi, \mathsf{view}\rangle$
5:     else
6:         send $\langle$“blame”$, k, v, \ell, \pi, \mathsf{lock}_i, \mathsf{lock\_val}_i, \mathsf{view}\rangle$ to every party $j$
7:         $\mathsf{view}_i \leftarrow \mathsf{view}_i + 1$
8: upon receiving an $\langle$“echo”$, k, v, \ell, \pi, \mathsf{view}\rangle$ broadcast from $j$, do
9:     upon $\mathsf{VLEVerify}_{i,\mathsf{view}}(\ell, \pi)$ terminating with the output 1, do
10:        if $(\ell, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$ then
11:            $\mathsf{echoes}_{i,\mathsf{view}} \leftarrow \mathsf{echoes}_{i,\mathsf{view}} \cup \{(j, k, v, \ell, \pi)\}$
12:            if $\exists (j', k', v', \ell', \pi') \in \mathsf{echoes}_i$ $s.t.$ $(k, v) \neq (k', v')$ then
13:                send $\langle$“equivocation”$, k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view}\rangle$ to every party $j$
14:                $\mathsf{view}_i \leftarrow \mathsf{view}_i + 1$
15:            else if $|\mathsf{echoes}_{i,\mathsf{view}}| = n - t$ then
16:                $\mathsf{key}_i \leftarrow \mathsf{view}, \mathsf{key\_val}_i \leftarrow v$
17:                broadcast $\langle$“key”$, v, \mathsf{view}\rangle$
18: upon receiving a $\langle$“key”$, v, \mathsf{view}\rangle$ broadcast from $j$, do
19:     upon $\mathsf{keyCorrect}_{i,\mathsf{view}+1}(\mathsf{view}, v)$ terminating with the output 1, do
20:         $\mathsf{keys}_{i,\mathsf{view}} \leftarrow \mathsf{keys}_{i,\mathsf{view}} \cup \{(j, v)\}$
21:         if $\left|\mathsf{keys}_{i,\mathsf{view}}\right| = n - t$ then
22:             $\mathsf{lock}_i \leftarrow \mathsf{view}, \mathsf{lock\_val}_i \leftarrow v$
23:             send $\langle$“lock”$, v, \mathsf{view}\rangle$ to every party $j$
24: upon receiving the first $\langle$“lock”$, v, \mathsf{view}\rangle$ message from $j$, do
25:     upon $\mathsf{lockCorrect}_i(\mathsf{view}, v)$ terminating with the output 1, do
26:         $\mathsf{locks}_{i,\mathsf{view}} \leftarrow \mathsf{locks}_{i,\mathsf{view}} \cup \{(j, v)\}$
27:         if $|\mathsf{locks}_{i,\mathsf{view}}| = n - t$ then
28:             send $\langle$“commit”$, v\rangle$ to every party $j$
```

Protocol 7.3: $\mathsf{leaderCorrect}_{i,\mathsf{view}}(\ell)$

```
1: upon there being a tuple of the form $(\ell, (k, v))$ in $\mathsf{proposals}_{i,\mathsf{view}}$, do
2:     output 1 and terminate
```

Protocol 7.4: $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$

```
1: if $\mathsf{view} > k$ then
2:     upon $\mathsf{validate}_i(v)$ terminating with the output 1, do
3:         if $k = 0$ then
4:             output 1 and terminate
5:         else
```

```
6:         upon $|\{j|\exists k', \ell, \pi \ s.t. \ (j, k', v, \ell, \pi) \in \mathsf{echoes}_{i,k}\}| \geq n - t$, do
7:             output 1 and terminate
```

Protocol 7.5: $\mathsf{lockCorrect}_i(k, v)$

```
1: if $k = 0$ then
2:     output 1 and terminate
3: else
4:     upon $\left|\{j|(j, v) \in \mathsf{keys}_{i,k}\}\right| \geq n - t$, do
5:         output 1 and terminate
```

Protocol 7.6: $\mathsf{checkTermination}()$

```
1: upon receiving a ⟨“commit”, $v$⟩ message with the same value $v$ from $t + 1$ parties, do
2:     send ⟨“commit”, $v$⟩ to every party $j$ if no such message has been previously sent
3: upon receiving a ⟨“commit”, $v$⟩ message with the same value $v$ from $n - t$ parties, do
4:     output $v$ from the AVABA protocol and terminate AVABA
```

Protocol 7.7: $\mathsf{viewChange}(\mathsf{view})$

```
1: $\mathsf{suggestions} \leftarrow \emptyset$                                  ▷ suggestions is a multiset
2: send ⟨“suggest”, $\mathsf{key}_i$, $\mathsf{key\_val}_i$, $\mathsf{view}$⟩ to every party $j$
3: upon receiving the first ⟨“suggest”, $k, v, \mathsf{view}$⟩ message from party $j$ such that $k < view$, do
4:     upon $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$ terminating with the value 1, do
5:         $\mathsf{suggestions} \leftarrow \mathsf{suggestions} \cup \{(k, v)\}$
6:         if $|\mathsf{suggestions}| = n - t$ then
7:             $(k, v) \leftarrow argmax_{(k,v) \in \mathsf{suggestions}}\{k\}$                ▷ break ties arbitrarily
8:             if $k = 0$ then
9:                 $(k, v) \leftarrow (0, x_i)$
10:            broadcast ⟨“proposal”, $k, v, \mathsf{view}$⟩
11: upon receiving a ⟨“proposal”, $k, v, \mathsf{view}$⟩ broadcast from $j$, do
12:     upon $\mathsf{keyCorrect}_{i,\mathsf{view}}((k, v))$ terminating with the output 1, do
13:         $\mathsf{proposals}_{i,\mathsf{view}} \leftarrow \mathsf{proposals}_{i,\mathsf{view}} \cup \{(j, (k, v))\}$
14:         if $j = i$ then
15:             call $\mathsf{VLE}_{i,\mathsf{view}}()$ with the validity predicate $\mathsf{leaderCorrect}_{i,\mathsf{view}}$
```

Protocol 7.8: $\mathsf{processFaults}(\mathsf{view})$

```
1: upon receiving the first ⟨“blame”, $k, v, leader, \pi, l, lv, \mathsf{view}$⟩ message from $j$, do
2:     upon $\mathsf{lockCorrect}_i(l, lv)$ and $\mathsf{VLEVerify}_{i,\mathsf{view}}(leader, \pi)$ terminating with the output 1, do
3:         if $k < l$ and $(leader, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$ then
4:             send ⟨“blame”, $leader, \pi, l, lv, \mathsf{view}$⟩ to every party $j$
5:             $\mathsf{view}_i \leftarrow \mathsf{view}_i + 1$
6: upon receiving the first ⟨“equivocation”, $k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view}$⟩ message from $j$, do
```

```
    upon $\mathsf{VLEVerify}_{i,\mathsf{view}}(\ell,\pi)$ and $\mathsf{VLEVerify}_{i,\mathsf{view}}(\ell',\pi')$ terminating with the output 1, do
        if $(\ell,(k,v)),(\ell',(k',v')) \in \mathsf{proposals}_{i,\mathsf{view}}$ and $(k,v) \neq (k',v')$ then
            send $\langle$"equivocation"$, k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view}\rangle$ to every party $j$
            $\mathsf{view}_i \leftarrow \mathsf{view}_i + 1$
```

7.4 Security Analysis

In this section, we will show that AVABA is an Asynchronously Validated Asynchronous Byzantine Agreement protocol in Theorem 7.19. We start by proving several lemmas. Lemma 7.10 and Lemma 7.11 are instrumental for showing the *safety* of the protocol. By that we mean that if some honest party outputs a value v, no other honest party outputs a differing value $v' \neq v$. The Correctness property of the protocol is then an immediate consequence of Lemma 7.17.

The remaining lemmas deal with the *liveness* of the protocol. By that we mean that eventually some progress is made, leading to the termination of the protocol. More specifically, we start by showing that parties don't get stuck in any view without being able to output a value or progress to the next view. We then show that once an honest party is chosen as a leader which is the unique verifiable output from the VLE protocol (which happens with constant probability), all honest parties will commit at the end of the view.

We start by defining what it means for a key or lock to be correct.

Definition 7.9. *A "key" message of the form $\langle$"key"$, v, \mathsf{view}\rangle$ is said to be correct if for some honest i, $\mathsf{keyCorrect}_{i,\mathsf{view}'}(\mathsf{view}, v) = 1$ holds for every $\mathsf{view}' > \mathsf{view}$. Similarly, a "lock" message of the form $\langle$"lock"$, v, \mathsf{view}\rangle$ is said to be correct if $\mathsf{lockCorrect}_i(\mathsf{view}, v) = 1$ for an honest i. In addition, the value of each such message is said to be the field v.*

As stated above, the following two lemmas are used in the proof that the protocol is safe. First, we show that in any given view only one value can proceed into the later rounds, meaning that any two values committed to in a single view must be the same. Following that, we show that if an honest party committed to a value, there are $t+1$ honest parties that won't send "echo" messages for any other value in any subsequent view. This prevents any other value from being included in correct "key" or "lock" messages, thus preventing other values from being committed to in later views. This idea is explored more fully and proved in Lemma 7.17. The proofs for the lemmas are provided in Section 7.5.

Lemma 7.10. *If two messages from a given view are correct, they both have the same value v.*

Lemma 7.11. *If an honest party sends a $\langle$"commit"$, v\rangle$ message in line 28 of $\mathsf{processMessages}(\mathsf{view})$, then for any $\mathsf{view}' \geq \mathsf{view}$ there exist $t+1$ honest parties that never send an $\langle$"echo"$, k', v', \ell', \pi', \mathsf{view}'\rangle$ message with $v' \neq v$.*

We now turn to deal with the liveness of the protocol, showing that parties either progress through views or terminate.

