Perfectly Secure Asynchronous Agreement on a Core Set in Constant Expected Time

Ittai Abraham[1], Gilad Ashsarov[2], Arpita Patra[3], and Gilad Stern[4]

[1] Intel Labs
[2] Bar Ilan University
[3] Indian Institute of Science
[4] The Hebrew University of Jerusalem

Abstract. A major challenge of any *asynchronous* MPC protocol is the need to reach 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. This leads to a fundamental barrier of expected $\Omega(\log n)$ rounds for any asynchronous MPC protocol (even for constant depth circuits).
We provide a new solution for Agreement on a Core Set that runs in expected $O(1)$ rounds, is perfectly secure, and resilient to $t < \frac{n}{4}$ corruptions. Our solution 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. We show a similar result with statistical security for $t < \frac{n}{3}$.

1 Introduction

One of the core challenges for MPC protocols in the *asynchronous* setting is that they must reach *agreement* on which private inputs to use as input for the circuit. Ben-Or, Canetti and Goldreich [8] call this problem Agreement on a Core Set (ACS). In this paper we consider the most demanding setting: *perfect security with optimal-resilience in the asynchronous model*. From the lower bound of [2,8,10], perfect security for MPC implies that the number of corruptions in this setting is at most $t < \frac{n}{4}$, so optimal resilience is when $n = 4t + 1$ (this is in contrast to $n = 3t+1$ optimality in the synchronous setting). The seminal result of [8,13] is the first work to obtain perfect security with optimal-resilience in the asynchronous model.

In terms of round complexity, the best one can hope for is reaching agreement in constant expectation [19]. However, to the best of our knowledge, all results in the asynchronous information-theoretic setting run $O(n)$ parallel asynchronous binary Byzantine agreements instances in order to reach an agreement on a core set. For perfect security, each asynchronous binary Byzantine agreement runs some variant of the seminal agreement protocol of Feldman and Micali

[18]. Composing n parallel agreement protocols, where each protocol runs in constant expect time means that the expectation of the maximum is $\Omega(\log n)$. So for over 30 years, the best expected round complexity for asynchronous MPC has $\Omega(\log n)$ overhead (even for constant depth circuits). 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?*

Constant expected round complexity. Our main contribution is an agreement on a core set in constant expected time via a new multi-valued agreement protocol with an *asynchronous validity predicate*. Our Agreement protocol is perfectly-secure and resilient to $t < \frac{n}{4}$ corruptions. For inputs of size $\mathcal{O}(n)$ words, it runs in $\mathcal{O}(1)$ expected time and requires $\mathcal{O}(n^5)$ expected communication complexity. Parties are guaranteed to reach agreement on an input of one of the parties and the value is guaranteed to pass an asynchronous validity predicate. In the MPC setting this predicate checks that the input contains $n - t$ parties who verifiably completed the input sharing phase. To the best of our knowledge, the most efficient agreement protocols [7,18] with constant expected rounds and $t < \frac{n}{4}$ currently require $\mathcal{O}(n^6)$ words 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 (Asynchronously Validated Asynchronous Byzantine Agreement (informal)). *There exists a perfectly-secure protocol for asynchronous Byzantine agreement with an asynchronous validity predicate that is resilient to $t < \frac{n}{4}$ Byzantine corruptions. Each party has a valid input of size $\mathcal{O}(n)$ words. The protocol runs in constant expected time and $\mathcal{O}(n^5)$ expected communication complexity.*

Using this AVABA protocol and an asynchronous validity predicate for checking which parties completed sharing their inputs, we implement a perfectly-secure, $t < \frac{n}{4}$ resilient constant expected time protocol for Agreement on a Core Set (ACS) with an expected $\mathcal{O}(n^5)$ communication complexity.

More generally our AVABA protocol requires $O(n)$ secrets to be shared per party per round and can be generalized to a protocol resilient to $t < \frac{n}{3}$ corruptions. Our protocol uses packed AVSS to generate randomness. As proven in [2,10], when $n < 4t$, any AVSS protocol must have some non-zero probability of non-termination. The work of [14] constructs such a 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 (General Asynchronously Validated Asynchronous Byzantine Agreement (informal)). *Let $c \in [3, 4]$. Given a $n > ct$ resilient protocol*

for asynchronous verifiable secret sharing that runs in $S(n, \epsilon)$ communication complexity and has error $\epsilon \geq 0$ and probability of termination of $1-\epsilon$. 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 $1 - \tilde{\mathcal{O}}(\epsilon)$ (and in particular for $\epsilon = 0$ is almost-surely terminating), the protocol runs in constant expected time and has $\mathcal{O}(n^3 \log n + S(n, \epsilon))$ communication complexity.

To avoid the use of n different instances of binary agreement, we adopt the approach of external validity to the information theoretic setting. External validity [12] is a very successful framework in the authenticated setting for multi-valued agreement. To adopt to the information theoretic setting we define the notion of an asynchronous validity predicate, which is an information-theoretic asynchronous alternative to external validity functions. Such predicates act "like functions", but the results are delivered asynchronously. That is, if a value is valid, all parties will eventually see that it is, but otherwise they might not receive any output from the predicate. We then construct an agreement with an asynchronous validity predicate. To adopt this approach of using an asynchronous validity predicate we first extend the use of information theoretic protocols [5] from partial synchrony to asynchrony. This requires using asynchronous verifiable secret sharing to randomly choose leaders. Since we do not have a perfect leader election mechanism, we use the techniques of [3] that build a weaker notion of proposer election and adopt them to the information theoretic setting. Among other things, this requires re-formulating the gather primitive to support non-cryptographic primitives and changing the HotStuff variant to support an non-cryptographic asynchronous view change protocol.

1.1 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 to reach *agreement* on the set of parties whose private input is used as the input for the MPC circuit. To solve this, [8] suggest a protocol called Agreement on a Core Set (ACS). To the best of our knowledge, all previous asynchronous MPC protocols (in the perfect and statistical security setting) use the same ACS protocol suggested by [8]. This ACS is based on running n parallel binary agreements. Roughly speaking, parties enter binary agreement i with the value 1 when they see that party i has completed secret sharing its input and enter with the value 0 to all remaining instances 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 each constant number of rounds, 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 [7,18] in this setting require $\mathcal{O}(n^6)$ words to be sent in expectation, meaning that the expected total is $\mathcal{O}(n^7)$ for the ACS.

On Ben-Or and El-Yaniv's work A work by Ben-Or and El-Yaniv [9] deals with executing n concurrent instances of Byzantine Agreement. The first part of [9] is for the synchronous model and we believe can we 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 number of rounds. We note that [9] explicitly do **not** mention that they can solve ACS in the asynchronous model. Indeed, we believe that [9] and the techniques of [9] do **not** provide a way to solve ACS (as needed for asynchronous MPC) in constant expected rounds.

In [10]'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, leaders whose VSS instances have been completed). Only after seeing $n-t$ instances with output 1, parties input 0 to the remaining instances. Trying to naively apply the techniques of [9] 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 can 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 upon value for each party to be 1 before proceeding (while dealing with $\log(n)$ BA instances per party), and possibly having several synchronization steps using Select.

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 nonfaulty parties, meaning that each party has at least one nonfaulty 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 which 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.

1.2 Our Agreement on a Core Set

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

Instead of separating ACS into n instances, we run a single multi-valued instance, where the input of each party is a set of parties that completed their

input sharing. The main challenge is that corrupt parties can suggest incorrect inputs. In the authenticated case (computational settings), this is overcome using a Validated Asynchronous Byzantine Agreement which uses an authenticated external validity function. Here we use a new information theoretic *asynchronous validity predicate* (AVP) and provide a new type of multi-valued agreement in the perfect security setting which we call *asynchronously validated asynchronous Byzantine agreement* (AVABA). The validity property of an AVABA protocol is that the output is an (asynchronously validated) input of one of the parties. The asynchronous validation guarantees that even if agreement is reached on the input of a corrupted party, this output (agreement value) is sufficient for agreeing on a core set of parties that completed their input sharing.

Implementing an AVABA protocol. The construction of an AVABA protocol follows the ideas and construction of the No Waitin' HotStuff (NWH) protocol of [3]. Seeing as the NWH protocol is designed in the authenticated setting and uses a signature scheme, this work adapts these ideas to the information theoretic setting, removing the need for cryptography. An important consideration when adapting these protocols, is that parties need to wait in order for their outputs to be dynamically validated before outputting them. This is done so that they can to check for validity without waiting for an asynchronous event that may never happen after outputting a value. Roughly speaking, our AVABA protocol has two parts, each with its separate challenges. The first part is a weak validated leader election protocol. This protocol is inspired by the synchronous leader election protocol of [21], its efficiency improvements [1], and the asynchronous authenticated proposal election protocol of [3]. As in [1,3], in order to efficiently generate randomness we used packed secret sharing. Here we use the packed secret sharing of [16]. As in [3] we use a gather based protocol to make sure a large core of potential leaders is common to all parties. Unlike [3], which relies on signatures and authentication, we implement an information theoretic version of gather whose inputs comply with an asynchronous validity predicate and whose output can be verified in a verification protocol. Morover, our gather protocol is unique in that it outputs a set of parties, while their values are inferred via an asynchronous validity predicate.

The second part is the asynchronous agreement protocol given a weak leader election. Here the main challenge is working with asynchronously validated inputs and maintaining both safety and liveness over the views. For safety, we use a common approach in authenticated protocols [15,22] of using lock certificates and adapt them to the information theoretic setting inspired by the approach of [5]. The protocol of [5] modifies the cryptographic protocol of [22] to the information theoretic setting in partial synchrony. Here we show how to obtain liveness under fully asynchronous network conditions. For liveness in asynchrony, there are two major challenges. The first is to guarantee that if a unique honest leader is elected, then all honest parties will reach agreement on its proposal. For this, we use the key certificate approach of [4,22] and adapt it to the information theoretic setting. The second, more challenging problem is guaranteeing that honest parties eventually proceed to a new view if the current view does not lead to

agreement. As in [3] we observe that there are two triggers: the first is when two different parties have different leaders (equivocation event) and the second is when the leader sends a proposal whose key is lower than a lock held by some party (blame event). In [3] these two events can simply be verified via an external validity function, so any honest party that observes this event simply forwards it to all parties. In our setting we adapt these two events into asynchronously validated predicates. Roughly speaking, when an equivocation or blame message is sent, parties record it and wait for it to be asynchronously validated.

2 Definitions and Assumptions

2.1 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 Byzantine adversary controlling up to $t < \frac{n}{3}$ parties, and can thus be used when $t < \frac{n}{4}$ as well. Furthermore, the adversary is adaptive in the sense that it can choose to corrupt a party at any time given that it hasn't already corrupted t parties. In the discussion of the efficiency of protocols we use the notion of "words", that can contain field elements, indices and counters.

2.2 Asynchronous Validity Predicate

We generalize the notion of dynamic predicates of [10] to a new notion of *asynchronous validity predicates* (or validity predicates for short). This type of predicate behaves like the asynchronous counterpart to the notion of an external validity function [12]. An external validity function, receives an input and returns 1 or 0, corresponding to a Boolean $true$ or $false$. Similarly, in an asynchronous validity predicate, party i holds a predicate $\mathsf{validate}_i$, and the value of $\mathsf{validate}_i(v)$ can depend on i's internal state. Unlike an external validity function, $\mathsf{validate}_i$ could potentially not terminate. Hence a validity predicate is an asynchronous version of an external validity function checking the party's internal state, and only returning a value when specific conditions hold. Similar notions of stateful predicates have also been discussed in other works requiring a dynamic property to be checked before outputting a value, such as the ones in [10,17,20].

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

Formally, these properties are captured in the following definition:

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

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

For brevity, when we say that some property holds for the predicate validate, we mean that it holds for every $\mathsf{validate}_i$. For example, if for every $i \in [n]$, $\mathsf{validate}_i$ is a predicate from the domain X to the range Y, then we simply say that $\mathsf{validate} : X \to Y$. As a shorthand we say that $\mathsf{validate}_i(v) = b$ at a given time for an honest i if $\mathsf{validate}_i(v)$ had already terminated at that time and output b or if it would immediately terminate with the output b upon being called at that time.

