ABEL: Perfect Asynchronous Byzantine Extension from List-Decoding

Ittai Abraham
Intel Labs, Petah Tikva, Israel

Gilad Asharov
Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel

Abstract

Asynchronous byzantine agreement extension studies the message complexity of L-bit multivalued asynchronous byzantine agreement given access to a binary asynchronous Byzantine agreement protocol.

We prove that asynchronous byzantine agreement extension can be solved with perfect security and optimal resilience in $O(nL + n^2 \log n)$ total communication (in bits) in addition to a single call to a binary asynchronous Byzantine agreement protocol. For $L = O(n \log n)$, this gives an asymptotically optimal protocol, resolving a question that remained open for nearly two decades.

List decoding is a fundamental concept in theoretical computer science and cryptography, enabling error correction beyond the unique decoding radius and playing a critical role in constructing robust codes, hardness amplification, and secure cryptographic protocols. A key novelty of our perfectly secure and optimally resilient asynchronous byzantine agreement extension protocol is that it uses list decoding - making a striking new connection between list decoding and asynchronous Byzantine agreement.

2012 ACM Subject Classification Security and privacy; Security and privacy → Cryptography; Security and privacy → Information-theoretic techniques; Security and privacy → Distributed systems security

Keywords and phrases Asynchronous Byzantine Agreement, Perfect Security

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.1

Related Version *Full Version*: `https://eprint.iacr.org/2025/1488` [2]

Funding *Gilad Asharov*: Research supported by the Israel Science Foundation (grant No. 2439/20), and by the European Union (ERC, FTRC, 101043243). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

The *agreement* problem is perhaps the quintessential problem in fault-tolerant distributed computing. In agreement, each party has an input and the goal is for all honest parties to output (*liveness*) the same value (*agreement*), and that if all of them have the same input, then this is the output (*weak validity*). In *Byzantine* agreement, the challenge is to do this while tolerating a strongly adaptive adversary that can corrupt up to t parties. It is known that for perfect security, or in asynchrony, byzantine agreement requires $n > 3t$, so we call a protocol that can tolerate $n = 3t + 1$ *optimally resilient*. In *asynchronous* byzantine agreement the network is asynchronous. In this setting infinite execution must exist, but using randomization, protocols for binary asynchronous byzantine agreement with an expected $O(1)$ rounds and expected $O(n^2)$ communication are known (for example, given access to a weak coin that has a constant probability of success) [18, 5].

The $O(n^2)$ cost for agreement on one bit is optimal due to lower bounds of Dolev and Reischuk [12] even in synchrony and for omission failures and even with randomization against a strongly adaptive adversary [6].

For the multivalued case, where the input is L bits, the obvious additional lower bound is $\Omega(nL)$ bits that are required when $\Omega(n)$ parties do not have the output as their input, need to learn the output. Thus, we say that a multivalued asynchronous byzantine agreement has *asymptotically optimal complexity* if the communication complexity is $O(nL + n^2)$. For $L = \Omega(n \log n)$, we say a multivalued asynchronous byzantine agreement that has $O(nL + n^2 \log n)$ communication complexity has *asymptotically optimal complexity.*

The goal of the asynchronous byzantine agreement extension problem, as suggested in [9, 15], is to study the costs of multivalued asynchronous byzantine agreement given access to a binary byzantine agreement protocol.

In this setting, for $L = \Omega(n \log n)$, results with optimal resilience and asymptotically optimal complexity are known under cryptographic assumptions and a PKI [17, 19] (in fact, using a PKI allows for stronger external validity properties). In synchrony, the breakthrough of [7] obtains perfect security (also see improvements [3] and statistical security with fewer rounds [1]).

Recently, the results in the asynchronous model have seen great progress: using just a cryptographic hash function [14]; near optimal resilience and statistical security [13]; and perfect security with $O(\log n)$ overhead in communication and time, or perfect with $n = 5t+1$ [8] or near optimal resilience [13].

Despite nearly two decades and considerable recent work, the following remains open: for $L = \Omega(n \log n)$, does there exist an asynchronous byzantine agreement extension protocol that is:

1. Optimally resilient ($n = 3t + 1$);
2. Asymptotically optimal (for $L = \Omega(n \log n)$ has complexity $O(Ln)$);
3. Perfectly secure (is error free).

The main result of this paper is a positive answer to this open question.

▶ **Theorem 1.1** (Main). *For $L = \Omega(n \log n)$, there exists a protocol that solves multivalued asynchronous byzantine agreement with optimal resilience, asymptotically optimal complexity, and perfect security, given access to a single instance of binary asynchronous byzantine agreement.*

In other words, our protocol achieves $O(nL + n^2 \log n)$ for inputs of size L in addition to a single call to a binary agreement. A key novelty of our perfectly secure and optimally resilient Asynchronous Byzantine Agreement Extension protocol is that it uses **list decoding** - making a striking new connection between list decoding and asynchronous byzantine agreement. List decoding [20, 16] is a fundamental concept in theoretical computer science and cryptography, allowing error correction beyond the unique decoding radius and playing a critical role in many areas of theoretical computer science. In this paper, we show its criticality in enabling optimal asynchronous extension protocols with perfect security.