Definition 7.12. *An honest party i is said to reach a view if at any point its local view_i field equals view. Similarly, an honest party i is said to be in view if its local view_i field equals view at that time.*

We will start by showing that the methods used for validating leaders, keys, and locks are asynchronous validity predicates and that keys and locks are always correct according to the party holding them. This means that parties can use the VLE protocol with $\mathsf{leaderCorrect}$ as an asynchronous validity predicate. In addition, this means that every honest party will be convinced of the correctness of other parties' keys and locks, allowing them to progress through views in the case that "blame" messages are sent. The proofs of the lemmas are provided in Section 7.6

Lemma 7.13. *Let* $\mathsf{keyCorrect}_{\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{\mathsf{view}}$ *be the predicates defined by* $\mathsf{keyCorrect}_{i,\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{i,\mathsf{view}}$ *for every* i *respectively.* $\mathsf{keyCorrect}_{\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{\mathsf{view}}$ *are asynchronous validity predicates for every* view*. Furthermore, for any* $\mathsf{view} > \mathsf{key}_i$, $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i) = 1$ *at any point in time.*

Lemma 7.14. $\mathsf{lockCorrect}$ *is an asynchronous validity predicate. Furthermore, at any point in time* $\mathsf{lockCorrect}_{i,\mathsf{view}}(\mathsf{lock}_i, \mathsf{lock\_val}_i) = 1$*.*

The next lemmas show that progress is made. We start in Lemma 7.15 by showing that parties don't get stuck in a view. More precisely, if no honest party completes the protocol in a given view, every honest party eventually reaches the next view. Lemma 7.16 then shows that if an honest party is chosen as the unique verifiable leader using the VLE protocol, the adversary cannot convince any honest party to proceed to the next view using a "blame" message. We then show in Lemma 7.17 that if some honest party terminates, every honest party does so as well. Finally, Lemma 7.18 shows that there is a constant probability of all parties terminating in any given view, using the fact that there is a $\frac{1}{3}$ probability that an honest party is elected in that view. The proofs of the following lemmas are straightforward and mostly consist of showing that parties eventually send the required messages and reach agreement. The lemmas are stated here but proved in Section 7.7.

Lemma 7.15. *If every honest party* i *has an input* x_i *such that* $\mathsf{validate}_i(x_i) = 1$ *at the time it calls* AVABA*, all honest parties participate in the protocol, and no honest party terminates during any* view' *such that* $\mathsf{view}' < \mathsf{view}$*, then all honest parties reach* view*.*

Lemma 7.16. *If an honest* j *broadcasts a* $\langle$*"proposal"*$, k, v, \mathsf{view}\rangle$ *message, then no honest party sends a* $\langle$*"blame"*$, k, v, j, \pi, l, lv, \mathsf{view}\rangle$ *message for any* π, l, lv*.*

Lemma 7.17. *If some honest party outputs* v *and terminates, then all honest parties eventually do so as well.*

Lemma 7.18. *If all honest parties start* view *and every honest* i *has an input* x_i *such that at the time it calls the* AVABA *protocol* $\mathsf{validate}_i(x_i) = 1$*, then with constant probability all honest parties terminate during* view*.*

Using these lemmas we prove the following theorem in Section 7.8. We analyze its efficiency in Appendix A.4.

Theorem 7.19. *Protocol* AVABA *is a Validated Asynchronous Byzantine Agreement protocol resilient to* $t < \frac{n}{4}$ *Byzantine parties.*

7.5 Proofs of AVABA Safety

Lemma 7.20. *If two messages from a given* view *are correct, they both have the same value* v.

Proof. First, observe two correct messages $\langle$"key"$, v, \mathsf{view}\rangle$ and $\langle$"key"$, v', \mathsf{view}\rangle$. The messages are correct, so $\mathsf{keyCorrect}_{i,\mathsf{view}+1}(\mathsf{view}, v) = 1$ for some honest i. Because $\mathsf{view} > 0$, this must mean that $|\{j|\exists \ell, k', \pi \ s.t.(j, k', v, \ell, \pi) \in \mathsf{echoes}_{i,\mathsf{view}}| \geq n - t$. Party i adds a tuple (j, k', v, ℓ, π) to $\mathsf{echoes}_{i,\mathsf{view}}$ after receiving a broadcasted $\langle$"echo"$, k', v, \ell, \pi, \mathsf{view}\rangle$ message from j. This means that i received such a broadcast with the same value v from at least $n - t$ parties. For similar reasons, j also received similar broadcasts with the value v' from $n - t$ parties. Since $n \geq 3t + 1$ at least $t + 1$ of those broadcasts must have been received from the same parties, and thus the received values v, v' are the same value.

Now observe a correct "lock" message $\langle$"lock"$, v', \mathsf{view}\rangle$. Similarly to the case above, for some honest i, $\mathsf{lockCorrect}_i(\mathsf{view}, v') = 1$ with $\mathsf{view} > 0$, so $\left|\{j|(j, v') \in \mathsf{keys}_{i,\mathsf{view}}\right| \geq n - t$. Following similar logic to above, this means that i received correct $\langle$"key"$, v', \mathsf{view}\rangle$ broadcasts from $n - t$ parties before adding those tuples to $\mathsf{keys}_{i,\mathsf{view}}$. As shown above, all of those messages have the same value v, and thus also $v' = v$. □

Lemma 7.21. *If an honest party sends a* $\langle$*"commit"*$, v\rangle$ *message in line 28 of* $\mathsf{processMessages}(\mathsf{view})$*, then for any* $\mathsf{view}' \geq \mathsf{view}$ *there exist* $t+1$ *honest parties that never send an* $\langle$*"echo"*$, k', v', \ell', \pi', \mathsf{view}'\rangle$ *message with* $v' \neq v$ *.*

Proof. We will prove inductively that for any $\mathsf{view}' \geq \mathsf{view}$, there must exist $t + 1$ such honest parties. First observe $\mathsf{view}' = \mathsf{view}$. Since some honest i sends a $\langle$"commit"$, v\rangle$ in line 28, it added $n - t$ tuples (j, v) to $\mathsf{locks}_{i,\mathsf{view}}$ and saw that $|\mathsf{locks}_{i,\mathsf{view}}| = n - t$. An honest i only does so after receiving $\langle$"lock"$, v, \mathsf{view}\rangle$ messages from $n-t$ parties and seeing that $\mathsf{lockCorrect}_i(\mathsf{view}, v) = 1$. Out of those parties, at least $t+1$ were honest, and they sent their $\langle$"lock"$, v, \mathsf{view}\rangle$ broadcast after seeing that $\left|\mathsf{keys}_{j,\mathsf{view}}\right| \geq n - t$. Following similar logic, they received $n - t$ $\langle$"key"$, v, \mathsf{view}\rangle$ messages and saw that they are correct. At least one of those messages was sent by an honest i that added $n - t$ tuples of the form (j, k, v, ℓ, π) to its echoes set after receiving $\langle$"echo"$, k, v, \ell, \pi, \mathsf{view}\rangle$ broadcasts from $n - t$ parties. Note that i sent a "key" message, so it did not change view before sending the message, meaning that it did not send an "equivocation" message at that time, and thus at that time every tuple (j, k, v, ℓ, π) in $\mathsf{echoes}_{i,\mathsf{view}}$ had the same k and v. In other words, it sent its $\langle$"key"$, v, \mathsf{view}\rangle$ message after receiving an "echo" broadcast with the same value v from $n - t$ parties. Out of those parties, at least $t + 1$ are honest and they only send one "echo" broadcast per view. From Lemma 7.10, all correct "key" and "lock" messages from view had the same value v, and thus the "commit" message had the same value as well.

Assume the claim holds for every view'' such that $\mathsf{view}' > \mathsf{view}'' \geq \mathsf{view}$. As shown above, there are at least $t + 1$ honest parties that send $\langle$"lock"$, v, \mathsf{view}\rangle$ broadcasts. Every honest party j only sends such a message after setting its lock_j field to view. Let the set of those honest parties be I. It is important to note that the field lock_j only grows throughout the protocol, so every one of the parties j such that $j \in I$ has $\mathsf{lock}_j \geq \mathsf{view}$ from that point on. Now assume by way of contradiction that some party $j \in I$ sent an $\langle$"echo"$, k', v', \ell', \pi', \mathsf{view}'\rangle$ message with $v' \neq v$. Before doing that, it output ℓ', π' from $\mathsf{VLE}_{i,\mathsf{view}'}$. From the Completeness and Validity properties of the VLE protocol, $\mathsf{leaderCorrect}_{i,\mathsf{view}'}(\ell', \pi) = 1$ at that time, so there was a tuple $(\ell', (k', v')) \in \mathsf{proposals}_{i,\mathsf{view}'}$, and $k' \geq \mathsf{lock}_j \geq \mathsf{view}$ because i did not send a "blame" broadcast. Party i adds such a tuple after receiving a $\langle$"proposal"$, k', v', \mathsf{view}'\rangle$ from ℓ' and seeing that $\mathsf{keyCorrect}_{i,\mathsf{view}'}((k', v')) = 1$, so

$\mathsf{view}' > k'$ and $\left|\{j'|\exists k'', \ell', \pi'' \ s.t. \ (j', k'', v', \ell', \pi'') \in \mathsf{echoes}_{j,k'}\}\right| \geq n - t$. As discussed above, each honest party only adds a tuple $(j', k'', v', \ell', \pi'')$ to $\mathsf{echoes}_{j,k'}$ after receiving an "echo" message with the value v' from j'. However, $\mathsf{view}' > k' \geq \mathsf{view}$, so by assumption there exist $t + 1$ parties that never send such a message in view k'. Any set of $n - t$ parties that sent the "echo" broadcasts must have at least one party in common with the parties in I, reaching a contradiction. □

7.6 Proofs of the Correctness of AVABA's Asynchronous Validity Predicates

Lemma 7.22. *Let* $\mathsf{keyCorrect}_{\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{\mathsf{view}}$ *be the predicates defined by* $\mathsf{keyCorrect}_{i,\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{i,\mathsf{view}}$ *for every i respectively.* $\mathsf{keyCorrect}_{\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{\mathsf{view}}$ *are asynchronous validity predicates for every* view. *Furthermore, for any* $\mathsf{view} > \mathsf{key}_i$, $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i) = 1$ *at any point in time.*

Proof. Let i, j be two honest parties and assume that at some point in time $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v) = b$ and $\mathsf{leaderCorrect}_{i,\mathsf{view}}(\ell) = b$. Note that both $\mathsf{keyCorrect}$ and $\mathsf{leaderCorrect}$ only output 1, so $b = 1$.