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

2.3 Verifiable Party Gather

Verifiable Party Gather is a variation of the Verifiable Gather protocol of [3]. The first difference is that we only output a set of parties (no values) and the second difference is that the cryptographic external validity function is replaced by an information theoretic asynchronous validity predicate, as defined in Section 2.2 that takes as input a party index. Intuitively, the goal of a *party gather* protocol is to have some common *core of parties* such that each honest party outputs a set of parties that is a super-set of this core. Intuitively, the goal of a *verifiable party gather* protocol is to make sure that the set of parties that are output by an honest party can be verified to be correct outputs of the protocol. Observe that different parties may output different super-sets of the core and there is no agreement on who is in the core.

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

The properties of Gather:

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

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

The termination properties of Verify:

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

The correctness properties of the Verify protocol:

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

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

2.4 Verifiable Leader Election

A perfect leader election would allow each have all parties 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 the output of VLE is one common randomly elected proposal coming from an honest proposer. As in the Verifiable Party Gather protocol, we also add a verification protocol. Crucially, in the good event mentioned above, the only value that passes verification is this commonly elected leader. In the remaining cases, the adversary can control the output and even cause different parties to have different outputs. However, even in these cases we force the adversary to allow all parties to eventually output some verifying value. This VLE is weak enough to be efficiently implementable, even in an information theoretic setting and we will later show that it is strong enough to enable an efficient constant expected round VABA protocol.

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

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

In addition, the VLE protocol has a natural termination property (assuming all honest start):

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

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

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

Finally, the correctness properties of Verify:

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

2.5 Asynchronously Validated Asynchronous Byzantine Agreement

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

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

2.6 Agreement on a Core Set

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

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

In this work, we assume the existence of protocols for Reliable Broadcast protocol and a Packed Asynchronous Verifiable Secret Sharing (packed AVSS). We use the protocols of [6,16]. For formal definitions of those protocols, see Appendix A.

3 Verifiable Party Gather

As part of our proposal election protocol we require a "reliable party gather". Throughout the protocol, parties broadcast values, which are later used to choose a winning proposal from among them. Ideally, we would like the parties to agree on an exact set of parties that broadcasted in order to make sure that they all elect a value from the same set. However, exactly agreeing on the set is non-trivial and potentially expensive. Therefore we slightly relax our requirements: there exists some core C of size $n - t$ or greater such that the output of every honest party contains C. Furthermore, we would like parties to be able to prove that they "acted correctly" and included C in their output.

Throughout the protocol, parties broadcast messages. The protocol takes place in two rounds. In the beginning, all parties broadcast their inputs $S_i \subseteq [n]$, which are sets of parties. We also assume that all parties have access to an asynchronous validity predicate $\mathsf{validate}$, and that for every honest i, $\mathsf{validate}_i(x) = 1$ for every $x \in S_i$ at the time it calls the Gather protocol. After an honest i receives such a set S_j from party j, it waits to see that $\mathsf{validate}_i(x) = 1$ for every $x \in S_j$.

When this condition holds, i records j as a party from whom it received a set and stores j in T_i. In addition, it adds all indices in S_i to its eventual output value, R_i. In the second round, i sends its set T_i after its size is at least $n-t$. When receiving a set T_j from party j, i waits until it sees that $T_j \subseteq T_i$. After seeing that this is the case for $n-t$ parties, i terminates and outputs R_i.

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

Algorithm 1 $\mathsf{Gather}_i(S_i)$

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

Algorithm 2 $\mathsf{GatherVerify}_i(X)$

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

3.1 Security Analysis

We start by proving that many parties send T sets with an index i^*, which will later be used for defining the common core of the protocol. We then show

that parties eventually have consistent views, and conclude with proving that (Gather, GatherVerify) is a Verifiable Party Gather protocol in Theorem 3. The proofs of the following lemmas are provided in Appendix B.

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

Lemma 2. *Let i, j be two honest parties. Observe the sets T_i, U_i at any time throughout the protocol. Eventually $T_i \subseteq T_j$ and $U_i \subseteq U_j$.*

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

Proof. Each property is proven separately.

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

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

Agreement on Verification. Assume that some honest i completes the $\mathsf{GatherVerify}_i(X)$ protocol on some set X and outputs $b \in \{0, 1\}$. Honest parties never output 0 from the GatherVerify protocol, and thus i output 1. This means that it saw that $|\{k|\exists(k, V_k) \in U_i, V_k \subseteq X\}| \geq n - t$ and that for every $x \in S_k$, $\mathsf{validate}_i(x) = 1$. Let j be some honest party that called $\mathsf{GatherVerify}_j(X)$. From Lemma 2, eventually $U_j \subseteq U_i$ and thus eventually $|\{k|\exists(k, V_k) \in U_j, V_k \subseteq X\}| \geq n-t$. In addition, from the Consistency property of validate, j will also eventually

see that $\mathsf{validate}_j(x) = 1$ for every $x \in X$. After both of those conditions hold, j outputs 1 from the GatherVerify protocol and terminates.

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

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

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

3.2 Efficiency

In the following discussion, we assume the existence of a broadcast protocol that terminates in $\mathcal{O}(1)$ rounds with $\mathcal{O}(b(m))$ words sent when broadcasting inputs of size $\mathcal{O}(m)$ words. Concretely, we use the broadcast protocol of [6] in which parties send $\mathcal{O}(n^2 \log n + n \cdot m)$ words 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 $\mathcal{O}(n^3 \log n)$ words and $\mathcal{O}(1)$ rounds.

Theorem 4. *The total number of words sent in the* Gather *protocol is* $\mathcal{O}(nb(n))$ *and all parties terminate after* $\mathcal{O}(1)$ *rounds.*

Proof. In the protocol, every party sends a constant number of broadcasts, totalling in $\mathcal{O}(n \cdot b(n))$ words 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.

4 Verifiable Leader Election

This section describes the construction of a verifiable leader election, which is related to the idea of a weak common coin and proposal election. With constant probability all honest parties output an honest leader from VLE, but in other cases parties might disagree on the leader. In both cases, every party's output must be an asynchronously validated leader according to an asynchronous validity predicate validate. Every honest party starts the protocol believing that it is a valid leader, i.e. with $\mathsf{validate}_i(i) = 1$. Since the predicate is consistent, honest parties will eventually agree that other honest parties are also valid leaders. In addition, parties can verify each other's output with a verification protocol, VLEVerify. In the constant probability event described above, in which a single honest leader is elected, this is the only leader that will pass verification in the VLEVerify protocol. Our construction uses techniques inspired by synchronous weak leader election [21] and cryptographic proposal election [3].

The protocol proceeds in 5 rounds described below:

Round 1: In the first round, every party shares n random values using a packed AVSS protocol, one for each party. Parties then participate in the packed AVSS instances with every party as dealer.

Round 2: In the second round, after completing the Share protocol for $t+1$ dealers, party i broadcasts an "attach" message with the set $\mathsf{dealers}_i$ for which it completed the share protocol. After receiving such a message from party j with a set $\mathsf{dealers}_j$, i checks that it also completed the Share protocol for the dealers in $\mathsf{dealers}_j$ and waits until it considers j to be valid according to $\mathsf{validate}_i$. That is, it checks that j is a valid leader that has committed to a random value, which is the sum of the j'th secrets shared by the dealers in $\mathsf{dealers}_j$.

Round 3: In the third round, i waits to see that $n-t$ parties committed to their random values and then inputs the set of those parties, $\mathsf{attached}_i$, to the Gather protocol. It does so with an asynchronous validity predicate checking that each party in $\mathsf{attached}_i$ is a valid candidate that actually committed to a random secret and.

Round 4: After completing the Gather protocol, i outputs a set of parties that it considers to be viable candidates who can be chosen as leaders and output from the VLE protocol. In order to be able to choose a single leader, i broadcasts a "candidates" message with that set of candidates, asking for parties to help reconstruct their attached random value.

Round 5: After receiving a "candidates" message from party j with a set $\mathsf{candidates}_j$, i checks that $\mathsf{candidates}_j$ is a valid output from the Gather protocol by calling the GatherVerify protocol. After the GatherVerify protocol returns 1 on the set $\mathsf{candidates}_j$, i starts reconstructing the sum of the k'th secrets shared by dealers in $\mathsf{dealers}_k$ for every $k \in \mathsf{candidates}_j$. In other words, it helps reconstruct the random value for each candidate. Note that parties only start reconstructing the secrets associated with a party k after seeing that k broadcasted its set of dealers. Since the set of dealers must be at least of size $t+1$, one of those secrets was shared by an honest dealer. This guarantees that the sum will be completely random and unknown to k, who hasn't seen any k'th secret reconstructed yet.

Output: Finally, after reconstructing the random values associated with each of its own candidates, i outputs the candidate with the highest random value. As proof, it also outputs the set of candidates $\mathsf{candidates}_i$.

Intuitively, every party outputs a set of candidates from the Gather protocol who have already committed to their random value. If the party with the maximal random value happens to be an honest party ℓ^* in the binding core of the Gather protocol, then all honest parties will see that random value and pick ℓ^* as their output. Since the values are sampled uniformly in an unbiased manner, this means that every party has the same probability of having the maximal evaluation being associated with it. When counting the number of honest parties in the common core, we find that the probability of the aforementioned event is at least $\frac{1}{3}$. This mechanism also allows to check whether a given proposal could have been the correct output from the VLE protocol. In order to convince an honest party that a value is a correct output from the VLE protocol, parties can provide their output from the Gather protocol. Parties will then be able to check if that set of parties is a possible output from Gather (i.e. if it contains the core), and if the correct leader was elected based on that set. If the maximal random value is associated with a party in the core, then only sets containing that party will verify, which means that only the honest leader ℓ^* will verify.

4.1 Security Analysis

We start by proving that parties' views are eventually consistent and that checkValidity, which is used an asynchronous validity predicate for the Gather protocol is indeed one. Following that, we prove that the (VLE, VLEVerify) are indeed a Verifiable Leader Election protocol in Theorem 5. The proofs of the following lemmas are provided in Appendix C

Lemma 3. *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* $\mathsf{ranks}_i \subseteq \mathsf{ranks}_j$.

Lemma 4. checkValidity *is an asynchronous validity predicate.*

In the following theorem, the threshold $t < \frac{n}{4}$ stems from using AVSS protocols, for which this threshold is necessary in order to guarantee their termination [10,2]. Similar results can be achieved by using AVSS protocols with an ϵ probability of failure or non-termination, yielding results with only probabilistic guarantees.

Theorem 5. *The pair* (VLE, VLEVerify) *is a Verifiable Leader Election protocol resilient to $t < \frac{n}{4}$ parties with $\alpha = \frac{1}{3}$.*

Proof. Each property is proven separately.

Termination of Output. If all honest parties participate in the VLE protocol, then they all sample n values and share them using packed AVSS. From

Algorithm 3 $\mathsf{VLE}_i()$

$\mathsf{dealers}_i \leftarrow \emptyset, \mathsf{attached}_i \leftarrow \emptyset, \mathsf{candidates}_i \leftarrow \emptyset, \mathsf{ranks}_i \leftarrow \emptyset$
$s_1, \ldots, s_n \xleftarrow{\$} \mathbb{F}$
share $s_1, \ldots, s_n$ using a packed AVSS protocol and participate in the PAVSS instances with every party as dealer
upon completing all Share calls with j as dealer, **do**
    $\mathsf{dealers}_i \leftarrow \mathsf{dealers}_i \cup \{j\}$
    if $|\mathsf{dealers}_i| = t + 1$ **then**
        broadcast $\langle$"attach", $\mathsf{dealers}_i\rangle$
upon receiving an $\langle$"attach", $\mathsf{dealers}_j\rangle$ broadcast from j, **do**
    upon $\mathsf{dealers}_j \subseteq \mathsf{dealers}_i$, $|\mathsf{dealers}_j| \geq t + 1$ and $\mathsf{validate}_i(j)$ terminating with the output 1, **do**
        $\mathsf{attached}_i \leftarrow \mathsf{attached}_i \cup \{(j, \mathsf{dealers}_j)\}$
        if $|\mathsf{attached}_i| = n - t$ **then**
            call $\mathsf{Gather}_i(\{k | \exists (k, \mathsf{dealers}_k) \in \mathsf{attached}_i\})$ with the validity predicate $\mathsf{checkValidity}_i$
upon Gather_i outputting the set X_i, **do** ▷ continue updating state according to Gather
    $\mathsf{candidates}_i \leftarrow X_i$
    broadcast $\langle$"candidates", $\mathsf{candidates}_i\rangle$
upon receiving a $\langle$"candidates", $\mathsf{candidates}_j\rangle$ broadcast from j, **do**
    upon $\mathsf{GatherVerify}_i(\mathsf{candidates}_j)$ terminating with the output 1 and Gather_i terminating, **do**
        for all $k \in \mathsf{candidates}_j$ **do**
            call $\mathsf{Sum-Reconstruct}(k, \mathsf{dealers}_k)$ for $(k, \mathsf{dealers}_k) \in \mathsf{attached}_i$
upon $\mathsf{Sum-Reconstruct}(j, \mathsf{dealers}_j)$ terminating with the output r_j, **do**
    $\mathsf{ranks}_i \leftarrow \mathsf{ranks}_i \cup \{(j, r_j)\}$
upon $\mathsf{candidates}_i \neq \perp$ and $\forall j \in \mathsf{candidates}_i\ \exists (j, r_j) \in \mathsf{ranks}_i$, **do**
    $\ell \leftarrow argmax\{r_j | j \in \mathsf{candidates}_i, (j, r_j) \in \mathsf{ranks}_i\}$ ▷ ℓ is the party with the highest rank r_ℓ
    $\pi_i \leftarrow \mathsf{candidates}_i$
    output (ℓ, π_i), but continue updating internal sets and sending messages