As a warm-up, we show a variant with statistical security (also see [13]) to highlight some of the challenges before going to the perfect security case.

▶ **Theorem 1.2** (Statistical). *For an error parameter $0 < \epsilon < 1$, and $L = \Omega(n \log n/\epsilon)$, there exists a protocol that solves multivalued asynchronous byzantine agreement with optimal resilience, asymptotically optimal complexity, and statistical security (with error ϵ), given access to a single instance of binary asynchronous byzantine agreement.*

1.1 Discussion and open questions

Our extension is asymptotically optimal when $L = \Omega(n \log n)$ bits. Obtaining asymptotic optimality for $L = O(n)$ remains an open question. Our reduction is nearly quadratic, but the currently best binary byzantine agreement protocol with perfect security, assuming private channels, is $\Omega(n^4)$ [4]. Improving this bound, or finding other assumptions where binary ABA with perfect security is (nearly) quadratic, is another open question.

Just like [7], we obtain byzantine agreement with a weak form of validity that allows parties to agree on a special symbol $\perp$ if the inputs from the honest parties do not agree. Getting similar results for stronger notions of validity (or proving this to be impossible) also remains an open question.

1.2 High level ideas

We start with an overview of the breakthrough COOL protocol of [7] and how it can be used in asynchrony. Then describe a new functionality called BOOST. The next step is a new general extension framework that combines BOOST, COOL, and binary asynchronous byzantine agreement to achieve asynchronous byzantine agreement extension. We then overview BOOST with statistical security, and then BOOST with perfect security.

COOL in asynchrony. A modular view of the COOL protocol [7, 3] is that it is a weak variant of asynchronous verifiable information dissemination and dispersal [10, 11, 3]. In asynchrony, we can combine the two phases to get a *reliable agreement* protocol that we will call *COOL reliable agreement* or simply COOL. Roughly speaking, COOL gives: (Validity) If all honest start with the same input, they all output this value; (Totality) If an honest outputs, then all honest will output; (Agreement) The output of all honest is the same.

Note that COOL reliable agreement does not guarantee termination in all cases. It is safe, but live only in the good case (validity property). Can we use COOL along with a binary asynchronous agreement protocol to get multivalued asynchronous agreement? This is where the BOOST protocol comes in.

BOOST in asynchrony. Unlike COOL, which is safe but not live, BOOST is live but not safe. All parties eventually output a value, or eventually output $\perp$ - but these are not necessarily mutually exclusive. A party might output twice in the protocol: a value, and later also $\perp$.

In BOOST, every party will eventually output a value or output $\perp$ or do both. Roughly speaking, BOOST gives: (Validity) If all honest start with the same input, they all output this value and never output $\perp$; (Value Totality) If an honest outputs a value then all honest will output a value; ($\perp$ Totality) If an honest outputs $\perp$ then all honest will output $\perp$; and finally the correctness property: (Correct or Detect) Either all parties output the same value, or all parties output $\perp$.

In the *correct* case, all parties output the same value from BOOST. But what if they do not? Then we are in the *detect* case: we are guaranteed that all parties will eventually output $\perp$. However, parties do not know what case they are in. They may output a value first and may not be certain if they will ever eventually also output $\perp$.

This leads to the following natural protocol: run BOOST. If it outputs $\perp$ then enter the binary agreement with 0. If BOOST outputs a value, then enter COOL with this value. If COOL outputs a value, then enter the binary agreement with 1. Finally, if the binary agreement ends with 0, then output $\perp$. Otherwise, wait for the output of COOL and output that.

Roughly speaking, (Validity) If all honest have F then BOOST will only output F and COOL will output F and agreement will output 1; (Liveness) If BOOST outputs $\perp$ then all will eventually output $\perp$ so the binary agreement will complete (note, may not always be on 0). Otherwise, if there is no output of $\perp$, then there is agreement (from the correct or detect property in BOOST), so all will output the same and enter COOL with the same and from the validity of COOL and the liveness of the binary agreement, all will output the same value; (Agreement) If the binary outputs 0 then it is immediate and otherwise due to the agreement property of COOL.

Why is it called BOOST? From BOOST validity, if all honest have $F(x)$ and the t malicious have $G(x) \neq F(x)$, then the output must be $F(x)$. Consider a world where $t+1$ honest have $F(x)$, t honest have $G(x)$, and the malicious are silent. This world is indistinguishable from the validity world with t slow parties with $F(x)$ instead of the silent malicious. Hence, BOOST cannot output $\perp$ because of validity, so it must output a value. That means it must output $F(x)$.