Finality. The first things that i does in $\mathsf{keyCorrect}_i(k, v)$ are checking that $\mathsf{view} > k$ and that $\mathsf{validate}_i(v) = 1$. From the Finality property of $\mathsf{validate}$, $\mathsf{validate}_i(v)$ will output 1 in the future as well. This means that if $k = 0$, $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$ terminates with the output 1 at any time in the future. If $k \neq 0$, then $|\{j|\exists k', \ell, \pi \ s.t. \ (j, k', v, \ell, \pi) \in \mathsf{echoes}_{i,k},\}| \geq n - t$. Honest parties do not remove values from $\mathsf{echoes}_{i,k}$, so this will continue to hold in the future and $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$ will output 1 and terminate in any future call. Additionally, if $\mathsf{leaderCorrect}_{i,\mathsf{view}}(\ell)$ returned 1, then there was a tuple of the form $(\ell, (k, v))$ in $\mathsf{proposals}_{i,\mathsf{view}}$. Parties don't remove tuples from $\mathsf{proposals}_{i,k}$ either, so this will continue to hold as well and thus $\mathsf{leaderCorrect}_{i,\mathsf{view}}(\ell)$ will return 1 in the future.

Consistency. As shown above, $\mathsf{view} > k$ and $\mathsf{validate}_i(v) = 1$. Therefore, from the Consistency property of $\mathsf{validate}$, $\mathsf{validate}_j(v) = 1$ will also eventually hold. If $k = 0$, then $\mathsf{keyCorrect}_{j,\mathsf{view}}(k, v)$ will terminate at that time and output 1. We will now prove by induction on view that any call $\mathsf{keyCorrect}_{j,\mathsf{view}}(k, v)$ eventually terminates and outputs 1 if $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$ does and that any call $\mathsf{leaderCorrect}_{j,\mathsf{view}}(\ell)$ eventually terminates and outputs 1 if $\mathsf{leaderCorrect}_{i,\mathsf{view}}(\ell)$ does. For $\mathsf{view} = 1$, since $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v) = 1$, $\mathsf{view} > k$, and thus $k = 0$. In this case, we've already shown above that $\mathsf{keyCorrect}_{j,\mathsf{view}}(k, v)$ will terminate with the output 1. In addition, if $\mathsf{leaderCorrect}_{i,\mathsf{view}}(\ell)$ terminates with the output 1, then there exists a tuple of the form $(\ell, (k', v'))$ in $\mathsf{proposals}_{i,\mathsf{view}}$. i adds such a tuple after receiving a $\langle\text{"proposal"}, k', v', \mathsf{view}\rangle$ broadcast from ℓ and seeing that $\mathsf{keyCorrect}_{i,\mathsf{view}}(k', v') = 1$. j will receive the same broadcast and as shown above, eventually see that $\mathsf{keyCorrect}_{j,\mathsf{view}}(k', v') = 1$. Following that j will add $(\ell, (k', v'))$ to $\mathsf{proposals}_{i,\mathsf{view}}$ and return 1 from $\mathsf{leaderCorrect}_{j,\mathsf{view}}(\ell)$.

Now assume that the claim holds for every $\mathsf{view}' < \mathsf{view}$. This means that for every $\mathsf{view}' < \mathsf{view}$, $\mathsf{keyCorrect}_{i,\mathsf{view}'}$ and $\mathsf{leaderCorrect}_{i,\mathsf{view}}$ have both the Finality and Consistency properties, and are thus asynchronous validity predicates. As above, $|\{j|\exists k', \ell, \pi \ s.t. \ (j, k', v, \ell, \pi) \in \mathsf{echoes}_{i,k}\}| \geq n - t$. Party i adds a tuple of the form (j, k', v, ℓ, π) to $\mathsf{echoes}_{i,k}$ after receiving an $\langle\text{"echo"}, k', v, \ell, \pi, k\rangle$ broadcast from a party ℓ, having $\mathsf{VLEVerify}_{i,k}(\ell, \pi)$ terminate with the output 1 and seeing that $(\ell, (k', v)) \in \mathsf{proposals}_{i,k}$. Party j will receive the same broadcasts and call $\mathsf{VLEVerify}_{j,k}(\ell, \pi)$. Note that $\mathsf{view} > k$ and thus $\mathsf{leaderCorrect}_k$ is an asynchronous validity predicate, so from the Agreement on Verification property of $\mathsf{VLEVerify}$, eventually $\mathsf{VLEVerify}_{j,k}(\ell, \pi)$ will terminate with the output 1 for every such tuple. From the Validity property of $\mathsf{VLEVerify}$, at that time $\mathsf{leaderCorrect}_{j,k}(\ell) = 1$, so there is a tuple $(\ell, (k'', v')) \in \mathsf{proposals}_{j,k}$. i and j add those tuples to $\mathsf{proposals}_{j,k}$ after receiving the same $\langle$"proposal"$, k', v, k\rangle$ broadcast from ℓ. This means that j adds the same tuples to $\mathsf{echoes}_{i,k}$ and eventually sees that the same condition holds, at which point it will output 1 from $\mathsf{keyCorrect}_{i,\mathsf{view}}$.

As for $\mathsf{leaderCorrect}_{\mathsf{view}}$, similarly to above, there exists a tuple of the form $(l, (k', v'))$ in $\mathsf{proposals}_{i,\mathsf{view}}$ because $\mathsf{leaderCorrect}_{i,\mathsf{view}}(\ell) = 1$, so i received a $\langle$"proposal"$, k', v', \mathsf{view}\rangle$ broadcast from j and saw that $\mathsf{keyCorrect}_{i,\mathsf{view}}(k', v') = 1$. j will receive the same broadcast, and from the Consistency property of $\mathsf{keyCorrect}_{\mathsf{view}}$, eventually see that $\mathsf{keyCorrect}_{j,\mathsf{view}}(k', v') = 1$. At that point, j will add $(\ell, (k', v'))$ to $\mathsf{proposals}_{j,\mathsf{view}}$ and return 1 from $\mathsf{leaderCorrect}_{j,\mathsf{view}}(\ell)$.

We will now show that $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i) = 1$ for any $\mathsf{view} > \mathsf{key}_i$ at any point in time. First, by definition $\mathsf{view} > \mathsf{key}_i$ so the first condition checked in $\mathsf{keyCorrect}_{i,\mathsf{view}}$ holds. If i has not updated $\mathsf{key}_i, \mathsf{key\_val}_i$ throughout the protocol, then $\mathsf{key}_i = 0, \mathsf{key\_val}_i = x_i$. By assumption, $\mathsf{validate}_i(\mathsf{key\_val}_i) = 1$ at the time i calls AVABA, so i will immediately see that $\mathsf{key}_i = 0$, output 1 and terminate. Otherwise, i updated both fields in the view key_i in line 16 after seeing that $\left|\mathsf{echoes}_{i,\mathsf{key}_i}\right| = n - t$. Party i only does so after receiving an $\langle$"echo"$, k', v', \ell', \pi', \mathsf{key}_i\rangle$ broadcast and seeing $(\ell', (k', v')) \in \mathsf{proposals}_{i,\mathsf{view}}$. It then updates its $\mathsf{key\_val}_i$ field to be v'. Note that before adding such a tuple to $\mathsf{proposals}_{i,\mathsf{key}_i}$, i checks that $\mathsf{keyCorrect}_{i,\mathsf{key}_i}(k', v') = 1$, and thus $\mathsf{validate}_i(v') = 1$ at that time. At that time for every $(j'', k'', v'', l'', \pi'') \in \mathsf{echoes}_{i,\mathsf{key}_i}$, $(k'', v'') = (k', v')$. Otherwise, j would have seen an equivocation in line 12 and proceeded to the next view before updating key_i. Therefore, $|\{j | \exists k'', \ell, \pi \ s.t. \ (j, k'', v', \ell, \pi) \in \mathsf{echoes}_{i,\mathsf{key}_i}\}| \geq n - t$ at that time, so i will output 1 and terminate from $\mathsf{keyCorrect}_i(\mathsf{key}_i, \mathsf{key\_val}_i)$. □

Lemma 7.23. $\mathsf{lockCorrect}$ *is an asynchronous validity predicate. Furthermore, at any point in time* $\mathsf{lockCorrect}_{i,\mathsf{view}}(\mathsf{lock}_i, \mathsf{lock\_val}_i) = 1$.

Proof. Let i, j be two honest parties and assume $\mathsf{lockCorrect}_i(k, v) = b$ at some point in time. Note that $\mathsf{lockCorrect}$ only outputs 1, so $b = 1$.

Finality. If $k = 0$, then i will always immediately output 1 and terminate. Otherwise, i saw that $\left|\{j | (j, v) \in \mathsf{keys}_{i,k}\}\right| \geq n - t$. Honest parties do not remove elements from their keys sets, so this condition will continue to hold and thus i will output 1 and terminate from $\mathsf{lockCorrect}_i(k, v)$.