Algorithm 4 $\mathsf{checkValidity}_i(k)$

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

Algorithm 5 $\mathsf{VLEVerify}_i(k, \pi)$

upon $\forall j \in \pi\ \exists (j, r_j) \in \mathsf{ranks}_i$, **do**
    upon $\mathsf{GatherVerify}_i(\pi)$ terminating with the output 1 and Gather_i terminating, **do**
        $\ell \leftarrow argmax\{r_j | j \in \pi, (j, r_j) \in \mathsf{ranks}_i\}$
        if $k = \ell$ **then**
            output 1 and **terminate**

the Termination property of AVSS, every honest party i will complete those calls, add every honest j to $\mathsf{dealers}_i$ and broadcast $\langle$"attach", $\mathsf{dealers}_i\rangle$ when it $|\mathsf{dealers}_i| = t+1$. After an honest i receives an "attach" message from an honest j, it sees that $|\mathsf{dealers}_j| \geq t+1$. In addition, from Lemma 3, eventually $\mathsf{dealers}_j \subseteq \mathsf{dealers}_i$. By assumption, $\mathsf{validate}_j(j) = 1$ for every honest j at the time it starts the VLE protocol, so from the Consistency property of the predicate $\mathsf{validate}_i(j)$, will eventually output 1 for every honest i. When these conditions hold, i adds $(j, \mathsf{dealers}_j)$ to $\mathsf{attached}_i$. After adding such a tuple for every honest j, i sees that $|\mathsf{attached}_i| = n - t$ and it calls Gather_i with the input $S_i = k|\exists(k, \mathsf{dealers}_k) \in \mathsf{attached}_i$. Clearly, for every $k \in S_i$ there is a tuple of the form $(k, \mathsf{dealers}_k)$ in $\mathsf{attached}_i$, so $\mathsf{checkValidity}_i(k) = 1$. From Lemma 4, checkValidity is an asynchronous validity predicate and all honest parties eventually call the Gather protocol, so from the Termination protocol of Gather they all eventually complete the protocol. When an honest i completes the Gather protocol with an output X_i, it updates its $\mathsf{candidates}_i$ set to X_i and broadcasts $\langle$"candidates", $\mathsf{candidates}_i\rangle$. Every honest j receives that broadcast and from the Completeness property, $\mathsf{Gather}_j(\mathsf{candidates}_i)$ eventually terminates with the output 1. At that point, j calls $\mathsf{Sum-Reconstruct}(k, \mathsf{dealers}_k)$ for every $k \in \mathsf{candidates}_i$ with $(k, \mathsf{dealers}_k) \in \mathsf{attached}_j$. Note that honest parties add those tuples after receiving the same $\langle$"attach", $\mathsf{dealers}_k\rangle$ broadcast, so they all call $\mathsf{Sum-Reconstruct}$ with the same set of dealers. From the Termination property of the AVSS protocol, i completes the $\mathsf{Sum-Reconstruct}_i(k, \mathsf{dealers}_k)$ call for every $k \in \mathsf{candidates}_i$ and adds a tuple (k, r_k) to ranks_i. Finally, after having $\mathsf{candidates} \neq \perp$ and there being a tuple $(k, r_k) \in \mathsf{ranks}_i$ for every $k \in \mathsf{candidates}_i$, i performs local computations, outputs some value and terminates.

Completeness. Assume some honest party i outputs the index ℓ and proof π from VLE. Party i chooses ℓ to be the index ℓ such that $\ell = argmax\{r_j | j \in \mathsf{candidates}_i, (j, r_j) \in \mathsf{ranks}_i\}$ and sets π to $\mathsf{candidates}_i$, which was i's output from the Gather_i protocol. Observe some honest party j that calls $\mathsf{VLEVerify}_j(\ell, \pi)$. Note that i only completes the VLE protocol after seeing that for every $k \in \mathsf{candidates}_i$ there exists a tuple $(k, r_k) \in \mathsf{ranks}_i$. From Lemma 3, eventually $\mathsf{ranks}_j \subseteq \mathsf{ranks}_i$ and thus $\forall k \in \pi\ \exists(k, r_k) \in \mathsf{ranks}_j$. In addition, from the Completeness property of GatherVerify, $\mathsf{GatherVerify}_j(\pi)$ eventually terminates with the output 1. As shown above, j eventually completes Gather_j and proceeds to compute ℓ. Both i and j compute ℓ to be $argmax\{r_k | k \in \mathsf{candidates}_i\}$, with r_k being the output from $\mathsf{Sum-Reconstruct}(k, \mathsf{dealers}_k)$. From the Correctness property of AVSS, both i and j receive the same output r_k for every k, and thus they compute the same ℓ as the index with maximal r_k. Therefore, $\mathsf{VLEVerify}_j(\ell, \pi)$ outputs 1 and terminates.

α-Binding. From the Includes Core property of the Gather protocol, at the time the first honest party completes the Gather protocol, there exists a binding core X^* of at least $n - t$ indices in $[n]$ such that if $\mathsf{GatherVerify}_i(X)$ terminates with the output 1 for an honest party i, then $X^* \subseteq X$. Note that at least $n - 2t \geq t + 1 > \frac{n}{3}$ of those indices are honest parties' indices. Let I be the set of all parties k for which at least one honest party j called $\mathsf{Sum-Reconstruct}_j(k, \mathsf{dealers}_k)$.

Before calling the protocol, j completes Gather_j, calls $\mathsf{GatherVerify}_j(\mathsf{candidates})$ and sees that it terminates with the output 1 for some set $\mathsf{candidates}$ which includes k. From the Validity property of the Gather protocol, there already existed a tuple $(k, \mathsf{dealers}_k) \in \mathsf{attached}_i$ at that time and from the Binding Core property of the Gather protocol X^* is already defined at that time. Note that from the Correctness property of $\mathsf{Sum-Reconstruct}$, r_k is the sum of the k'th secrets shared by the dealers in $\mathsf{dealers}_k$. For each $(k, \mathsf{dealers}_k) \in \mathsf{attached}_j$, $\mathsf{dealers}_k$ has at least $t+1$ indices, and thus at least one of the dealers was honest. That honest dealer shared a uniformly sampled value, and no honest party started reconstructing r_k or the uniformly sampled secret shared by the honest dealer before receiving an "attach" broadcast from k. Therefore, from the Secrecy property of the AVSS protocol, the value shared by the honest dealer is sampled uniformly and independently of the adversary's view at that time. This means that for every $k \in I$, r_k is sampled uniformly and independently from all other values and from the set X^*. Therefore, the probability of a given party $k \in I$ having the maximal rank r_k is $\frac{1}{|I|} \geq \frac{1}{n}$[5]. This means that the probability that there exists a nonfaulty party ℓ^* such that $\ell^* \in X^*$ and r_{ℓ^*} is the maximal rank among all r_k such that $k \in I$ is at least $\frac{n}{3} \cdot \frac{1}{n} = \frac{1}{3}$. If that is the case, define ℓ^* to be that party's index, otherwise define it to be $\perp$.

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

Agreement on Verification Let i, j be two honest parties and ℓ, π be two values such that $\mathsf{VLEVerify}_i(\ell, \pi)$ terminates with the output b. Honest parties only output 1 from the $\mathsf{VLEVerify}$ protocol so $b = 1$. Party i starts $\mathsf{VLEVerify}$ by waiting until $\forall k \in \pi$, there exists a tuple $(k, r_k) \in \mathsf{ranks}_i$. From Lemma 3, eventually $\mathsf{ranks}_i \subseteq \mathsf{ranks}_j$, so j will see that this condition holds. Following that, i sees that $\mathsf{GatherVerify}_i(\pi)$ terminates with the output 1 and that Gather_i terminates. From the Agreement on Verification and Termination properties of the Gather protocol, j sees that these conditions hold as well. At that time, $\mathsf{checkValidity}_i(k) = 1$ was true for every $k \in \pi$ and thus there was a tuple of the form $(k, \mathsf{dealers}_k) \in \mathsf{attached}_i$. The same holds for j, which received the same broadcast and added the same tuples to $\mathsf{attached}_j$. In addition, both i and j add the tuples (k, r_k) to their respective ranks sets after reconstructing r_k in the call to $\mathsf{Sum-Reconstruct}(k, \mathsf{dealers}_k)$. From the Correctness property of the protocol, they both reconstruct the same value, so they have the same tuples

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

for every $k \in \pi$. They then compute ℓ in the same way with regard to the same tuples (k, r_k). For that index, i saw that $k = \ell$, and thus j will see that the same condition holds, output 1 and terminate.

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

4.2 Efficiency

As above, we assume the existence of a broadcast protocol that terminates in $\mathcal{O}(1)$ rounds with $\mathcal{O}(b(n))$ words sent when broadcasting inputs of size $\mathcal{O}(n)$ words. 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.

We also assume a packed AVSS protocol with a constant number of rounds in both share and reconstruct protocols which requires $\mathcal{O}(s(n))$ words to be sent when a dealer shares $\mathcal{O}(n)$ values and $\mathcal{O}(r(n))$ words for reconstructing the sum of $\mathcal{O}(n)$ secrets. When assuming a resilience threshold of $t < \frac{n}{4}$, we can use the packed AVSS protocol described in [16]. This protocol requires $\mathcal{O}(n^4)$ words while sharing $\mathcal{O}(n)$ values by calling the protocol a constant number of times, each time sharing a constant fraction of the n secrets. We sum-reconstruct secrets by simply reconstructing each secret individually and summing the results. This results in $\mathcal{O}(n^3)$ words sent while reconstructing each sum.

Using the values $b(n) = n^2 \log n$, $s(n) = n^4$ and $r(n) = n^3$, the theorem below results in a complexity of $\mathcal{O}(n^5)$.

Theorem 6. *The total number of words sent in the* VLE *protocol is* $\mathcal{O}(n \cdot (b(n) + s(n) + r(n)))$ *and all parties terminate after* $\mathcal{O}(1)$ *rounds.*

Proof. Each party starts by sharing n values, requiring $\mathcal{O}(ns(n))$ sent words. Each party broadcasts a constant number of messages of size $\mathcal{O}(n)$ resulting in $\mathcal{O}(nb(n))$ sent words. The parties then run the Gather protocol in which $\mathcal{O}(nb(n))$ more words are sent. Following that, parties reconstruct $\mathcal{O}(n)$ sums of secrets, requiring a final $\mathcal{O}(nr(n))$ words. In total, parties send $\mathcal{O}(n(b(n) + s(n) + r(n)))$ words. Each call to the broadcast, share, reconstruct or gather protocols terminates after $\mathcal{O}(1)$ rounds, requiring a constant number of rounds overall.

5 Asynchronously Validated Asynchronous Byzantine Agreement

This section deals with constructing our AVABA protocol which is built upon ideas in [3] and [5] and adapts them to the asynchronous information theoretic

setting. The protocol of [3] heavily relies on cryptographic primitives (signatures) to obtain externally valid outputs. Here we use the framework of asynchronous validity predicates to replace external validity with an information theoretic counterpart. This requires redefining and adopting new information theoretic variants of verifiable gather (party gather) and verifiable leader election. The protocol of [5] modifies the cryptographic protocol of [22] 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". In each view, parties propose values to agree upon and then try to choose an honest leader using the VLE protocol. Our VLE protocol has α-Quality for $\alpha = \frac{1}{3}$, so this event should take place with probability $\frac{1}{3}$ or greater. Once this happens, the AVABA protocol guarantees that all parties will terminate with the proposal suggested by that honest party.

