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

Ittai Abraham* Gilad Asharov† Arpita Patra‡ Gilad Stern§

December 5, 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 and statistical security setting used this same paradigm for solving ACS. Using all known asynchronous binary Byzantine agreement protocols, this type of ACS has $\Omega(\log n)$ expected round complexity, which results in such a bound on the round complexity of asynchronous MPC protocols as well (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. Along the way, we also construct a new perfectly secure packed asynchronous verifiable secret sharing (AVSS) protocol with just $\mathcal{O}(n^3 \log n)$ communication complexity, improving the state of the art by a factor of $\mathcal{O}(n)$. This leads to a more efficient asynchronous MPC that matches the state-of-the-art synchronous MPC.

*Intel Labs, Ittai.abraham@intel.com

†Bar-Ilan University, Gilad.Asharov@biu.ac.il. Supported by the Israel Science Foundation (grant No. 2439/20), and by JPM Faculty Research Award, and by the European Union (ERC, FTRC, 101043243). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

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

§Tel Aviv University, giladstern@tauex.tau.ac.il. Supported in part by ISF 2338/23, AFOSR Award FA9550-23-1-0387, AFOSR Award FA9550-23-1-0312, and an Algorand Foundation grant. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government, AFOSR or the Algorand Foundation.

Contents

1 Introduction

Broadly, there are two main network conditions where secure multiparty computation protocols were studied. The first is the synchronous setting, where all messages sent between honest parties arrive after some known bounded delay. The choice of this delay bound is critical: setting a large delay causes the protocol to be inefficient and slow, while setting a small delay might lead to non-termination. The second category is the asynchronous model, where each message sent between honest parties arrives after some finite delay. This model allows protocols to dynamically adjust to adversarial network conditions and terminate even when the adversary can adaptively manipulate the delays.

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) [11] 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 [5, 11, 13], *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 [11, 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 successfully completes the AVSS, then all honest parties will eventually also receive valid shares. However, some instance of corrupted dealers might never terminate, and some instances of honest dealers might be very slow (due to adversarial dealys). 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. 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 terminated or will eventually terminate 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 will never terminate.

In terms of round complexity, the best one can hope for is reaching agreement in constant expectation [25]. 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. All known asynchronous binary agreement protocols follow a geometric distribution, and composing n such protocols in parallel, 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] As opposed to some claims in the literature, the work of [12] does not provide an $O(1)$ expected time ACS; see Section 1.2.

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 *asynchronous validation*; (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 a new notion that we introduce, called "AVABA". This is an information-theoretic version of Validated Asynchronous Byzantine Agreement (VABA [7]), where the external validity function is replaced with asynchronous validation. Our AVABA protocol is perfectly secure and resilient to $t < n/4$ corruptions. For inputs of size $\mathcal{O}(n)$ 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 validation. In the MPC setting, this validation 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 [10, 24] 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 asynchronous validation (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 validation 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 validation 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. We also extend our results to the statistical settings, and derive resilience to $t < n/3$ corruptions. Set Section 2.3.1 for further details.

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

Theorem 1.3. *There exists a perfectly secure protocol for asynchronous verifiable secret sharing resilient to $t < n/4$ malicious 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 [11] achieves total communication complexity of $\mathcal{O}(n^4 \log n)$ for sharing *one* secret, and the one by [30, 19] obtains a 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 [30, 19], improving the communication by a factor of $\mathcal{O}(n)$.

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

Corollary 1.4. *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 [30, 19], the cost of the entire MPC is $\mathcal{O}((Cn + Dn^2 + n^7) \log n)$ and $\mathcal{O}(D + \log n)$ expected-time. Our work improves both the communication and round complexities. See Section 2.3.2.

Synchronous vs. asynchronous MPC. We conclude this section by noting an accepted claim regarding the relationship between synchronous and asynchronous MPC is that (see, e.g., Nielsen [29]):

> *"Synchronous MPC has higher security and requires less communication than asynchronous MPC."*

The first is part of the sentence is due to the different optimal bounds: Optimal resilient in perfect MPC in the synchronous setting is $t < n/3$ whereas in the asynchronous setting is $t < n/4$. The second part, as claimed in [29] is due to the fact that "Another advantage of synchronous MPC is that we know how to construct them with much less communication in terms of bits sent than the asynchronous ones".

Our work shows that there is no support for the second part of the claim, at least, for perfect MPC. Specifically, the current most efficient perfect *synchronous* MPC [2] protocol achieves the **exact same complexity** as ours: $\mathcal{O}((Cn + Dn^2 + n^4) \log n)$ with $\mathcal{O}(D)$ expected rounds.[2] Note that while one could run asynchronous protocols in synchronous networks, asynchronous MPC protocols are designed for lower corruption thresholds and only take $n - t$ inputs into consideration. This even gives rise to the hope that perhaps, one can even construct a more efficient asynchronous MPC protocol than a synchronous protocol, and our understanding of the relationship between synchronous and asynchronous protocols is still lacking.

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 the asynchronous setting is reaching *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 [11] suggest a protocol called Agreement on a Core Set (ACS). To the best of our knowledge, all previous asynchronous

[2] A somewhat incomparable result [26] 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. We are not aware of a comparable result in the asynchronous case.

MPC protocols (in the perfect and statistical security setting) use the same ACS protocol suggested by [11]. 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 P_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 [10, 24] in this setting require the expected total of $\mathcal{O}(n^7 \log n)$ communication bits for the ACS.

On Ben-Or and El-Yaniv's work. The work of [11] 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 [12] (published 10 years after [11] cites them) deals with executing n concurrent instances of Byzantine Agreement. The first part of [12] 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 [12] explicitly do **not** mention that they can solve ACS in the asynchronous model. Indeed, the stated results of [12] and the techniques of [12] 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 [13]. In [13]'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 [12] 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.* [23] 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, [23] requires a strong leader election, which, to the best of our knowledge, requires a DKG and a computationally bounded adversary for the required complexity. An additional recent work in the cryptographic setting is that of Das *et al.* [22]. Their work only relies on a cryptographic hash function without additional setup assumptions and achieves $\mathcal{O}(\lambda n^3)$ bit complexity and $\mathcal{O}(1)$ time complexity.

Cohen *et al.* [21] 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 [21] 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.

A newer version of [21] was made public after an earlier version of our work appeared online [4]. In this newer version, the authors use the information theoretic Gather protocol defined in our paper in order to solve ACS (the gather protocol in [6] relied on cryptographic assumptions). The resulting ACS of [21] uses our techniques, and requires at least $\Omega(n^5\kappa^2 + n^6)$ bits of communication and is statistically secure, where κ is a statistical security parameter. This bound stems from each party having to share n secrets. Using the packed secret sharing protocol of [20], each such sharing requires $\Omega(n^4\kappa^2 + n^5)$ bits of communication. Constructing a statistical ACS protocol using our techniques will also have the same bottleneck of n packed sharings, resulting in the same communication complexity. In comparison, our perfect ACS protocol can be used to solve ACS in just $\mathcal{O}(n^4 \log n)$ bits of communication with perfect security.

2 Technical Overview

We provide a high level overview of our main techniques. In Section 2.1 we provide a brief overview of our AVSS protocol, which might be of an independent interest. In Section 2.2 we provide the overall structure of the ACS protocol, and our AVABA protocol. We also provide some extensions to our result in Section 2.3.

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

Protocol	Bit Complexity Complexity	Expected Time Complexity	Security
BCG [11]	$\mathcal{O}(n^7 \log n)$	$\mathcal{O}(\log n)$	perfect
BKR [13]	$\Omega(n^{13}k^2)$	$\mathcal{O}(\log n)$	statistical
FIN [23]	$\mathcal{O}(\lambda n^3)$	$\mathcal{O}(1)$	computational with DKG
Cohen et al. [21]	$\mathcal{O}(n^5\kappa^2 + n^6)$	$\mathcal{O}(1)$	statistical
Das et al. [22]	$\mathcal{O}(\lambda n^3)$	$\mathcal{O}(1)$	computational
This work (statistical)	$\mathcal{O}(n^5\kappa^2 + n^6)$	$\mathcal{O}(1)$	statistical
This work (perfect)	$\mathcal{O}(n^4 \log n)$	$\mathcal{O}(1)$	perfect

Table 1: A comparison of ACS schemes, in terms of: (1) the total number of bits sent; (2) the expected number of rounds; (3) the type of security provided by the protocol (either perfect security, statistical security, or computational security reliant on cryptographic assumptions). All of the works are optimally resilient, i.e. assume $n > 4t$ for perfectly secure schemes and $n > 3t$ for statistical and computational schemes. κ and λ are statistical and cryptographic security parameters respectively. Both [21] and our statistical protocol are evaluated using the packed ACSS protocol of [20] for n instances of packed secret sharing, assuming sharing $\mathcal{O}(n)$ secrets costs $\mathcal{O}(n^4\kappa^2 + n^5)$.

receive messages from more than $n - t$ parties only when they are certain that not all messages from honest parties have been received.

Notation. To describe the ith party, we use i or P_i interchangeably. In Section 2.1 we overview our improvement in the communication complexity of AVSS. In Section 2.2 we overview our new asynchronously validated asynchronous Byzantine Agreement.

2.1 Packed Asynchronous Verifiable Secret Sharing

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 [11]. We start with a quick overview of the AVSS protocol of Ben-Or, Canetti and Goldreich [11]. 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 each party P_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 P_i forwards the values $f_i(j), g_i(j)$ to every P_j.
3. When a party P_i receives a forwarded pair of values $f_j(i), g_j(i)$ from some party P_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 P_i broadcasts $\langle \texttt{ok}, i, j \rangle$, signifying that P_i agrees with P_j.
4. The dealer initiates a graph G with $V = [n]$ and $E = \emptyset$. Then, upon receiving broadcasted messages $\langle \texttt{ok}, i, j \rangle$ and $\langle \texttt{ok}, j, i \rangle$ it adds the edge (i, j) to E. It then looks for a clique $K \subseteq [n]$ in G. If found, it broadcasts $\langle \texttt{clique}, K \rangle$. 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 $\langle \texttt{clique}, K \rangle$, 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 P_j terminates, all honest parties will eventually terminate. This is because P_j terminates only after the dealer has broadcasted a clique and P_j has verified the same clique in its respective graph. Since all the messages that P_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 (i.e., there is a well-defined secret at the end of the sharing phase), 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 that finds such a STAR if there exists a clique of size $n - t$. The dealer then looks for a STAR in the graph instead of a clique, and once found, it broadcasts $\langle \texttt{star}, C, D \rangle$. Each party verifies that (C, D) is a STAR in its graph, and terminates if so.

This guarantees validity and binding: **Validity:** If the dealer is honest, then the protocol should terminate, and reconstruction should be the secret that the dealer shared. When the dealer is honest, eventually, there will be a clique in the graph of size $n - t$. The STAR algorithm then outputs a star (C, D), and all honest parties will eventually verify that (C, D) is a STAR. All honest parties receive shares that are consistent with the dealer's polynomial. **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, by utilizing Reed Solomon decoding.

Cost. When considering the costs of broadcasting `ok` messages, the above protocol requires $\mathcal{O}(n^4 \log n)$ bits to be 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 P_i broadcasts $(\texttt{ok}, i, j)$ 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 [30] 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 P_i broadcasts $\langle \texttt{ok}, i, j \rangle$ 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 yeilds an overhead of $\mathcal{O}(n)$ 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 $\mathcal{O}(n^3)$.

The difficulty is that there are n^2 "short messages" to be broadcasted ($\langle \texttt{ok}, i, j \rangle$), and the overhead of each broadcast is $\mathcal{O}(n^2)$. One might try to amortize these costs by having parties send $\mathcal{O}(n)$ of these `ok` messages at the same time (in which case, a broadcast with an overhead of $\mathcal{O}(n)$ can be used). However, parties do not even know how many `ok` they might broadcast. For instance, in the case of an honest dealer, parties know that they will eventually broadcast `ok`s for the $n - t$ honest parties, but they do not know how many corrupted parties would send them correct sub-shares. So they can wait and broadcast one large $n - t$ `ok` message, but then they will still have to broadcast the remaining (even up to $t = \mathcal{O}(n)$) messages one-by-one, as they arrive one-by-one. Moreover, all parties must hear all the edges between honest parties, to verify that the graph has a star. Thus, total communication of $\Omega(n^4)$ looks like a natural barrier.

Breaking the $\Omega(n^4)$-barrier. We achieve $\mathcal{O}(n)$-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.

To overcome this difficulty, we instruct each party P_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.,[30]) (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 different parties might have different graphs. Nevertheless, we claim that the following holds: (1) Validity: If the dealer is honest, then all honest parties will eventually find extended stars in their respective graphs; (2) Binding: Any pair of extended stars found by honest parties define the exact same bivariate polynomial, even they do not necessarily 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 each honest party. An extended star will then always be found.
- **Binding:** We claim that for every honest party P_j, the honest parties in the sets that P_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 P_j and P_k that might have distinct extended stars (C_j, D_j, E_j, F_j) and (C_k, D_k, E_k, F_k) that define the bivariate polynomials S_j and S_k respectively, $S_j = S_k$. 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 $2t+1$ and thus have at least $t+1$ honest parties in their intersection. The f-shares of those parties uniquely define S_j and S_k, respectively, and thus 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 reconstruct their shares. Since the shares of all honest parties that have an extended star must define the same bivariate polynomial, we get that, eventually, all parties 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. We refer to Section 5 for full specification and proofs.

2.2 ACS and AVABA

We now describe how we construct the ACS protocol, and our new notion called "AVABA", which is an information-theoretic version of Validated Asynchronous Byzantine Agreement (VABA [7]). This is a multivalued agreement protocol that allows parties to agree on a value which is "validated". The notion of validated will become clearer soon.

Our ACS. Recall that the main goal of ACS is to agree on a common core set of $n-t$ parties whose AVSS successfully terminated. Parties can asynchronously validate which parties can be considered: A party P_i validates P_j when the AVSS of P_j as a dealer terminates. When a party P_i validates a set of $n-t$ parties, it broadcasts its set S_i containing those $n-t$ parties; However, it continue

to store and update S_i as more parties are being validated (i.e., their AVSS has terminated). The parties now run an instance of AVABA – and try to reach an agreement on the sets S_i; the main difference is that this set S_i is dynamic, and as P_i validates additional parties, say, some P_k, it allows the agreed output to contain such additional parties that were dynamically added – such as P_k. The AVABA protocol guarantees that all honest parties will reach an agreement on the set, and whoever appears in the output was validated by an honest party. See Section 8.

AVABA. The construction of an AVABA protocol follows the ideas and construction of the No Waitin' HotStuff (NWH) protocol of [6], and replaces the cryptographic validation function with an asynchronous validation notion, as described above. Seeing as the NWH protocol is designed in the authenticated setting and uses a signature scheme, our work adapts these ideas to the information-theoretic setting, removing the need for cryptography.

Our AVABA protocol proceeds in iterations called "views". Parties start each view by exchanging suggestions for possible outputs from the protocol. These values are either derived from messages they saw in previous views, or simply their inputs if no suitable previous values exist. Every party then chooses the suggestion from the most recent view, and broadcasts it as its proposal for the current view. After parties broadcast their proposals, one of them is chosen *retroactively and obliviously* using a Verifiable Leader Election (VLE) protocol. Following that, parties check whether the proposal can be safely output from the protocol without contradicting values output by parties previously. If that is the case, they do so. We emphasize that when an honest leader is elected, this is always the case. Otherwise, they proceed to the next view, while ensuring other parties can proceed as well. We now elaborate on how the parties pick the leader.

Verifiable leader election. The first challenge is constructing a verifiable leader election (VLE) protocol. In ordinary leader election, the goal is for all parties to agree on the identity of an honest leader with some constant probability. In our setting, the chosen leader might be validated by some asynchronous validation process (specifically, in our AVABA, a party is validated once it broadcasted a proposal for an output). Our protocol is inspired by the synchronous leader election protocol of [27], its efficiency improvements [1], and the asynchronous authenticated proposal election protocol in the computational setting of [6].

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

We borrow ideas from [1, 6] 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 instances in which each dealer can share $\mathcal{O}(n)$ secrets at once. This reduces the number of instances to $\mathcal{O}(n)$. 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" of $n-t$ parties.

Since all of the c_i ranks 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 rank 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 $\frac{n-2t}{n} \geq \frac{1}{2}$. See more in Section 6.

Our verifiable party gather protocol is inspired by [6]. Unlike [6], which relies on signatures and authentication, we implement an information theoretic version of gather whose inputs comply with asynchronous validation. Moreover, our gather protocol is unique in that it outputs a set of parties, while their values are inferred via asynchronous validation. See more on the gather protocol in Section Section 4.

Verifiable leader election $\Longrightarrow$ AVABA. We now slightly elaborate on how we move from VLE to AVABA. Here the main challenge is working with asynchronously validated inputs and maintaining both safety (that all honest parties output the same value) and liveness (that the protocol terminates) over the different iterations (views). Our protocol is an information-theoretic adaptation of [6].

For safety, we use a common approach in authenticated protocols [18, 31] of using lock certificates and adapt them to the information-theoretic setting. Some of these techniques are inspired by the approach of [8] that adapts the protocol of [31] to 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 [7, 31] and adapt it to the information-theoretic setting. The second, more challenging problem is guaranteeing that honest parties eventually proceed to the next iteration (view) if the current iteration does not lead to agreement. As in [6], we observe that there are two triggers to changing views (i.e., giving up on the current iteration/leader):

- The first is when two different honest parties decided on different leaders (failure of the VLE); or
- The second is when the leader sends a proposal whose key is lower than a lock held by some party (blame event): parties check whether the leader proposed values that are "acceptable"

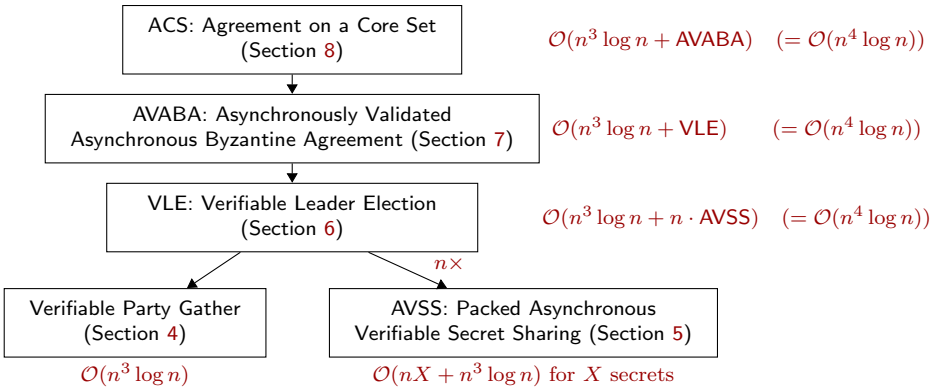

Figure 1: The structure of our ACS protocol; In red: The communication complexity of each primitive.

based on their current state, and "blame" the leader publicly if that is not the case.

In [6] 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 validate the correctness of messages asynchronously instead of using cryptographic tools. Roughly speaking, when the VLE fails or a blame message is sent, parties record it and wait for it to be asynchronously validated. We refer the reader to Section 7 for further details.

Structure. Figure 1 shows the structure of our ACS protocol, including the different building blocks and showing their complexity.

2.3 Extensions

We conclude the overview and the preliminary part of the paper by discussing two extensions: The statistical settings (Section 2.3.1), and the ramifications of our ACS and AVSS to asynchronous MPC (Section 2.3.2).