Correctness. If $k = 0$, then j immediately outputs 1 from $\mathsf{lockCorrect}_j(k, v)$ as well. Otherwise, $k > 0$. Since i output 1 from $\mathsf{lockCorrect}_i(k, v)$, it saw that $\left|\{j | (j, v) \in \mathsf{keys}_{i,k}\}\right| \geq n - t$. i only adds a tuple (j, v) to its $\mathsf{keys}_{i,k}$ sets after receiving a $\langle$"key"$, v, k\rangle$ broadcast from j and seeing that $\mathsf{keyCorrect}_{i,k+1}(k, v) = 1$. From Lemma 7.13, $\mathsf{keyCorrect}_{k+1}$ is an asynchronous validity predicate, so eventually $\mathsf{keyCorrect}_{j,k+1}(k, v) = 1$ as well. At that point, j will add the same tuple (j, v) to $\mathsf{keys}_{j,k}$. After adding all of those tuples, j will see that the same condition holds and return 1 from $\mathsf{lockCorrect}_j(k, v)$.

Finally, we will show that $\mathsf{lockCorrect}_i(\mathsf{lock}_i, \mathsf{lock\_val}_i) = 1$ at any point in time. If i did not update those fields, then $\mathsf{lock}_i = 0, \mathsf{lock\_val}_i = \perp$. In that case, when running $\mathsf{lockCorrect}_i$, i will immediately see that $\mathsf{lock}_i = 0$ and output 1. Otherwise, i updated its lock_i and $\mathsf{lock\_val}_i$ fields after adding a tuple (j, v) to $\mathsf{keys}_{i,\mathsf{view}}$ and seeing that $\left|\mathsf{keys}_{i,\mathsf{view}}\right| = n - t$. This happens after receiving a $\langle$"key"$, v, \mathsf{view}\rangle$ broadcast from j and seeing that $\mathsf{keyCorrect}_{i,\mathsf{view}+1}(\mathsf{view}, v) = 1$. From Lemma 7.10, those messages have the same value v, and thus $|\{j | (j, v) \in \mathsf{keys}_{i,\mathsf{view}}\}| \geq n - t$ at that time, meaning that $\mathsf{lockCorrect}_i(\mathsf{view}, v) = 1$. □

7.7 Proofs of AVABA Liveness

Lemma 7.24. *If every honest party i has an input x_i such that $\mathsf{validate}_i(x_i) = 1$ at the time it calls* AVABA*, all honest parties participate in the protocol, and no honest party terminates during any view' such that $\mathsf{view}' < \mathsf{view}$, then all honest parties reach view.*

Proof. We will prove the claim inductively on view. First, all honest parties start in $\mathsf{view} = 1$. Now observe some $\mathsf{view} > 1$ and assume no honest party terminated in any $\mathsf{view}' < \mathsf{view}$, and that they all reached $\mathsf{view} - 1$. Since they reached $\mathsf{view} - 1$, they started off broadcasting "suggest" messages with their current $\mathsf{key}, \mathsf{key\_val}$ fields. An honest i only updates its key_i field to the view it is currently in, and thus at the beginning of $\mathsf{view} - 1$, $\mathsf{key}_i < \mathsf{view} - 1$. From Lemma 7.13, for every honest j, $\mathsf{keyCorrect}_{j,\mathsf{view}-1}(\mathsf{key}_j, \mathsf{key\_val}_j)$ at the time it sent those fields, and from the Consistency property of $\mathsf{keyCorrect}_{\mathsf{view}-1}$, eventually $\mathsf{keyCorrect}_{i,\mathsf{view}-1}(\mathsf{key}_j, \mathsf{key\_val}_j)$ terminates with the output 1 for every honest i. After receiving such a message from every honest j and seeing that the suggested $\mathsf{key}_j, \mathsf{key\_val}_j$ are correct, every honest i adds a tuple to $\mathsf{suggestions}$. After adding a tuple for each honest party, i broadcasts $\langle\text{"proposal"}, k, v, \mathsf{view}\rangle$ with (k, v) either being a tuple from $\mathsf{suggestions}$ or $(k, v) = (0, x_i)$. Note that (k, v) is only added to $\mathsf{suggestions}$ after i sees that $\mathsf{keyCorrect}_{i,\mathsf{view}-1}(k, v) = 1$. In addition, 0 and x_i are the first values to which key_i and $\mathsf{key\_val}_i$ are set, so as argued in Lemma 7.13, $\mathsf{keyCorrect}_{i,\mathsf{view}-1}(0, x_i) = 1$ at that time. This means that when receiving its own broadcast, i adds $(i, (k, v))$ to $\mathsf{proposals}_{i,\mathsf{view}}$ and calls $\mathsf{VLE}_{i,\mathsf{view}}$ with the asynchronous validity predicate $\mathsf{leaderCorrect}_{i,\mathsf{view}-1}$. Since there is a tuple $(i, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$ at that time, $\mathsf{leaderCorrect}_{i,\mathsf{view}}(i) = 1$. From the Termination of Output property of VLE, every honest i outputs some index ℓ and a proof π from $\mathsf{VLE}_{i,\mathsf{view}-1}$.

First, we will show that if some honest party sends an "equivocation" or a "blame" message, then the claim holds. If some honest party i sends a $\langle\text{"blame"}, k, v, \mathit{leader}, \pi, l, lv, \mathsf{view} - 1\rangle$ message, then it either did so in line 6 or in line 4. In the first case, it did so after outputting leader, π from $\mathsf{VLE}_{i,\mathsf{view}-1}$ and seeing that there is a tuple $(\mathit{leader}, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$ such that $k < \mathsf{lock}_i$. It then sent the "blame" message with $l = \mathsf{lock}_i, lv = \mathsf{lock\_val}_i$, and from Lemma 7.14, $\mathsf{lockCorrect}_i(\mathsf{lock}_i, \mathsf{lock\_val}_i) = 1$ at that time. Every honest j will eventually receive the message and see that $k < l$. Then, from the Consistency property of $\mathsf{lockCorrect}$ and from the Completeness property of $\mathsf{VLEVerify}$, j will see that $\mathsf{lockCorrect}_j(l, lv) = 1$ and that $\mathsf{VLEVerify}_{j,\mathsf{view}-1}(\mathit{leader}, \pi) = 1$. From the Validity property of $\mathsf{VLEVerify}$, there exists a tuple $(\mathit{leader}, (k', v')) \in \mathsf{proposals}_{j,\mathsf{view}-1}$. Both i and j added their respective $(\mathit{leader}, (k, v))$ and $(\mathit{leader}, (k', v'))$ after receiving the same broadcast, so $(k, v) = (k', v')$ and thus j will also proceed to the next view. Otherwise, i sent the message in line 4, after seeing that $k < l$ and having $\mathsf{lockCorrect}_i(l, lv) = 1$ and $\mathsf{VLEVerify}_{i,\mathsf{view}-1}(\mathit{leader}, \pi) = 1$. Similarly, from the Consistency property of $\mathsf{lockCorrect}$ and Agreement on Verification property of $\mathsf{VLEVerify}$, j will see that the same conditions hold. In addition, i saw that $(\mathit{leader}, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}-1}$, so j will see that the same holds and proceed to the next view.

Following similar arguments, i can either send an "equivocation" message in line 13 or in line 9. In the first case, it does so after having received two "echo" messages with values k, v, ℓ, π and k', v', ℓ', π' such that $(k, v) \neq (k', v')$, having $\mathsf{VLEVerify}_{i,\mathsf{view}-1}(\ell, \pi)$ and $\mathsf{VLEVerify}_{i,\mathsf{view}-1}(\ell', \pi')$ terminate with the output 1, and that $(\ell, (k, v)), (\ell', (k', v')) \in \mathsf{proposals}_{i,\mathsf{view}-1}$ and then sending the message. In the second case, it received an $\langle\text{"equivocation"}, k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view} - 1\rangle$ message directly and saw that the same conditions hold, after which it forwarded the message. Every honest j will then receive the $\langle\text{"equivocation"}, k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view} - 1\rangle$ message sent by i and see that

$(k,v) \neq (k',v')$. From the Agreement on Verification property of $\mathsf{VLEVerify}$, j will eventually see that $\mathsf{VLEVerify}_{j,\mathsf{view}-1}(\ell,\pi) = 1$ and that $\mathsf{VLEVerify}_{j,\mathsf{view}-1}(\ell',\pi') = 1$. From the Validity property of $\mathsf{VLEVerify}$, at that time $\mathsf{leaderCorrect}_{j,\mathsf{view}-1}(\ell) = 1$ and $\mathsf{leaderCorrect}_{j,\mathsf{view}-1}(\ell') = 1$. Therefore there are tuples of the form $(\ell,(k'',v''))$ and $(\ell',(k''',v'''))$ in $\mathsf{proposals}_{j,\mathsf{view}-1}$. Following the same logic as above, those are the same tuples that i added to $\mathsf{proposals}_{i,\mathsf{view}-1}$, so $(\ell,(k,v)),(\ell',(k',v')) \in \mathsf{proposals}_{j,\mathsf{view}-1}$, and thus j proceeds to the next view. In other words, if some honest party sends either a "blame" message or an "equivocation" in $\mathsf{view}-1$, and no honest party completes the protocol in this view, then all honest parties proceed to view.