AVABA uses the "Key-Lock-Commit" paradigm used in previous HotStuff protocols (VABA, IT-HS and NWH) in order maintain safety and liveness. As explained in [3]:

> **Key:** Parties set a local key field that indicates that no other value was committed to in previous rounds. The keys help maintain liveness: if at any point some party sets a lock (to be explained later) in a view where no commitment takes place, then they will eventually see a key from that view (or a later view), that will convince them to participate in the current view. A key consists of two values: key, which is a view number and key_val which is the suggested value.
> **Lock:** Before committing to a value in a given view, parties will wait to hear that enough other parties have set a lock on the same value in that view. Before parties set a lock in a given view, they make sure that enough other parties have set a local key field that indicates that no other value was committed to in previous rounds. Parties that are locked on a value won't be willing to participate in a later view in which another value was suggested, unless key from a later view is provided. This mechanism helps in guaranteeing the safety of decision values. If a commitment took place, then there will be a large number of honest parties that are locked on that value. Those parties won't be willing to participate in views with different values, which will prevent any party from setting a key in a later view with a different value. This in turn will guarantee that no party will be able to provide erroneous proof that the locks can be opened.
> A lock consists of two values: lock, which is a view number, and lock_val which is the value seen when setting the lock.
> **Commit:** If an honest party commits to a value no other honest party ever commits to another value, using the locking mechanism. Before terminating, parties make sure that every party will hear a large number of commit messages, which will make sure they can also commit and terminate.

The parties proceed in 5 rounds in each view. The general idea is that parties will first confirm that they all agree on the leader elected in the VLE protocol, set

a lock to elected leader's proposal and confirm that they are all locked, commit to the lock and terminate. If at any point they see that the VLE failed, then they move onto a new view and announce that they are doing so (with proof). In the NWH protocol, parties provided cryptographic proofs for their keys and locks in the form of signatures on "echo" and "key" messages respectively. These signatures are inherently transferable since they can be sent to any party which can verify those signatures on their own. In order 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" and "echo" messages, which are used to inform parties of a failed VLE session.

In more detail:

Round 1: The first round in each view begins with a viewChange protocol. In the viewChange protocol parties choose their proposals and broadcast them to all parties. 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, using $\mathsf{leaderCorrect}_i$ as an asynchronous validity predicate. This guarantees that any chosen leader has already broadcasted a proposal.

Round 2: In the second round, parties check whether the VLE was successful or not. If it was they continue in the view, but if it wasn't 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 then echo that message to all other parties and include a proof that the this is the leader elected in the VLE protocol.
- If the leader's proposed value is incorrect, send a "blame" message and a proof that the this is the leader elected in the VLE protocol and that its proposed value is incorrect and proceed to the next view. In this context, by an incorrect proposal we mean that its key wasn't high enough to open the receiving party's current lock. Every party can check that the purported lock could have been set in a later view by waiting to receive the same broadcasted "key" messages required to set a key.

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

  Upon receiving an "equivocation" message with different values and correct proofs, forward that message, and proceed 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 an "equivocation" message. This also means that no other value was committed to in an earlier view, meaning that a key can be formed. Upon receiving $n-t$ "echo" messages, update the key and key_val fields before sending a "key" message to all parties.
Round 4: Upon receiving $n-t$ "key" messages, update the lock and lock_val fields before sending a "lock" message to all parties. 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. Such a message is sent after having received "lock" messages from $n-t$ parties, guaranteeing that $t+1$ parties have set their lock in the current view. These parties will not be willing to 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 [11], 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.

Algorithm 6 AVABA(x_i)

$\mathsf{key}_i \leftarrow 0, \mathsf{key\_val}_i \leftarrow x_i$
$\mathsf{lock}_i \leftarrow 0, \mathsf{lock\_val}_i \leftarrow \perp$
$\forall v \in \mathbb{N}\ \mathsf{proposals}_i \leftarrow \emptyset, \mathsf{echoes}_{i,v} \leftarrow \emptyset, \mathsf{keys}_{i,v} \leftarrow \emptyset, \mathsf{locks}_{i,v} \leftarrow \emptyset$
$\mathsf{view}_i \leftarrow 1$
continually run checkTermination()
while true **do**
    cur_view $\leftarrow \mathsf{view}_i$
    as long as cur_view $= \mathsf{view}_i$, **run**
        delay any message from any view v such that $v > \mathsf{view}_i$
        call viewChange(view_i) and continually run the **upon** commands within it
        continually run processMessages(view_i) and processFaults(view_i)
        continue updating sets and participating in broadcasts from older views, but do not send news messages or broadcasts or update $\mathsf{key}_i, \mathsf{key\_val}_i, \mathsf{lock}_i, \mathsf{lock\_val}_i$ in previous views

5.1 Security Analysis

In this section we will show that AVABA is an Asynchronously Validated Asynchronous Byzantine Agreement protocol in Theorem 7. We start by proving

Algorithm 7 processMessages(view)

upon $\mathsf{VLE}_{i,\mathsf{view}}$ outputting ℓ, π, **do** ▷ continue updating state according to $\mathsf{VLE}_{i,\mathsf{view}}$
  let (k, v) be a tuple such that $(\ell, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$
  if $k \geq \mathsf{lock}_i$ **then**
    broadcast $\langle$"echo", $k, v, \ell, \pi, \mathsf{view}\rangle$
  else
    send $\langle$"blame", $k, v, \ell, \pi, \mathsf{lock}_i, \mathsf{lock\_val}_i, \mathsf{view}\rangle$ to every party j
    $\mathsf{view}_i \leftarrow \mathsf{view}_i + 1$
upon receiving an $\langle$"echo", $k, v, \ell, \pi, \mathsf{view}\rangle$ broadcast from j, **do**
  upon $\mathsf{VLEVerify}_{i,\mathsf{view}}(\ell, \pi)$ terminating with the output 1, **do**
    if $(\ell, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$ **then**
      $\mathsf{echoes}_{i,\mathsf{view}} \leftarrow \mathsf{echoes}_{i,\mathsf{view}} \cup \{(j, k, v, \ell, \pi)\}$
      if $\exists (j', k', v', \ell', \pi') \in \mathsf{echoes}_i$ *s.t.* $(k, v) \neq (k', v')$ **then**
        send $\langle$"equivocation", $k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view}\rangle$ to every party j
        $\mathsf{view}_i \leftarrow \mathsf{view}_i + 1$
      else if $|\mathsf{echoes}_{i,\mathsf{view}}| = n - t$ **then**
        $\mathsf{key}_i \leftarrow \mathsf{view}, \mathsf{key\_val}_i \leftarrow v$
        broadcast $\langle$"key", $v, \mathsf{view}\rangle$
upon receiving a $\langle$"key", $v, \mathsf{view}\rangle$ broadcast from j, **do**
  upon $\mathsf{keyCorrect}_{i,\mathsf{view}+1}(\mathsf{view}, v)$ terminating with the output 1, **do**
    $\mathsf{keys}_{i,\mathsf{view}} \leftarrow \mathsf{keys}_{i,\mathsf{view}} \cup \{(j, v)\}$
    if $\left|\mathsf{keys}_{i,\mathsf{view}}\right| = n - t$ **then**
      $\mathsf{lock}_i \leftarrow \mathsf{view}, \mathsf{lock\_val}_i \leftarrow v$
      send $\langle$"lock", $v, \mathsf{view}\rangle$ to every party j
upon receiving the first $\langle$"lock", $v, \mathsf{view}\rangle$ message from j, **do**
  upon $\mathsf{lockCorrect}_i(\mathsf{view}, v)$ terminating with the output 1, **do**
    $\mathsf{locks}_{i,\mathsf{view}} \leftarrow \mathsf{locks}_{i,\mathsf{view}} \cup \{(j, v)\}$
    if $|\mathsf{locks}_{i,\mathsf{view}}| = n - t$ **then**
      send $\langle$"commit", $v\rangle$ to every party j

Algorithm 8 $\mathsf{leaderCorrect}_{i,\mathsf{view}}(\ell)$

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

Algorithm 9 $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$

if $\mathsf{view} > k$ **then**
  upon $\mathsf{validate}_i(v)$ terminating with the output 1, **do**
    if $k = 0$ **then**
      output 1 and **terminate**
    else
      upon $|\{j | \exists k', \ell, \pi \ s.t. \ (j, k', v, \ell, \pi) \in \mathsf{echoes}_{i,k}\}| \geq n - t$, **do**
        output 1 and **terminate**

Algorithm 10 $\mathsf{lockCorrect}_i(k, v)$

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

Algorithm 11 $\mathsf{checkTermination}()$

upon receiving a $\langle$"commit", $v\rangle$ message with the same value v from $t + 1$ parties, **do**
  send $\langle$"commit", $v\rangle$ to every party j if no such message has been previously sent
upon receiving a $\langle$"commit", $v\rangle$ message with the same value v from $n - t$ parties, **do**
  output v from the AVABA protocol and **terminate** AVABA

Algorithm 12 $\mathsf{viewChange}(\mathsf{view})$

$\mathsf{suggestions} \leftarrow \emptyset$ ▷ suggestions is a multiset
send $\langle$"suggest", $\mathsf{key}_i, \mathsf{key\_val}_i, \mathsf{view}\rangle$ to every party j
upon receiving the first $\langle$"suggest", $k, v, \mathsf{view}\rangle$ message from party j such that $k < view$, **do**
  upon $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$ terminating with the value 1, **do**
    $\mathsf{suggestions} \leftarrow \mathsf{suggestions} \cup \{(k, v)\}$
    if $|\mathsf{suggestions}| = n - t$ **then**
      $(k, v) \leftarrow argmax_{(k,v)\in\mathsf{suggestions}}\{k\}$ ▷ break ties arbitrarily
      if $k = 0$ **then**
        $(k, v) \leftarrow (0, x_i)$
      broadcast $\langle$"proposal", $k, v, \mathsf{view}\rangle$
upon receiving a $\langle$"proposal", $k, v, \mathsf{view}\rangle$ broadcast from j, **do**
  upon $\mathsf{keyCorrect}_{i,\mathsf{view}}((k, v))$ terminating with the output 1, **do**
    $\mathsf{proposals}_{i,\mathsf{view}} \leftarrow \mathsf{proposals}_{i,\mathsf{view}} \cup \{(j, (k, v))\}$
    if $j = i$ **then**
      call $\mathsf{VLE}_{i,\mathsf{view}}()$ with the validity predicate $\mathsf{leaderCorrect}_{i,\mathsf{view}}$

Algorithm 13 $\mathsf{processFaults}(\mathsf{view})$

upon receiving the first $\langle$"blame", $k, v, leader, \pi, l, lv, \mathsf{view}\rangle$ message from j, **do**
  upon $\mathsf{lockCorrect}_i(l, lv)$ and $\mathsf{VLEVerify}_{i,\mathsf{view}}(leader, \pi)$ terminating with the output 1, **do**
    if $k < l$ and $(leader, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$ **then**
      send $\langle$"blame", $leader, \pi, l, lv, \mathsf{view}\rangle$ to every party j
      $\mathsf{view}_i \leftarrow \mathsf{view}_i + 1$
upon receiving the first $\langle$"equivocation", $k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view}\rangle$ message from j, **do**
  upon $\mathsf{VLEVerify}_{i,\mathsf{view}}(\ell, \pi)$ and $\mathsf{VLEVerify}_{i,\mathsf{view}}(\ell', \pi')$ terminating with the output 1, **do**
    if $(\ell, (k, v)), (\ell', (k', v')) \in \mathsf{proposals}_{i,\mathsf{view}}$ and $(k, v) \neq (k', v')$ **then**
      send $\langle$"equivocation", $k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view}\rangle$ to every party j
      $\mathsf{view}_i \leftarrow \mathsf{view}_i + 1$

several lemmas. Lemma 5 and Lemma 6 are instrumental for showing the *safety* of the protocol. By that we mean that if some honest party outputs a value v, no other honest party outputs a differing value $v' \neq v$. The Correctness property of the protocol is then an immediate consequence of Lemma 11.

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

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

Definition 2. *A "key" message of the form* $\langle \text{"key"}, v, \mathsf{view} \rangle$ *is said to be correct if for some honest* i, $\mathsf{keyCorrect}_{i,\mathsf{view}'}(\mathsf{view}, v) = 1$ *holds for every* $\mathsf{view}' > \mathsf{view}$. *Similarly, a "lock" message of the form* $\langle \text{"lock"}, v, \mathsf{view} \rangle$ *is said to be correct if* $\mathsf{lockCorrect}_i(\mathsf{view}, v) = 1$ *for an honest* i. *In addition, the value of each such message is said to be the field* v.