2.3.1 The Statistical Settings

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 [5, 13], when $n < 4t$, any AVSS protocol must have some non-zero probability of non-termination. The work of [17] 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 2.1 (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 2.2. *There exists a statistically secure protocol for asynchronous agreement on a core set with asynchronous validation resilient to $t < n/3$ Byzantine corruptions. Conditioned upon the protocol succeeding with probability $1 - \epsilon$, the protocol runs in constant expected time.*

It is also possible to directly construct our AVABA protocol using a single call to a verifiable leader election protocol per round instead. This means that any construction of such a protocol will immediately yield an AVABA protocol and consequently an ACS protocol as well. More precisely:

Theorem 2.3 (General Asynchronously Validated Asynchronous Byzantine Agreement (informal))**.** *Let $c \in [3, 4]$. Given a $n > ct$ resilient protocol for verifiable leader election that has $LE(n, \epsilon)$ 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 $\mathcal{O}(n^3 \log n + LE(n, \epsilon))$ expected communication complexity.*

2.3.2 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 [19], 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 [19]. 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.

3 Definitions and Assumptions

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, for simplicity, our modeling of the functionalities and the simulation proofs assume a static adversary, but we believe that our construction can be extended to the adaptive adversary. We also provide proofs for property-based definitions that are secure against an adaptive adversary that can choose to corrupt a party at any time given that it hasn't already corrupted t parties.

3.2 Asynchronous Secure Computation and SUC

We model our protocols in the simplified universally composable setting (SUC), formalized by Canetti, Cohen, and Lindell [16], which implies UC security. We briefly overview the model here, where this overview is taken almost verbatim from [3].

We consider an asynchronous network where the parties are $\{P_1, \ldots, P_n\}$. The parties are connected via pairwise ideal private channels. To model asynchrony, messages sent on a channel can be arbitrarily delayed, however, they are guaranteed to be eventually received after some finite number of activations of the adversary. In general, the order in which messages are received might be different from the order in which they were sent. Yet, to simplify notation and improve readability, we assume that the messages that a party receives from a channel are guaranteed to be delivered in the order they were sent. This can be achieved using standard techniques – counters, and acknowledgements, and so we just make this simplification assumption.

Main difference from SUC. The SUC model allows the adversary to also drop messages, and the adversary is not limited to eventually delivering all messages. To model "eventual delivery" (which is the essence of the asynchronous model), we limit the capabilities of the adversary and quantify over adversaries that eventually transmit each message in the network (i.e., they do not drop messages). Formally, any message sent must be delivered after some finite number of activations of the adversary.

As in SUC, the parties are modeled as interactive Turing machines, with code tapes, input tapes, outputs tapes, incoming communication tapes, outgoing communication tape, random tape and work tape.

Communication. In each execution there is an *environment* $\mathcal{Z}$, an *adversary* $\mathcal{A}$, participating *parties* $P_1, \ldots, P_n$, and possibly an *ideal functionality* $\mathcal{F}$ and a *simulator* $\mathcal{S}$. The parties, adversary

and ideal functionality are connected in a star configuration, where all communication is via an additional *router machine* that takes instructions from the adversary. That is, each entity has one outgoing channel to the router and one incoming channel. When P_i sends a message to P_j, it sends it to the router, and the message is stored by the router. The router delivers general information about the message to the adversary (i.e., "a header" but not the "content". That is, the adversary can know the type of the message and its size, but cannot see its content). When the adversary allows the delivery of the message, the router delivers the message to P_j. As mentioned, we quantify only over all adversaries that eventually deliver all messages. In particular, even in an execution with an ideal functionality, communication between the parties and this functionality is done via the router machine and is subject to (finite) delivery delays imposed by the adversary.

Note that the router machine is also part of the ideal model. When the functionality gives for instance, some output to party P_j, then this is performed via the router, and the simulator is notified. Thus, if the adversary, for instance, delays the delivery of the output of some party P_j, we do not explicitly mention that in the functionality (e.g., "wait to receive OK_j from the adversary and then deliver the output to P_j"), yet it is captured by the model.

Finally, the environment $\mathcal{Z}$ communicates with the adversary directly and not via the router. In particular, the environment can communicate with the adversary (and it cannot communicate even with the ideal functionality $\mathcal{F}$). In addition, $\mathcal{Z}$ can *write* inputs to the honest parties' input tapes and can *read* their output tapes.

Execution in the ideal model. In the ideal model we consider an execution of the environment $\mathcal{Z}$, dummy parties $P_1, \ldots, P_n$, the router, a functionality $\mathcal{F}$ and a simulator $\mathcal{S}$. In the ideal model with a functionality $\mathcal{F}$ the parties follow a fixed ideal-model protocol. The execution is as follows:

1. The environment is first activated with some input z.
2. The environment delivers the inputs to the dummy honest parties, which forward the inputs to the functionality (recall that this is done via the router, which then gives some leakage about the message header to $\mathcal{S}$, which can adaptively delay the delivery by any finite amount). Moreover, $\mathcal{Z}$ can also give some initial inputs to the corrupted parties via $\mathcal{S}$.
3. At a later stage, where the dummy parties receive output from the functionality $\mathcal{F}$, they just write the outputs on their output tapes (and $\mathcal{Z}$ can read those outputs). Again, these messages go through the router, and the simulator can delay them.
4. At the end of the interaction, $\mathcal{Z}$ outputs some bit b.

The simulator $\mathcal{S}$ can send messages to $\mathcal{Z}$ and to the functionality $\mathcal{F}$. The simulator cannot directly communicate with the participating parties. We stress that in the ideal model, the simulator $\mathcal{S}$ interacts with $\mathcal{Z}$ in an online way, and the environment can essentially read the outputs of the honest parties and query the simulator (i.e., can ask to receive a simulated transcript of the adversary's view) at any point of the execution. We denote by $\mathsf{ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(z)$ an execution of this ideal model of the functionality $\mathcal{F}$ with a simulator $\mathcal{S}$ and environment $\mathcal{Z}$, which starts with an input z.

Execution in the real model with protocol π. In the real model, there is no ideal functionality and the participating parties are $\mathcal{Z}$, the parties $P_1, \ldots, P_n$, the router and the real-world adversary $\mathcal{A}$.

1. The environment is first activated with some input z, and it can give inputs to the honest parties, as well as some initial inputs to the corrupted parties controlled by the adversary $\mathcal{A}$.

2. The parties run the protocol π as specified, while the corrupted parties are controlled by $\mathcal{A}$. The environment can see at any point the outputs of the honest parties, and communicate directly with the adversary $\mathcal{A}$ (and see, without loss of generality, its partial view).
3. All messages go through the router and the adversary gets notified. The adversary can decide when to deliver each message.
4. At the end of the execution, the environment outputs some bit b.

We denote by $\mathsf{real}_{\pi,\mathcal{A},\mathcal{Z}}(z)$ an execution of this real model with the protocol π, the real-world adversary $\mathcal{A}$ and the environment $\mathcal{Z}$, which starts with some input z.

Definition 3.1. *We say that an adversary $\mathcal{A}$ is an* asynchronous adversary *if for any message that it receives from the router, it allows its delivery within some finite number of activations of $\mathcal{A}$.*

Definition 3.2. *Let π be a protocol and let $\mathcal{F}$ be an ideal functionality. We say that π securely computes $\mathcal{F}$ in the asynchronous setting if for every real-model asynchronous adversary $\mathcal{A}$ there exists an ideal-world adversary $\mathcal{S}$ that runs in polynomial time in $\mathcal{A}$'s running time, such that for every $\mathcal{Z}$:*

$$\{\mathsf{ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(z)\}_z \equiv \{\mathsf{real}_{\pi,\mathcal{A},\mathcal{Z}}(z)\}_z$$

3.3 Cooperative Adversaries

Most of the functionalities discussed in this paper do not involve private inputs. Specifically, in many of these functionalities, the adversary is aware of all parties' inputs and outputs. The adversary's primary capability is influencing these outputs to some degree. Such functionalities often include a command like setOutput, allowing the adversary to set the output for certain honest parties under specific conditions verified by the functionality. This can lead to a scenario where the adversary might choose not to cooperate, failing to send any inputs and forcing the functionality to "get stuck".

To simplify matters, we model the adversary is always "cooperative". This is done for simplicity of exposition, to enhance readability and provide a more concise description of the functionalities. In this model, the adversary eventually responds and cooperates with the functionality.

We justify this modeling by noting that any functionality assuming a cooperative adversary can be transformed to handle a non-cooperative adversary. In such a case, the functionality would specify default outputs for honest parties. This involves the functionality calling the setOutput command that the ideal adversary would typically invoke. This message is routed to itself via the router, and the adversary is notified. The adversary can decide to rush and reply with its own setOutput command, or, it must eventually pass the first command to the functionality (recall that we assume eventual delivery), which then sends the output to the honest parties. In the case of two setOutput commands, the second one will be ignored. When describing a functionality that works for every adversary, not necessarily a cooperative one, the eventual output will be the one that it sent by the functionality to itself.

In other words, by modeling functionalities with cooperative adversary, we effectively "shift" some of the functionality's description to the simulator, resulting in more concise functionalities that precisely describe the adversary's capabilities. For instance, in F_{Gather} (Section 4), the adversary might choose different outputs for different parties, but the functionality ensures that all outputs share a significant common subset. We just define that the adversary is capable of choosing different outputs to different parties, without necessarily describing what the outputs are when the adversary

does not cooperate. This means that a simulator can use its access to define outputs, or give up this right by being uncooperative and allow the functionality to define outputs instead.

3.4 Reliable Broadcast

We assume the existence of a Reliable Broadcast protocol. 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.

Concretely, we use the reliable broadcast protocol of [9] in which parties send $\mathcal{O}(n^2 \log n + n \cdot m)$ bits when broadcasting a message of size $\mathcal{O}(m)$.

4 Verifiable Party Gather

In the leader election protocol, we wish to agree on a large set of parties that actively participated and elect a leader among them. However, since some instances might terminate earlier than others for some parties, exactly agreeing on the set is non-trivial and potentially expensive. We know, however, that if an instance has terminated for one honest party then it eventually terminates for all parties. The gather functionality comes to "synchronize" the parties. Exactly agreeing on all terminating instances is rather expensive (in fact, this is the end goal of "agreement on a core set" – of Section 8). At this point, we slightly relax our requirements, the parties might output different sets, but under the following condition:

> **CORE:** there exists some core C^* of size $n - t$ or greater such that the output of every honest party contains C^*.

The parties also give their outputs to one another, and whenever P_j receives the output of P_i it verifies that the output was computed correctly. As we will see, all the messages sent are public, and therefore the view of the different parties should be the same. The only difference is the different scheduling of messages. Thus, it should be possible to eventually verify an honestly generated output, at least eventually.

4.1 Property-Based Definition

To aid understanding, we first give some property based definition of this primitive. We later describe its functionality (Functionality 4.1). (Our proof refers to the functionality, the property-based definition is given just for completeness.)

- **Syntax:** Each party eventually P_j eventually sets its output to be $(\mathsf{output}, j, C_j)$; Moreover, it receives the sets of the other parties and verifies them (and $(\mathsf{output}, k, C_k)$ will also be part of its output for other parties P_k).

- **Core:** Once the first honest party outputs a value from the protocol, there exists a core set C^* such that $|C^*| \geq n - t$. If an honest party P_j outputs a set $(\mathsf{output}, j, C_j)$ then it holds that $C^* \subseteq C_j$.
- **Completeness:** For every honest P_j, if P_j outputs $(\mathsf{output}, j, C_j)$ then all honest parties would verify its set and have $(\mathsf{output}, j, C_j)$ as part of their output.
- **Agreement on verification:** For a corrupted P_i, if some honest party outputs $(\mathsf{output}, i, C_i)$, then all honest parties eventually output $(\mathsf{output}, i, C_i)$.
- **Validation of verification:** For a corrupted P_i, if some honest party P_j outputs $(\mathsf{output}, i, C_i)$, then it holds that $C^* \subseteq C_i$ and each element in C_i has been validated by P_j.

4.2 Gather Functionality

We now describe the gather functionality. As described in Section 3.3, we describe the functionality with a cooperative adversary - namely, we assume that it always replies to the functionality.

Functionality 4.1: Gather Functionality

The functionality is reactive. Initialize $C_1, \ldots, C_n = C^* = \emptyset$.

- $\mathsf{validate}(i, j)$: Whenever the functionality received this command from some party P_i (via the router), forward $(\mathsf{validate}, i, j)$ to the ideal adversary and record the message.
- $\mathsf{setCore}(C)$: Whenever the ideal adversary sends this command, verify that $|C| \geq n - t$ and that for every $x \in C$, there exists a recorded $\mathsf{validate}(\ell, x)$ for an honest P_ℓ. If so, store $C^* = C$; Otherwise, ignore the command.
- $\mathsf{setOutput}(i, C'_i)$: Whenever the ideal adversary sends this command, verify that $|C'_i| \geq n - t$, $C^* \subseteq C'_i$ and for every $x \in C'_i$, there exists a recorded $\mathsf{validate}(\ell, x)$ for an honest P_ℓ. If $C_i = \emptyset$ replace it with $C_i = C'_i$, and send $(\mathsf{output}, j, C_j)$ to all parties. Otherwise ($C_i \neq \emptyset$), ignore the command.

Input Assumption 4.2. *We prove that the protocol implements the functionality for* restricted *environments. In particular, we assume the following:*

1. *For every pair of honest parties (j, k), the environment issues* $\mathsf{validate}(j, k)$.
2. *If the functionality issues* $\mathsf{validate}(j, i)$ *from an honest P_j and corrupted P_i, for any other honest P_k, the environment will also issue* $\mathsf{validate}(k, i)$.

4.3 Gather Protocol

The protocol consists of the following steps. The parties receive some validations of parties from the environment (e.g., P_j validates P_i when the secret sharing of P_i terminates). Then:

1. In the first phase, parties try to establish sets $S_i \subseteq [n]$ of parties that were successfully validated. Once the set S_i is of size at least $n - t$, P_i broadcasts $\langle 1, S_i \rangle$ to all parties.
2. In the second phase, after an honest P_i receives such a set S_j from party P_j, it validates that set. That is, it verifies that S_j is of size $n - t$, and for every $k \in S_j$, waits until $\mathsf{validate}_i[k]$ is 1. Once the message is validated, it adds the content of S_j to US_i, and j to $\mathsf{VerifiedBc\text{-}1}_i$. Once $n - t$ broadcast-1 messages are validated (i.e., $|\mathsf{VerifiedBc\text{-}1}_i| \geq n - t$), P_i broadcasts its broadcast-2 message, consisting of US_i and $\mathsf{VerifiedBc\text{-}1}_i$.

3. In the third round, parties verify the broadcast-2 messages. Once $n - t$ such messages are verified, the party sets $C_i = \mathsf{US}_i$ as its output, and broadcast C_i as a broadcast-3 message.
4. Finally, each party continues to listen to broadcast-3 message, and validates them. This allows each party to know what is the output of each other party.

It is important to notice that when some honest party P_i sets its output C_i, then from counting argument, there exists some i^* that appears in $t + 1$ $\mathsf{VerifiedBc\text{-}1}_i$ sets (recall that those sets are being broadcasted in broadcast-2 messages). The CORE can be defined to be S_{i^*}, and the proof shows that the sets C_j that parties output must contain S_{i^*}.

Protocol 4.3: Implementing Gather

Each party P_i intializes $S_i, \mathsf{US}_i, \mathsf{VerifiedBc\text{-}1}_i, \mathsf{VerifiedBc\text{-}2}_i, C_1, \ldots, C_n \leftarrow \emptyset$, $\mathsf{validate}_i = 0^n$ and works as follows:

$\mathsf{validate}(i, j)$: **Upon** receiving the command $\mathsf{validate}(i, j)$ from the environment, set $\mathsf{validate}_i[j] = 1$.

Main process:

1. **Build initial set of validated parties S_i and broadcast-1 message:**
   (a) For every $j \in [n]$, **upon** $\mathsf{validate}_i[j] = 1$: Add j to S_i.
   (b) For the first time when $|S_i| \geq n - t$, broadcast $\langle 1, S_i \rangle$.
2. **Validate broadcast-1 messages and send broadcast-2 message:**
   (a) **Upon** receiving $\langle 1, S_j \rangle$ from j, verify that the message $\langle 1, S_j \rangle$ is valid:
   Verify that $|S_j| \geq n - t$, and for every $k \in S_j$, wait until $\mathsf{validate}_i[k]$ becomes 1.
   (b) **Upon** the message $\langle 1, S_j \rangle$ being validated, perform $\mathsf{US}_i = \mathsf{US}_i \cup S_j$, and add j to $\mathsf{VerifiedBc\text{-}1}_i$.
   (c) For the first time when $|\mathsf{VerifiedBc\text{-}1}_i| \geq n - t$, broadcast $\langle 2, \mathsf{VerifiedBc\text{-}1}_i, \mathsf{US}_i \rangle$.
3. **Validate broadcast-2 messages and send broadcast-3 message:**
   (a) **Upon** receiving $\langle 2, \mathsf{VerifiedBc\text{-}1}_j, \mathsf{US}_j \rangle$ from P_j, validate that the message is valid:
      i. Verify that $|\mathsf{VerifiedBc\text{-}1}_j| \geq n - t$.
      ii. Verify that $\mathsf{VerifiedBc\text{-}1}_j \subseteq \mathsf{VerifiedBc\text{-}1}_i$ (i.e., all the parties that P_j considered as broadcasting message-1, P_i also heard those broadcasts and could verify that their message-1 is valid);
      iii. Consider $\mathsf{US}'_j = \bigcup_{k \in \mathsf{VerifiedBc\text{-}1}_j} S_k$ (i.e., simulate the set US_j that P_j was supposed to send according to its reported $\mathsf{VerifiedBc\text{-}1}_j$). Verify that $\mathsf{US}_j = \mathsf{US}'_j$.
   (b) **Upon** the message $\langle 2, \mathsf{VerifiedBc\text{-}1}_j, \mathsf{US}_j \rangle$ being validated, add the pair (j, US_j) to $\mathsf{VerifiedBc\text{-}2}_i$.
   (c) **Upon** $|\mathsf{VerifiedBc\text{-}2}_i| = n - t$ (i.e., $n - t$ broadcast-2 messages were verified), then set $C_i = \mathsf{US}_i$. Broadcast $\langle 3, C_i \rangle$. Append $(\mathsf{output}, i, C_i)$ to the output tape.
4. **Validate broadcast-3 messages and outputs:**
   (a) **Upon** receiving a broadcast $\langle 3, C_j \rangle$ message from P_j, and **Upon** the following conditions being satisfied, append $(\mathsf{output}, j, C_j)$ to the output tape: (1) $C_i \neq \emptyset$; (2) Let num be the number of $(k, \mathsf{US}_k) \in \mathsf{VerifiedBc\text{-}2}_i$ such that $\mathsf{US}_k \subseteq C_j$. Then it should hold that $num \geq n - t$, and for every $x \in C_j$, $\mathsf{validate}_i[x] = 1$;

4.4 Security Analysis

We start with a few lemmas and proceed to the simulation-based proof. The efficiency of the protocol is analyzed in Appendix A.