Now assume no honest party sends a "blame" or an "equivocation" message in $\mathsf{view}-1$. In that case, after completing the call to $\mathsf{VLE}_{j,\mathsf{view}}$ with the output ℓ,π every honest j broadcasts an $\langle$"echo"$, k, v, \ell, \pi, \mathsf{view}-1\rangle$ message since it did not send a "blame" message instead. Every honest i receives that message, and from the Completeness property of $\mathsf{VLEVerify}$ eventually sees that $\mathsf{VLEVerify}_{i,\mathsf{view}-1}(\ell,\pi) = 1$ and adds a tuple to $\mathsf{echoes}_{i,\mathsf{view}-1}$. By assumption, i does not send an "equivocation" in $\mathsf{view}-1$, so after adding such a tuple for each honest party, i has $|\mathsf{echoes}_{i,\mathsf{view}-1}| \geq n-t$, and thus i updates key_i to $\mathsf{view}-1$ and $\mathsf{key\_val}_i$ to v and broadcasts a $\langle$"key"$, v, \mathsf{view}-1\rangle$ message during $\mathsf{view}-1$. From Lemma 7.13, at that time $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{view}-1,v) = 1$. Every honest j receives that message and from the Consistency property of $\mathsf{keyCorrect}_{\mathsf{view}}$, eventually sees that $\mathsf{keyCorrect}_{j,\mathsf{view}}(\mathsf{view}-1,v) = 1$ as well. After that, j adds a tuple to $\mathsf{keys}_{j,\mathsf{view}-1}$ for every honest party and sees that $\left|\mathsf{keys}_{j,\mathsf{view}-1}\right| \geq n-t$, so j sends a "lock" message to every party. Following identical reasoning, every honest i receives the "lock" message from every honest party, eventually sees that $\mathsf{lockCorrect}_i(\mathsf{view},v) = 1$ and updates its $\mathsf{locks}_{i,\mathsf{view}}$ set. After doing so for all honest parties, it sends a "commit" message. From Lemma 7.10, all correct "lock" messages contain the same value, so all honest parties sent "commit" messages with the same value. Finally, after receiving those messages from all honest parties, every honest i sees that it received $n-t$ such messages and completes the AVABA protocol in line 4. In other words, every honest party completes the protocol, reaching a contradiction. □

Lemma 7.25. *If an honest j broadcasts a $\langle$"proposal"$, k, v, \mathsf{view}\rangle$ message, then no honest party sends a $\langle$"blame"$, k, v, j, \pi, l, lv, \mathsf{view}\rangle$ message for any π, l, lv.*

Proof. Assume by way of contradiction some honest party i sends such a message. It either does so in line 6 or in line 4. In both cases, it first checked that $k < l$ and that $(j,(k,v)) \in \mathsf{proposals}_{i,\mathsf{view}}$. i adds such a tuple to $\mathsf{proposals}_{i,\mathsf{view}}$ after seeing that $\mathsf{keyCorrect}_{i,\mathsf{view}}((k,v)) = 1$. Since $\mathsf{keyCorrect}_{i,\mathsf{view}}(k,v) = 1$, either $k = 0$ or there exist at least $n-t$ tuples in $\mathsf{echoes}_{i,k}$ with $k > 0$ and thus $k \geq 0$. In addition, if i sent the message in line 6, then $l = \mathsf{lock}_i, lv = \mathsf{lock\_val}_i$, and from Lemma 7.14, $\mathsf{lockCorrect}_i(l,lv) = 1$ at that time. If i sent the message in line 4, then it first checked that $\mathsf{lockCorrect}_i(l,lv) = 1$ at that time. It cannot be the case that $l = 0$, because then $k \geq l$, reaching a contradiction. Therefore, $\left|\{j' | (j',v) \in \mathsf{keys}_{i,l}\}\right| \geq n-t$. Each tuple (j',v) was added to $\mathsf{keys}_{i,l}$ after receiving a $\langle$"key"$, v, l\rangle$ message from j'. At least $t+1$ of those tuples were added after receiving a "key" message from honest parties. Note that an honest j' sends such a message after updating $\mathsf{key}_{j'}$ to l and $\mathsf{key\_val}_{j'}$ to v. Since the $\mathsf{key}_{j'}$ field only increases throughout the protocol, $\mathsf{key}_{j'} \geq l$ from this point on. Let I be the indices of the honest parties j' that sent those "key" messages, for whom it is guaranteed that $\mathsf{key}_{j'} \geq l$ from this point on. Now observe the pair (k,v) that i chose to input into $\mathsf{VLE}_{i,\mathsf{view}}$. At the time it chose (k,v), i had $|\mathsf{suggestions}| = n-t$, so it received $\langle$"suggest"$, k', v', \mathsf{view}\rangle$ from $n-t$ parties and added corresponding tuples to $\mathsf{suggestions}$. As shown above, $|I| \geq t+1$, so at least one of those messages was received from a party j' such that $j \in I$, for whom $k' = \mathsf{key}_j \geq l$. Party i chooses the tuple (k, v) to be the one with the maximal k in $\mathsf{suggestions}$. Therefore, $k \geq k' \geq l$, reaching a contradiction. $\square$

Lemma 7.26. *If some honest party outputs v and terminates, then all honest parties eventually do so as well.*

Proof. Assume some honest party output v and terminated. It first received $\langle$"commit"$, v\rangle$ messages from $n - t$ parties, with $t + 1$ of them being honest. Let i be the first honest party that sent such a message. First we will show that no honest party sends a $\langle$"commit"$, v'\rangle$ message with any other value $v' \neq v$. Assume by way of contradiction that some honest party sends such a message, and let j be the first honest party to send such a message. Since both i and j were the first honest parties to send such messages, at the time they sent the message they received "commit" messages from at most t parties. This means that both i and j sent their respective "commit" messages in line 28 at the end of view and view' respectively. Assume without loss of generality that $\mathsf{view} \leq \mathsf{view}'$. From Lemma 7.11, in view', there are $t + 1$ honest parties that never send an "echo" message with any value $v' \neq v$. If some honest party sends a "key" message in view', then it does so after receiving $n - t$ "echo" messages with the same value (i.e. without detecting equivocation and proceeding to the next view). At least one of those messages was sent by the $t + 1$ honest parties described above, so any "key" message sent by an honest party in view' has the value v. For similar reasons, any "lock" message sent by an honest party in view' has the value v. Before sending a "commit" message, j receives $n - t$ correct "lock" messages and sends a "commit" message with the value v' of a received correct "lock" message. From Lemma 7.10, those messages had the value v, and thus $v = v'$, reaching a contradiction. Therefore, if two honest parties send "commit" messages, they send messages with the same value v.

We will now turn to show that if i completes the protocol with the output v, every honest party will do so as well. Since i completed the protocol, it received $\langle$"commit"$, v\rangle$ messages from $n - t$ parties, with $t + 1$ of them being honest. Those honest parties send their "commit" messages to all parties, and thus every honest party receives $\langle$"commit"$, v\rangle$ messages from at least $t + 1$ parties. Once that happens, every honest party sends the same message to all parties in line 2. every honest party then receives those messages from at least $n - t$ honest parties and outputs v and terminates in line 4. Note that if some honest party terminated before receiving the "commit" messages from the $t + 1$ honest parties specified above, it must have received "commit" messages from $n - t$ other parties with the same value v'. At least one of those was sent by an honest party, so $v = v'$. Therefore, before completing the protocol every honest party also receives $\langle$"commit"$, v\rangle$ messages from some $n - t$ parties and also sends a $\langle$"commit"$, v\rangle$ message as described above. $\square$

Lemma 7.27. *If all honest parties start* view *and every honest i has an input x_i such that at the time it calls the* AVABA *protocol* $\mathsf{validate}_i(x_i) = 1$*, then with constant probability all honest parties terminate during* view*.*

Proof. If at any point some honest party terminates with the value v, then from 7.17 every honest party will do so as well. From this point on, we will not deal with the case that some of the parties terminate early in view and some do not terminate at all. The first thing that an honest party does in view is calling $\mathsf{viewChange}$ and sending a "suggest" message to every party with the local fields key_i and $\mathsf{key\_val}_i$. From Lemma 7.13, $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i) = 1$ at that time, and from the Consistency of $\mathsf{keyCorrect}_{\mathsf{view}}$, for every honest j eventually $\mathsf{keyCorrect}_{j,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i) = 1$ as well. Therefore, when an honest party j receives that message, it eventually adds a tuple to

$\mathsf{suggestions}$. After receiving such a message from every honest party, j finds that $|\mathsf{suggestions}| \geq n-t$, and it broadcasts a $\langle$"proposal"$, k, v, \mathsf{view}\rangle$ message. At that time it either has $(k, v) = (0, x_j)$ and as shown in Lemma 7.13 $\mathsf{keyCorrect}_{j,\mathsf{view}}((k, v)) = 1$, or it has chosen a tuple $(k, v) \in \mathsf{suggestions}$ for which it checked that $\mathsf{keyCorrect}_{j,\mathsf{view}}((k, v)) = 1$. Therefore, when receiving its own "proposal" broadcast, it adds a tuple $(j, (k, v))$ to $\mathsf{proposals}_{j,\mathsf{view}}$ and thus $\mathsf{leaderCorrect}_{j,\mathsf{view}}(j) = 1$ at that time. j then calls $\mathsf{VLE}_{j,\mathsf{view}}$.

Before an honest i sends a "blame" or an "equivocation" message it must either output a value from $\mathsf{VLE}_{i,view}$, or find that $\mathsf{VLEVerify}_{i,view}$ terminates with the output 1 for some value. Both of those things only happen after completing $\mathsf{VLE}_{i,view}$. In other words, all honest parties participate in VLE and wait for it to terminate before any of them proceed to the next view. From the Termination of Output property of VLE, all honest parties eventually output some value when running VLE. We now prove that if the binding value ℓ^* of $\mathsf{VLE}_{\mathsf{view}}$ as defined in the α-Binding property of the VLE protocol is the index of some honest party that acted honestly when it started the VLE protocol, then all parties terminate during view. From the α-Binding property of VLE this event happens with probability $\alpha = \frac{1}{3}$, so all parties terminate during view with a constant probability.