As stated above, the following two lemmas are used in the proof that the protocol is safe. First, we show that in any given view only one value can proceed into the later rounds, meaning that any two values committed to in a single view must be the same. Following that, we show that if an honest party committed to a value, there are $t + 1$ honest parties that won't send "echo" messages for any other value in any subsequent view. This prevents any other value from being included in correct "key" or "lock" messages, thus preventing other values from being committed to in later views. This idea is explored more fully and proved in Lemma 11. The proofs for the lemmas are provided in Appendix D.

Lemma 5. *If two messages from a given* view *are correct, they both have the same value* v.

Lemma 6. *If an honest party sends a* $\langle \text{"commit"}, v \rangle$ *message in line 28 of* processMessages(view)*, then for any* $\mathsf{view}' \geq \mathsf{view}$ *there exist* $t + 1$ *honest parties that never send an* $\langle \text{"echo"}, k', v', \ell', \pi', \mathsf{view}' \rangle$ *message with* $v' \neq v$.

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

Definition 3. *An honest party* i *is said to reach a* view *if at any point its local* view_i *field equals* view. *Similarly, an honest party* i *is said to be in* view *if its local* view_i *field equals* view *at that time.*

We will start by showing that the methods used for validating leaders, keys and locks are asynchronous validity predicates and that keys and lock are always correct according to the party holding them. This means that parties can use the VLE protocol with leaderCorrect as an asynchronous validity predicate. In addition, this means that every honest party will be convinced of the correctness

of other parties' keys and locks, allowing them to progress through views in the case that "blame" messages are sent. The proofs of the lemmas are provided in Appendix E

Lemma 7. *Let* $\mathsf{keyCorrect}_{\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{\mathsf{view}}$ *be the predicates defined by* $\mathsf{keyCorrect}_{i,\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{i,\mathsf{view}}$ *for every* i *respectively.* $\mathsf{keyCorrect}_{\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{\mathsf{view}}$ *are asynchronous validity predicates for every* view*. Furthermore, for any* $\mathsf{view} > \mathsf{key}_i$*,* $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i) = 1$ *at any point in time.*

Lemma 8. $\mathsf{lockCorrect}$ *is an asynchronous validity predicate. Furthermore, at any point in time* $\mathsf{lockCorrect}_{i,\mathsf{view}}(\mathsf{lock}_i, \mathsf{lock\_val}_i) = 1$.

The next lemmas show that progress is made. We start in Lemma 9 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 10 then shows that if an honest party is chosen as the unique verifiable leader using the VLE protocol, the adversary cannot convince any honest party to proceed to the next view using a "blame" message. We then show in Lemma 11 that if some honest party terminates, every honest party does so as well. Finally, Lemma 12 shows that there is a constant probability of all parties terminating in any given view, using the fact that there is a $\frac{1}{3}$ probability that an honest party is elected in that view. The proofs of the following lemmas are straightforward, and mostly consist of showing that parties eventually send the required messages and reach agreement. The lemmas are stated here, but proved in Appendix F.

Lemma 9. *If every honest party* i *has an input* x_i *such that* $\mathsf{validate}_i(x_i) = 1$ *at the time it calls* AVABA*, all honest parties participate in the protocol, and no honest party terminates during any* view' *such that* $\mathsf{view}' < \mathsf{view}$*, then all honest parties reach* view.

Lemma 10. *If an honest* j *broadcasts a* $\langle \text{"proposal"}, k, v, \mathsf{view} \rangle$ *message, then no honest party sends a* $\langle \text{"blame"}, k, v, j, \pi, l, lv, \mathsf{view} \rangle$ *message for any* π, l, lv.

Lemma 11. *If some honest party outputs* v *and terminates, then all honest parties eventually do so as well.*

Lemma 12. *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.

As in the description of the VLE protocol, the resilience threshold of $t < \frac{n}{4}$ 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.

Theorem 7. *Protocol* AVABA *is a Validated Asynchronous Byzantine Agreement protocol resilient to* $t < \frac{n}{4}$ *Byzantine parties.*

Proof. Each property is proven individually.

Correctness. This property was proven in Lemma 11.

Validity. Before some honest i outputs a value v, it sends $\langle$"commit", v, view$\rangle$ message. As discussed in the proof of Lemma 11, at least $n - t$ parties sent "key" messages in view with the value v as well. At least one of those parties is honest. Party i only sends a $\langle$"key", v, view$\rangle$ message after receiving an $\langle$"echo", k, v, ℓ, π, view$\rangle$ message such that $\mathsf{VLEVerify}_{i,view}(\ell, \pi)$ terminates and $(\ell, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$. Party i adds a tuple $(\ell, (k, v))$ to $\mathsf{proposals}_{i,\mathsf{view}}$ after receiving a $\langle$"proposal", k, v, view$\rangle$ from ℓ and having $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v) = 1$. Before $\mathsf{keyCorrect}_{i,\mathsf{view}}(k, v)$ terminates with the output 1, i sees that $\mathsf{validate}_i(v) = 1$. Therefore, $\mathsf{validate}_i(v) = 1$ at that time.

Termination. If at any point an honest party terminates, from Lemma 11, all honest parties do so as well. Now assume that every honest party i has an input x_i such that $\mathsf{validate}_i(x_i) = 1$ at the time it calls AVABA and that all honest parties participate in the protocol. Observe some view, and assume no honest party terminated during view$'$ for any view$' <$ view. In that case, from Lemma 9 all honest parties eventually reach view. Then, from Lemma 12, with constant probability all honest parties terminate during view. In order for an honest party not to terminate by view, that constant probability event must not have happened in each one of the previous views. The honest parties run the VLE protocol with independent randomness in each view and thus for any adversary's strategy, there is an independent constant probability of terminating in each view. Therefore, the probability of reaching a given view decreases exponentially with the view number and thus approaches 0 as view grows. In other words, all honest parties almost-surely terminate.

Quality. Assume some honest party i completed the protocol, otherwise the claim holds trivially. This means that it at least completed the VLE protocol in view $= 1$. From the α-Binding property of VLE, with probability α or greater the binding value is the index ℓ^* of some party that behaved in an honest manner when starting VLE. From the Completeness and Validity properties of VLE, at that time $\mathsf{leaderCorrect}_{i,\mathsf{view}}(\ell^*) = 1$ and thus there exists some tuple $(\ell^*, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$ at that time, which was added after receiving a $\langle$"proposal", $k, , v$, view$\rangle$ broadcast from ℓ^*. Using the same arguments as the ones made in Lemma 12, in that case no honest party sends a "blame" or an "equivocation" message during view. Then, following similar logic to the one in Lemma 12, every honest party that hasn't committed due to a message from an earlier view eventually terminates after sending a "commit" message with the value v proposed by party i. No party can commit due to a message from an earlier view because there is no earlier view. Therefore, every honest party that participates in view and outputs a value from VLE, terminates and outputs the value v that ℓ^* proposed. Before sending its proposal, ℓ^* sees that $|\mathsf{suggestions}| = n - t$. ℓ^* only adds a tuple to suggestions after receiving the first $\langle$"suggest", k, v, view$\rangle$ message from each party. Each of those tuples must have

$k < \mathsf{view} = 1$ because $\mathsf{keyCorrect}_{i,1}(k, v) = 1$. At that time no honest party updated its key_j and $\mathsf{key\_val}_j$ fields, so they send messages with $k = 0$. Since at least one of the $n - t$ messages was sent by an honest party, there exists some $(k, v) \in \mathsf{suggestions}$ such that $k = 0$, and as shown above there is no such tuple with $k > 0$. Therefore, when computing choosing the tuple (k, v), i sees that the tuple with maximal k in $\mathsf{suggestions}$ has $k = 0$. Party i then sets $(k, v) = (0, x_i,)$, with x_i being its input to the AVABA protocol. As shown above, with constant probability all honest parties that start view output x_i, completing the proof.

5.2 Efficiency

We set m to be the size of inputs to the protocol and we use the same broadcast and packed AVSS protocols as described in the previous efficiency sections. Similarly to above, define $\mathcal{O}(b(m))$ to be the number of words sent when broadcasting messages with $\mathcal{O}(m)$ values, and $\mathcal{O}(s(n)), \mathcal{O}(r(n))$ to be the number of words sent while sharing n secrets and reconstructing the sum of secrets $\mathcal{O}(n)$ secrets respectively. Using the same protocols as above we get $b(m) = n^2 \log n + n \cdot m, s(n) = n^4, r(n) = n^3$. Using these values in the theorem below, we get an Asynchronously Validated Asynchronous Byzantine Agreement protocol with an efficiency of $\mathcal{O}(n^5 + n^2 \cdot m)$.

Theorem 8. *The expected total number of words sent in the* AVABA *protocol is* $\mathcal{O}(n \cdot (b(n+m) + s(n) + r(n) + n^2 + n \cdot m))$ *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, totalling in $\mathcal{O}(n^3+n^2 \cdot m)$ words. In addition, each party broadcasts messages of size $\mathcal{O}(n+m)$, totalling in $\mathcal{O}(nb(n+m))$ additional sent words. Finally, each party calls VLE once in each view, adding $\mathcal{O}(n \cdot (b(n)+s(n)+r(n)))$ total words. Summing all of these terms gives the result of $\mathcal{O}(n \cdot (b(n+m) + s(n) + r(n) + n^2 + n \cdot m))$ total words 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 $\mathcal{O}(n \cdot (b(n+m)+s(n)+r(n)+n^2+n \cdot m))$ words to be sent in expectation overall.

6 Agreement on a Core Set

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

A functionality for Agreement on a Core Set, using the above protocol, is also provided in Appendix G

Algorithm 14 $\mathsf{ACS}_i()$

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

Algorithm 15 $\mathsf{ACSValidity}_i(S)$

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

6.1 Security Analysis

We will start by showing that ACSValidity is indeed an asynchronous validity predicate, and then show that ACS is a protocol for agreeing on a core set.

Lemma 13. ACSValidity *is an asynchronous validity predicate.*

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

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

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

Theorem 9. ACS *is an Agreement on a Core Set protocol resilient to* $t < \frac{n}{3}$ *Byzantine parties.*

Proof. We will prove each property independently.

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

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

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

Note that parties simply call the AVABA protocol with inputs of size $\mathcal{O}(n)$ without sending additional messages, and thus the ACS protocol has the same efficiency as the AVABA protocol, yielding an ACS protocol with $\mathcal{O}(n^5)$ communication complexity.

7 Acknowledgments

We would like to thank the anonymous reviewers of Crypto 2022 for pointing out that [9] claimed to solve parallel broadcast also in asynchrony and the folklore assumption that this implies they also solve Agreement on a Core Set in asynchrony. This caused us to be clearer that we believe [9] does not claim and cannot be used directly to solve Agreement on a Core Set in asynchrony.

References

1. Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. Asymptotically free broadcast in constant expected time via packed vss. In *Theory of Cryptography: 20th International Conference, TCC 2022, Chicago, IL, USA, November 7–10, 2022, Proceedings, Part I*, pages 384–414. Springer, 2023.
2. Ittai Abraham, Danny Dolev, and Gilad Stern. Revisiting asynchronous fault tolerant computation with optimal resilience. *Distributed Computing*, 35(4):333–355, 2022.
3. Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching consensus for asynchronous distributed key generation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 363–373, 2021.
4. Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, page 337–346, New York, NY, USA, jul 2019. ACM.

5. 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.
6. 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.
7. 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.
8. 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.
9. Michael Ben-Or and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Comput.*, 16(4):249–262, 2003.
10. 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.
11. Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.
12. Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 524–541, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
13. Ran Canetti. *Studies in secure multiparty computation and applications*. PhD thesis, Citeseer, 1996.
14. 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.
15. 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.
16. Ashish Choudhury and Arpita Patra. An efficient framework for unconditionally secure multiparty computation. *IEEE Transactions on Information Theory*, 2016.
17. Thomas Dinsdale-Young, Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. Afgjort: A partially synchronous finality layer for blockchains. In Clemente Galdi and Vladimir Kolesnikov, editors, *Security and Cryptography for Networks - 12th International Conference, SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings*, volume 12238 of *Lecture Notes in Computer Science*, pages 24–44. Springer, 2020.
18. Paul Neil Feldman. *Optimal algorithms for Byzantine agreement*. PhD thesis, Massachusetts Institute of Technology, 1988.
19. 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.

20. Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-ng: Fast asynchronous BFT consensus with throughput-oblivious latency. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1187–1201. ACM, 2022.
21. 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.
22. Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery.