Lemma 4.4. *Assume some honest party P_i sets C_i. There exists some i^* such that at least $t+1$ parties sent broadcasts of the form $\langle 2, \mathsf{VerifiedBc\text{-}1}, \mathsf{US}\rangle$ with $i^* \in \mathsf{VerifiedBc\text{-}1}$.*

Proof. Before setting C_i in Step 3c, P_i found that $|\mathsf{VerifiedBc\text{-}2}_i| \geq n-t$, and thus it received $n-t$ broadcasts of the form $\langle 2, \mathsf{VerifiedBc\text{-}1}_j, \mathsf{US}_j\rangle$ such that $|\mathsf{VerifiedBc\text{-}1}_j| \geq n-t$. Let J be the set of parties who sent those broadcasts. Now assume by way of contradiction that there is no such i^*, i.e., every index k appears in at most t of the broadcasted sets $\mathsf{VerifiedBc\text{-}1}_j$ such that $j \in J$. 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 $\langle 2, T, R\rangle$ broadcasts sent, $i^* \in T$. □

Lemma 4.5. *Let P_i, P_j be two honest parties. Observe the sets $\mathsf{VerifiedBc\text{-}1}_i, \mathsf{VerifiedBc\text{-}2}_i$ at any time throughout the protocol. Eventually $\mathsf{VerifiedBc\text{-}1}_i \subseteq \mathsf{VerifiedBc\text{-}1}_j$ and $\mathsf{VerifiedBc\text{-}2}_i \subseteq \mathsf{VerifiedBc\text{-}2}_j$.*

Proof. Observe some $k \in \mathsf{VerifiedBc\text{-}1}_i$. Party P_i added k to $\mathsf{VerifiedBc\text{-}1}_i$ after receiving a $\langle 1, S_k\rangle$ broadcast from P_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, P_j eventually receives the same message, and from the input assumption of the validity, it will eventually see that $\mathsf{validate}_i[x] = 1$ for every $x \in S_k$. At that point, P_j adds k to $\mathsf{VerifiedBc\text{-}1}_j$ as well. Similarly, observe some $(k, \mathsf{US}_k) \in \mathsf{VerifiedBc\text{-}2}_i$. Party P_i received a $\langle 2, \mathsf{VerifiedBc\text{-}1}_k, \mathsf{US}_k\rangle$ message such that $|\mathsf{VerifiedBc\text{-}1}_k| \geq n-t$ and saw that $\mathsf{VerifiedBc\text{-}1}_k \subseteq \mathsf{VerifiedBc\text{-}1}_i$. Party P_j will also receive that same message and eventually see that $\mathsf{VerifiedBc\text{-}1}_k \subseteq \mathsf{VerifiedBc\text{-}1}_i \subseteq \mathsf{VerifiedBc\text{-}1}_j$, at which point it will add a tuple (k, US'_k) to $\mathsf{VerifiedBc\text{-}2}_i$. Note that both parties compute US_k and R'_k to be the union of the S_l sets such that $l \in \mathsf{VerifiedBc\text{-}1}_k$. 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 $(k, R'_k) = (k, \mathsf{US}_k)$ to its $\mathsf{VerifiedBc\text{-}2}_j$ set. □

Lemma 4.6. *There exists a core C^* of cardinality $\geq n-t$, such that for any $(\mathsf{output}, i, C_i)$ that any honest party might output, it holds that $C^* \subseteq C_i$. Moreover, this core C^* can be efficiently extracted from the view of the first honest party P_j that sets C_j.*

Proof. Assume the first honest party that sets its output C_j is P_j, and observe the index i^* as defined in Lemma 4.4. Party P_j only adds a tuple (k, US_k) to $\mathsf{VerifiedBc\text{-}2}_j$ after receiving a $\langle 2, \mathsf{VerifiedBc\text{-}1}_k, \mathsf{US}_k \rangle$ message from party P_k. Before completing the protocol, P_j received $n-t$ such broadcasts and found that $\mathsf{VerifiedBc\text{-}1}_k \subseteq \mathsf{VerifiedBc\text{-}1}_j$. From Lemma 4.4, $t+1$ of the parties broadcast some message $\langle 2, \mathsf{VerifiedBc\text{-}1}_k, \mathsf{US}_k \rangle$ such that $i^* \in \mathsf{VerifiedBc\text{-}1}_k$. Therefore, for at least one party P_k, $i^* \in \mathsf{VerifiedBc\text{-}1}_k \subseteq \mathsf{VerifiedBc\text{-}1}_j$. Before adding i^* to $\mathsf{VerifiedBc\text{-}1}_j$, P_j 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 C^* be S_{i^*}. Clearly $|C^*| \geq n-t$ because $|S_{i^*}| \geq n-t$. The fact that C^* 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. □

Lemma 4.7. *For every honest P_j, if P_j outputs $(\mathsf{output}, j, C_j)$ then all honest parties would verify its set and have $(\mathsf{output}, j, C_j)$ as part of their output.*

Proof. Assume the input assumption as per Input Assumption 4.2. When an honest party P_j sets S_j, it holds that $\mathsf{validate}_j[x] = 1$ for all $x \in S_j$. P_j then broadcasts $\langle 1, S_j \rangle$.

Every honest P_k receives the message $\langle 1, S_j \rangle$, and from the input assumption, if P_j validated some $x \in S_j$, then P_k will also eventually see that x is validated. When all of S_j is validated, P_k adds j to $\mathsf{VerifiedBc\text{-}1}_k$. After adding such an index for every honest party, it must be that $|\mathsf{VerifiedBc\text{-}1}_k| \geq n-t$ and P_k can broadcast $\langle 2, \mathsf{VerifiedBc\text{-}1}_k, \mathsf{US}_k \rangle$ (it might broadcast that message also earlier, depending on the messages that the adversary sends).

Similarly, every honest party P_ℓ eventually receives that broadcast and sees that $|\mathsf{VerifiedBc\text{-}1}_k| \geq n-t$. From Lemma 4.5, eventually $\mathsf{VerifiedBc\text{-}1}_\ell \subseteq \mathsf{VerifiedBc\text{-}1}_k$, at which point P_ℓ adds a tuple (k, US_k) to $\mathsf{VerifiedBc\text{-}2}_\ell$.

After adding such a tuple for every honest P_k, every honest P_ℓ sees that $|\mathsf{VerifiedBc\text{-}2}_\ell| \geq n-t$, and it can set $C_\ell = \mathsf{US}_\ell$. We therefore conclude that each honest party, P_ℓ, eventually sets its own set C_ℓ. It then broadcasts $\langle 3, C_\ell \rangle$ to all parties. Verifying this message holds for P_ℓ, and therefore each honest party would verify that message, as per Lemma 4.8.

□

Lemma 4.8 (Agreement on verification.)**.** *For a corrupted P_i, if some honest party outputs $(\mathsf{output}, i, C_i)$, then all honest parties eventually output $(\mathsf{output}, i, C_i)$. Moreover, it holds that the core $C^* \subseteq C_i$.*

Proof. Assume that some honest P_j completes the verification and adds $(\mathsf{output}, i, C_i)$ to its output. This means that it received a broadcast message $\langle 3, C_i \rangle$ by P_i, and it saw that $|\{k \mid \exists (k, X) \in \mathsf{VerifiedBc\text{-}2}_j, X \subseteq C_i\}| \geq n-t$ and that for every $x \in C_i$ it holds that $\mathsf{validate}_j[x] = 1$. Let P_ℓ be some other honest party. From Lemma 4.5, eventually $\mathsf{VerifiedBc\text{-}2}_\ell \subseteq \mathsf{VerifiedBc\text{-}2}_j$ and thus eventually $|\{k \mid \exists (k, X) \in \mathsf{VerifiedBc\text{-}2}_\ell, X \subseteq C_i\}| \geq n-t$. In addition, from the input assumption of validate, it will also hold that $\mathsf{validate}_\ell[x] = 1$ for every $x \in C_i$. Thus, P_ℓ eventually adds $(\mathsf{output}, i, C_i)$ to its output. Moreover, we have at least $t+1$ honest parties P_ℓ for which (k, US_k), it holds that $\mathsf{US}_k \subseteq C_i$. From Lemma 4.6 it holds that $C^* \subseteq C_i$. □

We proceed with the main theorem of this section:

Theorem 4.9. *Protocol 4.3 securely realizes Functionality 4.1 in the presence of a malicious adversary controlling at most $t < n/3$ parties, assuming the Input Assumption 4.2.*

Proof. The simulator $\mathcal{S}$ works as follows:

1. Since the parties have no inputs, the simulator just runs all honest parties with the adversary $\mathcal{A}$, allowing the adversary to also schedule the delivery of the messages in the simulated execution.
2. Whenever the simulator receives $(\mathsf{validate}, j, i)$ from the functionality, it notifies the simulated P_j in the simulated protocol (while the adversary $\mathcal{A}$ has the ability to delay the delivery of the message).
3. When the first simulated honest party P_j sets the value C_j, extract the core C^* from the view of the simulated honest parties, as elaborated in Lemma 4.6. Then, send $\mathsf{setCore}(C^*)$ to the functionality.
4. Whenever a simulated honest party P_j sets the value C_j, send $\mathsf{setOutput}(j, C_j)$ to the functionality. The functionality then sends a message $(\mathsf{output}, j, C_j)$ to all parties, and this message is passed through the router. Deliver the message $(\mathsf{output}, j, C_j)$ to P_k only when $(\mathsf{output}, j, C_j)$ is obtained by the simulated P_k.
5. Whenever a corrupted party P_i broadcasts $\langle 3, X\rangle$ then verify that $X \subseteq C^*$ and that for every $k \in X$ there exists some honest P_ℓ that sent $\mathsf{validate}(\ell, k)$ command. If so, send $\mathsf{setOutput}(i, X)$ to the functionality. Otherwise, wait until the above conditions hold. (In particular, note that an honest party P_j never outputs C_i before setting its own C_i; thus, no honest party would output C_i yet). When $\mathsf{setOutput}(i, X)$ is sent to the functionality, the functionality then sends a message $(\mathsf{output}, j, C_j)$ to all parties, and the simulator allows the delivery of that message only when $(\mathsf{output}, j, C_j)$ is written to the output tape of the simulated P_k.

We now show that the real and ideal execution are identically distributed. The view of the adversary is clearly identical, as there are no private inputs, and the simulator just runs the protocol with the adversary as the honest parties do in the real. We now show that the outputs of the honest parties are the same in the real and ideal:

1. When the first simulated honest party sets its C_j, it can extract a core $C^* \subseteq C_j$ as follows from Lemma 4.6. Therefore, the functionality would accept C_j.
2. Moreover, for a corrupted P_i, an honest party would output $(\mathsf{output}, i, C_i)$ only if P_i has broadcasted a $\langle 3, C_i\rangle$ message, and C_i has been verified. In that case, in the ideal, the simulator sends $\mathsf{setOutput}(i, C_i)$ to the functionality as specified in Step 5 in the simulation. The functionality then verifies a few checks, and if they hold it sends $(\mathsf{output}, i, C_i)$ to all parties, and the simulator delivers those messages in the exact same scheduling as in the simulated protocol (which corresponds to the real). We now show that all those checks hold:
   (a) $C^* \neq \emptyset$; This must hold as an honest party P_j outputs $(\mathsf{output}, i, C_i)$ only after it sets C_j. As such, it must hold that the simulator already sent a core C^* to the functionality.
   (b) $C^* \subseteq C_i$: This holds as part of Lemma 4.8.
   (c) For every $x \in C_i$ there exists a recorded $\mathsf{validate}(\ell, x)$ command for some honest P_ℓ: This is also part of the verification that P_j performs before setting C_i.

□

5 Packed AVSS

5.1 The Functionality

The protocol we present implements a complete secret sharing, where all parties receive output. The dealer inputs a bivariate polynomial. If the polynomial is of the appropriate degree, all parties receive shares on that polynomial and the functionality can terminate. If the polynomial is not from the expected degree, the functionality does not terminate.

Functionality 5.1: Sharing phase of AVSS

1. The dealer sends the functionality a bivariate polynomial $S(X, Y)$.
2. The functionality verifies that $S(X, Y)$ is of degree at most $2t$ in X and degree at most t in Y. If not - the functionality does not terminate.
3. Otherwise, it sends all the shares $S(X, i), S(i, Y)$ for each corrupted party P_i to the ideal adversary. Moreover, it sends the shares $S(X, j), S(j, Y)$ to all honest parties $j \notin I$. Recall that the adversary has the ability to delay those messages.

5.2 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:

- **Input:** An undirected graph $G = (V, E)$ with $|V| = n$. (The parameters n, t are implict)
- **Output:** $C, D \subseteq V$ such that $C \subseteq D$ and there exists an edge $(u, v) \in E$ for every $u \in C, v \in D$, such that $|C| \geq n - 2t$ and $|D| \geq n - t$. If not such (C, D) sets were found, the algorithm outputs $(\emptyset, \emptyset)$.

It has been shown that if G contains a clique of size $n - t$, then the algorithm always finds such (C, D)-star. Moreover, the set C will contain at least $n - 2t$ vertices from the clique. This property was not originally formulated in [15] but is proven in [30].

RobustInt. The RobustInt algorithm is a decoding algorithm for the Reed-Solomon encoding [28]:

- **Input:** S, d, e such that S is a set of tuples (j, y_j), d is the degree, and e is the number of allowed errors.
- **Output:** It ouptuts a polynomial $p(x)$ of degree d that agree with all but e points in S. If there is no such polynomial, it outputs $\perp$.

The algorithm always outputs $\perp$ if $|S| < d + e + 1$ because there is no unique polynomial in that case. Moreover, if $|S| \geq d + e + m + 1$ for some $m \leq e$, and for some polynomial $p(x)$ there are at most m tuples $(j, y_j) \in S$ such that $p(j) \neq y_j$, then RobustInt outputs $p(x)$.

We are now ready to describe our AVSS protocol.

Protocol 5.2: Sharing phase of AVSS

Input: The dealer inputs a bivariate polynomial $S(X, Y)$. Each other party has no input.
The protocol:

1. **Initialization:** (All parties)
   (a) Initialize $f_i \leftarrow \perp$, $g_i \leftarrow \perp$, $p_i \leftarrow \perp$, $q_i \leftarrow \perp$.
   (b) $\mathsf{stars}_i \leftarrow \emptyset, \mathsf{interpolated}_i \leftarrow \emptyset, \mathsf{points}_i \leftarrow \emptyset$ $C_i \leftarrow \emptyset, D_i \leftarrow \emptyset, E_i \leftarrow \emptyset, F_i \leftarrow \emptyset$
   (c) Initialize an (undirected) graph G_i where the vertices are $[n]$ and the set of edges is empty.
2. **Sharing:** (The dealer only)
   (a) For every $i \in [n]$, the dealer sets $f_i(X) = S(X, i)$, $g_i(Y) = S(i, y)$. It then sends $\langle \texttt{polynomials}, f_j, g_j \rangle$ to P_i.
3. **Exchange sub-shares:** (Each party P_i)
   (a) **Upon** receiving $\langle \texttt{polynomials}, f'_j, g'_j \rangle$ message from the dealer, verify that f_j is of degree at most $2t$ and g_j is of degree at most t. Then, store $f_i \leftarrow f'_i$ and $g_i \leftarrow g'_i$, and send $\langle \texttt{values}, f_i(j), g_i(j) \rangle$ for every $j \in [n]$.
   (b) **Upon** receiving a message $\langle \texttt{values}, f_j(i), g_j(i) \rangle$ message from P_j, and **Upon** $f_i, g_i \neq \perp$, verify that $f_i(j) = g_j(i)$ and $g_i(j) = f_j(i)$. If so, then send $\langle \texttt{ok}, i, j \rangle$ to all parties.
4. **Look for an extended star:** (Each party P_i)
   (a) **Upon** receiving a message $\langle \texttt{ok}, j, k \rangle$ from P_j and $\langle \texttt{ok}, k, j \rangle$ from P_k, add (j, k) to G_i. Then, as long as an extended star is still not found, look for an extended star:
      i. Run $C_i, D_i \leftarrow \mathsf{FindStar}(G_i)$.
      ii. Let $F_i \leftarrow \{j \in [n] \mid |N(j) \cap C_i| \geq n - 2t\}$, i.e., all vertices in G_i that have at least $n - 2t$ neighbors in C_i.
      iii. Let $E_i \leftarrow \{j \in [n] \mid |N(j) \cap F_i| \geq n - t\}$, i.e., all vertices in G_i that have at least $n - t$ neighbors in F_i.
      iv. If $|C_i| \geq n - 2t$ and $|D_i| \geq n - t$, $|E_i| \geq n - t$ and $|F_i| \geq n - t$ then an extended star has been found. Send $\langle \texttt{star}, i, C_i, D_i, E_i, F_i \rangle$ to all parties.
   (b) **Upon** receiving a $\langle \texttt{star}, j, C_j, D_j, E_j, F_j \rangle$ from P_j, add $(\texttt{star}, j, C_j, D_j, E_j, F_j)$ to the set stars_i.
5. **Reconstruct the final column polynomial:** (Each party P_i)
   (a) **Upon** a new `star` message has been received (as in Step 4b), or a new `values` message has been received (as in Step 3b),
      i. For every $(j, C_j, D_j, E_j, F_j) \in \texttt{star}_i$ call $\mathsf{RobustInt}$, while considering only the points (received in Step 3b) of parties in E_j, looking for a polynomial of degree-t with at most t errors. If there is a reconstructed polynomial $g_{i,j}(y)$, then add $(j, g_{i,j})$ to $\mathsf{interpolated}_i$.
   (b) **Upon** $\mathsf{interpolated}_i$ being updated, if there is the same polynomial q'_i that appears at least $t + 1$ times in $\mathsf{interpolated}_i$, then set $q_i \leftarrow q'_i$, and send $\langle \texttt{col}, q_i(j) \rangle$ to every $j \in [n]$.
6. **Reconstruct the final row polynomial:** (Each party P_i)
   (a) **Upon** receiving $\langle \texttt{col}, q_j(i) \rangle$, add the point $(j, q_j(i))$ to points_i.
   (b) Run $p'_i \leftarrow \mathsf{RobustInt}(\mathsf{points}_i)$ for a polynomial of degree-$2t$ with at most t errors. If $p'_i \neq \perp$, then set $p_i \leftarrow p'_i$.
7. **Termination:**
   (a) **Upon** $|\texttt{star}_i| = n - t$ or receiving $\langle \texttt{done} \rangle$ from $t + 1$ parties, send $\langle \texttt{done} \rangle$ to all parties.
   (b) **Upon** receiving $\langle \texttt{done} \rangle$ from $n - t$ parties and $p_i \neq \perp$, $q_i \neq \perp$: **terminate**, while outputting $(p_i(X), q_i(Y))$.

There are a few security lemmas that we show, followed by simulatability. The efficiency of the protocol is analyzed in Appendix A.

Lemma 5.3. *Assume some honest party i sent a $\langle \texttt{star}, C_i, D_i, E_i, F_i \rangle$ message. Then every honest party $j \in C_i \cup D_i \cup E_i \cup F_i$ has $f_j \neq \perp, g_j \neq \perp$. In addition, there exists a unique polynomial S of degree $2t$ in X and t in Y such that for every honest $j \in C_i \cup E_i$ $f_j(X) = S(X, j)$ and for every honest $j \in D_i \cup F_i$ $g_j(X) = S(j, Y)$.*

Proof. Since P_i sent the `star` message in Step 4(a)iv, 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 the graph G_i. Party P_i only adds such an edge to G_i after receiving an $\langle \texttt{ok}, k, j \rangle$ message from P_k and an $\langle \texttt{ok}, j, k \rangle$ message from P_j. Note that if P_j and P_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 $R(X, Y)$ of degree $2t$ in X and t in Y such that for every such j, k, $f_j(X) = R(X, j)$ and $g_k(Y) = R(k, Y)$.