So the $t+1$ that hold $F(x)$ must convince the t honest parties that hold $G(x)$ to also output $F(x)$. So we need to **boost** the value $F(x)$ from $t+1$ inputs to the output of all $2t+1$ honest parties. Hence, we call this protocol BOOST because its main challenge is to identify a value that appears in $t+1$ honest parties and **boost** it to all parties (or detect a problem while trying to do that).

BOOST with statistical security. For statistical security, one can test whether two polynomials are equal (with statistical error) by evaluating them at a random point. Extending this idea, a party P_i can determine whether P_j and P_k hold the same polynomial by receiving a single evaluation at a common random challenge point. This approach, used to reduce rounds in reliable agreement for statistical security [1], treats an evaluation as a hash of the polynomial. The only requirement is that the challenge point be chosen after the adversary fixes the polynomial; for simplicity in our statistical case, we assume all inputs are fixed before the protocol begins (See [1] for removing this assumption).

In the following, define the *F-case* as the case there are at least $t+1$ parties that have the same input F.

Statistical BOOST protocol in 7 phases:

1. *Exchange phase*: parties send a random challenge point, and get responses that are evaluations at the challenge point. In this phase parties also detect if they see $t+1$ evaluations that disagree with their input.
2. *Support phase*: If a party sees $t+1$ responses on the same point, it identifies a group of $t+1$ parties with the same input. It sends them a *support* message. Note that you can send support messages for at most two groups (because $3t+3 > 3t+1$).
3. *Sending YourPoint phase*: If a party hears $2t+1$ support messages on its input then it sends each party its point on this polynomial.
   In the F case, if a party has input $G(x) \neq F(x)$ and hears $2t+1$ support, this means that all parties will see at least $t+1$ support for $G(x)$, and hence all $t+1$ parties with input $F(x)$ will detect.
4. *Sending MyPoint phase*: If a party hears $t+1$ parties send it the same YourPoint, then it sends back this point as a MyPoint message. Critically, a party sends only one MyPoint message.
   So in the F-case, if no $G(x) \neq F(x)$ has $2t+1$ support then eventually all honest parties will send their MyPoint for $F(x)$.

5. *Reconstruct phase*: Parties now wait to robustly reconstruct at least $2t + 1$ MyPoint values that agree on the same polynomial.
6. *Totality and Abundance phase*: standard techniques to ensure that one termination implies all terminate and that at least $t + 1$ terminate with a value.
7. *Disseminate phase*: standard techniques to ensure that if at least $t + 1$ have the same value, and all other have this value or $\perp$ then all honest will output this value. This part is done as a separate sub-protocol.

BOOST with perfect security. Moving from statistical to perfect security introduces several new challenges.

First, in the statistical setting, either $t + 1$ evaluations agree on your input or $t + 1$ disagree, and you detect. With perfect security, detecting from $2t + 1$ responses is much harder. Moreover, sending support requires finding a set of $t + 1$ parties that have the same polynomial. How can you do this if you only hear two deterministic evaluation points from each party?

Second, unlike the statistical setting, we cannot get strong detection properties so early in the protocol, hence we cannot stop after sending one MyPoint and we may need to send several MyPoints. But this complicates decoding: how can you decode if each party sends two points? multiple polynomials may fit. Even more problems arise since a support message may help several polynomials, not just one with high probability as in the statistical setting.

Third, is the most subtle, in the F-case we must ensure that if some party outputs $G(x) \neq F(x)$ then all parties will eventually detect. But party i may have $G(x)$ such that $G(i) = F(i)$, so detecting a conflict seems challenging.

To address the first problem, we use a more subtle detection rule (see Claim 5.2). This detection uses *List Decoding*: we observe that identifying $t + 1$ points that agree on a degree $< t/7$ polynomial is exactly a List Decoding instance and prove the list is at most of size 3.

For the second problem, list decoding helps again: instead of interpolating, we wait to see if there are $2t + 1$ points that agree with a polynomial in our list. This may yield a list of reconstructed polynomials, unlike the statistical case. This adds complexity and forces us to do several non-trivial rounds of detection in order to make sure that in the F-case, if a party outputs $G(x) \neq F(x)$ then all parties eventually detect.

Obtaining this detection is the crux of our protocol, and is done in three phases. The *Filter* phase detects one type of disagreement and then the *Simple Detect* phase detects another. After these two phases there may still be a party that wants to output $G(x) \neq F(x)$ but has $G(i) = F(i)$. In this case, we need to add an additional *Resolve Conflict* phase.

This phase again uses list decoding, allowing every party to learn about $F(x)$. So parties holding $G(x)$ with $G(i) = F(i)$ learn that evaluations at i cannot differentiate parties holding $G(x)$ from parties holding $F(x)$, so they send a new challenge point i' such that $G(i') \neq F(i')$ and using responses from this new challenge point we can finally get the desired detection property.