A Definitions for Protocols Used in Our Constructions

A.1 Reliable Broadcast

A *Reliable Broadcast* 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.

A.2 Packed Asynchronous Verifiable Secret Sharing

A packed asynchronous verifiable secret sharing protocol (packed AVSS) over a finite field $\mathbb{F}$ consists of a pair of protocols $(\mathsf{Share}, \mathsf{Reconstruct})$ with a designated party acting as dealer. The dealer has inputs $s_1, \ldots, s_m \in \mathbb{F}$ to the Share protocol for some $m \in \mathbb{N}$, whereas the rest of the parties have no input. On the other hand, each party can call $\mathsf{Reconstruct}(k)$ for each $k \in [m]$, and output some value from the protocol. Honest parties only call the $\mathsf{Reconstruct}$ protocol after having completed the Share protocol. A packed AVSS protocol has the following properties:

- **Correctness.** Once the first honest party completes the Share protocol, there exist values $r_1, \ldots, r_m$ such that:
  - if the dealer is honest then $\forall k \in [m]$ $r_k = s_k$; and
  - if some honest party completes $\mathsf{Reconstruct}(k)$ for some $k \in [m]$, then its output is r_k.
- **Termination.** If all honest parties participate in the Share protocol, then:
  - if the dealer is honest, all honest parties complete the Share protocol; and
  - if some honest party completes the Share protocol, then all honest parties complete the share protocol.

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

In addition, we are interested in protocols that allow reconstructing sums of secrets. That is, packed AVSS schemes with an additional $\mathsf{Sum} - \mathsf{Reconstruct}$ protocol that takes an input $\mathsf{dealers} \subseteq [n]$ and an input $k \in \mathbb{N}$ and outputs

a value. Parties call $\mathsf{Sum}-\mathsf{Reconstruct}(k, \mathsf{dealers})$ only after having completed the Share protocol with i as dealer for every $i \in \mathsf{dealers}$. The $\mathsf{Sum}-\mathsf{Reconstruct}$ protocol has the following two properties:

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

Note that it is possible to trivially construct a packed AVSS protocol from an AVSS protocol (i.e. a protocol with $m = 1$) by simply sharing each value independently. However, some protocols allow to share several values more efficiently than sharing them independently. In addition, it is possible to reconstruct the sum $\sum_{i \in \mathsf{dealers}} r_{i,k}$ by simply reconstructing each of the $r_{i,k}$ values individually and then summing the outputs. However, some protocols allow for more efficient reconstruction of sums as well.

B Proofs of Lemmas for the Gather Protocol

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

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

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

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

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

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

Lemma 2. *Let i, j be two honest parties. Observe the sets T_i, U_i at any time throughout the protocol. Eventually $T_i \subseteq T_j$ and $U_i \subseteq U_j$.*

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

C Proofs of Lemmas for the VLE Protocol

Lemma 3. *Let i and j be honest parties. Observe the sets* $\mathsf{dealers}_i$,$\mathsf{attached}_i$, *and* ranks_i *at any time throughout the protocol. Eventually* $\mathsf{dealers}_i \subseteq \mathsf{dealers}_j$, $\mathsf{attached}_i \subseteq \mathsf{attached}_j$ *and* $\mathsf{ranks}_i \subseteq \mathsf{ranks}_j$.

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

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

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

Lemma 4. checkValidity *is an asynchronous validity predicate.*

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

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

Consistency. Assume that $\mathsf{checkValidity}_i(k) = 1$ for some honest k. Since $\mathsf{checkValidity}_i(k) = 1$, i saw that there exists a tuple of the form $(k, \mathsf{dealers}_k)$ in $\mathsf{attached}_i$. From Lemma 3, eventually $(k, \mathsf{dealers}_k)$ will be added to $\mathsf{attached}_j$ as well, and $\mathsf{checkValidity}_j(k)$ will terminate with the output 1.

D Proofs of **AVABA** Safety

Lemma 5. *If two messages from a given* view *are correct, they both have the same value v.*

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

Now observe a correct "lock" message $\langle$"lock", v', view$\rangle$. Similarly to the case above, for some honest i, $\mathsf{lockCorrect}_i(\mathsf{view}, v') = 1$ with $\mathsf{view} > 0$, so $\left|\{j | (j, v') \in \mathsf{keys}_{i,\mathsf{view}}\right| \geq n - t$. Following similar logic to above, this means that i received correct $\langle$"key", v', 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 6. *If an honest party sends a $\langle$"commit", $v\rangle$ message in line 28 of* processMessages(view)*, then for any* $\mathsf{view}' \geq \mathsf{view}$ *there exist $t + 1$ honest parties that never send an $\langle$"echo", k', v', ℓ', π', $\mathsf{view}'\rangle$ message with $v' \neq v$.*

Proof. We will prove inductively that for any $\mathsf{view}' \geq \mathsf{view}$, there must exist $t + 1$ such honest parties. First observe $\mathsf{view}' = \mathsf{view}$. Since some honest i sends a $\langle$"commit", $v\rangle$ in line 28, it added $n - t$ tuples (j, v) to $\mathsf{locks}_{i,\mathsf{view}}$ and saw that $|\mathsf{locks}_{i,\mathsf{view}}| = n - t$. An honest i only does so after receiving $\langle$"lock", v, view$\rangle$ messages from $n - t$ parties and seeing that $\mathsf{lockCorrect}_i(\mathsf{view}, v) = 1$. Out of those parties, at least $t + 1$ were honest, and they sent their $\langle$"lock", v, view$\rangle$ broadcast after seeing that $\left|\mathsf{keys}_{j,\mathsf{view}}\right| \geq n - t$. Following similar logic, they

received $n-t$ $\langle$"key", v, view$\rangle$ messages and saw that they are correct. At least one of those messages was sent by an honest i that added $n-t$ tuples of the form (j,k,v,ℓ,π) to its echoes set after receiving $\langle$"echo", k,v,ℓ,π, view$\rangle$ broadcasts from $n-t$ parties. Note that i sent a "key" message, so it did not change view before sending the message, meaning that it did not send an "equivocation" message at that time and thus at that time there were every tuple (j,k,v,ℓ,π) in $\mathsf{echoes}_{i,\mathsf{view}}$ had the same k and v. In other words, it sent its $\langle$"key", v, view$\rangle$ message after receiving an "echo" broadcast with the same value v from $n-t$ parties. Out of those parties, at least $t+1$ are honest and they only send one "echo" broadcast per view. From Lemma 5, all correct "key" and "lock" messages from view had the same value v, and thus the "commit" message had the same value as well.

Assume the claim holds for every view'' such that $\mathsf{view}' > \mathsf{view}'' \geq \mathsf{view}$. As shown above, there are at least $t+1$ honest parties that send $\langle$"lock", v, view$\rangle$ broadcasts. Every honest party j only sends such a message after setting its lock_j field to view. Let the set of those honest parties be I. It is important to note that the field lock_j only grows throughout the protocol, so every one of the parties j such that $j \in I$ has $\mathsf{lock}_j \geq \mathsf{view}$ from that point on. Now assume by way of contradiction that some party $j \in I$ sent an $\langle$"echo", $k',v',\ell',\pi',\mathsf{view}'\rangle$ message with $v' \neq v$. Before doing that, it output ℓ',π' from $\mathsf{VLE}_{i,\mathsf{view}'}$. From the Completeness and Validity properties of the VLE protocol, $\mathsf{leaderCorrect}_{i,\mathsf{view}'}(\ell',\pi)=1$ at that time, so there was a tuple $(\ell',(k',v')) \in \mathsf{proposals}_{i,\mathsf{view}'}$, and $k' \geq \mathsf{lock}_j \geq \mathsf{view}$ because i did not send a "blame" broadcast. Party i adds such a tuple after receiving a $\langle$"proposal", $k',v',\mathsf{view}'\rangle$ from ℓ' and seeing that $\mathsf{keyCorrect}_{i,\mathsf{view}'}((k',v'))=1$, so $\mathsf{view}' > k'$ and $|\{j' | \exists k'',\ell',\pi''\ s.t.\ (j',k'',v',\ell',\pi'') \in \mathsf{echoes}_{j,k'}\}| \geq n-t$. As discussed above, each honest party only adds a tuple (j',k'',v',ℓ',π'') to $\mathsf{echoes}_{j,k'}$ after receiving an "echo" message with the value v' from j'. However, $\mathsf{view}' > k' \geq \mathsf{view}$, so by assumption there exist $t+1$ parties that never send such a message in view k'. Any set of $n-t$ parties that sent the "echo" broadcasts must have at least one party in common with the parties in I, reaching a contradiction.

E Proofs of the Correctness of AVABA' Asynchronous Validity Predicates

Lemma 7. *Let* $\mathsf{keyCorrect}_{\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{\mathsf{view}}$ *be the predicates defined by* $\mathsf{keyCorrect}_{i,\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{i,\mathsf{view}}$ *for every* i *respectively.* $\mathsf{keyCorrect}_{\mathsf{view}}$ *and* $\mathsf{leaderCorrect}_{\mathsf{view}}$ *are asynchronous validity predicates for every* view*. Furthermore, for any* $\mathsf{view} > \mathsf{key}_i$*,* $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i) = 1$ *at any point in time.*

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

Finality. The first things that i does in $\mathsf{keyCorrect}_i(k,v)$ are checking that $\mathsf{view} > k$ and that $\mathsf{validate}_i(v)=1$. From the Finality property of validate,

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

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

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

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

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

Lemma 8. lockCorrect *is an asynchronous validity predicate. Furthermore, at any point in time* $\mathsf{lockCorrect}_{i,\mathsf{view}}(\mathsf{lock}_i, \mathsf{lock\_val}_i) = 1$.

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

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

Correctness. If $k = 0$, then j immediately outputs 1 from $\mathsf{lockCorrect}_j(k, v)$ as well. Otherwise, $k > 0$. Since i output 1 from $\mathsf{lockCorrect}_i(k, v)$, it saw that $\left|\{j|(j, v) \in \mathsf{keys}_{i,k}\}\right| \geq n - t$. i only adds a tuple (j, v) to its $\mathsf{keys}_{i,k}$ sets after receiving a $\langle$"key", $v, k\rangle$ broadcast from j and seeing that $\mathsf{keyCorrect}_{i,k+1}(k, v) = 1$. From Lemma 7, $\mathsf{keyCorrect}_{k+1}$ is an asynchronous validity predicate, so eventually $\mathsf{keyCorrect}_{j,k+1}(k, v) = 1$ as well. At that point, j will add the same tuple (j, v) to $\mathsf{keys}_{j,k}$. After adding all of those tuples, j will see that the same condition holds and return 1 from $\mathsf{lockCorrect}_j(k, v)$.

Finally, we will show that $\mathsf{lockCorrect}_i(\mathsf{lock}_i, \mathsf{lock\_val}_i) = 1$ at any point in time. If i did not update those fields, then $\mathsf{lock}_i = 0, \mathsf{lock\_val}_i = \perp$. In that case, when running $\mathsf{lockCorrect}_i$, i will immediately see that $\mathsf{lock}_i = 0$ and output 1. Otherwise, i updated its lock_i and $\mathsf{lock\_val}_i$ fields after adding a tuple (j, v) to $\mathsf{keys}_{i,\mathsf{view}}$ and seeing that $\left|\mathsf{keys}_{i,\mathsf{view}}\right| = n - t$. This happens after receiving a $\langle$"key", $v, \mathsf{view}\rangle$ broadcast from j and seeing that $\mathsf{keyCorrect}_{i,\mathsf{view}+1}(\mathsf{view}, v) = 1$. From Lemma 5, 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 $\mathsf{lockCorrect}_i(\mathsf{view}, v) = 1$.