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

Lemma 5.4. *Assume two honest parties i, j sent `star` messages and let S_i, S_j be the polynomials defined for them in Lemma 5.3. Then, $S_i = S_j$.*

Proof. As in Lemma 5.3, 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 $k \in E_i \cap E_j$, $S_i(X, k) = f_k(X) = S_j(X, k)$. Therefore, for honest $k \in E_i \cap E_j$ and for every possible v, $S_i(v, k) = S_j(v, k)$. Both S_i and S_j have degree t in Y, and for every value v, $S_i(v, Y)$ agrees with $S_j(v, Y)$ at $t + 1$ points. Therefore, for every value v, $S_i(v, Y) = S_j(Y)$ and thus the polynomials are equal. □

Lemma 5.5. *If the dealer is honest, then all honest parties complete the* Share *protocol. Moreover, if the input of the dealer is $S(X, Y)$ then the output of each honest party P_j is $S(X, j), S(j, Y)$.*

Proof. Assume the dealer is honest. In that case, the dealer starts by inputting a bivariate polynomial $S(X, Y)$ of degree $2t$ in X and t in Y and sends every party i the polynomials $f_i(X) = S(X, i), g_i(Y) = S(i, Y)$. From inspection of the protocol, an edge (j, k) is added to the graph for every pair of honest parties P_j and P_k. As such, every honest party finds an extended star, and sends it to all other parties. The polynomial that P_i interpolates upon receiving a star (j, C_j, D_j, E_j, F_j) must be $S(j, Y)$ for every honest party P_j. Therefore, P_i adds at least $t+1$ times

the polynomial $S(j, Y)$ to $\mathsf{interpolated}_i$, and sends `col` messages. Each honest party P_i later receives at least $3t + 1$ points on its row-polynomial $S(X, i)$, and since at most t errors can be introduced, Reed-Solomon decoding guarantees a unique decoding of $S(X, i)$. Moreover, after receiving $n - t$ `star` messages, it sends `done` messages to all, and therefore all honest parties will eventually receive $n - t$ `done`, and therefore, P_i eventually must terminate and output $S(X, i), S(i, Y)$. □

Lemma 5.6. *If the dealer is corrupted and one honest party completes the* Share *protocol, then all honest parties will eventually complete* Share*. Moreover, there exists a unique bivariate polynomial $S'(X, Y)$ of degree at most $2t$ in X and at most t in Y, such that each honest P_j outputs $S'(X, j), S'(j, Y)$.*

Proof. Assume that some honest party completes the Share protocol. It does so after having its p and q polynomials not equal $\perp$ 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 5.3 and 5.4 all of those messages define the same bivariate polynomial $S(X, Y)$ 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 $g_{i,j}(Y) = S(i, Y)$ for every $(j, C_j, D_j, E_j, F_j) \in \mathsf{stars}_i$ such that j is honest and adds $(j, S(i, Y))$ to $\mathsf{interpolated}_i$. In addition, $\mathsf{interpolated}_i$ contains at most t tuples $(j, g_{i,j}(Y))$ such that $g_{i,j}(Y) \neq S(i, Y)$. 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 $(j, S(i, Y))$ in $\mathsf{interpolated}_i$ and updates $q_i(Y)$ to $S(i, Y)$. Now, following the exact same argument as above, every honest party eventually updates $p_i(X)$ to $S(X, i)$. 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 $p_i \neq \perp, q_i \neq \perp$ and receives `done` messages from at least $n - t$ parties, and thus it completes the Share protocol as well. □

Lemma 5.7. *Assume some honest party P_i has $p_i \neq \perp$ or $q_i \neq \perp$. Then some honest party P_j sent a $\langle \texttt{star}, C_j, D_j, E_j, F_j \rangle$, and P_i has $p_i(X) = S(X, i)$ or $q_i(Y) = S(i, Y)$ respectively for the S defined in Lemma 5.3.*

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 $\langle \texttt{star}, C_j, D_j, E_j, F_j \rangle$ 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 S be the polynomial defined for j in Lemma 5.3.

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 $\langle \texttt{star}, k, C_k, D_k, E_k, F_k \rangle$ 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 P_i adds tuples (l, y_l) to $\mathsf{points}_{g,i}$ after receiving a $\langle \texttt{values}, v_l, y_l \rangle$ message from l, and honest parties send such a message with $y_l = g_l(i)$. From Lemma 5.4, for every such honest l, $g_l(Y) = S(l, Y)$ and thus $y_l = S(l, i)$. Therefore, for every $(l, y_l) \in S$ such that l is honest, $y_l = S(l, y_l)$. In other words, $g_{i,k}(Y) = S(Y, i)$ 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 $(k, S(Y, k))$ for every honest k, and thus there are at most t tuples $(k, g_{i,k}(Y))$ for which $g_{i,k}(Y) \neq S(Y, k)$. Finally, this means that when i updated q_i it did so to the polynomial $q_i(Y) = S(Y, i)$.

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 $\langle \texttt{col}, y_k \rangle$ message from party k, and honest parties only send such a message with $y_k = q_k(i) = S(k, i)$. 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 $(k, S(k, i))$ added after receiving messages from honest parties. This means that the polynomial $S(X, i)$ 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, $p_i(X) = S(X, i)$. □

Theorem 5.8. *Protocol 5.2 implements Functionality 5.1 in the presence of an (unbounded) malicious adversary corrupting at most $t < n/4$ parties.*

Proof. We show a simulation for the case of an honest dealer and a corrupted dealer.

The case of an honest dealer. The simulator works as follows:

1. It receives from the trusted party the shares of the corrupted parties $f_i(X), g_i(Y)$ for every $i \in I$.
2. It fixes secrets $(s_0, \ldots, s_t) = (0, \ldots, 0)$.
3. It chooses an arbitrary bivariate polynomial $S(X, Y)$ of degree $2t$ in X and degree t in Y, under the constraints that:
   (a) $S(-k, 0) = s_k$ for every $k \in \{0, \ldots, t\}$;
   (b) $S(X, i) = f_i(X)$ and $S(i, y) = g_i(Y)$ for all $i \in I$.
4. Run the protocol when the dealer inputs $S(X, Y)$, all the other honest parties have no input, and communicate with the adversary $\mathcal{A}$. Note that with each message that some P_j sends to P_k, the adversary $\mathcal{A}$ is notified that the message has been sent and can decide when to deliver that message. Moreover, whenever $\mathcal{A}$ allows the delivery of the last message that causes some simulated P_j to terminate in the simulated protocol, the simulator then allows the transmission of the output of P_j from the trusted party to the honest P_j in the ideal world.

We show that the view of the environment $\mathcal{Z}$ in the real execution is identical to its view in the ideal execution via a sequence of hybrid experiments:

1. $\mathbf{Hyb}_0$: This is the ideal execution.
2. $\mathbf{Hyb}_1$: We assume a modified ideal execution in which the simulator receives from the trusted party the polynomial $S(X, Y)$ that the honest sender has sent the trusted party. Then, the

simulator, instead of choosing a polynomial $S'(X,Y)$ arbitrarily that agrees with $f_i(X)$ and $g_i(Y)$, simply uses the polynomial $S'(X,Y) = S(X,Y)$. Note, however, that the outputs of the honest parties in this execution are determined by the trusted party.

3. $\mathbf{Hyb}_2$: This is the real execution. In particular, the honest dealer uses $S(X,Y)$, the other honest parties have no input, and the output of all honest parties is determined by the protocol (and in particular, there is no trusted party involved).

We now show that each consecutive hybrids are identically distributed. Specifically:

$\mathbf{Hyb}_0$ and $\mathbf{Hyb}_1$: The difference between the two hybrids is that in $\mathbf{Hyb}_1$ the dealer uses $S(X,Y)$ as input, whereas in $\mathbf{Hyb}_0$ it uses some $S'(X,Y)$ under the constraint that $S(X,i) = S'(X,i)$ and $S(i,Y) = S'(i,Y)$ for every $i \in I$. By inspection, all the points that the adversary receives throguout the protocol are those points/polynomials. Therefore, the view of the adversary is exactly the same in both executions.

$\mathbf{Hyb}_1$ and $\mathbf{Hyb}_2$: In both hybrids the dealer uses the exact same input $S(X,Y)$. The difference between the two hybrids is that in $\mathbf{Hyb}_1$ the outputs of the honest parties is determined by the trusted party; Whenever the simulator sees that the simulated P_j terminates, it allows the delivery of the output message $S(X,j), S(j,Y)$ from the trusted party to P_j in the ideal world. On the other hand, $\mathbf{Hyb}_2$ is the real world; Since the protocol is correct and terminates in case of an honest dealer (see Lemma 5.5), we have that each honest party terminates with the shares $S(X,j), S(j,Y)$ where $S(X,Y)$ is the input of the dealer. Moreover, this also implies that each honest party in the simulated execution terminates, and according to the specification of the simulator, it then delivers the message sent from the trusted party to the honest P_j in the ideal world (and this message is the output – $S(X,j), S(j,Y)$. Therefore, all the honest parties in the ideal world also eventually receive their outputs.

The case of a corrupted dealer. The simulator works as follows:

1. Since the dealer is corrupted, all honest parties have no input in the real execution of the protocol. The simulator, therefore, can perfectly simulate a real execution of the protocol by simply running the code of the honest parties.
2. If, and when, the first honest party P_k terminates in the simulated execution, we interpolate the unique bivariate polynomial $S'(X,Y)$ of degree-$2t$ in X and degree-t in Y that is defined from the f and g shares of all honest parties in E_k, where E_k is part of the extended star that P_k has found in Step 4(a)iv. Send $S'(X,Y)$ to the trusted party and allow the delivery of the output to P_k in the ideal world.
3. Whenever a simulated honest party P_j terminates, the simulator allows the delivery of the output of P_j (which is $S'(X,j), S'(j,Y)$ from the trusted party to the honest P_j in the ideal world.

Clearly, the view of the adversary $\mathcal{A}$ is the same in both executions. The view of the environment $\mathcal{Z}$ in the real and ideal executions (which contains the view of the adversary and it can see what and when each honest party receives its output and terminates), we also claim that:

- First, if one honest party terminates, then eventually, all honest parties terminate (see Lemma 5.6).
- Let P_k the honest party that terminates first; Let $S'(X,Y)$ be the bivariate polynomial defined from the shares of honest parties in E_k. Then, all honest parties eventually terminate with shares on $S'(X,Y)$, as per Lemma 5.6.

□

5.3 Reconstruction

We now turn our attention to the reconstruction protocol of the AVSS.

Functionality 5.9: Reconstruction Functionality

1. Upon receiving $(\mathsf{point}, i, k, p_i(-k))$ from all honest parties with the same index $k \in \{0, \ldots, -t\}$, forward the message to the ideal adversary .
2. After receiving these messages from $t+1$ parties, reconstruct the unique degree-t univariate polynomial $q(Y)$ satisfying $q(j) = p_j(-k)$ for each honest P_j received.
3. Send $q(Y)$ to all parties.

Input Assumption 5.10. *It is assumed that all the shares of the honest parties lie on a unique degree-t univariate polynomial $g(Y)$.*

Protocol 5.11: $\mathsf{Reconstruct}(k)$

Input: The input of each honest party is $(\mathsf{point}, i, k, p_i(-k))$, where there exists a unique bivariate polynomial of degree $2t$ in X and degree t in Y such that $S(X, i) = p_i(X)$.
The protocol: (each party P_i)

1. Initialize $\mathsf{points}_i \leftarrow \emptyset$
2. Send $\langle \mathtt{rec}, k, p_i(-k) \rangle$ to all parties.
3. **Upon** receiving a point $\langle \mathtt{rec}, k, y_j \rangle$ message from P_j:
   (a) Add the point (j, y_j) to points_i.
   (b) Once $|\mathsf{points}_i| \geq n - t$, attempt to reconstruct $q(Y) \leftarrow \mathsf{RobustInt}(\mathsf{points}_i, t, t)$.
   (c) If $q(Y) \neq \perp$ then output $q(Y)$ and **terminate**.

Theorem 5.12. *Protocol 5.11 securely realizes Functionality 5.9 assuming Input Assumption 5.10, against a malicious adversary corrupting $t < n/4$ parties.*

Proof. The simulator works as follows:

1. Whenever an honest party sends its input to the functionality, allow the delivery of that message.
2. When the functionality sends $g(Y)$ to the adversary as the output, simulate the protocol with $\mathcal{A}$, while simulating the honest parties; The input of each honest party is $g(i)$. When a new honest parties joins (by sending its inputs to the functionality) - allow the delivery of their messages to the functionality and simulate them as well.
3. Whenever a simulated honest party P_j terminates, allow the delivery of the output of the functionality to P_j in the ideal world.

The view of the adversary is identical in the real and ideal, since the adversary learns all the inputs of all honest parties. The outputs of the honest parties in the ideal and real are identical: Since there is a unique degree-t univariate polynomial $g(Y)$ such that $g(i) = p_i(-k)$ for every i, $\mathsf{RobustInt}$ eventually outputs $g(Y)$ as there are at least $2t+1$ correct points and at most t incorrect points on that polynomial. A party might learn the output earlier; in which case, the ideal adversary also allows the delivery of the message earlier, at the exact same time. □

5.4 Putting it All Together

We now show a reactive functionality that combines the share and reconstruct phases.

Functionality 5.13: Reactive AVSS - F_{AVSS}

The functionality is parameterized by the identify of the dealer.

- $\mathsf{Share}(s_0, \ldots, s_t)$: When the dealer transmits this message to the functionality with input $(s_0, \ldots, s_t)$, forward Share to the ideal adversary, and record $(s_0, \ldots, s_t)$. Reply to all parties with the message shared.
- $\mathsf{Reconstruct}(k)$ with $k \in \{0, \ldots, -t\}$: Whenever this message is received from some party P_i, record that message. Once $t+1$ honest parties sent $\mathsf{Reconstruct}(k)$, reply with (k, s_k) to all parties that sent that command and all following commands.

Protocol 5.14: Reactive AVSS

- $\mathsf{Share}(s_0, \ldots, s_t)$:
  1. When the dealer calls this command, choose a random bivariate polynomial $S(X, Y)$ of degree-$2t$ in X and degree t in Y such that $S(-k, 0) = s_k$. Call Functionality 5.1 with input $S(X, Y)$.
  2. Each party P_j receives the output $(p_i(X), q_i(Y)) = (S(X, i), S(i, Y))$ from Functionality 5.1. Store these polynomials as private state, and output shared.
- $\mathsf{Reconstruct}(k)$ with $k \in \{0, \ldots, -t\}$:
  1. When receiving this command from the environment, call Functionality 5.9 with input $(\mathsf{point}, i, k, q_i(-k))$.
  2. When receiving an output $(k, g(Y))$ from Functionality 5.9, output $(k, g(0))$.

Theorem 5.15. *Protocol 5.14 securely implements Functionality 5.13 in the presence of a malicious adversary for any $t < n/4$.*

Proof. We separate between the case of an honest dealer and a corrupted dealer.

Honest dealer. In the case of an honest dealer, the simulator chooses a random polynomial $S(X, Y)$ of degree $2t$ in X and degree t in Y with the constraint of $S(-k, 0) = 0$ for all $k \in \{0, \ldots, t\}$. It then runs the sharing protocol with this input. In the real, the difference is that $S(X, Y)$ is chosen uniformly at random with the constraints that $S(-k, 0) = s_k$ for every $k \in \{0, \ldots, t\}$. However, the adversary sees just $S(X, i), X(i, Y)$ for every $i \in I$, and from simple counting, the same view is obtained in both executions with the exact same probability $(1/|\mathbb{F}|^{|I|(2t+1)+|I|(t+1-|I|)})$.

In the reconstruction protocol, whenever it learns some s_k from the functionality, it chooses a random degree t polynomial $q(Y)$ such that $S(-k, i) = q(k)$ for every $i \in I$, and $q(0) = s_k$. Since $|I| \leq t$, there exist $|\mathbb{F}|^{t-|I|}$ such polynomials. It then sends points on those polynomial as coming from the honest parties. The enviorment sees just $s_0, \ldots, s_t$, and the view of the adversary, which consists first of $S(X, i), S(i, Y)$, and later being "fixed" to a different polynomial $S'(X, Y)$ satisfying:

1. $S'(X, i) = S(X, i)$ and $S'(i, Y) = S(i, Y)$ for every $i \in I$;

2. $S(-k, 0) = s_k$ for every $k \in \{0, \ldots, t\}$.

which is exactly as in the real.

Corrupted dealer. In the case of a corrupted dealer, the simulator receives from the adversary the polynomial $S(X, Y)$ as it being sent to Functionality 5.1. The simulator then extracts (if indeed S is of degree-$2t$ in X and degree t in Y) – $s_0 = S(0, 0)$,...,$S(-k, 0) = s_k$ for every $k \in \{0, \ldots, t\}$. It sends $(s_0, \ldots, s_t)$ to the trusted party. The honest parties, the aiding functionalities - are all deterministic and have no inputs, and therefore, simulating the adversary's view is straightforward. □

Remark 5.16. *In the following section, each dealer participats in $\ell = \lceil n/(t+1) \rceil$ AVSS as a dealer, to distribute n secrets in total. To improve readability, we abuse notation and assume that we have one instance of F_{AVSS} (Functionality 5.13) that can accommodate n secrets. This can be implemented by distributing ℓ bivariate polynomials instead of just one.*

6 Verifiable Leader Election

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 that all parties elect the same honest party. 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 value. We proceed to define the VLE functionality in Section 6.1, followed by the protocol (Section 6.2).

6.1 The Functionality

Following Section 3.3, we define the functionality for cooperative adversaries. Once again, there is an external validity command that can be invoked, with the same input assumption as Input Assumption 4.2. The functionality samples a random rank for each party. Ideally, we would like all parties to consider all parties as possible candidates, and to choose the one with the highest rank as leader. As we will see, by delaying the delivery of messages, the adversary can make some parties not consider other parties as possible candidates. However, there is a large core of parties that is considered by all parties. After describing the functionality, we prove that this suffices in order to elect an agreed leader with a constant probability.

Functionality 6.1: F_{VLE} – The Verifiable Leader Election Functionality

1. $\mathsf{validate}(i, j)$: Whenever the command is received from some party P_i, forward $(\mathsf{validate}, i, j)$ to the ideal adversary and record the message.
2. $\mathsf{setCore}(C^*)$: Upon receiving this command from the ideal adversary, with $C^* \subseteq n$, and $|C^*| \geq n - t$, verify that for every $k \in C^*$ there exists an honest j such that $\mathsf{validate}(j, k)$ was recorded. Then, the functionality stores C^* and chooses a random rank to every party $r_1, \ldots, r_n$.
3. $\mathsf{setCandidates}(i, C_i)$: Upon receiving this command from the ideal adversary, verify that that $C^* \subseteq C_i$, and that for every $k \in C_i$ there exists an honest j such that $\mathsf{validate}(j, k)$ was

recorded. The functionality sets $\ell_i = argmax\{r_k \mid k \in C_i\}$. Add $(\mathsf{output}, i, \ell_i)$ to all parties, and send $(r_k)_{k \in C_i}$ to the ideal adversary.
It is assumed that the adversary sends $\mathsf{setCandidates}(j, C_j)$ for every honest party P_j, and it might also send such commands for some corrupted parties, according to its choice.

We assume the exact same input assumption as in Input Assumption 4.2.

Properties of the ideal functionality. We show that the functionality satisfies the following properties. In particular, we show that with a probability of at least 1/3, all honest parties agree on an honest leader, and the output of the corrupted parties (if it exists) also defines the same leader.

Claim 6.2. *The following properties hold:*

1. *With probability at least* 1/3*, there exists an index of an honest party* ℓ^**, such that* $\ell_i = \ell^*$ *for all* i *for which* $(\mathsf{output}, i, \ell_i)$ *has been sent to all parties (no matter whether* P_i *is honest or corrupted). Furthermore,* ℓ^* *is defined at the time that the first* $(\mathsf{output}, i, \ell_i)$ *has been sent by the functionality.*
2. *For every* $(\mathsf{output}, i, \ell_i)$ *that has been transmitted by the functionality to all parties, there exists an honest* j *such that the command* $\mathsf{validate}(j, \ell_i)$ *has been received.*

Proof. From the properties of the functionality, at the first time the functionality sent $(\mathsf{output}, j, C_j)$, there exists a core C^* of at least $n - t$ indices, such that every $C_i \subseteq C^*$. Note that the ranks are chosen only after the core C^* is fixed. Moreover, the adversary cannot see the rank r_i of some party P_i, unless i is included in some C_j in some $\mathsf{setCandidates}(j, C_j)$ command.