BOOST with perfect security - Overview of the protocol. Recall that the *F-case* is when there are at least $t + 1$ parties that have the same input F.

1. *Exchange phase*: here each party i with input $f_i(x)$ sends to each party j the two points $f_i(i)$ and $f_i(j)$. There are two non-trivial aspects when receiving these points:
   a. The first is that a party detects if the number of parties that send detect plus the number of parties that send points that do not agree with your input is more than $t + 1$. This rule is used in a subtle manner to prove the detect property in Claim 5.2 in the not F-case.

   b. The second is that we need to give support to any set of $t+1$ parties that agree on the same degree at most $d < t/7$ polynomial, but we cannot use a random challenge point for perfect security. So the only way to find this is to use (online) List Decoding to generate a dynamic set S^1. Since our degree is $< t/7$, the list has at most 3 elements (see Claim 5.2).
2. *Filter phase*: each party sends its support to at most 3 polynomials by sending back to them their point on this polynomial. Now a party may see 3 different values from each other party, how can it know if it has a support of 2t+1? The new idea here is to check which of the polynomials in S^1 has $2t+1$ points that agree with it. Parties put those polynomials in a dynamic set S^2.
   Parties also detect if a polynomial other than their input is in S^2. In the F-case, we prove that all parties whose input disagrees with $F(x)$ will eventually detect. Going forward, let $|B|$ be the number of parties detected this way.
   This detection is not enough, because some party may get $G(x) \in \mathsf{S}^2$ with $G(x) \neq F(x)$ but we may still not have $t+1$ detections. The next two phases solve this.
3. *Simple detect phase*: In this phase if any party i adds $G(x) \neq F(x)$ to S^3 and $G(i) \neq F(i)$ then we prove all parties will eventually detect. We do this by proving that at least $t+1-|B|$ parties that have $F(x)$ as input must detect (see Claim 5.4) and use the $|B|$ detections from the previous phase.
4. *Resolve conflict phase*: Here we want to detect if a party i outputs $G(x) \neq F(x)$ but has $G(i) = F(i)$. We observe that if the previous detect does not trigger, then *all* honest parties have points that agree with $F(x)$ (even if the polynomial they have is not $F(x)$). So how can a party holding $G(x) \neq F(x)$ with $G(i) = F(i)$ know that it is not holding F(x) in this case?
   This observation implies that party i can list decode the values it receives at this stage, and it must include $F(x)$ in this list after waiting for $2t+1$ messages. Now party i, that is holding $G(x) \neq F(x)$, is aware of $F(x)$.
   Party i sees that $G(i) = F(i)$ so index i is not suitable for disambiguating between $F(x)$ and $G(x)$. So party i chooses a new index i' such that $G(i') \neq F(i')$ and sends that as a *new challenge point*. With this new challenge point, in the F-case, if a party outputs $G(x) \neq F(x)$ then it must be that all parties output detect (see Claim 5.5). This is somewhat reminiscent of the random point challenge used in the statistical security setting.
5. *Totality and Abundance phase*: standard techniques.
6. *Disseminate phase*: standard techniques.

2 Preliminaries and Building Blocks

2.1 Notations

We let n denote the total number of participants. We let t denote the total number of corrupted parties. We assume Byzantine faults, and that $n \geq 3t+1$. Here we focus on the case where $n = 3t+1$. We assume communication channels are identifiable so parties can identify who sent them a message.

Reed Solomon codes. Let $\mathbb{F}$ be a finite field such that $|\mathbb{F}| > n$, where n is the number of parties. Without loss of generality, we assume that $1, \ldots, n \in \mathbb{F}$, while this is just to ease convention. To encode a message $m = (m_0, \ldots, m_d)$ where each $m_i \in \mathbb{F}$, the encoding algorithm defines a polynomial $P(x) = m_0 + m_1 x + \ldots + m_d x^d$, and outputs $(P(1), \ldots, P(n))$.

Since two polynomials of degree d can agree on at most d points, the distance is $n-d$. Thus, we can correct up to $(n-d-1)/2$ errors. When $d < n/3$, this means that we can correct, in particular, t errors. For a set S of points, we use $\mathsf{RSDec}(S, d)$ as the unique decoding procedure of Reed-Solomon codes, which returns a unique polynomial of degree d that agrees with all but at most d points in S (if exists).

List decoding. List decoding procedure of Reed-Solomon codes allows the return of all polynomials that agree with a set S when the distance is larger than the minimum distance of the code. We use $\mathsf{ListDecode}(S, d, p)$ to denote the procedure that returns all polynomials of degree d that agree with S on p points. Looking ahead, we will use it with $d = t/7$ and $p = t+1$, and our analysis would show that when $2t+1 \leq |S| \leq 3t+1$, there are no more than three possible polynomials that satisfy that, i.e., no more than three polynomials of degree $t/7$ that agree with $t+1$ points on S.