If the binding value is indeed the index of a party that acted honestly when it started VLE, then from the Binding Verification property of VLE there is exactly one index ℓ^* for which it is possible that $\mathsf{VLEVerify}_{i,view}(\ell^*, \pi)$ terminates with the output 1 for an honest i. If an honest i adds a tuple $(j', k', v', \ell', \pi')$ to $\mathsf{echoes}_{i,\mathsf{view}}$, then it did so after receiving an "echo" message and seeing that $\mathsf{VLEVerify}_{i,\mathsf{view}}(\ell', \pi) = 1$ and that $(\ell', (k', v')) \in \mathsf{proposals}_{i,\mathsf{view}}$. Therefore, $\ell' = \ell^*$ for every such tuple. An honest i adds a tuple $(\ell^*, (k', v'))$ to $\mathsf{proposals}_{i,\mathsf{view}}$ after receiving a $\langle$"proposal"$, k', v', \mathsf{view}\rangle$ broadcast from ℓ^*, and thus all tuples in the set $\mathsf{echoes}_{i,\mathsf{view}}$ have the same values k', v', which prevents an honest party from sending an "equivocation" message in line 13. In addition, no honest i sends an "equivocation" message in line 9 after receiving an $\langle$"equivocation"$, k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view}\rangle$ message because $\ell = \ell' = \ell^*$, and thus $(k, v) = (k', v')$. We would now like to show that no honest party i sends a "blame" message in view. If an honest party sends a $\langle$"blame"$, k, v, \ell, \pi, l, lv\rangle$ message, it does so either in line 6 or in line 4. In the first case, it did so after outputting ℓ, π from $\mathsf{VLE}_{i,\mathsf{view}}$ and from the Completeness and Binding Verification properties of VLE, $\ell = \ell^*$. In the second case, it checked that $\mathsf{VLEVerify}(\ell, \pi) = 1$ and for the same reasons $\ell = \ell^*$. Before starting VLE, ℓ^* broadcasts $\langle$"proposal"$, k, v, \mathsf{view}\rangle$. For similar reasons as above, if an honest i has a tuple $(\ell^*, (k', v')) \in \mathsf{proposals}_{i,\mathsf{view}}$, then $(k', v') = (k, v)$. Therefore, i sends the message $\langle$"blame"$, k, v, \ell^*, \pi, l, lv, \mathsf{view}\rangle$, contradicting Lemma 7.16.

Honest parties only proceed to $\mathsf{view} + 1$ after sending either a "blame" or an "equivocation" message, so no honest party proceeds to $\mathsf{view} + 1$. Since no honest i sends a "blame" message, each one sends an $\langle$"echo"$, k, v, \ell^*, \pi, \mathsf{view}\rangle$ message after completing the $\mathsf{VLE}_{i,view}$ call. From the Completeness property of VLE, $\mathsf{VLEVerify}_{i,view}(\ell^*, \pi)$ eventually terminates with the output 1. Since i doesn't send an "equivocation" message in view, it then adds a tuple to $\mathsf{echoes}_{i,\mathsf{view}}$. After such a tuple is added for every honest party, i sees that $|\mathsf{echoes}_{i,\mathsf{view}}| = n - t$ and it sends a message $\langle$"key"$, v, \mathsf{view}\rangle$ to all parties after updating key_j to view and $\mathsf{key\_val}_j$ to v. From Lemma 7.13, at that time $\mathsf{keyCorrect}_{i,\mathsf{view}+1}(\mathsf{view}, v) = 1$ so eventually $\mathsf{keyCorrect}_{j,\mathsf{view}+1}(\mathsf{view}, v) = 1$ for every honest j from the consistency property of $\mathsf{keyCorrect}_{\mathsf{view}+1}$. Therefore, when receiving that message, every honest j eventually sees that the message is correct and adds a pair (i, v) to $\mathsf{keys}_{i,\mathsf{view}}$. After adding such a pair for every honest party, j has $\left|\mathsf{keys}_{i,\mathsf{view}}\right| = n - t$ and it sends a "lock" message. Using identical arguments, eventually every honest party sends a "commit" message. Finally, after receiving a "commit" from $n-t$ parties, every honest party terminates. $\square$

7.8 Proof of AVABA

Theorem 7.28. *Protocol* AVABA *is a Validated Asynchronous Byzantine Agreement protocol resilient to* $t < \frac{n}{4}$ *Byzantine parties.*

Proof. Each property is proven individually.

Correctness. This property was proven in Lemma 7.17.

Validity. Before some honest i outputs a value v, it sends $\langle$"commit"$, v, \mathsf{view}\rangle$ message. As discussed in the proof of Lemma 7.17, at least $n-t$ parties sent "key" messages in view with the value v as well. At least one of those parties is honest. Party i only sends a $\langle$"key"$, v, \mathsf{view}\rangle$ message after receiving an $\langle$"echo"$, k, v, \ell, \pi, \mathsf{view}\rangle$ message such that $\mathsf{VLEVerify}_{i,view}(\ell, \pi)$ terminates and $(\ell, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$. Party i adds a tuple $(\ell, (k, v))$ to $\mathsf{proposals}_{i,\mathsf{view}}$ after receiving a $\langle$"proposal"$, k, v, \mathsf{view}\rangle$ from ℓ and having $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v) = 1$. Before $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$ terminates with the output 1, i sees that $\mathsf{validate}_i(v) = 1$. Therefore, $\mathsf{validate}_i(v) = 1$ at that time.

Termination. If at any point an honest party terminates, from Lemma 7.17, all honest parties do so as well. Now assume that every honest party i has an input x_i such that $\mathsf{validate}_i(x_i) = 1$ at the time it calls AVABA and that all honest parties participate in the protocol. Observe some view, and assume no honest party terminated during view' for any $\mathsf{view}' < \mathsf{view}$. In that case, from Lemma 7.15 all honest parties eventually reach view. Then, from Lemma 7.18, with constant probability all honest parties terminate during view. In order for an honest party not to terminate by view, that constant probability event must not have happened in each one of the previous views. The honest parties run the VLE protocol with independent randomness in each view and thus for any adversary's strategy, there is an independent constant probability of terminating in each view. Therefore, the probability of reaching a given view decreases exponentially with the view number and thus approaches 0 as view grows. In other words, all honest parties almost surely terminate.

Quality. Assume some honest party i completed the protocol, otherwise the claim holds trivially. This means that it at least completed the VLE protocol in $\mathsf{view} = 1$. From the α-Binding property of VLE, with probability α or greater the binding value is the index ℓ^* of some party that behaved honestly when starting VLE. From the Completeness and Validity properties of VLE, at that time $\mathsf{leaderCorrect}_{i,\mathsf{view}}(\ell^*) = 1$ and thus there exists some tuple $(\ell^*, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$ at that time, which was added after receiving a $\langle$"proposal"$, k, , v, \mathsf{view}\rangle$ broadcast from ℓ^*. Using the same arguments as the ones made in Lemma 7.18, in that case, no honest party sends a "blame" or an "equivocation" message during view. Then, following similar logic to the one in Lemma 7.18, every honest party that hasn't committed due to a message from an earlier view eventually terminates after sending a "commit" message with the value v proposed by party i. No party can commit due to a message from an earlier view because there is no earlier view. Therefore, every honest party that participates in view and outputs a value from VLE, terminates and outputs the value v that ℓ^* proposed. Before sending its proposal, ℓ^* sees that $|\mathsf{suggestions}| = n - t$. ℓ^* only adds a tuple to $\mathsf{suggestions}$ after receiving the first $\langle$"suggest"$, k, v, \mathsf{view}\rangle$ message from each party. Each of those tuples must have $k < \mathsf{view} = 1$ because $\mathsf{keyCorrect}_{i,1}(k, v) = 1$. At that time no honest party updated its key_j and $\mathsf{key\_val}_j$ fields, so they send messages with $k = 0$. Since at least one of the $n - t$ messages was sent by an honest party, there exists some $(k, v) \in \mathsf{suggestions}$ such that $k = 0$, and as shown above there is no such tuple with $k > 0$. Therefore, when computing choosing the tuple (k, v), i sees that the tuple with maximal k in $\mathsf{suggestions}$ has $k = 0$. Party i

then sets $(k, v) = (0, x_i,)$, with x_i being its input to the AVABA protocol. As shown above, with constant probability all honest parties that start view output x_i, completing the proof. □

8 Agreement on a Core Set

8.1 Definition

An Agreement on a Core Set protocol is a protocol in which parties have no input, but they have access to an asynchronous validity predicate $\mathsf{validate} : [n] \to \{0, 1\}$. Furthermore, it is guaranteed that for each honest party i, there is a set S_i such that $|S_i| \geq n - t$ and eventually $\forall k \in S_i$, $\mathsf{validate}_i(k)$ terminates with the output 1. Each party outputs a set $S \subseteq [n]$ from the protocol. An Agreement on a Core Set protocol has the following properties:

- **Agreement.** All honest parties that complete the protocol output the same set S from the protocol.
- **Validity.** If an honest party i outputs S from the protocol then $S \subseteq [n]$, $|S| \geq n - t$, and $\mathsf{validate}_i(k) = 1$ at that time for every $k \in S$.
- **Termination.** All parties almost-surely terminate.

Using the above AVABA protocol, we construct a protocol ACS for agreement on a core set. Each party i has access to an asynchronous validity predicate $\mathsf{validate}_i$ such that eventually for at least $n - t$ indices $k \in [n]$, $\mathsf{validate}_i(k) = 1$.