Each party $i \in C^*$ has probability $1/n$ to have the maximal rank[3]. among all $[n]$, i.e., including all supersets of C^*. Since $|C^*| \geq n-t$, we have at least $n-2t$ honest parties in C^*, and therefore the probability that some honest party ℓ^* in C^* has the maximal rank in $[n]$ is at least $(n-2t)/n \geq 1/3$. Since each C_i must include C^*, all output messages have ℓ^* as the leader. This is true even though the adversary chooses C_i adaptively, as C_i is chosen after C^* and all the ranks are determined.

For the second part of the claim, whenever the functionality receives some $\mathsf{setCandidates}(i, C_i)$ message it verifies that all parties in C_i has been validated (i.e., for each $k \in C_i$, some honest P_j has sent $\mathsf{validate}(j, k)$). The chosen ℓ_i is some index in C_i, and therefore ℓ_i must have been validated. □

6.2 The Protocol

The protocol has the following phases:

1. Each party contributes a sub-rank to every other party, using AVSS, which serves as a commitment scheme. This is to ensure that the adversary cannot bias the ranks.
2. Since some AVSS might not terminate, each party can decide which dealers that contributed to it will be considered. It is required to have $t + 1$ dealer, so its final rank will have at least one honest contribution, and therefore is random. It then broadcast its choices.

[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 $\frac{n-2t}{n} \geq \frac{n}{3} + \frac{1}{n}$

3. Each broadcasted message is verified, i.e., other parties have to see that the AVSS instances of those dealers does indeed terminate. If verified, that party can be considered as a candidate.
4. To ensure better agreement, the parties use the F_{Gather} functionality to have a large intersection between their possible candidates.
5. After receiving an output from F_{Gather}, which is a set of candidates, the parties reconstruct the sub-ranks of the dealers that contributed to each candidate, and then learn the ranks of all candidates. Each party outputs the party with the maximal rank among its candidate set as its leader.
6. In addition, each party can compute the output of each other party.

We are now ready for the formal details.

Protocol 6.3: VLE_i **in the** $(F_{\mathsf{AVSS}}, F_{\mathsf{Gather}})$**-hybrid model**

---

1. **Initialization:** Initialize $\mathsf{dealers}_i \leftarrow \emptyset, \mathsf{attached}_i \leftarrow \emptyset, \mathsf{candidates}_i \leftarrow \emptyset, \mathsf{ranks}_i \leftarrow \emptyset$, and $p_1, \ldots, p_n$. Moreover, initialize $\mathsf{validate}_i = 0^n$ and $\ell_1, \ldots, \ell_n = \perp$.
2. The parties invoke the F_{Gather} functionality (Functionality 4.1).
3. **Upon** receiving the command $\mathsf{validate}(i, j)$, set $\mathsf{validate}_i[j] = 1$.
4. **Contributing random sub-rank to all parties:**
   (a) Each dealer P_j chooses random sub-ranks $c_{j\to 1}, \ldots, c_{j\to n}$. For every $j \in [n]$ the parties call $F^j_{\mathsf{AVSS}}.\mathsf{Share}$ (Functionality 5.13) where P_j is the dealer (see also Remark 5.16).
5. **Choosing which sub-ranks will be considered for computing your rank:**
   (a) **Upon** completing $F^j_{\mathsf{AVSS}}.\mathsf{Share}$ sharing calls with P_j as dealer do: Add j to $\mathsf{dealers}_i$. If $|\mathsf{dealers}_i| = t + 1$ then **broadcast** $\langle \mathtt{attach}, \mathsf{dealers}_i \rangle$.
6. **Verify the choices of chosen sub-ranks of other parties:**
   Upon receiving an $\langle \mathtt{attach}, \mathsf{dealers}_j \rangle$ broadcast from P_j:
   (a) **Upon** $\mathsf{dealers}_j \subseteq \mathsf{dealers}_i$, $|\mathsf{dealers}_j| \geq t + 1$ and $\mathsf{validate}_i[j] = 1$:
      i. Set $\mathsf{attached}_i[j] \leftarrow \mathsf{dealers}_j$ and call $F_{\mathsf{Gather}}.\mathsf{validate}(i, j)$.
7. **Learning candidate sets, reconstructing their ranks, and compute outputs:**
   (a) **Upon** receiving $(\mathsf{output}, j, \mathsf{candidates}_j)$ from F_{Gather}:
      i. For every $k \in \mathsf{candidates}_j$:
         A. Set $r_k = 0$, and retrieve $\mathsf{dealers}_k \leftarrow \mathsf{attached}_i[k]$.
         B. For every $j \in \mathsf{dealers}_k$: Call $F^j_{\mathsf{AVSS}}.\mathsf{Reconstruct}(k)$ – i.e., P_j as the dealer and the secret corresponding to P_k. **Upon** retrieving an output $c_{j\to k}$ from F^j_{AVSS}, perform $r_k \leftarrow r_k + c_{j\to k}$.
      ii. **Upon** completing all reconstructions in $\mathsf{dealers}_k$: add (k, r_k) to ranks_i.
      iii. Moreover, compute $\ell_j = argmax\{r_k \mid k \in \mathsf{candidates}_j \ \wedge \ (k, r_k) \in \mathsf{ranks}_i\}$, and append $(\mathsf{output}, j, \ell_j)$ to the output tape.

6.3 Security Analysis

We show that the protocol securely realizes Functionality 6.1. The efficiency of the protocol is analyzed in Appendix A.

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. We show each one of the three claims.

$\mathsf{dealers}_i \subseteq \mathsf{dealers}_j$: Let k be some index in $\mathsf{dealers}_i$. Party P_i adds P_k to $\mathsf{dealers}_i$ after completing all Share calls with P_k as dealer. From the F_{AVSS} functionality, P_j completes those calls as well and adds k to $\mathsf{dealers}_j$.

$\mathsf{attached}_i \subseteq \mathsf{attached}_j$: Let $(k, \mathsf{dealers}_k)$ be a tuple in $\mathsf{attached}_i$. Party P_i adds the tuple to $\mathsf{attached}_i$ after receiving an $\langle \texttt{attach}, \mathsf{dealers}_k \rangle$ broadcast from P_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, P_j receives the same broadcast and as shown above and eventually sees that $\mathsf{dealers}_k \subseteq \mathsf{dealers}_i \subseteq \mathsf{dealers}_j$ and from the Input Assumption of validate, $\mathsf{validate}(j,k)$ eventually occurs. P_j then adds $(k, \mathsf{dealers}_k)$ to $\mathsf{attached}_j$.

$\mathsf{ranks}_i \subseteq \mathsf{ranks}_j$: Finally, let (k, r_k) be a tuple in ranks_i. P_i adds a tuple (k, r_k) after computing r_k:

1. That is, $r_k = \sum_{j \in \mathsf{dealers}_k} c_{j \to k}$.
2. Each $c_{j \to k}$ is a result of $F^j_{\mathsf{AVSS}}.\mathsf{Reconstruct}(k)$, for every $j \in \mathsf{dealers}_k$.
3. The set $\mathsf{dealers}_k = \mathsf{attached}_i[k]$, i.e., it is a result of a message $\langle \texttt{attach}, \mathsf{dealers}_k \rangle$ has been previously broadcasted by P_k.
4. P_i computes the rank of P_k, i.e., r_k only if k exists in $\mathsf{candidates}_l$ for some l, that is, P_i received an output $(\mathsf{output}, l, \mathsf{candidates}_l)$ from F_{Gather}, and $k \in \mathsf{candidates}_l$.
5. P_i receives an output from F_{Gather} only after having $|\mathsf{attached}_i| = n - t$.

We now show that P_j also adds the tuple (k, r_k) to ranks_j. As shown above, eventually $\mathsf{attached}_j \subseteq \mathsf{attached}_i$, so P_j will also see that $|\mathsf{attached}_j| = n - t$ at some point. As a result, it will also eventually receive outputs from F_{Gather}, and in addition, it will receive the same $(\mathsf{output}, l, \mathsf{candidates}_l)$ output from F_{Gather}. Since $k \in \mathsf{candidates}_l$, and since $\langle \texttt{attach}, \mathsf{dealers}_k \rangle$ has been broadcasted by P_k, it will see the same set of $\mathsf{dealers}_k$ as P_i. It will then call to all $F^d_{\mathsf{AVSS}}.\mathsf{Reconstruct}(k)$ for all $d \in \mathsf{dealers}_k$, and will receive $c_{d \to k}$, and thus reconstruct r_k. It adds (k, r_k) to ranks_j. □

Lemma 6.5. *The Input Assumption of* F_{Gather} *is satisfied.*

Proof. We show that for every pair of honest parties P_j, P_k, we have that P_j sent the command $F_{\mathsf{Gather}}.\mathsf{validate}(j,k)$. Since the sharing phase of honest parties must terminate, each honest party P_k eventually broadcast $\langle \texttt{attach}, \mathsf{dealers}_k \rangle$ message. From Lemma 6.4, P_j will validated that message, and from the input assumption of VLE, also $\mathsf{validate}(j,k)$ will be invoked. As such, P_j will invoke $F_{\mathsf{Gather}}.\mathsf{validate}(j,k)$ in Step 6(a)i.

Moreover, if some honest party P_j called $F_{\mathsf{Gather}}.\mathsf{validate}(j,i)$ for some corrupted P_i, then eventually each other honest party P_k will call $F_{\mathsf{Gather}}.\mathsf{validate}(k,i)$. If P_j called $F_{\mathsf{Gather}}.\mathsf{validate}(j,i)$, then P_i must have broadcasted $\langle \texttt{attach}, \mathsf{dealers}_i \rangle$ message that has been validated by P_j. This implies that for P_k, it will also hold that $\mathsf{dealers}_k \subseteq \mathsf{dealers}_i$, it holds that $|\mathsf{dealers}_i| \geq t+1$, and from the input assumption, since some party (P_j) called $\mathsf{validate}(j,i)$, then eventually also P_k will call $\mathsf{validate}(k,i)$. As such P_k will call $F_{\mathsf{Gather}}.\mathsf{validate}(k,i)$. □

Lemma 6.6. *The simulator satisfies the assumptions in Functionality 6.1: It first sends* $\mathsf{setCore}(C^*)$ *where* C^* *is valid (will be accepted by the functionality). Moreover, the simulator sends* $\mathsf{setCandidates}(j, C_j)$ *for every honest party* P_j*.*

Proof. This follows from the assumption on the adversary: Since the input assumption of F_{Gather} is satisfied (Lemma 6.5), then the adversary must call $F_{\mathsf{Gather}}.\mathsf{setCore}(C^*)$ for C^* that is valid. The simulator then calls to its own functionality with $\mathsf{setCore}$. Moreover, the adversary must invoke $F_{\mathsf{Gather}}.\mathsf{setOutput}(C_j)$ for every honest party C_j. In that case, the simulator calls $\mathsf{setCandidates}(j, C_j)$. □

Lemma 6.7. *Each honest party* P_j *appends* $(\mathsf{output}, k, \ell_k)$ *to its output tape for every other honest party* P_k*.*

Proof.

1. Since all honest parties call to $F_{\mathsf{AVSS}}.\mathsf{Share}$, and since each such call must terminate and return $\mathsf{receipt}$, we have that every honest P_k will add at least $t+1$ indices to $\mathsf{dealers}_k$, and broadcast $\langle \mathtt{attach}, \mathsf{dealers}_k \rangle$ in Step 5a.
2. From Lemma 6.4, each honest party P_l will eventually have $\mathsf{dealers}_j \subseteq \mathsf{dealers}_l$. Moreover, according to the input assumption, there would be $\mathsf{validate}(l, j)$ command. As such, P_l will add $\mathsf{dealers}_j$ to $\mathsf{attached}_l$, and will call $F_{\mathsf{Gather}}.\mathsf{validate}(l, j)$. Since this holds for every j, l, the input assumption of F_{Gather} is satisfied.
3. From the guarantees of F_{Gather}, all honest parties will eventually receive $(\mathsf{output}, k, \mathsf{candidates}_k)$.
4. For every $l \in \mathsf{candidates}_k$, it must have been that P_l has broadcasted $\langle \mathtt{attach}, \mathsf{dealers}_l \rangle$ message (otherwise, P_k would have not considered P_l). Then, it must have been that P_l received $\mathsf{receipt}$ for each $o \in \mathsf{dealers}_l$ and therefore the parties can reconstruct all the necessary values and compute r_l.
5. Since all the ranks in $\mathsf{candidates}_k$ can be recovered, P_j will compute $\ell_k = argmax\{r_l \mid l \in \mathsf{candidates}_k \ \wedge \ (l, r_l) \in \mathsf{ranks}_j\}$, and appends $(\mathsf{output}, k, \ell_k)$ to its output tape.

□

Lemma 6.8. *If some honest party appends* $(\mathsf{output}, i, \ell_i)$ *to its output tape for some corrupted party* P_i*, then all honest parties will eventually append* $(\mathsf{output}, i, \ell_i)$ *to their output tape.*

Proof. The proof follows from the same arguments as in Lemma 6.7. □

Theorem 6.9. *Protocol 6.3 securely implements Functionality 6.1 in the presence of a malicious adversary for every* $t < n/4$*, in the* $F_{\mathsf{AVSS}}, F_{\mathsf{Gather}}$*-hybrid model, assuming Input Assumption 4.2.*

Proof. The simulator works as follows:

1. the simulator runs the protocol with the adversary $\mathcal{A}$ while simulating also the functionalities F_{AVSS} and F_{Gather}. Moreover, it runs the code of all honest parties in the protocol, with one difference: Instead of picking $c_{j\to 1}, \ldots, c_{j\to n}$ uniformly at random, each honest party P_j sends nothing.
2. Whenever a message $\mathsf{validate}(i, j)$ is received from the functionality, the simulator invokes this command in the simulated protocol.
3. Whenever a party sends $F^j_{\mathsf{AVSS}}.\mathsf{Share}$ (and recall that by our modification, honest parties provide no input to that invocation), the simulator sends $\mathsf{receipt}$ to all parties.

4. Whenever the adversary calls $F^i_{\mathsf{AVSS}}.\mathsf{Share}$ in the name of a corrupted dealer P_i, the simulator receives its input – $c_{i\to 1}, \ldots, c_{i\to n}$. Simulate $F^i_{\mathsf{AVSS}}.\mathsf{Share}$ replying with $\mathsf{receipt}$ to all parties.
5. According to the scheduling of the delivery $\mathsf{receipt}$ messages from $F^j_{\mathsf{AVSS}}.\mathsf{Share}$, the simulated honest parties generate the $\langle \mathtt{attach}, \mathsf{dealers}_j \rangle$ messages. Accordingly, simulate honest parties call to $F_{\mathsf{Gather}}.\mathsf{validate}(j,k)$ as per Step 6(a)i in the protocol (which notifies the adversary).
6. We will show that the input assumption of F_{Gather} is satisfied, which implies that eventually, the cooperative adversary must send a valid $\mathsf{setCore}(C^*)$ to F_{Gather}. The simulator verifies that $|C^*| \geq n - t$ and that for every $i \in C^*$ there exists an honest j such that $F_{\mathsf{Gather}}.\mathsf{validate}(i,j)$ has been called.
7. Likewise, since the input assumption of F_{Gather} is satisfied, then the adversary must send $\mathsf{setOutput}(\mathsf{candidates}_i)$ with a valid $\mathsf{candidates}_i$. Whenever the adversary sends such a command:
   (a) Verify that $C^* \subseteq \mathsf{candidates}_i$, and that for every $k \in C_i$, there exists an honest j such that $F_{\mathsf{Gather}}.\mathsf{validate}(j,k)$ has been called.
   (b) Send to the functionality the command $\mathsf{setCandidates}(j, \mathsf{candidates}_i)$.
   (c) Receive back ℓ_i and the ranks $(r_k)_{k\in\mathsf{candidates}_i}$ from the functionality. The functionality also sends $(\mathsf{output}, i, \ell_i)$ as output to all parties. The adversary still does not allow the delivery of those messages in the ideal (this will happen in Step 7e).
   (d) For every $k \in \mathsf{candidates}_i$:
      i. Retrieve $\mathsf{dealers}_k \leftarrow \mathsf{attached}_i[k]$.
      ii. If $(c_{j\to k})_{j\in\mathsf{dealers}_k}$ is still not set, choose $c_{j\to k}$ uniformly at random under the constraint that $r_k = \sum_{j\in\mathsf{dealers}_k} c_{j\to k}$. Since $|\mathsf{dealers}_k| = t+1$, it must contain at least one honest party, for which we can choose its sub-rank (i.e., it was not determined in the $F_{\mathsf{AVSS}}.\mathsf{Share}$-phase).
      iii. For every $j \in \mathsf{dealers}_k$: Simulate calling to $F^j_{\mathsf{AVSS}}.\mathsf{Reconstruct}(k)$ – i.e., P_j as the dealer and the secret corresponding to P_k.
   (e) According to the delivery of the messages, whenever a simulated honest P_j appends $(\mathsf{output}, i, \ell_i)$ to its output tape, allow the delivery of the message $(\mathsf{output}, i, \ell_i)$ to P_j in the ideal.

Since the honest parties have no inputs (those are just the $\mathsf{validate}$ commands coming from the environment) the view of the adversary is clearly the same in both executions. We now show that the outputs are the same in the real in the ideal. Specifically, we show that:

1. The input assumption of F_{Gather} is satisfied. This implies that the adversary must send $\mathsf{setCore}$ message to the simulated F_{Gather}, followed by valid $\mathsf{setOutput}$. We formalized this in Lemma 6.5.
2. The simulator sends $\mathsf{setCandidates}(j, \mathsf{candidates}_j)$ for every honest j. The functionality then sends an output to all honest parties. We show that in the real, all honest parties have the output of all other honest parties, and that the outputs are the same. This is formalized in Lemma 6.7.
3. Likewise, if the adversary sends $\mathsf{setCandidates}(i, \mathsf{candidates}_i)$ for a corrupted party i, then we show that all honest parties append some $(\mathsf{output}, i, \ell_i)$ to their output tape, both in the real and ideal worlds. This is formalized in Lemma 6.8.

By combining those lemmas, we conclude that the output of the honest parties in the real and ideal

executions are identical. □

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 [5, 13]. Similar results can be achieved by using AVSS protocols with an ϵ probability of failure or non-termination, resulting in probabilistic guarantees and a resilience threshold of $t < \frac{n}{3}$.

7 Asynchronously Validated Asynchronous Byzantine Agreement

This section deals with constructing our AVABA protocol, which is built upon ideas in [6] and [8] and adapts them to the asynchronous information-theoretic setting.

7.1 The Functionality

Assuming once again a cooperative adversary (see Section 3.3), the functionality is relatively simple to describe: There is an external validity command, where each party might validate some value $v \in \mathcal{V}$. At some point, the adversary decides on a value x and sends it to the functionality. If x has been validated by some honest party, then this value is accepted and is sent to all parties as output.

Functionality 7.1: F_{AVABA}

The functionality is parameterized with a domain $\mathcal{V}$, and support the following commands:

$\mathsf{validate}(i, x)$: Upon receiving this command from party P_i with a value $v \in \mathcal{V}$, forward the command to the adversary and store this command.

$\mathsf{setInput}(j, x_j)$ Upon receiving this command from an honest P_j with $x_j \in \mathcal{V}$, forward the command to the adversary and store this command.

$\mathsf{setOutput}(x)$: Upon receiving this command from the ideal adversary with $x \in \mathcal{V}$, verify that there exists an honest j such that $\mathsf{validate}(j, x)$ or $\mathsf{setInput}(j, x_j)$ is recorded, send (output, x) to all parties and terminate.