Input length. In our protocols, we assume that the input is a polynomial of degree d (looking ahead, t in the statistical, and $t/7$ in perfect). For the general case of an L-bit message, the parties segment the L-bit message into $L = \lceil \frac{L}{(d+1)\log|\mathbb{F}|} \rceil$ blocks of size $(d+1)\log|\mathbb{F}|$ bits each (with padding when necessary). Generalizing the protocols to L-blocks input can be done in a natural way, e.g., instead of sending a single point, we would send L points, each is associated with a different polynomial. We omit such details. Additionally, in the perfect security case, we assume $\log|\mathbb{F}| \in O(\log n)$.

2.2 Dispersal

▶ **Definition 2.1** (Dispersal). *A protocol for parties $P_1, \ldots, P_n$ where the input of each party P_i is some $f_i(x) \in \mathbb{F}^{d+1}$, is an* asynchronous dispersal protocol tolerating t corrupted parties *if the following properties hold:*

- **Termination:** *If one honest party terminates, then all honest parties terminate.*
- **Weak agreement:** *If an honest party terminates, then at least $t+1$ honest parties terminate with the same output $F(x)$, and the rest might terminate with either $F(x)$ or $\perp$.*
- **Weak Validity:** *If all honest parties start with the same polynomial $F(x)$ of degree at most d, then termination and weak agreement hold with respect to $F(x)$.*

It is important to note that this protocol might never terminate. In particular, termination is guaranteed only in the case where all honest parties have the exact same input. We might also terminate with other outputs, in which case weak agreement is guaranteed ($t+1$ honest parties output the same output, but the rest output $\perp$).

Protocol 2.1: Dispersal

Input: Each party P_i holds $f_i(x)$ of degree at most d over $\mathbb{F}$.
The protocol:

1. Initialization: Each party initializes $S_i = \emptyset$, $\mathsf{A}_i^1 = \mathsf{A}_i^2 = \emptyset$.
2. **Exchange:**
   a. P_i sends $(\texttt{Exchange}, f_i(i), f_i(j))$ to each P_j.
3. **Dynamic set A_i^1:**
   a. **Upon** receiving $(\texttt{Exchange}, u_j, v_j)$ from P_j, if $f_i(j) = u_j$ and $f_i(i) = v_j$ then add j to A_i^1.
   b. **Upon** $|\mathsf{A}_i^1| \geq n - t$, send OK_1 to all.

4. **Dynamic set A_i^2:**
   a. **Upon** receiving message OK_1 from P_j for which $j \in \mathsf{A}_i^1$, then add j to A_j^2.
   b. **Upon** $|\mathsf{A}_i^2| \geq n - t$, send OK_2 to all.
5. **Sending** `Done`:
   a. If P_i sent OK_2, and **upon** receiving $2t + 1$ OK_2 messages, send `Done` messages to all.
   b. **Upon** receiving $t + 1$ `Done` messages from distinct parties, send `Done` to everyone.
   c. **Upon** receiving $2t + 1$ `Done` messages, if P_i sent OK_2 message, then terminate and output $f_i(x)$. If P_i did not send OK_2 message, then terminate and output $\perp$.

The following theorem is proven in [3], and the protocol is based on the protocol of [7]:

▶ **Theorem 2.2** ([3]). *Protocol 2.1 is a perfectly-secure asynchronous dispersal protocol tolerating $t < n/3$ malicious parties (as per Definition 2.1). The protocol takes 4 rounds and a total communication of $O(nL + n^2 \log n)$ bits where each party starts with an input of size L.*

We remark that Protocol 2.1 is described where the input of each party is a polynomial $f_i(x)$ of degree d, whereas the above theorem refers to a general input of size L. See remark about input length in Section 2.1. We also remark that in [3], the protocol assumes that $d < t/3$.

2.3 Asynchronous Data Dissemination

▶ **Definition 2.3.** *A protocol for parties $P_1, \ldots, P_n$ where the input of each party P_i is some $f_i(x) \in \mathbb{F}^{d+1}$ is a* Asynchronous Data Dissemination protocol *tolerating t corrupted parties if the following properties hold:*

- **Termination:** *If one honest party terminates, then all parties terminate.*
- **Validity:** *If at least $t + 1$ honest parties start with the same input $F(x)$, and all other honest parties start with $\perp$, then all honest parties output $F(x)$.*

Protocol 2.2: Asynchronous Data Dissemination

Input: Each party P_i holds $f_i(x)$ as input, of degree at most d. Some P_i might have input $\perp$.