F Proofs of AVABA Liveness

Lemma 9. *If every honest party i has an input x_i such that* $\mathsf{validate}_i(x_i) = 1$ *at the time it calls* AVABA*, all honest parties participate in the protocol, and no*

honest party terminates during any view' *such that* $\mathsf{view}' < \mathsf{view}$*, then all honest parties reach* view*.*

Proof. We will prove the claim inductively on view. First, all honest parties start in $\mathsf{view} = 1$. Now observe some $\mathsf{view} > 1$ and assume no honest party terminated in any $\mathsf{view}' < \mathsf{view}$, and that they all reached $\mathsf{view} - 1$. Since they reached $\mathsf{view} - 1$, they started off broadcasting "suggest" messages with their current $\mathsf{key}, \mathsf{key\_val}$ fields. An honest i only update its key_i field to the view it is currently in, and thus in the beginning of $\mathsf{view} - 1$, $\mathsf{key}_i < \mathsf{view} - 1$. From Lemma 7, for every honest j, $\mathsf{keyCorrect}_{j,\mathsf{view}-1}(\mathsf{key}_j, \mathsf{key\_val}_j)$ at the time it sent those fields, and from the Consistency property of $\mathsf{keyCorrect}_{\mathsf{view}-1}$, eventually $\mathsf{keyCorrect}_{i,\mathsf{view}-1}(\mathsf{key}_j, \mathsf{key\_val}_j)$ terminates with the output 1 for every honest i. After receiving such a message from every honest j and seeing that the suggested $\mathsf{key}_j, \mathsf{key\_val}_j$ are correct, every honest i adds a tuple to $\mathsf{suggestions}$. After adding a tuple for each honest party, i broadcasts $\langle$"proposal"$, k, v, \mathsf{view}\rangle$ with (k, v) either being a tuple from $\mathsf{suggestions}$ or $(k, v) = (0, x_i)$. Note that (k, v) is only added to $\mathsf{suggestions}$ after i sees that $\mathsf{keyCorrect}_{i,\mathsf{view}-1}(k, v) = 1$. In addition, 0 and x_i are the first values to which key_i and $\mathsf{key\_val}_i$ are set, so as argued in Lemma 7, $\mathsf{keyCorrect}_{i,\mathsf{view}-1}(0, x_i) = 1$ at that time. This means that when receiving its own broadcast, i adds $(i, (k, v))$ to $\mathsf{proposals}_{i,\mathsf{view}}$ and calls $\mathsf{VLE}_{i,\mathsf{view}}$ with the asynchronous validity predicate $\mathsf{leaderCorrect}_{i,\mathsf{view}-1}$. Since there is a tuple $(i, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$ at that time, $\mathsf{leaderCorrect}_{i,\mathsf{view}}(i) = 1$. From the Termination of Output property of VLE, every honest i outputs some index ℓ and a proof π from $\mathsf{VLE}_{i,\mathsf{view}-1}$.

First we will show that if some honest party sends an "equivocation" or a "blame" message, then the claim holds. If some honest party i sends a $\langle$"blame"$, k, v, \mathit{leader}, \pi, l, lv, \mathsf{view} - 1\rangle$ message, then it either did so in line 6 or in line 4. In the first case, it did so after outputting leader, π from $\mathsf{VLE}_{i,\mathsf{view}-1}$ and seeing that there is a tuple $(\mathit{leader}, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}}$ such that $k < \mathsf{lock}_i$. It then sent the "blame" message with $l = \mathsf{lock}_i, lv = \mathsf{lock\_val}_i$, and from Lemma 8, $\mathsf{lockCorrect}_i(\mathsf{lock}_i, \mathsf{lock\_val}_i) = 1$ at that time. Every honest j will eventually receive the message and see that $k < l$. Then, from the Consistency property of $\mathsf{lockCorrect}$ and from the Completeness property of $\mathsf{VLEVerify}$, j will see that $\mathsf{lockCorrect}_j(l, lv) = 1$ and that $\mathsf{VLEVerify}_{j,\mathsf{view}-1}(\mathit{leader}, \pi) = 1$. From the Validity property of $\mathsf{VLEVerify}$, there exists a tuple $(\mathit{leader}, (k', v')) \in \mathsf{proposals}_{j,\mathsf{view}-1}$. Both i and j added their respective $(\mathit{leader}, (k, v))$ and $(\mathit{leader}, (k', v'))$ after receiving the same broadcast, so $(k, v) = (k', v')$ and thus j will also proceed to the next view. Otherwise, i sent the message in line 4, after seeing that $k < l$ and having $\mathsf{lockCorrect}_i(l, lv) = 1$ and $\mathsf{VLEVerify}_{i,\mathsf{view}-1}(\mathit{leader}, \pi) = 1$. Similarly, from the Consistency property of $\mathsf{lockCorrect}$ and Agreement on Verification property of $\mathsf{VLEVerify}$, j will see that the same conditions hold. In addition, i saw that $(\mathit{leader}, (k, v)) \in \mathsf{proposals}_{i,\mathsf{view}-1}$, so j will see that the same holds and proceed to the next view.

Following similar arguments, i can either send an "equivocation" message in line 13 or in line 9. In the first case, it does so after having received two "echo" messages with values k, v, ℓ, π and k', v', ℓ', π' such that $(k, v) \neq (k', v')$, seeing

that $\mathsf{VLEVerify}_{i,\mathsf{view}-1}(\ell,\pi)$ and $\mathsf{VLEVerify}_{i,\mathsf{view}-1}(\ell',\pi')$ terminate with the output 1, and that $(\ell,(k,v)),(\ell',(k',v')) \in \mathsf{proposals}_{i,\mathsf{view}-1}$ and then sending the message. In the second case it received an $\langle$"equivocation", $k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view}-1\rangle$ message directly and saw that the same conditions hold, after which it forwarded the message. Every honest j will then receive the $\langle$"equivocation", $k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view}-1\rangle$ message sent by i and see that $(k,v) \neq (k',v')$. From the Agreement on Verification property of $\mathsf{VLEVerify}$, j will eventually see that $\mathsf{VLEVerify}_{j,\mathsf{view}-1}(\ell,\pi) = 1$ and that $\mathsf{VLEVerify}_{j,\mathsf{view}-1}(\ell',\pi') = 1$. From the Validity property of $\mathsf{VLEVerify}$, at that time $\mathsf{leaderCorrect}_{j,\mathsf{view}-1}(\ell) = 1$ and $\mathsf{leaderCorrect}_{j,\mathsf{view}-1}(\ell') = 1$. Therefore there are tuples of the form $(\ell,(k'',v''))$ and $(\ell',(k''',v'''))$ in $\mathsf{proposals}_{j,\mathsf{view}-1}$. Following the same logic as above, those are the same tuples that i added to $\mathsf{proposals}_{i,\mathsf{view}-1}$, so $(\ell,(k,v)),(\ell',(k',v')) \in \mathsf{proposals}_{j,\mathsf{view}-1}$, and thus j proceeds to the next view. In other words, if some honest party sends either a "blame" message or an "equivocation" in $\mathsf{view}-1$, and no honest party completes the protocol in this view, then all honest parties proceed to view.

Now assume no honest party sends a "blame" or an "equivocation" message in $\mathsf{view}-1$. In that case, after completing the call to $\mathsf{VLE}_{j,\mathsf{view}}$ with the output ℓ,π every honest j broadcasts an $\langle$"echo", $k, v, \ell, \pi, \mathsf{view}-1\rangle$ message since it did not send a "blame" message instead. Every honest i receives that message, and from the Completeness property of $\mathsf{VLEVerify}$ eventually sees that $\mathsf{VLEVerify}_{i,\mathsf{view}-1}(\ell,\pi) = 1$ and adds a tuple to $\mathsf{echoes}_{i,\mathsf{view}-1}$. By assumption, i does not send an "equivocation" in $\mathsf{view}-1$, so after adding such a tuple for each honest party, i has $|\mathsf{echoes}_{i,\mathsf{view}-1}| \geq n-t$, and thus i updates key_i to $\mathsf{view}-1$ and $\mathsf{key\_val}_i$ to v and broadcasts a $\langle$"key", $v, \mathsf{view}-1\rangle$ message during $\mathsf{view}-1$. From Lemma 7, at that time $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{view}-1,v) = 1$. Every honest j receives that message and from the Consistency property of $\mathsf{keyCorrect}_{\mathsf{view}}$, eventually sees that $\mathsf{keyCorrect}_{j,\mathsf{view}}(\mathsf{view}-1,v) = 1$ as well. After that, j adds a tuple to $\mathsf{keys}_{j,\mathsf{view}-1}$ for every honest party and sees that $\left|\mathsf{keys}_{j,\mathsf{view}-1}\right| \geq n-t$, so j sends a "lock" message to every party. Following identical reasoning, every honest i receives the "lock" message from every honest party, eventually sees that $\mathsf{lockCorrect}_i(\mathsf{view},v) = 1$ and updates its $\mathsf{locks}_{i,\mathsf{view}}$ set. After doing so for all honest party, it sends a "commit" message. From Lemma 5, all correct "lock" messages contain the same value, so all honest parties sent "commit" messages with the same value. Finally, after receiving those messages from all honest parties, every honest i sees that it received $n-t$ such messages and complete the AVABA protocol in line 4. In other words, every honest party completes the protocol, reaching a contradiction.

Lemma 10. *If an honest j broadcasts a $\langle$"proposal", $k, v, \mathsf{view}\rangle$ message, then no honest party sends a $\langle$"blame", $k, v, j, \pi, l, lv, \mathsf{view}\rangle$ message for any π, l, lv.*

Proof. Assume by way of contradiction some honest party i sends such a message. It either does so in line 6 or in line 4. In both cases, it first checked that $k < l$ and that $(j,(k,v)) \in \mathsf{proposals}_{i,\mathsf{view}}$. i adds such a tuple to $\mathsf{proposals}_{i,\mathsf{view}}$ after seeing that $\mathsf{keyCorrect}_{i,\mathsf{view}}((k,v)) = 1$. Since $\mathsf{keyCorrect}_{i,\mathsf{view}}(k,v) = 1$, either $k = 0$ or there exist at least $n-t$ tuples in $\mathsf{echoes}_{i,k}$ with $k > 0$ and thus

$k \geq 0$. In addition, if i sent the message in line 6, then $l = \mathsf{lock}_i, lv = \mathsf{lock\_val}_i$, and from Lemma 8, $\mathsf{lockCorrect}_i(l, lv) = 1$ at that time. If i sent the message in line 4, then it first checked that $\mathsf{lockCorrect}_i(l, lv) = 1$ at that time. It cannot be the case that $l = 0$, because then $k \geq l$, reaching a contradiction. Therefore, $\left|\{j' | (j', v) \in \mathsf{keys}_{i,l}\}\right| \geq n - t$. Each tuple (j', v) was added to $\mathsf{keys}_{i,l}$ after receiving a $\langle$"key", $v, l\rangle$ message from j'. At least $t + 1$ of those tuples were added after receiving a "key" message from honest parties. Note that an honest j' sends such a message after updating $\mathsf{key}_{j'}$ to l and $\mathsf{key\_val}_{j'}$ to v. Since the $\mathsf{key}_{j'}$ field only increases throughout the protocol, $\mathsf{key}_{j'} \geq l$ from this point on. Let I be the indices of the honest parties j' that sent those "key" messages, for whom it is guaranteed that $\mathsf{key}_{j'} \geq l$ from this point on. Now observe the pair (k, v) that i chose to input into $\mathsf{VLE}_{i,\mathsf{view}}$. At the time it chose (k, v), i had $|\mathsf{suggestions}| = n - t$, so it received $\langle$"suggest", $k', v', \mathsf{view}\rangle$ from $n - t$ parties and added corresponding tuples to $\mathsf{suggestions}$. As shown above, $|I| \geq t + 1$, so at least one of those messages was received from a party j' such that $j \in I$, for whom $k' = \mathsf{key}_j \geq l$. Party i chooses the tuple (k, v) to be the one with the maximal k in $\mathsf{suggestions}$. Therefore, $k \geq k' \geq l$, reaching a contradiction.

Lemma 11. *If some honest party outputs v and terminates, then all honest parties eventually do so as well.*

Proof. Assume some honest party output v and terminated. It first received $\langle$"commit", $v\rangle$ messages from $n - t$ parties, with $t + 1$ of them being honest. Let i be the first honest party that sent such a message. First we will show that no honest party sends a $\langle$"commit", $v'\rangle$ message with any other value $v' \neq v$. Assume by way of contradiction that some honest party sends such a message, and let j be the first honest party to send such a message. Since both i and j were the first honest parties to send such messages, at the time they sent the message they received "commit" messages from at most t parties. This means that both i and j sent their respective "commit" messages in line 28 at the end of view and view' respectively. Assume without loss of generality that $\mathsf{view} \leq \mathsf{view}'$. From Lemma 6, in view', there are $t+1$ honest parties that never send an "echo" message with any value $v' \neq v$. If some honest party sends a "key" message in view', then it does so after receiving $n - t$ "echo" messages with the same value (i.e. without detecting equivocation and proceeding to the next view). At least one of those messages was sent by the $t + 1$ honest parties described above, so any "key" message sent by an honest party in view' has the value v. For similar reasons, any "lock" message sent by an honest party in view' has the value v. Before sending a "commit" message, j receives $n - t$ correct "lock" messages and sent a "commit" message with the value v' of a received correct "lock" message. From Lemma 5, those messages had the value v, and thus $v = v'$, reaching a contradiction. Therefore, if two honest parties send "commit" messages, they send messages with the same value v.