Input Assumption 7.2. *We prove that the protocol implements the functionality for restricted environments that follow the following assumptions:*

1. *If for some honest party P_j the environment issued $\mathsf{setInput}(j, x_j)$ or $\mathsf{validate}(j, x_j)$ then for every other honest party P_k a command $\mathsf{setInput}(k, x_k)$ or $\mathsf{validate}(k, x_k)$ will be issued.*

We remark that our protocol actually satisfies a stronger property, named "α-quality": With probability α, all parties output the input x_j of a party P_j that was honest when starting the protocol. This is not reflected in our functionality, and we will prove this property as a property-based definition. This property is unnecessary for achieving the final ACS in Section 8, and we prove it for completeness.

7.2 The Protocol

The protocol of [6] heavily relies on cryptographic primitives (signatures) to obtain externally valid outputs. Here we use asynchronous external validity instead. This requires redefining and adapting new information-theoretic variants of verifiable gather (party gather) and verifiable leader election. The protocol of [8] modifies the cryptographic protocol of [31] to the information-theoretic setting in partial synchrony. Here we show how to extend this to full asynchronous network conditions, which in turn requires a new information-theoretic view change protocol and consistency checks for sent values.

In the AVABA protocol, parties proceed in "views", which are just an iteration number (asynchronous "phases"). In each view, parties propose values to agree upon and then try to choose an honest leader using the VLE protocol. Recall that the VLE protocol might fail: either by not agreeing on the chosen leader, or by electing a corrupt leader. However, once an honest leader is chosen, its value will be adopted and the parties will terminate.

AVABA uses the "Key-Lock-Commit" paradigm used in previous HotStuff protocols (VABA, IT-HS and NWH) to maintain safety and liveness. Given a value, a party first puts a "key" on it (this is just an indicator on the view number), then a "lock", and finally "commits" to it. Intuitively, locks help guarantee safety by forcing parties to ignore old values, and ensuring that parties can commit. Keys are used to ensure progress by convincing parties to accept proposed values in spite of their locks if no commitment was made. In more detail:

- **Commit:** When a party commits to a value, it knows that this value will be the output, and it pushes toward termination. It sends to other parties that it is ready to terminate.
- **Lock:** A lock consists of two values: lock, which is a view number, and lock_val which is the value seen when setting the lock (which is the potential output).
  A party might lock a value, but this lock can still be later removed. However, it indicates that this is a value that the parties should agree on, and it tells it to all other parties. Once it hears $n - t$ locks on the same value from other parties, it changes its status to "commit".
- **Key:** This indicates that a party is witness to a current value; If it hears enough "key" messages on the same value – it moves to "lock". Just like a lock, a key consists of two values: key, which is a view number, and key_val which is the suggested value.

Overview. The parties proceed in 5 (asynchronous) rounds in each view. The general idea is that parties first confirm that they all agree on the leader elected in the VLE protocol, set a key, set a lock to the elected leader's proposal, confirm that they are all locked, commit to the lock, and terminate.

If at any point they see that the VLE failed, they move onto a new view and announce that they are doing so. Recall that in VLE, eventually each party sees the output of the other parties, and therefore if two leaders are elected, then all parties will see that eventually.

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 those signatures on their own. We cannot use signatures. 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 that it could have formed that key or lock. Similar techniques are employed when providing `blame` messages, which are used to inform parties of a failed VLE session.

In each view (iteration), the parties run two protocols in parallel:

1. viewChange: The parties send each other suggestions for the current view. Parties then compute their proposal for the current view. Every P_i broadcasts this proposal, which is later used if it is chosen as a leader. Initially, this is the input of P_i, but later in the protocol, P_i might adopt some other value based on other parties' suggestions.
2. processMessages: This is the key-lock-commit mechanism. The leader is chosen, and the parties proceed to see if they can agree on the proposal of the leader.

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 make sure that 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, while P_i validates P_j in VLE only if P_j has broadcasted a valid proposal. 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 the view, but if it was not, they inform each other and proceed to the next view.

- Upon electing a leader using the VLE protocol, if the leader's proposed value is correct, i.e. it contains a key and value pair that could have been set in a view later than the current lock, then send an `echo` message to all other parties.
- If the leader's proposed value is "incorrect", send a `blame` message and proceed to the next view. In this context, by an "incorrect proposal" we mean that its key is not high enough to open the receiving party's current lock. Since a party puts a lock on a value based on public messages, 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 that lock, and verify whether the `blame` message is valid (note, however, that we cannot verify that a particular `blame` message is false). Upon sending or receiving a valid `blame` message, reject the leader, and proceed to the next view.
- Since each party P_i can see the outputs of all other parties from the VLE protocol, it can verify if two different parties elected different leaders. In that case, the leader election fails, and the party proceeds to the next 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 that of two different elected leaders. 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. Before setting a lock, every party makes sure that at least $t + 1$ honest parties set their keys to the current value. By doing that, every party guarantees that when choosing which value and key to input to the VLE protocol, all honest parties will hear of the current value and will 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. Before committing to a value, every party makes sure that at least $t+1$ honest 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.

Protocol 7.3: AVABA**: Asynchronously Validated Asynchronous Byzantine Agreement**

Initialization: Intialize $\mathsf{validated}_i \leftarrow \emptyset$. $\mathsf{key}_i \leftarrow 0, \mathsf{lock}_i \leftarrow 0, \mathsf{lock\_val}_i \leftarrow \perp$. Moreover, define for all $v \in \mathbb{N}$: $\mathsf{proposals}_i \leftarrow \emptyset, \mathsf{echoes}_{i,v} \leftarrow \emptyset, \mathsf{keys}_{i,v} \leftarrow \emptyset, \mathsf{locks}_{i,v} \leftarrow \emptyset$.

$\mathsf{validate}(i, x)$: Upon receiving this command with $x \in \mathcal{V}$, add x to $\mathsf{validated}_i$.

$\mathsf{setInput}(x_i)$:

1. Store x_i; add x_i to $\mathsf{validated}_i$.
2. $\mathsf{view}_i \leftarrow 1$.
3. Set $\mathsf{cur\_view}_i \leftarrow \mathsf{view}_i$.
4. Run $\mathsf{checkTermination}()$ in the background.
5. While $\mathsf{cur\_view}_i = \mathsf{view}_i$:
   (a) Invoke $\mathsf{viewChange}(\mathsf{view}_i)$ and $\mathsf{processMessages}(\mathsf{view}_i)$ in parallel.
   (b) Delay any message from any view $v > \mathsf{cur\_view}_i$. In constrast, continue updating sets and participating in broadcasts from older views, but do not send new messages or broadcast, and do not update $\mathsf{key}_i, \mathsf{key\_val}_i, \mathsf{lock}_i, \mathsf{lock\_val}_i$ in previous views.

$\mathsf{viewChange}(\mathsf{view})$: This comes to compute my proposal for the view view. This is a value that P_i suggests if it is chosen as the leader.

1. Start a new session of $F_{\mathsf{VLE}}(\mathsf{view})$, for the current view.
2. $\mathsf{suggestions} \leftarrow \emptyset$ ($\mathsf{suggestions}$ is a multiset):
3. Send $\langle \mathtt{suggest}, \mathsf{key}_i, \mathsf{key\_val}_i, \mathsf{view} \rangle$ to all parties.
4. **Upon** receiving the first $\langle \mathtt{suggest}, \mathsf{key}'_j, \mathsf{key\_val}'_j, \mathsf{view} \rangle$ message from party j such that $\mathsf{key}'_j < \mathsf{view}$, and **Upon** $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{key}'_j, \mathsf{key\_val}'_j) = 1$:
   (a) Add $(\mathsf{key}'_j, \mathsf{key\_val}'_j)$ to $\mathsf{suggestions}$.
   (b) If $|\mathsf{suggestions}| = n - t$ then $(k, v) \leftarrow argmax_{(k,v) \in \mathsf{suggestions}}\{k\}$ (break ties arbitrarily). I.e., take as a suggestion the one that has the highest view number.
   (c) If $k = 0$: then $(k, v) \leftarrow (0, x_i)$.
   (d) **Broadcast** $\langle \mathtt{proposal}, k, v, \mathsf{view} \rangle$.
5. **Upon** receiving a $\langle \mathtt{proposal}, k, v, \mathsf{view} \rangle$ broadcast from P_j, and **upon** $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$ terminating with output 1:
   (a) Add $(j, (k, v))$ to $\mathsf{proposals}_{i,\mathsf{view}}$.
   (b) Call $F_{\mathsf{VLE}}(\mathsf{view}).\mathsf{validate}(i, j)$, i.e., P_i validates party P_j as a possible leader, since P_j provided a valid suggestion.

processMessages(view): Attempt to reach agreement on the value of the leader of the current view. If the leader is honest, then the parties will reach an agreement by the end of this view.

1. **Support or reject the leader's proposal:**
   Upon receiving $(\mathsf{output}, i, \ell)$ from $F_{\mathsf{VLE}}(\mathsf{view})$ and having a tuple $(\ell, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$:
   (a) If $k \geq \mathsf{lock}_i$ then support the leader's proposal. Broadcast $\langle \mathtt{echo}, \mathsf{view} \rangle$.
   (b) Otherwise, reject the leader's proposal: broadcast $\langle \mathtt{blame}, \mathsf{lock}_i, \mathsf{lock\_val}_i, \mathsf{view} \rangle$, and proceed to the next view ($\mathsf{view}_i \leftarrow \mathsf{view}_i + 1$).
2. **Verify the rejects of other parties:**
   Upon receiving the broadcast message $\langle \mathtt{blame}, \mathsf{lock}'_j, \mathsf{lock\_val}'_j, \mathsf{view} \rangle$ from party P_j:
   (a) **Upon** receiving $(\mathsf{output}, j, \ell_j)$ from $F_{\mathsf{VLE}}(\mathsf{view})$ and having a tuple $(\ell_j, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$, and **upon** $\mathsf{lockCorrect}_i(\mathsf{lock}'_j, \mathsf{lock\_val}'_j)$=1:
      i. If $k < \mathsf{lock}'_j$ then reject the leader and proceed to the next view. Set $\mathsf{view}_i \leftarrow \mathsf{view}_i + 1$.
3. **Verify that F_{VLE} defines a unique leader:**
   (a) **Upon** some $(\mathsf{output}, a, \ell_a)$ and $(\mathsf{output}, b, \ell_b)$ are received with $\ell_a \neq \ell_b$ from $F_{\mathsf{VLE}}(\mathsf{view})$– then reject the current view. Proceed to the next view ($\mathsf{view}_i \leftarrow \mathsf{view}_i + 1$).
4. **Verify supports of other parties, and once enough supports - send key:**
   Upon receiving an $\langle \mathtt{echo}, \mathsf{view} \rangle$ broadcast from P_j, and **Upon** receiving $(\mathsf{output}, j, \ell_j)$ from $F_{\mathsf{VLE}}(\mathsf{view})$ and having a tuple $(\ell_j, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$:
   (a) Add (j, k, v) to $\mathsf{echoes}_{i,\mathsf{view}}$.
   (b) If $|\mathsf{echoes}_{i,\mathsf{view}}| = n - t$ then $\mathsf{key}_i \leftarrow \mathsf{view}, \mathsf{key\_val}_i \leftarrow v$. Moreover, **broadcast** $\langle \mathtt{key}, v, \mathsf{view} \rangle$.
5. **Verify keys of other parties, and once enough keys - send lock:**
   Upon receiving a $\langle \mathtt{key}, v, \mathsf{view} \rangle$ broadcast from P_j, and **Upon** $\mathsf{keyCorrect}_{i,\mathsf{view}+1}(\mathsf{view}, v)$ terminating with the output 1:
   (a) Add (j, v) to $\mathsf{keys}_{i,\mathsf{view}}$.
   (b) If $\left|\mathsf{keys}_{i,\mathsf{view}}\right| = n - t$ then $\mathsf{lock}_i \leftarrow \mathsf{view}, \mathsf{lock\_val}_i \leftarrow v$; send $\langle \mathtt{lock}, v, \mathsf{view} \rangle$ to all parties.
6. **Verify locks of other parties, and once enough locks - send commit:**
   Upon receiving the first $\langle \mathtt{lock}, v, \mathsf{view} \rangle$ message from P_j, and **upon** $\mathsf{lockCorrect}_i(\mathsf{view}, v)$ terminating with the output 1:
   (a) Add (j, v) to $\mathsf{locks}_{i,\mathsf{view}}$.
   (b) If $|\mathsf{locks}_{i,\mathsf{view}}| = n - t$ then send $\langle \mathtt{commit}, v \rangle$ to every party.

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

1. If $\mathsf{view} > k$ then **upon** $v \in \mathsf{validate}_i$:
   (a) If $k = 0$ then output 1.
   (b) Else, **upon** $|\{j | \exists k' \ s.t. \ (j, k', v) \in \mathsf{echoes}_{i,k}\}| \geq n - t$: output 1.

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

1. If $k = 0$ then output 1;
2. Else, **upon** $\left|\{j | (j, v) \in \mathsf{keys}_{i,k}\}\right| \geq n - t$, output 1.

---

checkTermination():

1. **Upon** receiving a $\langle \texttt{commit}, v \rangle$ message with the same value v from $t+1$ parties: send $\langle \texttt{commit}, v \rangle$ to every party if no such message has been previously sent.
2. **Upon** receiving a $\langle \texttt{commit}, v \rangle$ message with the same value v from $n-t$ parties: **output** v from the AVABA protocol and **terminate**.

Security analysis. The protocol is analyzed by proving lemmas for the protocol's safety and liveness separately, and proving that the correctness checks for keys and locks output consistent values for different parties. Using these lemmas, it is possible to prove Theorem 7.4. As in the description of the VLE protocol, the resilience threshold is $t < \frac{n}{4}$ because of the use of a packed AVSS protocol with guaranteed termination. Similarly to above, any resilience threshold between $\frac{n}{4}$ and $\frac{n}{3}$ can be adopted by allowing an ϵ probability of error or non-termination in the AVSS protocol, yielding the result of Theorem 2.1. A full description of the relevant lemmas, as well as their proofs and proof of the below theorem are provided in Appendix B. The efficiency of the protocol is analyzed in Appendix A.

Theorem 7.4. *Protocol* AVABA *(Protocol 7.3) securely implements (Functionality 7.1) in the presence of $t < n/4$ corrupted parties, in the F_{VLE}-hybrid model, assuming Input Assumption 7.2.*

8 Agreement on a Core Set (ACS)

We now turn to the main functionality: agreement on a core set, which is a simple corollary of Section 7. Recall the main application for MPC: Each party secret shares its input. If the dealer is honest, it is guaranteed that the sharing phase will terminate and all honest parties will receive their shares. If the dealer is corrupted, and the sharing phase terminated for one honest party, then it will eventually terminate for all other honest parties. The goal of ACS is to agree on a set of parties whose sharing phase has terminated.

The input assumption is the same as in Input Assumption 4.2: All honest parties validate each other, and if some honest party validates a corrupted party P_i, then eventually all honest parties will validate P_i. The functionality has a $\mathsf{validate}(i, j)$ command ("P_i sees that the sharing phase of P_j has terminated"). The adversary eventually chooses a set C of $n-t$ parties, and all parties receive C as output. The guarantee is that for every $k \in C$, there is some honest party that validated k. Again, we assume a cooperative adversary; otherwise, the functionality is slightly more complicated. In that case, core set C is set to be the indices k for which there is some honest party that validated k, and the parties receive that output when $n-t$ honest parties validated $n-t$ parties each. We now proceed to the definition of the functionality and the protocol. The efficiency of the protocol is analyzed in Appendix A.

Functionality 8.1: $\mathcal{F}_{\mathsf{ACS}}$ – Agreement on a Core Set Functionality

---

- $\mathsf{validate}(i, j)$: Whenever this functionality receives this command from some party P_i (via the router), forward the command to the ideal adversary and record the command.
- $\mathsf{setOutput}(C)$: Whenever the ideal adversary sends this command, verify that $|C| \geq n-t$ and that for every $x \in C$, there exists a recorded $\mathsf{validate}(\ell, x)$ for an honest P_ℓ. Send (output, C) to all parties and terminate.

Protocol 8.2: Π_{ACS} **– protocol implementing** $\mathcal{F}_{\mathsf{ACS}}$

1. **Initialize:** Each party P_i initializes $S_i \leftarrow \emptyset$.
2. **Upon** receiving $\mathsf{validate}(i, k)$ for some $k \in [n]$:
   (a) Add k to S_i.
   (b) If $|S_i| = n - t$, then broadcast $\langle \mathtt{set}, S_i \rangle$ and call $F_{\mathsf{AVABA}}.\mathsf{setInput}(i, S_i)$.
3. **Upon** receiving a $\langle \mathtt{set}, S'_j \rangle$ broadcast from P_j and having $S'_j \subseteq S_i$:
   (a) If $|S'_j| \subseteq [n]$ and $|S'_j| \geq n - t$ then call $F_{\mathsf{AVABA}}.\mathsf{validate}(i, S'_j)$.
4. **Upon** receiving (output, i, S) from F_{AVABA}: Output (output, S) and terminate.

Theorem 8.3. *Protocol 8.2 securely realizes Functionality 8.1 in the presence of a malicious adversary corrupting $t < n/4$ parties in the F_{AVABA}-hybrid model.*

Proof. The simulator runs the protocol with the real-world adversary $\mathcal{A}$, while simulating the F_{AVABA}-functionality. The simulator receives $\mathsf{validate}(j, k)$ commands from the environment and forwards the commands to the relevant parties in the simulated execution. In particular, with each $\mathsf{validate}(j, k)$ command, it adds k to the simualted S_j set. Once $|S_j| \geq n - t$, it simulates P_j invoking $F_{\mathsf{AVABA}}.\mathsf{setInput}(j, S_j)$, which notifies the adversary, and simulates P_i broadcast $\langle \mathtt{set}, S_j \rangle$. Likewise, it simulates the rest of the protocol.

The adversary must eventually call to $F_{\mathsf{AVABA}}.\mathsf{setOutput}(C)$. The simulator then verifies that $|C| \geq n - t$ and that for every $x \in C$, there is a recorded $\mathsf{validate}(\ell, x)$ for an honest P_ℓ, and then it calls to $\mathcal{F}_{\mathsf{ACS}}.\mathsf{setOutput}(C)$. It allows the delivery of the message (output, C) to each honest P_j in the ideal world whenever the simulated P_j gets output in the simulated execution.

Clearly, the view of the adversary is the same in both executions. To show that the outputs are the same, we rely on the input assumption and on the functionality F_{AVABA}. Specifically, Input Assumption 7.2 is satisfied: if some honest party called to $\mathsf{setInput}(j, S_j)$, it then broadcasts the set S_j. Each other honest party eventually validates the set S_j and calls to $F_{\mathsf{AVABA}}.\mathsf{validate}(S_j)$. As such, the cooperative adversary must send $\mathsf{setOutput}$ to F_{AVABA}. All honest parties receive an output from F_{AVABA} and this is their output in the real execution, exactly as in the ideal. □

Acknowledgements

We would like to thank Victor Shoup for valuable discussions and feedback.

G. Asharov was supported by the Israel Science Foundation (grant No. 2439/20), and by JPM Faculty Research Award. A. Patra would like to acknowledge financial support from SONY Faculty Innovation Award 2022, J.P Morgan Chase Faculty Award 2023, Google Privacy Research Award 2023. G. Stern was supported in part by ISF 2338/23, AFOSR Award FA9550-23-1-0387, AFOSR Award FA9550-23-1-0312, and an Algorand Foundation grant. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government, AFOSR or the Algorand Foundation.