Protocol 8.1: $\mathsf{ACS}_i()$

```
1: $S_i \leftarrow \emptyset$
2: call $\mathsf{validate}_i(k)$ for every $k \in [n]$
3: upon $\mathsf{validate}_i(k)$ terminating with the output 1 for some $k \in [n]$, do
4:     $S_i \leftarrow S_i \cup \{k\}$
5:     if $|S_i| = n - t$ then
6:         call $\mathsf{AVABA}_i(S_i)$ with the asynchronous validity predicate $\mathsf{ACSValidity}_i$
7: upon AVABA terminating with the output $S$, do
8:     output $S$ and terminate
```

Protocol 8.2: $\mathsf{ACSValidity}_i(S)$

```
1: if $S \subseteq [n], |S| \geq n - t$ then
2:     upon $S \subseteq S_i$, do
3:         output 1 and terminate
```

8.2 Security Analysis

We start by showing that ACSValidity is indeed an asynchronous validity predicate and then show that ACS is a protocol for agreeing on a core set. We then prove that the protocol described in Protocol 8.1 is indeed a protocol for agreement on a core set.

Lemma 8.3. $\mathsf{ACSValidity}$ *is an asynchronous validity predicate.*

Proof. Let i, j be two honest parties and assume $\mathsf{ACSValidity}_i(S) = b$ at some point in time. Note that $\mathsf{ACSValidity}$ only outputs 1, so $b = 1$. We will show each property independently

Finality. If $\mathsf{ACSValidity}_i(S) = 1$, then i saw that $S \subseteq [n], |S| \geq n - t$ and that $S \subseteq S_i$. The first two conditions clearly continue to hold. Honest parties never remove indices from their S_i sets, so $S \subseteq S_i$ will continue to hold, and thus $\mathsf{ACSValidity}_i(S)$ will terminate with the output 1 in the future as well.

Correctness. Similarly to above, i saw that $S \subseteq [n], |S| \geq n - t$ and that $S \subseteq S_i$. Any honest j that calls $\mathsf{ACSValidity}_j(S)$ will also see that $S \subseteq [n]$ and that $|S| \geq n - t$. In addition, i added indices k to S_i after calling $\mathsf{validate}_i(k)$ and getting the output 1. When starting the ACS protocol, j also calls $\mathsf{validate}_j(k)$ for every $k \in [n]$, and from the Consistency of the $\mathsf{validate}$, $\mathsf{validate}_j(k)$ will eventually return 1 for every $k \in S_i$. After receiving such an output for every $k \in S_i$, $S_i \subseteq S_j$ and thus $S \subseteq S_j$ as well, at which point $\mathsf{ACSValidity}_j(S)$ terminates with the output 1 as well. □

Theorem 8.4. ACS *is an Agreement on a Core Set protocol resilient to* $t < \frac{n}{4}$ *Byzantine parties.*

Proof. We will prove each property independently.

Agreement. Assume two honest parties output sets from the ACS protocol. Those sets are the parties' output from the AVABA protocol, and thus from the Agreement protocol of AVABA they are equal.

Validity. Assume some honest party i outputs a set S from the ACS protocol. That set is its output from the AVABA protocol, so $\mathsf{ACSValidity}_i(S) = 1$ at that time from the Validity property of AVABA. This means that $S \subseteq [n]$, $|S| \geq n - t$ and that at that time $S \subseteq S_i$. Note that i only adds indices $k \in [n]$ for which $\mathsf{validate}_i(k) = 1$ to S_i, and thus $\forall k \in S$, $\mathsf{validate}_i(k) = 1$ at that time.

Termination. Every honest i starts the protocol by calling $\mathsf{validate}_i(k)$ for every $k \in [n]$. By assumption, there exists a set $S \subseteq [n]$ such that $|S| \geq n - t$ and for every $k \in S$, eventually $\mathsf{validate}_i(k) = 1$. After i sees that this is the case for every $k \in S$, it adds each of those indices to S_i and sees that $|S_i| = n - t$. Following that it calls the AVABA with the input S_i. Note that $S_i \subseteq [n]$, $|S_i| \geq n - t$ and that at that time $S_i \subseteq S_i$. In other words, every honest party calls AVABA with a valid input and uses the asynchronous validity predicate $\mathsf{ACSValidity}$. From the Termination property of AVABA, all honest parties complete the protocol, after which they output a value from the ACS protocol and terminate. □

9 Asynchronous Secure Computation

Plugging our new AVSS and the ACS protocols in the recent asynchronous MPC protocol of [3] leads to an efficiency improvement. Specifically, instead of MPC with $\mathcal{O}((Cn + Dn^2 + n^7) \log n)$ communication and $\mathcal{O}(D + \log n)$ expected-time using the ACS of [15] and the AVSS of [18], we obtain $\mathcal{O}((Cn + Dn^2 + n^4) \log n)$ communication and $\mathcal{O}(D)$ expected-time.

The MPC protocol of [3] has the following structure:

Offline: beaver triplets generation. The goal is to distribute (Shamir, univariate degree-t) shares of random secret values a, b, and c, such that $c = ab$. This is performed as follows:

1. **Triplets with and without a dealer.** Each party first distributes secrets a_i, b_i, c_i such that $c_i = a_i \cdot b_i$. If the computation requires C multiplications in total, each dealer has to generate

C/n such triplets. Using the previous best AVSS, this step requires $\mathcal{O}(n^4 \log n + C \log n)$ communication for each dealer, i.e., a total of $\mathcal{O}(n^5 \log n + Cn \log n)$ for all parties combined. Using our AVSS protocol, this step is automatically reduced to $\mathcal{O}(n^4 \log n + Cn \log n)$. Both protocols are constant expected number of rounds.

2. **Agreeing on a core set (ACS):** The parties then have to agree on a core set of parties whose beaver triplets generation terminated and will be considered in the sequel of the computation. The communication cost of the ACS from [15] is $\mathcal{O}(n^7 \log n)$ with $\mathcal{O}(\log n)$ rounds, which we reduce to $\mathcal{O}(n^4 \log n)$ and expected constant time.

3. **Triplets with no dealer:** Once agreed on the core, there is a way to extract $\mathcal{O}(n)$ triplets with no dealer (i.e., when no party knows the secrets a, b and c) from $\mathcal{O}(n)$ triplets with a dealer (where the dealer knows the secrets a, b and c). This step costs $\mathcal{O}(n^2 \log n + Cn \log n)$.

To conclude, generating C multiplication triplets costs a total of $\mathcal{O}(n^4 \log n + Cn \log n)$.

Online. The second step follows the standard structure where each party shares its input (using AVSS), and the parties evaluate the circuit gate-by-gate while consuming the multiplication triplets they have generated, using the method of [18]. Using our AVSS, the sharing phase is reduced from $\mathcal{O}(n^5 \log n)$ to $\mathcal{O}(n^4 \log n)$. The computation of the circuit using the multiplication triplets remains $\mathcal{O}((Cn + Dn^2) \log n)$ with an $\mathcal{O}(D)$ time requirement.

In total, using our ACS and AVSS, we obtain a protocol that requires $\mathcal{O}((Cn + Dn^2 + n^4) \log n)$ communication and $\mathcal{O}(D)$ time.

References

[1] Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. Asymptotically free broadcast in constant expected time via packed vss. In *Theory of Cryptography: 20th International Conference, TCC 2022, Chicago, IL, USA, November 7–10, 2022, Proceedings, Part I*, pages 384–414. Springer, 2023.

[2] Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. Detect, pack and batch: Perfectly-secure MPC with linear communication and constant expected time. In *Advances in Cryptology - EUROCRYPT 2023*, volume 14005 of *Lecture Notes in Computer Science*, pages 251–281. Springer, 2023.

[3] Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. Perfect Asynchronous MPC with Linear Communication Overhead. In *Advances in Cryptology - EUROCRYPT 2024*, Lecture Notes in Computer Science. Springer, 2024. to appear.

[4] Ittai Abraham, Danny Dolev, and Gilad Stern. Revisiting asynchronous fault tolerant computation with optimal resilience. *Distributed Computing*, 35(4):333–355, 2022.

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

[6] 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*, page 337–346, New York, NY, USA, jul 2019. ACM.

[7] Ittai Abraham and Gilad Stern. Information theoretic hotstuff. In *OPODIS*, volume 184 of *LIPIcs*, pages 11:1–11:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[8] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Balanced byzantine reliable broadcast with near-optimal communication and improved computation. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, page 399–417, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Laasya Bangalore, Ashish Choudhury, and Arpita Patra. Almost-surely terminating asynchronous byzantine agreement revisited. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 295–304, 2018.

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

[11] Michael Ben-Or and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Comput.*, 16(4):249–262, 2003.

[12] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, page 183–192, New York, NY, USA, 1994. Association for Computing Machinery.

[13] Gabriel Bracha. An asynchronous [(n - 1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on principles of distributed computing*, page 154–162, New York, NY, USA, 1984. Association for Computing Machinery.

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

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

[16] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 42–51, New York, NY, USA, 1993. Association for Computing Machinery.

[17] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999.

[18] Ashish Choudhury and Arpita Patra. An efficient framework for unconditionally secure multiparty computation. *IEEE Transactions on Information Theory*, 2016.

[19] Ran Cohen, Pouyan Forghani, Juan Garay, Rutvik Patel, and Vassilis Zikas. Concurrent asynchronous byzantine agreement in expected-constant rounds, revisited. *Cryptology ePrint Archive*, 2023.

[20] Thomas Dinsdale-Young, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. Afgjort: A partially synchronous finality layer for blockchains. In Clemente Galdi and Vladimir Kolesnikov, editors, *Security and Cryptography for Networks - 12th International Conference, SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings*, volume 12238 of *Lecture Notes in Computer Science*, pages 24–44. Springer, 2020.

[21] Sisi Duan, Xin Wang, and Haibin Zhang. Practical signature-free asynchronous common subset in constant time. *IACR Cryptol. ePrint Arch.*, page 154, 2023.