The protocol:

1. Initialize a multi-set $M_i = \emptyset$ and $S_i = \emptyset$. If $f_i(x) \neq \perp$, send to each P_j its point $(\mathsf{YourPoint}, f_i(j))$.
2. **Upon** receiving $(\mathsf{YourPoint}, u_j)$ from P_j, add u_j to M_i.
3. **Upon** some u_i appearing $t + 1$ times in M_i, send $(\mathsf{MyPoint}, u_i)$ to all parties.
4. **Upon** receiving $(\mathsf{MyPoint}, u_j)$ from P_j, add (j, u_j) to S_i. **Upon** $|S_i| \geq d + t + 1$ execute the following:
   a. Run $\mathsf{RSDec}(S_i)$ and try to decode a polynomial $f_i(x)$ of degree at most d that agrees with S_i on at least $d + t + 1$ values.
   b. If no such polynomial exists, then wait to receive more points in S_i and retry.
   c. If such a polynomial $f_i(x)$ is computed, set $f_i(x)$ to be the resultant value.
5. **Upon** unique decoding $f_i(x)$, terminate and output $f_i(x)$.

▶ **Theorem 2.4** ([10, 3]). *Protocol 2.2 is an asynchronous data-dissemination protocol tolerating $t < n/3$ malicious parties. The protocol takes 2 rounds and $O(nL + n^2 \log n)$ communication, where each party starts with an input of size L.*

2.4 Reliable Agreement

In reliable broadcast, there is a sender, and other parties have no inputs. The sender wishes to broadcast its message, and that all honest parties would output that value. Agreement and validity are guaranteed, but there is no guaranteed termination.

We now consider "reliable agreement". The difference from reliable broadcast is that all parties have an input, and there is no sender.

▶ **Definition 2.5.** *A protocol for parties $P_1, \ldots, P_n$, where the input of each party P_i is some $f_i(x) \in \mathbb{F}^{d+1}$ is a* Reliable Agreement *tolerating t corrupted parties, if the following properties hold:*

- **Termination:** *If one honest party terminates, then all parties terminate.*
- **Validity:** *If all honest parties start with the same input $F(x)$, then all honest parties terminate with output $F(x)$.*
- **Agreement:** *If one honest party terminates with output $F'(x)$, then it is guaranteed that all honest parties eventually terminate with output $F'(x)$.*

Protocol 2.3: Reliable Agreement

- **Input:** The input of each party is some polynomial $f_i(x)$ of degree d over $\mathbb{F}$.
- **The protocol:**
  1. Run Dispersal (Protocol 2.1 with input $f_i(x)$.
  2. **Upon** dispersal terminating with output $f_i^{(1)}(x)$, enter dissemination (Protocol 2.2) with that input.
  3. **Upon** dissemination terminating with output $f_i^{(2)}(x)$, output this polynomial.

The following is proven in the appendix:

▶ **Theorem 2.6.** *Protocol 2.3 is a reliable agreement protocol tolerating $t < n/3$ corrupted parties. It requires $O(nL + n^2 \log N)$ communication for an input of size L.*

2.5 Asynchronous Byzantine Agreement

In an asynchronous Byzantine Agreement protocol, each party has some input x_i, and the parties have to agree on some output. It is required that all parties must terminate, must agree on the output, and if all honest parties start with the same input x, then the output x.

▶ **Definition 2.7.** *A protocol for parties $P_1, \ldots, P_n$ where the input of each party P_i is some bit $b_i \in \{0, 1\}$ is a* Asynchronous Byzantine Agreement *tolerating t corrupted parties if the following properties are satisfied:*

- **Agreement:** *All honest parties must output the same value.*
- **Validity:** *If all honest parties enter with the same input x, then all output x.*
- **Termination:** *All honest parties must terminate.*

If $x \in \{0, 1\}$, then we call the protocol Binary Asynchronous Byzantine Agreement*; If the input is a general string, then we call the protocol* Multi-value Asynchronous Byzantine Agreement*. If a Multi-value Asynchronous Byzantine Agreement uses a Binary Asynchronous Byzantine Agreement, then we call it a* Asynchronous Byzantine Agreement Extension *protocol.*

3 From BOOST to Multivalue Byzantine Agreement

3.1 BOOST

We introduce a new primitive called BOOST. We build a multivalue Byzantine Agreement protocol from BOOST, which we define its properties next:

▶ **Definition 3.1.** *A protocol for parties $P_1, \ldots, P_n$, where the input of each party P_i is some polynomial $f_i(x)$ of degree d over $\mathbb{F}$, is a* BOOST *protocol, tolerating t corrupted parties, if the following properties are satisfied:*