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. 10

[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 Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology - EUROCRYPT 2023 - 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part II*, volume 14005 of *Lecture Notes in Computer Science*, pages 251–281. Springer, 2023. 3

[3] Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. Perfect asynchronous MPC with linear communication overhead. In *Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part V*, volume 14655 of *Lecture Notes in Computer Science*, pages 280–309. Springer, 2024. 3, 13, 14

[4] Ittai Abraham, Gilad Asharov, Arpita Patra, and Gilad Stern. Asynchronous agreement on a core set in constant expected time and more efficient asynchronous vss and mpc. Cryptology ePrint Archive, Paper 2023/1130, 2023. https://eprint.iacr.org/2023/1130. 5

[5] Ittai Abraham, Danny Dolev, and Gilad Stern. Revisiting asynchronous fault tolerant computation with optimal resilience. *Distributed Computing*, 35(4):333–355, 2022. 1, 12, 37

[6] 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. 5, 10, 11, 12, 37, 38

[7] 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. 2, 9, 11

[8] 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. 11, 37, 38

[9] 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. 17, 47

[10] 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. 2, 4

[11] 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. 1, 3, 4, 6

[12] Michael Ben-Or and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Comput.*, 16(4):249–262, 2003. 1, 4

[13] 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. 1, 4, 6, 12, 37

[14] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987. 40

[15] Ran Canetti. *Studies in secure multiparty computation and applications*. PhD thesis, Citeseer, 1996. 1, 2, 3, 7, 13, 23

[16] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In *Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II 35*, pages 3–22, 2015. 14

[17] 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. 12

[18] 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. 11

[19] Ashish Choudhury and Arpita Patra. An efficient framework for unconditionally secure multiparty computation. *IEEE Transactions on Information Theory*, 2016. 3, 13, 14

[20] Ashish Choudhury and Arpita Patra. On the communication efficiency of statistically-secure asynchronous MPC with optimal resilience. Cryptology ePrint Archive, Paper 2022/913, 2022. 5, 6

[21] Ran Cohen, Pouyan Forghani, Juan Garay, Rutvik Patel, and Vassilis Zikas. Concurrent asynchronous byzantine agreement in expected-constant rounds, revisited. *Cryptology ePrint Archive*, 2023. 5, 6

[22] Sourav Das, Sisi Duan, Shengqi Liu, Atsuki Momose, Ling Ren, and Victor Shoup. Asynchronous consensus without trusted setup or public-key cryptography. Cryptology ePrint Archive, Paper 2024/677, 2024. 5, 6

[23] Sisi Duan, Xin Wang, and Haibin Zhang. Practical signature-free asynchronous common subset in constant time. *IACR Cryptol. ePrint Arch.*, page 154, 2023. 5, 6

[24] Paul Neil Feldman. *Optimal algorithms for Byzantine agreement*. PhD thesis, Massachusetts Institute of Technology, 1988. 2, 4

[25] 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. 1

[26] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II*, volume 11693 of *Lecture Notes in Computer Science*, pages 85–114. Springer, 2019. 3

[27] 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. 10

[28] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*, volume 16. Elsevier, 1977. 23

[29] Jesper Buus Nielsen. Mpc techniques series, part 4: Beaver's trick, 2021. 3

[30] Arpita Patra, Ashish Choudhury, and C. Pandu Rangan. Efficient asynchronous verifiable secret sharing and multiparty computation. *J. Cryptol.*, 28(1):49–109, 2015. 3, 7, 8, 23

[31] 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. 11, 38

A Efficiency

The efficiency of the provided protocols is analyzed in this section.

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 $\mathcal{O}(m)$ bits. Concretely, we use the broadcast protocol of [9] 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 communication complexity of 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. □

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* $\mathcal{O}(n^2 \log n)$*, assuming a field element can be described in* $O(\log n)$ *bits. In addition, if the leader 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. 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 (described as bitmaps), resulting in a total communication complexity of $\mathcal{O}(n^3 \log n)$. 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. □

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 $\mathcal{O}(n)$ bits. 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 5. This protocol has $\mathcal{O}(n^3 \log n)$ complexity for sharing $\mathcal{O}(n)$ secrets. Reconstructing each secret in the sum individually achieves $\mathcal{O}(n^3 \log n)$ complexity per sum reconstructed.

Theorem A.3. *The total communication complexity of 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 $\mathcal{O}(n^4 \log n)$ communication. 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 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 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. □

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 $\mathcal{O}(n^4 \log n + n^2 \log n \cdot m)$.

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 F_{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 communication 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 $\mathcal{O}(n \cdot (b(n + m) + n^3 \log n + n \cdot m))$ bits to be sent in expectation overall. □

ACS Efficiency

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 ACS protocol with an efficiency of $\mathcal{O}(n^4 \log n)$.

Theorem A.5. *The expected total number of bits sent in the* ACS *protocol is* $\mathcal{O}(n \cdot (b(n) + n^3 \log n))$ *and all parties terminate after* $\mathcal{O}(1)$ *rounds in expectation.*

Proof. Parties simply broadcast $\mathcal{O}(n)$ bits and call the AVABA protocol with inputs of size $\mathcal{O}(n)$ (note that any subset of $[n]$ can be represented as a bitmap of n bits). The total communication complexity of these steps is $\mathcal{O}(n \cdot (b(n) + n^3 \log n))$. The broadcast protocols for all honest parties terminate in $\mathcal{O}(1)$ rounds. After completing these broadcasts, parties call AVABA and complete the protocol after $\mathcal{O}(1)$ expected rounds. In total this means that $\mathcal{O}(1)$ rounds are required in expectation. □

B Deferred Proofs for AVABA (Section 7)

In this section, we will show that AVABA is an Asynchronously Validated Asynchronous Byzantine Agreement protocol in Theorem 7.4. We start by proving several lemmas. Lemma B.2 and Lemma B.3 are instrumental for showing the *safety* of the protocol. By that we mean that:

> **Safety:** 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 B.9.

The remaining lemmas deal with the *liveness* of the protocol.

> **Liveness:** eventually some progress is made, leading to the (almost surely) 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 a unique chosen leader output from the F_{VLE} protocol (which happens with constant probability), all honest parties will commit at the end of that view.

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

Definition B.1. *Correctness of key/lock:*

- *A* `key` *message of the form* $\langle \texttt{key}, v, \mathsf{view} \rangle$ *is said to be correct if for some honest* P_j*, it holds that* $\mathsf{keyCorrect}_{j,\mathsf{view}'}(\mathsf{view}, v) = 1$ *for every* $\mathsf{view}' > \mathsf{view}$*.*
- *A* `lock` *message of the form* $\langle \texttt{lock}, v, \mathsf{view} \rangle$ *is said to be correct if* $\mathsf{lockCorrect}_j(\mathsf{view}, v) = 1$ *for an honest* P_j*.*

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 B.9.

Lemma B.2. *If two messages from a given* view *are correct, then they both have the same value v.*

Proof. First, observe two correct messages $\langle \texttt{key}, v, \mathsf{view}\rangle$ and $\langle \texttt{key}, v', \mathsf{view}\rangle$. 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 $|\{j | \exists k' \ s.t. (j, k', v) \in \mathsf{echoes}_{i,\mathsf{view}}| \geq n - t$. Party P_i adds a tuple (j, k', v) to $\mathsf{echoes}_{i,\mathsf{view}}$ after receiving a broadcasted $\langle \texttt{echo}, \mathsf{view}\rangle$ message from P_j, receiving $(\mathsf{output}, j, \ell)$ from $F_{\mathsf{VLE}}(\mathsf{view})$ for some ℓ and a $\langle \texttt{proposal}, k', v, \mathsf{view}\rangle$ broadcast from P_ℓ. This means that i received such broadcasts and outputs 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 $\langle \texttt{lock}, v', \mathsf{view}\rangle$. Similarly to the case above, for some honest P_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 $\langle \texttt{key}, v', \mathsf{view}\rangle$ 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 B.3. *If an honest party sends a $\langle \texttt{commit}, v\rangle$ message in line 6b of* processMessages(view)*, then for any* $\mathsf{view}' \geq \mathsf{view}$ *there exist $t+1$ honest parties P_j that only send an $\langle \texttt{echo}, \mathsf{view}'\rangle$ message if they output $(\mathsf{output}, j, \ell)$ from $F_{\mathsf{VLE}}(\mathsf{view})$ such that P_ℓ broadcast a message $\langle \texttt{proposal}, k, v, \mathsf{view}'\rangle$ for some k.*

Proof. We will prove inductively that for any $\mathsf{view}' \geq \mathsf{view}$, there must exist $t + 1$ such honest parties that only send `echo` messages as defined in the Lemma's statement.

First assume $\mathsf{view}' = \mathsf{view}$. Since some honest P_i sends a $\langle \texttt{commit}, v\rangle$ in line 6b, 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 P_i only does so after receiving $\langle \texttt{lock}, v, \mathsf{view}\rangle$ 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 $\langle \texttt{lock}, v, \mathsf{view}\rangle$ broadcast after seeing that $\left|\mathsf{keys}_{j,\mathsf{view}}\right| \geq n - t$. Following similar logic, they received $n-t$ $\langle \texttt{key}, v, \mathsf{view}\rangle$ messages and saw that they are correct. At least one of those messages was sent by an honest P_i that added $n-t$ tuples of the form (j, k, v) to its echoes set after receiving $\langle \texttt{echo}, \mathsf{view}\rangle$ broadcasts from $n - t$ parties $P_{j'}$ for whom P_i also received $(\mathsf{output}, j', \ell)$ from $F_{\mathsf{VLE}}(\mathsf{view})$ for some ℓ and a $\langle \texttt{proposal}, k, v, \mathsf{view}\rangle$ from P_ℓ. Note that P_i sent a `key` message, so it did not change the view number before sending the message, and thus at that time, every output $(\mathsf{output}, j', \ell)$ it had received from $F_{\mathsf{VLE}}(\mathsf{view})$ had the same leader ℓ. Since that leader broadcasts only one `proposal` message, every tuple (j, k, v) in $\mathsf{echoes}_{i,\mathsf{view}}$ had the same k and v. In other words, it sent its `key` message after receiving an `echo` broadcast for parties with the same leader ℓ that broadcasted the value v in its proposal.

Out of those parties, at least $t+1$ are honest and they only send one `echo` broadcast per view. From Lemma B.2, all correct `key` and `lock` messages from view have the same value v, and thus the `commit` message has 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 $\langle \texttt{lock}, v, \mathsf{view} \rangle$ broadcasts. Every honest party P_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 P_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 P_j such that $j \in I$ sent an $\langle \texttt{echo}, \mathsf{view}' \rangle$ message with after having received $(\mathsf{output}, j, \ell)$ from $F_{\mathsf{VLE}}(\mathsf{view}')$ and a broadcast $\langle \texttt{proposal}, k', v', \mathsf{view}' \rangle$ with $v' \neq v$. From the properties of the F_{VLE} functionality, some honest party P_i validated $P_{\ell'}$ 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 P_i did not send a `blame` broadcast. Party P_i adds such a tuple after receiving a $\langle \texttt{proposal}, k', v', \mathsf{view}' \rangle$ from ℓ' and seeing that $\mathsf{keyCorrect}_{i,\mathsf{view}'}(k', v') = 1$, so $\mathsf{view}' > k'$ and $\left|\{j' | \exists k'', \ell', \ s.t. \ (j', k'', v',) \in \mathsf{echoes}_{j,k'}\}\right| \geq n - t$. As discussed above, each honest party only adds a tuple (j', k'', v') to $\mathsf{echoes}_{j,k'}$ after receiving an `echo`, message, an output $(\mathsf{output}, j', \ell)$ from $F_{\mathsf{VLE}}(k')$ for some j', ℓ and a $\langle \texttt{proposal}, k'', v', k' \rangle$ broadcast from P_ℓ. 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. □

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

Definition B.4. *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 input assumption holds for F_{VLE}. In addition, we will show that parties always think that their own keys and lock are correct, and that if some honest party thinks that a key or lock is correct, all honest parties eventually think so as well. 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.

Lemma B.5. *The input assumption of F_{VLE} holds for any* view. *In addition, let* $\mathsf{key}_i, \mathsf{key\_val}_i$ *be the key set by an honest party P_i at some point in time. For any* $\mathsf{view} > \mathsf{key}_i$, *it holds that* $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i) = 1$ *at any point after setting the key (i.e. immediately returns 1). Furthermore, if for any pair (k, v) and honest party P_i,* $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$ *returns 1, then* $\mathsf{keyCorrect}_{j,\mathsf{view}}(k, v)$ *eventually returns 1 for any honest P_j.*

Proof. Let P_i, P_j be two honest parties and assume that at some point in time $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$ output 1. From the definition of $\mathsf{keyCorrect}$, $\mathsf{view} > k$ and $v \in \mathsf{validate}_i$. Note that P_i only adds v to $\mathsf{validated}_i$ after receiving $\mathsf{validate}(i, v)$ from the environment, and thus from the input assumption, every honest P_j will also validate v and add it to $\mathsf{validated}_j$. 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 P_j eventually calls $F_{\mathsf{VLE}}.\mathsf{validate}(j, \ell)$ if P_i calls $F_{\mathsf{VLE}}.\mathsf{validate}(i, \ell)$. 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 P_i calls $F_{\mathsf{VLE}}.\mathsf{validate}(j,\ell)$, then P_i received a $\langle \texttt{proposal}, k', v', \mathsf{view}\rangle$ broadcast from ℓ and seeing that $\mathsf{keyCorrect}_{i,\mathsf{view}}(k',v')=1$. P_j will receive the same broadcast and as shown above, eventually see that $\mathsf{keyCorrect}_{j,\mathsf{view}}(k',v')=1$. Following that P_j will also call $F_{\mathsf{VLE}}.\mathsf{validate}(j,\ell)$.

Now assume that the claim holds for every $\mathsf{view}' < \mathsf{view}$ and that P_i outputs 1 from $\mathsf{keyCorrect}_{i,\mathsf{view}}(k,v)$. If $k=0$, the claim holds immediately. Otherwise, $|\{j|\exists k' \ s.t. \ (j,k',v)\in \mathsf{echoes}_{i,k}\}| \geq n-t$. P_i adds a tuple of the form (j,k',v) to $\mathsf{echoes}_{i,k}$ after receiving an $\langle \texttt{echo}, k\rangle$ broadcast from a party $P_{j'}$, receiving a tuple $(\mathsf{output}, j', \ell)$ from $F_{\mathsf{VLE}}(k)$ and receiving a $\langle \texttt{proposal}, k', v, k$ broadcast from P_ℓ. Party P_j will receive the same broadcasts and call $\mathsf{VLEVerify}_{j,k}(\ell)$. Note that $\mathsf{view} > k$ and thus the input assumption holds for $F_{\mathsf{VLE}}(k)$, and thus P_j will eventually receive $(\mathsf{output}, j', \ell)$ as well and receive the same broadcasts. According to the functionality F_{VLE}, some party validated ℓ in $F_{\mathsf{VLE}}(k)$ and from the input assumption on $F_{\mathsf{VLE}}(k)$, P_j eventually does so as well. Before validating ℓ, P_j receives the same broadcast $\langle \texttt{proposal}, k', v, k$ from P_ℓ and adds $(\ell,(k',v))$ to $\mathsf{proposals}_{j,\mathsf{view}}$. This means that P_j adds the same tuples to $\mathsf{echoes}_{i,k}$ as P_i and eventually sees that the same condition holds, at which point it will output 1 from $\mathsf{keyCorrect}_{i,\mathsf{view}}$.

As for the input assumption of $F_{\mathsf{VLE}}(\mathsf{view})$, similarly to above, if P_i validated ℓ in $F_{\mathsf{VLE}}(\mathsf{view})$, it received a $\langle \texttt{proposal}, k', v', \mathsf{view}\rangle$ broadcast from P_ℓ and saw that $\mathsf{keyCorrect}_{i,\mathsf{view}}(k',v')=1$. P_j will receive the same broadcast, and from the above claims, P_j eventually outputs 1 from $\mathsf{keyCorrect}_{j,\mathsf{view}}(\ell)$ At that point, P_j will also validate ℓ in $F_{\mathsf{VLE}}(\mathsf{view})$. In addition, parties only broadcast `proposal` messages after receiving `suggest` messages, checking that the suggested keys are correct and choosing one of them. As argued above, all honest parties will see that the keys are correct as well, and thus accept the broadcasts and validate each other.

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 after P_i sets $\mathsf{key}_i, \mathsf{key\_val}_i$. First, by definition $\mathsf{view} > \mathsf{key}_i$ so the first condition checked in $\mathsf{keyCorrect}_{i,\mathsf{view}}$ holds. If P_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 definition, P_i's input is validated at the time P_i calls AVABA, so P_i will immediately see that $\mathsf{key}_i = 0$, output 1 and terminate. Otherwise, P_i updated both fields in the view key_i in line 4b after seeing that $\left|\mathsf{echoes}_{i,\mathsf{key}_i}\right| = n-t$. P_i only adds values to $\mathsf{echoes}_{i,\mathsf{key}_i}$ after receiving $\langle \texttt{echo}, \mathsf{key}_i\rangle$ broadcasts from parties P_j and $(\mathsf{output}, j, \ell')$ from $F_{\mathsf{VLE}}(\mathsf{key}_i)$ for some ℓ', and having a tuple $(\ell',(k',v'))$ in $\mathsf{proposals}_{i,\mathsf{view}}$. P_i does so after checking that $\mathsf{keyCorrect}_{i,\mathsf{key}_i}(k',v')=1$, and thus $v' \in \mathsf{validated}_i$ at that time. At that time for every $(j'',k'',v'') \in \mathsf{echoes}_{i,\mathsf{key}_i}$, $(k'',v'')=(k',v')$. That is because if that is not the case, P_i added tuples after receiving broadcasts and $(\mathsf{output}, j, \ell)$ for different leaders P_ℓ, and would have proceeded to the next view in line 3a before updating key_i. Therefore, $|\{j|\exists k'' \ s.t. \ (j,k'',v')\in \mathsf{echoes}_{i,\mathsf{key}_i}\}| \geq n-t$ at that time, so P_i will output 1 and terminate from $\mathsf{keyCorrect}_i(\mathsf{key}_i, \mathsf{key\_val}_i)$. □

Lemma B.6. *Let* $\mathsf{lock}_i, \mathsf{lock\_val}_i$ *be the lock set by an honest party* P_i *at some point in time. Then* $\mathsf{lockCorrect}_i(\mathsf{lock}_i, \mathsf{lock\_val}_i)=1$ *at any point after setting the lock. Furthermore, if for any pair* (l,v) *and honest party* P_i, $\mathsf{lockCorrect}_i(l,v)$ *returns* 1, *then* $\mathsf{lockCorrect}_j(k,v)$ *eventually returns* 1 *for any honest* P_j.

Proof. Let $P_{i,j}$ be two honest parties and assume $\mathsf{lockCorrect}_i(k,v)=1$ outputs 1 at some point in time. If $k=0$, then P_j immediately outputs 1 from $\mathsf{lockCorrect}_j(k,v)$ as well. Otherwise, $k>0$. Since P_i output 1 from $\mathsf{lockCorrect}_i(k,v)$, it saw that $\left|\{j|(j,v)\in \mathsf{keys}_{i,k}\}\right| \geq n-t$. P_i only adds a tuple (j,v) to its $\mathsf{keys}_{i,k}$ sets after receiving a $\langle \texttt{key}, v, k\rangle$ broadcast from j and seeing that $\mathsf{keyCorrect}_{i,k+1}(k,v)=1$. From Lemma B.5, $\mathsf{keyCorrect}_{k+1}$ eventually $\mathsf{keyCorrect}_{j,k+1}(k,v)$ outputs

1 as well. At that point, P_j will add the same tuple (j, v) to $\mathsf{keys}_{j,k}$. After adding all of those tuples, P_j will see that the same condition holds and return 1 from $\mathsf{lockCorrect}_j(k, v)$.