[22] Paul Neil Feldman. *Optimal algorithms for Byzantine agreement.* PhD thesis, Massachusetts Institute of Technology, 1988.

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

[24] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-ng: Fast asynchronous BFT consensus with throughput-oblivious latency. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1187–1201. ACM, 2022.

[25] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In *Advances in Cryptology - CRYPTO 2019*, volume 11693 of *Lecture Notes in Computer Science*, pages 85–114. Springer, 2019.

[26] Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *Journal of Computer and System Sciences*, 75(2):91–112, 2009.

[27] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*, volume 16. Elsevier, 1977.

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

[29] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery.

A Efficiency

In all of the discussions below, we consider sets of $\mathcal{O}(n)$ indices as being of size $\mathcal{O}(n)$ bits. While storing and sending these values as indices requires $\mathcal{O}(\log n)$ bits per index, it is possible to represent these sets as bitmaps of $\mathcal{O}(n)$ bits instead, with 1 indicating that a value is in the set and 0 indicating that it is not.

A.1 Verifiable Party Gather Efficiency

In the following discussion, we assume the existence of a broadcast protocol that terminates in $\mathcal{O}(1)$ rounds with $\mathcal{O}(b(m))$ bits sent when broadcasting inputs of size $\mathcal{O}(m)$ vits. Concretely, we use the broadcast protocol of [8] in which parties send $\mathcal{O}(n^2 \log n + n \cdot m)$ bits when broadcasting a message of size $\mathcal{O}(m)$. In addition, we assume that if $\mathsf{validate}_i(x) = 1$ for some honest i at some time, $\mathsf{validate}_j(x)$ will terminate a constant number of rounds after that time for every honest j.

When using inputs of size $\mathcal{O}(n)$ and setting $b(n) = n^2 \log n$, as achieved by the above protocol, we see in the following theorem that the Gather protocol requires $\mathcal{O}(n^3 \log n)$ bits and $\mathcal{O}(1)$ rounds.

Theorem A.1. *The total number of bits sent in the* Gather *protocol is* $\mathcal{O}(nb(n))$ *and all parties terminate after* $\mathcal{O}(1)$ *rounds.*

Proof. In the protocol, every party sends a constant number of broadcasts, totaling in $\mathcal{O}(n \cdot b(n))$ bits sent. In addition, every honest i will receive a broadcast $\langle 1, S_j \rangle$ from every honest j after $\mathcal{O}(1)$ rounds. By assumption, $\forall x \in S_j$ $\mathsf{validate}_j(x) = 1$ at the time j calls the protocol, and thus $\mathsf{validate}_i(x) = 1$ will hold $\mathcal{O}(1)$ rounds after that. Following that, every honest party will send a second broadcast up to $\mathcal{O}(1)$ rounds later, and terminate after receiving those broadcasts $\mathcal{O}(1)$ rounds after that. □

A.2 Packed AVSS Efficiency

Theorem A.2. *The total communication complexity of* Share *is* $\mathcal{O}(n^3 \log n)$*, and the communication complexity of each call to* $\mathsf{Reconstruct}(k)$ *is* $\mathcal{O}(n^2 \log n)$*. In addition, if the dealer is honest, all parties terminate after* $\mathcal{O}(1)$ *rounds, and if some honest party terminates for a Byzantine leader, all parties terminate* $\mathcal{O}(1)$ *rounds after it. Furthermore, all parties complete* $\mathsf{Reconstruct}(k)$ *after a constant number of rounds.*

Proof. In the Share protocol, the dealer starts by sending every party two polynomials of degree $\mathcal{O}(n)$, for a total of $\mathcal{O}(n^2 \log n)$ bits (assuming each field element in $\mathbb{F}$ can be represented in $\log n$ bits). In addition, parties send each other "values" and "col" messages with a constant number of field elements, "ok" messages with indices, "star" messages with a constant number of sets of size $\mathcal{O}(n)$, and finally "done" messages. The largest of those messages contains $\mathcal{O}(n)$ bits, resulting in a total of $\mathcal{O}(n^3)$ bits. In each call to $\mathsf{Reconstruct}(k)$, parties only send a single "rec" message.

As can be seen in the proof of termination, when the dealer is honest all parties receive the "polynomials" message, then send a "values" message, and then an "ok" message for every honest party in 3 rounds. After receiving all of those honest messages in one round, parties find a star and send a "star" message. After receiving those "star" messages and the "values" messages from all honest parties, every honest party interpolates a polynomial q_i and sends a "col" message, as well as a "done" message. Parties receive those messages and interpolate p_i, at which point they receive

$n - t$ "done" messages and have $p_i \neq \perp, q_i \neq \perp$, they then terminate after a constant number of rounds.

Now assume some honest party completes the Share protocol. As shown in the proof of termination, at that time it already received $n - t$ "done" messages and $n - 2t$ honest parties sent "star" messages. All parties receive these messages in a constant number of rounds, forward "done" message to all parties, and start interpolating a polynomial for each received star. For the same reasons as above, all parties now terminate in a constant number of rounds.

Finally, for the reconstruction protocol, all parties simply send a single "rec" message and terminate a single round later, after receiving all honest parties' messages. □

A.3 Verifiable Leader Election Efficiency

As above, we assume the existence of a broadcast protocol that terminates in $\mathcal{O}(1)$ rounds with $\mathcal{O}(b(n))$ bits sent when broadcasting inputs of size $\mathcal{O}(n)$. We use the same broadcast protocol with $b(n) = n^2 \log n$. In addition, we assume that if $\mathsf{validate}_i(x) = 1$ for some honest i at some time, $\mathsf{validate}_j(\ell)$ will terminate a constant number of rounds after that time for every honest j. In the VLE protocol, we use the packed AVSS protocol described in Section 4. This protocol has $\mathcal{O}(n^3 \log n)$ bit complexity for sharing $\mathcal{O}(n)$ secrets. We can either use the $\mathsf{Sum} - \mathsf{Reconstruct}$ protocol described in Protocol 4.10 to get an efficiency of $\mathcal{O}(n^2 \log)$ communication complexity per reconstruction or can simply reconstruct each secret in the sum individually to achieve $\mathcal{O}(n^3 \log n)$ complexity per sum reconstructed, which is good enough for our purposes.

Theorem A.3. *The total number of bits sent in the* VLE *protocol is* $\mathcal{O}(n \cdot (b(n) + n^3 \log n))$ *and all parties terminate after* $\mathcal{O}(1)$ *rounds.*

Proof. Each party starts by sharing n values, for a total of $\mathcal{O}(n^4 \log n)$ sent bits. Each party broadcasts a constant number of messages of size $\mathcal{O}(n)$ resulting in $\mathcal{O}(nb(n))$ sent bits. The parties then run the Gather protocol in which $\mathcal{O}(nb(n))$ more bits are sent. Following that, parties reconstruct $\mathcal{O}(n)$ sums of secrets, requiring a final $\mathcal{O}(n^3 \log n)$ bits. In total, parties send $\mathcal{O}(n(b(n) + n^3 \log n))$ bits. Each call to the broadcast, reconstruct, or gather protocols terminates after $\mathcal{O}(1)$ rounds. In addition, all honest parties complete the share invocations with honest dealers after a constant number of rounds, and if some party completes a share invocation with a Byzantine dealer before that (and adds it to its $\mathsf{dealers}$ set), every other honest party will complete it in a constant number of rounds after that. Therefore, every call to the share protocol which honest parties use in their $\mathsf{dealers}$ sets, and in their output from the gather protocol completes in a constant number of rounds, totaling in a constant number of rounds in the whole protocol. □

A.4 AVABA Efficiency

We set m to be the size of inputs to the protocol and we use the same broadcast protocol as described in the previous efficiency sections. Similarly to above, define $\mathcal{O}(b(m))$ to be the number of bits sent when broadcasting messages with $\mathcal{O}(m)$ values, and have $b(m) = n^2 \log n + n \cdot m$. In the theorem below, we get an Asynchronously Validated Asynchronous Byzantine Agreement protocol with an efficiency of $\mathcal{O}(n^4 \log n + n^2 \cdot m)$.

Theorem A.4. *The expected total number of bits sent in the* AVABA *protocol is* $\mathcal{O}(n \cdot (b(n + m) + n^3 \log n + n \cdot m))$ *and all parties terminate after* $\mathcal{O}(1)$ *rounds in expectation.*

Proof. In each view, every party sends a constant number of messages of size $\mathcal{O}(n+m)$ to all parties, totaling in $\mathcal{O}(n^3+n^2\cdot m)$ bits. In addition, each party broadcasts messages of size $\mathcal{O}(n+m)$, totaling in $\mathcal{O}(nb(n+m))$ additional sent bits. Finally, each party calls VLE once in each view, adding $\mathcal{O}(n\cdot(b(n)+n^3\log n))$ total bits. Summing all of these terms gives the result of $\mathcal{O}(n\cdot(b(n+m)+n^3\log n+n\cdot m))$ total bits in each view. In addition, each view consists of protocols that terminate in $\mathcal{O}(1)$ rounds, yielding a constant number of rounds per view.

As shown in the proof of termination, all parties terminate in a given view with probability $\frac{1}{3}$ or greater. This means that the expected number of views required in the protocol is at most 3, meaning that the protocol also requires an expected constant number of rounds and $\mathcal{O}(n\cdot(b(n+m)+n^3\log n+n\cdot m))$ words to be sent in expectation overall. $\square$

A.5 ACS Efficiency

In the ACS protocol, parties simply call the AVABA protocol with inputs of size $\mathcal{O}(n)$ without sending additional messages. Therefore, the ACS protocol has the same efficiency as the AVABA protocol, yielding an ACS protocol with $\mathcal{O}(n^4\log n)$ communication complexity and $\mathcal{O}(1)$ expected rounds.