- **Syntax:** *The protocol might not terminate, but each party eventually writes to (at least) one of two output tapes –* output_i*, or/and* detect_i*.*
- **Validity:** *If all honest parties enter with the same polynomial $F(x)$, then at least $t+1$ honest parties set their output to $F(x)$. Moreover, no honest party activates output* detect*.*
- **Set output:** *If an honest party sets output then all honest parties set their output.*
- **Detect or correct:** *If at least $t+1$ honest parties hold the same polynomial $F(x)$ as input, then: (a) either eventually all honest parties activate their detect output tape; or (b) at least $t+1$ honest parties set their output tape to $F(x)$, and the remaining set their output to* proceed*.*
- **Detect:** *If there is no set of $t+1$ honest parties with the same input, then all honest parties activate their detect tape.*

In fact, the crux is in designing this primitive. In the next subsection, we show how to utilize this primitive to obtain a multivalue byzantine agreement. In Section 4 we show how to implement BOOST with statistical security, which is shown mainly as a warmup. In Section 5 we show how to implement it with perfect security.

3.2 Main Protocol: Multivalue Byzantine Agreement

Protocol 3.1: Multivalue Byzantine Agreement

- **Input:** Each party P_i holds $f_i(x)$ of degree d as input.
- **The protocol:**
  1. **BOOST → Dissemination → Reliable Agreement:**
     a. Run BOOST protocol with input $f_i(x)$.
     b. **Upon** P_i sets $\mathsf{output}_i \neq \perp$ in BOOST, enter asynchronous data dissemination (Protocol 2.2). If $\mathsf{output}_i = f_i^{(1)}(x)$, then enter with that input; If $\mathsf{output}_i = \mathsf{proceed}$ then enter with $\perp$.
     c. **Upon** dissemination terminates with output $f_i^{(2)}(x)$, run reliable agreement (Protocol 2.3) with that input $f_i^{(2)}(x)$.
  2. **Asynchronous Binary Byzantine Agreement:** The parties run a single instance of Byzantine Agreement, where the input of each party is determined once, according to which event occurs first:
     a. **Upon** reliable agreement terminates with some output, enter BA with input 1.
     b. **Upon** BOOST activates $\mathsf{detect}_i = 1$, enter BA with input 0.
- **Output:**
  1. **Upon** BA terminates with output 0, terminate and output $\perp$.
  2. **Upon** BA terminates with output 1, output the output of reliable agreement and halt.

The following theorem is proven in the appendix:

▶ **Theorem 3.2.** *Protocol 3.1 is a multivalued byzantine agreement protocol, tolerating t corrupted parties.*

4 BOOST with Statistical Security

We refer to Section 1.2 for an overview of the following protocol.

Protocol 4.1: BOOST with Statistical Security

- **Input:** Each party enters with an input $f_i(x)$ – a polynomial of degree at most t.
- **Initialization:** Initialize $\mathsf{output}_i = \perp$, $g_i(x) = \perp$, $\mathsf{detect}_i = \perp$. Moreover, initialize an array C_i of size n, and a list T_i.

The protocol:

1. **Exchange:**
   a. Each party P_i chooses $r_i \leftarrow \mathbb{F}$ uniformly at random, and send to every P_j the message $(\texttt{challenge}, r_i)$.
   b. **Upon** receiving $(\texttt{challenge}, r_j)$ from P_j, set $C_i[j] = r_j$. Moreover, reply P_j with the message $(\texttt{Exchange}, f_i(r_j))$.
2. **Getting $2t+1$ support or send detect:**
   a. **Upon** receiving $(\texttt{Exchange}, u_j)$ from P_j, add (j, u_j) to the list T_i. If $u_j \neq f_i(r_i)$, then add j to DA_i.
   b. **Upon** $|\mathsf{DA}_i| \geq t+1$, then send to all parties the message detect.
   c. **Upon** T_i containing the same evaluation u_i from at least $t+1$ parties, send $(\texttt{support}, r_i, u_i)$ to all the parties.
   d. **Upon** receiving $2t+1$ messages $(\texttt{support}, r_j, u_j)$ such that $C_i[j] = r_j$ and $f_i(C_i[j]) = u_j$, then set $g_i(x) = f_i(x)$. Send to each P_k the message $(\mathsf{YourPoint}, g_i(C_i[k]))$.
   e. **Upon** receiving $t+1$ messages $(\texttt{support}, r_j, v_j)$ such that $C_i[j] \neq r_j$ or $v_j \neq f_i(r_j)$, then send detect to all parties.
3. **Reconstruction:**
   a. **Upon** receiving $t+1$ messages $(\mathsf{YourPoint}, u)$ with the same u, and P_i did not previously send $(\mathsf{MyPoint}, \cdot, \cdot)$ message, then send $(\mathsf{MyPoint}, r_i, u)$ to all. If $u \neq f_i(r_i)$, then send detect message to all (if not sent yet).
   b. **Upon** receiving a message $(\mathsf{MyPoint}, r_j, u_j)$ then add (r_j, u_j) to S_i. If $f_i(r_j) \neq u_j$, then add j to DA_i. (recall that **upon** $|\mathsf{DA}_i| \geq t+1$ then P_i sends detect message to all.)
   c. **Upon** $\mathsf{RSDec}(S_i, t) = f_i'(x)$, and $f_i'(r_j) = u_j$ for $2t+1$ points in S_i, then set $g_i(x) = f_i(x)$; Send $\texttt{HaveOutput}$ to all.
4. **Have output and termination:**
   a. **Upon** receiving $\texttt{HaveOutput}$ messages from $2t+1$ distinct parties, send $\texttt{done}$ to all.
   b. **Upon** receiving $\texttt{done}$ messages from $t+1$ distinct parties, send $\texttt{done}$ message to all.
5. **Detection:**
   a. **Upon** receiving detect messages from $t+1$ distinct parties, and if not sent detect message yet, send detect to all.
   b. **Upon** receiving detect messages from $2t+1$ distinct parties, set the output tape $\mathsf{detect}_i = 1$.
6. **Setting output:**
   a. **Upon** receiving $\texttt{done}$ messages from $2t+1$ distinct parties, then: If $g_i(x) \neq \perp$ then set $\mathsf{output}_i = g_i(x)$. Otherwise set $\mathsf{output}_i = \mathsf{proceed}$.