Next, we will show that $\mathsf{lockCorrect}_i(\mathsf{lock}_i, \mathsf{lock\_val}_i) = 1$ at any point after P_i sets $\mathsf{lock}_i, \mathsf{lock\_val}_i$. If P_i did not update those fields, then $\mathsf{lock}_i = 0, \mathsf{lock\_val}_i = \perp$. In that case, when running $\mathsf{lockCorrect}_i$, P_i will immediately see that $\mathsf{lock}_i = 0$ and output 1. Otherwise, P_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 $\langle \mathsf{key}, v, \mathsf{view} \rangle$ broadcast from j and seeing that $\mathsf{keyCorrect}_{i,\mathsf{view}+1}(\mathsf{view}, v) = 1$. From Lemma B.2, 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 $\mathsf{lockCorrect}_i(\mathsf{view}, v) = 1$ at that time. □

The next lemmas show that progress is made. We start in Lemma B.7 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 B.8 then shows that if an honest party is chosen as the unique leader using the F_{VLE} protocol, the adversary cannot convince any honest party to proceed to the next view using a `blame` message. We then show in Lemma B.9 that if some honest party terminates, every honest party does so as well. Finally, Lemma B.10 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.

Lemma B.7. *If the input assumption holds, 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 P_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 B.5, for every honest P_j, $\mathsf{keyCorrect}_{j,\mathsf{view}-1}(\mathsf{key}_j, \mathsf{key\_val}_j) = 1$ at the time it sent those fields, and eventually $\mathsf{keyCorrect}_{i,\mathsf{view}-1}(\mathsf{key}_j, \mathsf{key\_val}_j)$ terminates with the output 1 for every honest P_i. After receiving such a message from every honest P_j and seeing that the suggested $\mathsf{key}_j, \mathsf{key\_val}_j$ are correct, every honest P_i adds a tuple to $\mathsf{suggestions}$. After adding a tuple for each honest party, P_i broadcasts $\langle \texttt{proposal}, k, v, \mathsf{view} \rangle$ 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 P_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 B.5, $\mathsf{keyCorrect}_{i,\mathsf{view}-1}(0, x_i) = 1$ at that time. This means that when receiving its own broadcast, P_i adds $(i, (k, v))$ to and validates itself in $F_{\mathsf{VLE}}(\mathsf{view} - 1)$. As shown previously, the input assumption of $F_{\mathsf{VLE}}(\mathsf{view} - 1)$ holds, and thus every honest P_i eventually receives $(\mathsf{output}, j, \ell_j)$ for every honest P_j.

First, we will show that if some honest party proceeds to the next view, then the claim holds. If some honest party P_i sends a $\langle \texttt{blame}, \mathsf{lock}_i, \mathsf{lock\_val}_i, \mathsf{view} - 1 \rangle$ broadcast, then it output $(\mathsf{output}, i, \ell_i)$ from $F_{\mathsf{VLE}}(\mathsf{view} - 1)$ and saw that there is a tuple $(\ell_i, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$ such that $k < \mathsf{lock}_i$. It then sent the `blame` broadcast, and from Lemma B.6, $\mathsf{lockCorrect}_i(\mathsf{lock}_i, \mathsf{lock\_val}_i) = 1$ at that time. P_i added the tuple $(\ell_i, (k, v))$ to $\mathsf{proposals}_{i,\mathsf{view}-1}$ after receiving a $\langle \texttt{proposal}, k, v, \mathsf{view} - 1 \rangle$ broadcast from P_{ℓ_i} and outputting 1 from $\mathsf{keyCorrect}_{i,\mathsf{view}-1}(k, v)$. Every honest P_j eventually receives the same broadcast. From Lemma B.5, P_j will eventually output 1 from $\mathsf{keyCorrect}_{j,\mathsf{view}-1}(k, v)$

and add the same tuple to $\mathsf{proposals}_{i,\mathsf{view}-1}$. It will also eventually receive the `blame` message sent by P_i. P_j will see that $k < l$ and eventually receive $(\mathsf{output}, i, \ell_i)$ from $F_{\mathsf{VLE}}(\mathsf{view}-1)$ and 1 from $\mathsf{lockCorrect}_j(l, lv)$. After receiving all of these messages, P_j will proceed to the next view in line 2(a)i. On the other hand, if P_i proceeds to the next view in line 2(a)i, it received $(\mathsf{output}, j, \ell_j)$ from $F_{\mathsf{VLE}}(\mathsf{view}-1)$ and had a tuple $(\ell_j, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}-1}$. It then saw that $k < l$ and that $\mathsf{lockCorrect}_i(l, lv) = 1$. Similarly, from Lemma B.6, every honest party will see that $\mathsf{lockCorrect}_i(l, lv)$ outputs 1, and the same broadcasts that caused $(\ell_j, (k, v))$ to $\mathsf{proposals}_{i,\mathsf{view}-1}$. After that, every honest party will proceed to the next view. On the other hand, if an honest party P_i proceeds to the next view in line 3a, then it received $(\mathsf{output}, a, \ell_a), (\mathsf{output}, b, \ell_b)$ from $F_{\mathsf{VLE}}(\mathsf{view}-1)$ such that $\ell_a \neq \ell_b$. Every other honest party receives the same outputs from $F_{\mathsf{VLE}}(\mathsf{view}-1)$ and proceed to the next view. In other words, if some honest party proceeds to the next view, and no honest party completes the protocol in this view, then all honest parties proceed to view.

Now assume no honest party proceeds to view. In that case, every honest party P_i eventually outputs $(\mathsf{output}, i, \ell_i)$ from $F_{\mathsf{VLE}}(\mathsf{view}-1)$. Since ℓ_i was by some honest party P_j, it received a $\langle \texttt{proposal}, k, v, \mathsf{view}-1\rangle$ broadcast from P_{ℓ_i} and saw that $\mathsf{keyCorrect}_{j,\mathsf{view}-1}(k, v) = 1$. From Lemma B.5, P_i receives the same message and eventually sees that the same condition holds. This means that it broadcast a $\langle \texttt{echo}, \mathsf{view}-1\rangle$ message since it did not send a `blame` message instead. Every honest P_j receives that message, the same value $(\mathsf{output}, i, \ell_i)$ from $F_{\mathsf{VLE}}(\mathsf{view}-1)$, and the same `proposal` broadcast. After receiving these values, it adds a tuple to $\mathsf{echoes}_{i,\mathsf{view}-1}$. After adding such a tuple for each honest party, P_i has $|\mathsf{echoes}_{i,\mathsf{view}-1}| \geq n-t$, and thus it updates key_i to $\mathsf{view}-1$ and $\mathsf{key\_val}_i$ to v and broadcasts a $\langle \mathsf{key}, v, \mathsf{view}-1\rangle$ message during $\mathsf{view}-1$. From Lemma B.5, at that time $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{view}-1, v) = 1$. Every honest P_j receives that message and from Lemma B.5, eventually sees that $\mathsf{keyCorrect}_{j,\mathsf{view}}(\mathsf{view}-1, v) = 1$ as well. After that, P_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 P_j sends a `lock` message to every party. Following identical reasoning, every honest P_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 B.2, 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 P_i sees that it received $n-t$ such messages and completes the AVABA protocol in line 2. In other words, every honest party completes the protocol, reaching a contradiction. □

Lemma B.8. *No honest party P_i proceeds to the next view in lines 1b as a result of receiving $(\mathsf{output}, i, \ell_i)$ from $F_{\mathsf{VLE}}(\mathsf{view})$ if ℓ_i is honest or in 2(a)i as a result of receiving $\langle \texttt{blame}, l, lv, \mathsf{view}\rangle$ from P_j and $(\mathsf{output}, j, \ell_j)$ from $F_{\mathsf{VLE}}(\mathsf{view})$ if ℓ_j is honest.*

Proof. Assume by way of contradiction some honest party P_i does so. In both cases, it received $(\mathsf{output}, k, \ell)$ from $F_{\mathsf{VLE}}(\mathsf{view})$ for some honest P_ℓ, and had already added $(\ell, (k, v))$ to $\mathsf{proposals}_{i,\mathsf{view}}$ after receiving a $\langle \texttt{proposal}, k, v, \mathsf{view}\rangle$ broadcast from P_ℓ and 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 P_i proceeds in 1b, then $l = \mathsf{lock}_i, lv = \mathsf{lock\_val}_i$, and from Lemma B.6, $\mathsf{lockCorrect}_i(l, lv) = 1$ at that time. If P_i proceeded to the next view in line 2(a)i, 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 $\langle \mathsf{key}, v, l\rangle$ message from j'. At least $t+1$ of those tuples were added after receiving a `key` message from honest parties. Note that an honest $P_{j'}$ 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 P_ℓ broadcast in its `proposal` message. At the time it chose (k, v), P_ℓ had $|\mathsf{suggestions}| = n-t$, so it received $\langle \mathtt{suggest}, k', v', \mathsf{view} \rangle$ 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 $P_{j'}$ such that $j' \in I$, for whom $k' = \mathsf{key}_j \geq l$. P_ℓ 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. □

Lemma B.9. *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 $\langle \mathtt{commit}, v \rangle$ messages from $n-t$ parties, with $t+1$ of them being honest. Let P_i be the first honest party that sent such a message. First we will show that no honest party sends a $\langle \mathtt{commit}, v' \rangle$ message with any other value $v' \neq v$. Assume by way of contradiction that some honest party sends such a message, and let P_j be the first honest party to send such a message. Since both P_i and P_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 P_i and P_j sent their respective `commit` messages in line 6b at the end of view and view' respectively. Assume without loss of generality that $\mathsf{view} \leq \mathsf{view}'$. From Lemma B.3, 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 from parties P_k and receiving $(\mathsf{output}, k, \ell)$ for each one with the same leader P_ℓ. 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, P_j receives $n-t$ correct `lock` messages and sends a `commit` message with the value v' of a received correct `lock` message. From Lemma B.2, 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 P_i completes the protocol with the output v, every honest party will do so as well. Since P_i completed the protocol, it received $\langle \mathtt{commit}, v \rangle$ 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 $\langle \mathtt{commit}, v \rangle$ messages from at least $t+1$ parties. Once that happens, every honest party sends the same message to all parties in line 1. every honest party then receives those messages from at least $n-t$ honest parties and outputs v and terminates in line 2. 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 $\langle \mathtt{commit}, v \rangle$ messages from some $n-t$ parties and also sends a $\langle \mathtt{commit}, v \rangle$ message as described above. □

Lemma B.10. *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 B.9 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 P_i does in view is calling $\mathsf{viewChange}$ and sending a $\mathtt{suggest}$ message to every party with the local fields key_i and $\mathsf{key\_val}_i$. From Lemma B.5, $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i)$ at that time, and thus every honest P_j eventually has $\mathsf{keyCorrect}_{j,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i) = 1$ as well. Therefore, when an honest party P_j receives that message, it eventually adds a tuple to $\mathsf{suggestions}$. After receiving such a message from every honest party, P_j finds that $|\mathsf{suggestions}| \geq n - t$, and it broadcasts a $\langle \mathtt{proposal}, k, v, \mathsf{view} \rangle$ message. At that time it either has $(k, v) = (0, x_j)$ and as shown in Lemma B.5 $\mathsf{keyCorrect}_{j,\mathsf{view}}(k, v) = 1$, or it has chosen a tuple $(k, v) \in \mathsf{suggestions}$ for which it checked that $\mathsf{keyCorrect}_{j,\mathsf{view}}(k, v) = 1$.

Before an honest P_i proceeds to the next view, it must output some tuple $(\mathsf{output}, j, \ell)$ from $F_(\mathsf{VLE})(\mathsf{view})$. This means that all honest parties call $F_(\mathsf{VLE})(\mathsf{view})$ and wait to receive an output before any of them proceed to the next view. We now prove that if there is a party P^*_ℓ that acted honestly while broadcasting its $\mathtt{proposal}$ message such that every output $(\mathsf{output}, k, \ell)$ from $F_{\mathsf{VLE}}(\mathsf{view})$ has $\ell = \ell^*$, then all parties terminate during view. As shown in Claim 6.2, there is at least a $\frac{1}{3}$ probability of this event taking place.

If an honest P_i adds a tuple (j', k', v') to $\mathsf{echoes}_{i,\mathsf{view}}$, then it did so after receiving an $\mathtt{echo}$ message from P_j, an output $(\mathsf{output}, j, \ell)$ from $F_{\mathsf{VLE}}(\mathsf{view})$, and seeing a tuple $(\ell, (k', v'))$ in $\mathsf{proposals}_{i,\mathsf{view}}$. By assumption, $\ell = \ell^*$. An honest P_i adds a tuple $(\ell^*, (k', v'))$ to $\mathsf{proposals}_{i,\mathsf{view}}$ after receiving a $\langle \mathtt{proposal}, k', v', \mathsf{view} \rangle$ broadcast from ℓ^*, and thus all tuples in the set $\mathsf{echoes}_{i,\mathsf{view}}$ have the same values k', v'. No honest party proceeds to the next view in line 3a because all outputs from $F_{\mathsf{VLE}}(\mathsf{view})$ have the same leader. In addition, as shown in Lemma B.8, no honest party proceeds to the next view in either line 1b or line 2(a)i because they only receive outputs of the form $(\mathsf{output}, k, \ell^*)$ from $F_{\mathsf{VLE}}(\mathsf{view})$ for an honest P_{ℓ^*}.

Since no honest P_i sends proceeds to the next view in line 1b, each one sends an $\langle \mathtt{echo}, \mathsf{view} \rangle$ message after receiving $(\mathsf{output}, i, \ell^*)$ from $F_{\mathsf{VLE}}(\mathsf{view})$, and so do all other honest parties. After receiving the $\mathtt{echo}$ message and the same output from $F_{\mathsf{VLE}}(\mathsf{view})$, every honest P_i adds a tuple to $\mathsf{echoes}_{i,\mathsf{view}}$. After such a tuple is added for every honest party, P_i sees that $|\mathsf{echoes}_{i,\mathsf{view}}| = n - t$ and it sends a message $\langle \mathtt{key}, v, \mathsf{view} \rangle$ to all parties after updating key_j to view and $\mathsf{key\_val}_j$ to v. From Lemma B.5, 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 P_j. Therefore, when receiving that message, every honest P_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, P_j has $\left|\mathsf{keys}_{i,\mathsf{view}}\right| = n - t$ and it sends a $\mathtt{lock}$ message. Using identical arguments, eventually every honest party sends a $\mathtt{commit}$ message. Finally, after receiving a $\mathtt{commit}$ from $n - t$ parties, every honest party terminates. □

Main proof of AVABA

Theorem B.11. *Protocol 7.3 satisfies the following properties:*

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

Proof. Each property is proven individually.

Correctness. This property was proven in Lemma B.9.

Validity. Before some honest i outputs a value v, it sends $\langle \texttt{commit}, v, \mathsf{view} \rangle$ message. As discussed in the proof of Lemma B.9, at least $n - t$ parties sent `key` messages in view with the value v as well. At least one of those parties is honest. P_i only sends a $\langle \texttt{key}, v, \mathsf{view} \rangle$ message after receiving an $\langle \texttt{echo}, \mathsf{view} \rangle$ and seeing a corresponding tuple $(\ell, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$. P_i adds a tuple $(\ell, (k, v))$ to $\mathsf{proposals}_{i,\mathsf{view}}$ after receiving a $\langle \texttt{proposal}, k, v, \mathsf{view} \rangle$ 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, P_i sees that $v \in \mathsf{validated}_i$, which only happens after P_i received $(\mathsf{validate}, i, v)$.

Termination. If at any point an honest party terminates, from Lemma B.9, all honest parties do so as well. Observe some view, and assume no honest party terminated during view' for any $\mathsf{view}' < \mathsf{view}$. In that case, from Lemma B.7 all honest parties eventually reach view. Then, from Lemma B.10, 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 (at least $\frac{1}{3}$) 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. Furthermore, the expected number of views is bounded by 3.

Quality. Assume some honest party P_i completed the protocol, otherwise the claim holds trivially. This means that it called $F_{\mathsf{VLE}}(1)$ protocol in $\mathsf{view} = 1$. From Claim 6.2, with probability $\frac{1}{3}$ or greater all honest parties output $(\mathsf{output}, j, \ell^*)$ for a party P_{ℓ^*} that was honest when broadcasting a $\langle \texttt{proposal}, k, v, 1 \rangle$ message. Every honest party receives the message, eventually validates v and adds $(\ell^*, (k, v))$ to$\mathsf{proposals}_{i,\mathsf{view}}$. From Lemma B.10, 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 P_{ℓ^*}. No party can commit due to a message from an earlier view because there is no earlier view. Outputs the value v that P_{ℓ^*} proposed. Before sending its proposal, P_{ℓ^*} sees that $|\mathsf{suggestions}| = n - t$. P_{ℓ^*} only adds a tuple to $\mathsf{suggestions}$ after receiving the first $\langle \texttt{suggest}, k, v, \mathsf{view} \rangle$ 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), P_{ℓ^*} sees that the tuple with maximal k in $\mathsf{suggestions}$ has $k = 0$. P_{ℓ^*} 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. □

AVABA Simulation

Theorem B.12 (Theorem 7.4, restated)**.** *Protocol* AVABA *(Protocol 7.3) securely implements (Functionality 7.1) in the presence of $t < n/4$ corrupted parties, in the F_{VLE}-hybrid model, assuming Input Assumption 7.2.*

Proof. We provide the simulator $\mathcal{S}$: The simulator receives from the functionality notifications about the commands $\mathsf{validate}(i, x)$ and $\mathsf{setInput}(j, x_j)$. As such, there are no secret inputs for the

honest parties. It just simulated the honest parties in the ideal execution; whenever it receives a $\mathsf{setInput}(j, x_j)$ command for party P_j, it simulated the simulated P_j receiving this command from the environment. Whenever it receives a $\mathsf{validate}(j, x)$ command - it again forwards it to the simulated P_j. Moreover, it invokes the adversary $\mathcal{A}$ and simulates the protocol execution with the adversary. When the first simulated honest party P_j terminates (we will show that honest parties always terminate at some point), it sees its output. It takes its output y_j, and sends the command $\mathsf{setOutput}(y_j)$ to the functionality. The functionality verifies that y_j was validated by an honest party.If so, the functionality sends (output, y_j) to all parties. We will later show that the simulator always sends to the functionality an output that was validated by an honest party, and therefore, the functionality always accepts this value. The simulator allows the delivery of such to the honest P_k in the ideal world only when the simulated P_k terminates and receives its output. Note that along the way, the simulator also simulates the F_{VLE} functionality to the adversary $\mathcal{A}$, and follows the behavior of Functionality 6.1.

Since there are no secret inputs for the honest parties, the simulator perfectly simulates the adversary's view. To show that the outputs of the real execution and the ideal executions are identical, we have to show that:

1. Some honest party almost surely terminate in the simulated execution. This is proven in the Termination property in Theorem B. Moreover, this honest party outputs a value that is validated. This is the validity property in Theorem B. This guarantees that the simulator always sends some $\mathsf{setOutput}$ command to the functionality, and therefore, the simulator is "cooperative". Moreover, this guarantees that the functionality accepts the simulator value.
2. Once the simulator sends $\mathsf{setOutput}(y)$ with some value y, the functionality sends that value to all parties in the ideal execution. It allows the delivery of y to party P_j in the ideal world only when the simulated P_j terminates. To show that the simulator is valid, and that the outputs distribute as in the real, we show that all honest parties must terminate, and that their output is the same as the first honest party that terminated. This is formulated in Lemma B.9.

□