We will now turn to show that if i completed the protocol with the output v, every honest party will do so as well. Since i completed the protocol, it received $\langle$"commit", $v\rangle$ messages from $n - t$ parties, with $t + 1$ of them being honest.

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

Lemma 12. *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 11 every honest party will do so as well. From this point on, we will not deal with the case that some of the parties terminate early in view and some do not terminate at all. The first thing that an honest party does in view is calling viewChange and sending a "suggest" message to every party with the local fields key_i and $\mathsf{key\_val}_i$. From Lemma 7, $\mathsf{keyCorrect}_{i,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i) = 1$ at that time, and from the Consistency of $\mathsf{keyCorrect}_{\mathsf{view}}$, for every honest j eventually $\mathsf{keyCorrect}_{j,\mathsf{view}}(\mathsf{key}_i, \mathsf{key\_val}_i) = 1$ as well. Therefore, when an honest party j receives that message, it eventually adds a tuple to suggestions. After receiving such a message from every honest party, j finds that $|\mathsf{suggestions}| \geq n-t$, and it broadcasts a $\langle$"proposal", $k, v, \mathsf{view}\rangle$ message. At that time it either has $(k, v) = (0, x_j)$ and as shown in Lemma 7 $\mathsf{keyCorrect}_{j,\mathsf{view}}((k, v)) = 1$, or it has chosen a tuple $(k, v) \in \mathsf{suggestions}$ for which it checked that $\mathsf{keyCorrect}_{j,\mathsf{view}}((k, v)) = 1$. Therefore, when receiving its own "proposal" broadcast, it adds a tuple $(j, (k, v))$ to $\mathsf{proposals}_{j,\mathsf{view}}$ and thus $\mathsf{leaderCorrect}_{j,\mathsf{view}}(j) = 1$ at that time. j then calls $\mathsf{VLE}_{j,\mathsf{view}}$.

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

If the binding value is indeed the index of a party that acted in an honest manner when it started VLE, then from the Binding Verification property of VLE

there is exactly one index ℓ^* for which it is possible that $\mathsf{VLEVerify}_{i,view}(\ell^*, \pi)$ terminates with the output 1 for an honest i. If an honest i adds a tuple $(j', k', v', \ell', \pi')$ to $\mathsf{echoes}_{i,\mathsf{view}}$, then it did so after receiving an "echo" message and seeing that $\mathsf{VLEVerify}_{i,\mathsf{view}}(\ell', \pi) = 1$ and that $(\ell', (k', v')) \in \mathsf{proposals}_{i,\mathsf{view}}$. Therefore, $\ell' = \ell^*$ for every such tuple. An honest i adds a tuple $(\ell^*, (k', v'))$ to $\mathsf{proposals}_{i,\mathsf{view}}$ after receiving a $\langle$"proposal", $k', v', \mathsf{view}\rangle$ broadcast from ℓ^*, and thus all tuples in the set $\mathsf{echoes}_{i,\mathsf{view}}$ have the same values k', v', which prevents an honest party from sending an "equivocation" message in line 13. In addition, no honest i sends an "equivocation" message in line 9 after receiving an $\langle$"equivocation", $k, v, \ell, \pi, k', v', \ell', \pi', \mathsf{view}\rangle$ message because $\ell = \ell' = \ell^*$, and thus $(k, v) = (k', v')$. We would now like to show that no honest party i sends a "blame" message in view. If an honest party sends a $\langle$"blame", $k, v, \ell, \pi, l, lv\rangle$ message, it does so either in line 6 or in line 4. In the first case, it did so after outputting ℓ, π from $\mathsf{VLE}_{i,\mathsf{view}}$ and from the Completeness and Binding Verification properties of VLE, $\ell = \ell^*$. In the second case, it checked that $\mathsf{VLEVerify}(\ell, \pi) = 1$ and for the same reasons $\ell = \ell^*$. Before starting VLE, ℓ^* broadcasts $\langle$"proposal", $k, v, \mathsf{view}\rangle$. For similar reasons as above, if an honest i has a tuple $(\ell^*, (k', v')) \in \mathsf{proposals}_{i,\mathsf{view}}$, then $(k', v') = (k, v)$. Therefore, i sends the message $\langle$"blame", $k, v, \ell^*, \pi, l, lv, \mathsf{view}\rangle$, contradicting Lemma 10.

Honest parties only proceed to $\mathsf{view}+1$ after sending either a "blame" or an "equivocation" message, so no honest party proceeds to $\mathsf{view}+1$. Since no honest i sends a "blame" message, each one sends an $\langle$"echo", $k, v, \ell^*, \pi, \mathsf{view}\rangle$ message after completing the $\mathsf{VLE}_{i,view}$ call. From the Completeness property of VLE, $\mathsf{VLEVerify}_{i,view}(\ell^*, \pi)$ eventually terminates with the output 1. Since i doesn't send an "equivocation" message in view, it then adds a tuple to $\mathsf{echoes}_{i,\mathsf{view}}$. After such a tuple is added for every honest party, i sees that $|\mathsf{echoes}_{i,\mathsf{view}}| = n - t$ and it sends a message $\langle$"key", $v, \mathsf{view}\rangle$ to all parties after updating key_j to view and $\mathsf{key\_val}_j$ to v. From Lemma 7, at that time $\mathsf{keyCorrect}_{i,\mathsf{view}+1}(\mathsf{view}, v) = 1$ so from the consistency property of $\mathsf{keyCorrect}_{\mathsf{view}+1}$ eventually $\mathsf{keyCorrect}_{j,\mathsf{view}+1}(\mathsf{view}, v) = 1$ for every honest j. Therefore, when receiving that message, every honest j eventually sees that the message is correct and adds a pair (i, v) to $\mathsf{keys}_{i,\mathsf{view}}$. After adding such a pair for every honest party, j has $\left|\mathsf{keys}_{i,\mathsf{view}}\right| = n - t$ and it sends a "lock" message. Using identical arguments, eventually every honest party sends a "commit" message. Finally, after receiving a "commit" from $n - t$ parties, every honest party terminates.

G Modelling ACS as a Functionality

The notion of an agreement on a core set can also be modelled as a functionality $\mathcal{F}_{\mathsf{ACS}}$. In the functionality, each party i receives (record, i, k) commands with $k \in [n]$ and sends them to the functionality. Each party is guaranteed to receive at least $n-t$ such commands and if some honest i receives a (record, i, k) command, every honest j is guaranteed to eventually receive a (record, j, k) command as well. In addition, parties can receive $\mathsf{receiveS}()$ commands which they forward to the functionality. The functionality then returns a set $S \subseteq [n]$ as a response.

The functionality is fully described in Algorithm 16 Note that in either case,

Algorithm 16 $\mathcal{F}_{\mathsf{ACS}}$ – Agreement on a Core Set Functionality

The functionality is parameterized by the set of corrupted parties $I \subseteq [n]$. Initialize sets $S_i \leftarrow \emptyset$ for every $i \in [n]$ and $S \leftarrow \perp$. In addition, initialize $\mathsf{returned} \leftarrow 0$.

1. (record, i, k): Upon receiving this command from party i, add the index k to S_i. Forwards (record, i, k) to the adversary. If $|S_i| \geq n - t$ then set i as ready. If $n - t$ honest parties are ready and $\mathsf{returned} = 0$, set S to be the set of all indices $k \in [n]$ such that there exists some $\ell \notin I$ for which $(\mathsf{record}, \ell, k)$ was sent.
2. (advSet, S'): Upon receiving this command from the adversary, check that $S' \subseteq [n]$ and that $|S'| \geq n - t$. Moreover, check that for every $k \in S'$, there exists some $\ell \notin I$ for which $k \in S_\ell$ (i.e., P_ℓ has submitted $(\mathsf{record}, \ell, k)$). If all those conditions hold, and $\mathsf{returned} = 0$, then store $S \leftarrow S'$ and store $\mathsf{returned} \leftarrow 1$.
3. $\mathsf{receiveS}()$: Upon receiving this command from some party i, if $S \neq \perp$, set $\mathsf{returned} \leftarrow 1$. Return S.

all parties eventually receive the same set of indices $S \subseteq [n]$, such that for every $k \in S$ at least one honest party received a (record, i, k) command.

The following protocol implements the $\mathcal{F}_{\mathsf{ACS}}$ functionality:

Algorithm 17 Π_{ACS} – protocol implementing $\mathcal{F}_{\mathsf{ACS}}$

Each party i initializes a set $S_i \leftarrow \emptyset$.

1. (record, i, k): Party i executes this command by adding k to S_i.
2. $\mathsf{receiveS}()$: If this is the first $\mathsf{receiveS}$ command executed by i, call ACS_i (See Algorithm 14) with the validity predicate $\mathsf{validate}_i$ that returns 1 on an index k once $k \in S_i$. Wait for ACS_i to output a set S and then return it. If i has already completed the call to ACS_i with some output S, return S as well.

Lemma 14. $\mathsf{validate}$ *is a validity predicate.*

Proof. Note that $\mathsf{validate}$ only returns the value 1, so if $\mathsf{validate}_i(S) = b$ for some honest party, $b = 1$.

Finality. If $\mathsf{validate}_i(k) = 1$ for some honest i then $k \in S_i$. Honest parties don't remove indices from their S_i sets, so $\mathsf{validate}_i(k)$ will return 1 in any subsequent call.

Consistency. If $\mathsf{validate}_i(k) = 1$ for some honest i, $k \in S_i$. Party i adds indices $k \in [n]$ to S_i after receiving a (record, i, k) command. By assumption, every honest j will eventually receive a (record, j, k) command as well for every $k \in S_i$. This means that eventually $k \subseteq S_j$ as well, and thus $\mathsf{validate}_j(k) = 1$ will eventually hold as well.

Theorem 10. Π_{ACS} *securely computes the* $\mathcal{F}_{\mathsf{ACS}}$ *functionality in the presence of an adversary controlling up to* t *parties.*

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

1. The simulator invokes $\mathcal{A}$.
2. The simulator delivers all (record, i, k) messages to the functionality immediately upon them being sent and then receives the messages forwarded by the functionality.
3. The simulator initializes sets $S_i = \emptyset$ for every $i \notin I$. Upon receiving a (record, i, k) command from the functionality, the simulator adds k to S_i.
4. Upon an honest i receiving a $\mathsf{receiveS}$ command, simulate i acting honestly in the ACS_i protocol with the validity predicate $\mathsf{validate}_i$ defined in the $\mathsf{receiveS}$ description of the Π_{ACS} protocol. Once some honest i outputs a set S from the simulated AVABA call, send a (advSet, S) command to the functionality.
5. Deliver any $\mathsf{receiveS}()$ command after having delivered a (advSet, S) command to the functionality and then deliver the functionality's response.

The simulator simulates a run of the ACS protocol with every honest party acting honestly when interacting with the adversary, with the same validity predicate as the one in Π_{ACS}. The simulator then sends and delivers a (advSet, S') command to the functionality after some honest i outputs S'. From the validity property of the ACS protocol, at the time i outputs S', it has $\mathsf{validate}_i(k) = 1$ for every $k \in S'$ and $S' \subseteq [n]$, $|S'| \geq n - t$. From the definition of $\mathsf{validate}_i$, for every $k \in S'$, $k \in S_i$ as well. This means that i sent a (record, i, k) command to the functionality for every $k \in S'$. The simulator didn't deliver any $\mathsf{receiveS}()$ command yet and thus $\mathsf{returned} = 0$, so the functionality will update S to be S'.

Since the simulator honestly simulates all honest parties when interacting with $\mathcal{A}$, the adversary's view is distributed identically to the way it would be distributed in a run of the Π_{ACS} protocol. Furthermore, the $\mathsf{receiveS}()$ commands sent by an honest j are received after the simulator delivers the (advSet, S') command to the functionality. Therefore, j receives S' and outputs it from $\mathsf{receiveS}()$. From the Agreement property of the ACS protocol, all honest parties output the same set from the ACS protocol. This means that j outputs the same set S' from the ACS protocol, which it would then return from the $\mathsf{receiveS}$ command of the Π_{ACS} protocol. In other words the joint distributions of the adversary's view and the output from the $\mathsf{receiveS}$ calls are identical in Π_{ACS} and in the simulated protocol.