The following theorem is proven in the appendix:

▶ **Theorem 4.1.** *Protocol 4.1 is a* BOOST *protocol (as per Definition 3.1) where validity and set output hold unconditionally (with probability 1), and the Detect or correct and detect properties hold with probability* $1 - \frac{n^3}{|\mathbb{F}|}$.

5 BOOST with Perfect Security

In this section, we show how to implement the BOOST protocol for the perfect setting. We already provided an overview for this protocol in Section 1.2, and so we provide the formal description next.

5.1 The Entire Protocol

Protocol 5.1: BOOST **with Perfect Security**

- **Input:** Each party P_i starts with $f_i(x)$ of degree at most $t/7$ over $\mathbb{F}$.
- **Initialization:** The following variables are shared between the different sub-protocols. Initialize $\mathsf{Others}_i, \mathsf{My}_i, \mathsf{DA}_i = \emptyset$. Initialize $\mathsf{S}_i^1, \mathsf{S}_i^2, \mathsf{S}_i^3, \mathsf{LD}_i = \emptyset$.
- **The protocol:** Run all sub-protocols in parallel (Sub-protocol 5.2, 5.3, 5.4, 5.5, 5.7, 5.6), while using the same internal variables.
- **Output:** The protocol does not terminate, but each party P_i might eventually write to the output tapes output_i or detect_i (or both).

Sub-Protocol 5.2: Exchange Phase

1. Send to each P_j the message $(\texttt{Exchange}, f_i(i), f_i(j))$.
2. **Upon** receiving $(\texttt{Exchange}, u_j, v_j)$ from P_j:
   a. Add the points (j, u_j) to Others_i and (j, v_j) to My_i.
   b. If $|\mathsf{Others}_i| \geq 2t + 1$, run $L_i = \mathsf{ListDecode}(\mathsf{Others}_i, t/7, t + 1)$, and set $\mathsf{S}_i^1 = \mathsf{S}_i^1 \cup L_i$.
   c. If $u_j \neq f_i(j)$ or $v_j \neq f_i(i)$ then add j to DA_i.
3. **Upon** $|\mathsf{DA}_i| \geq t + 1$ send detect to all.
4. **Upon** receiving detect message from P_j, add j to DA_i.

Sub-Protocol 5.3: Filter Phase

1. **Upon** adding a polynomial $g_i(x)$ to S_i^1:
   a. Let $P[g_i(i)] = \{j \in [n] \mid (j, v) \in \mathsf{My}_i\}$.
   b. **Upon** $|P[g_i(i)]| \geq t + 1$, then send $(\texttt{MyPotentialPoint}, g_i(i))$ to all parties. If $g_i(i) \neq f_i(i)$, then send detect to all. Moreover, note that each party sends at most two $\texttt{MyPotentialPoint}$ messages.
2. **Upon** receiving $(\texttt{MyPotentialPoint}, u_j)$ from P_j, add (j, u_j) to T_i.
3. **Upon** (a) $g_i(x) \in \mathsf{S}_i^1$; (b) $g_i(x)$ agrees with $2t + 1$ points in T_i, then add $g_i(x)$ to S_i^2.
4. **Upon** a polynomial $g_i(x)$ added to S_i^2, but $g_i(x) \neq f_i(x)$, then send detect to all.

Sub-Protocol 5.4: Detect Simple Conflicts

1. **Upon** adding a polynomial $g_i(x)$ to S_i^2:
   a. Send to each party P_j the message $(\texttt{YourPoint}, g_i(j))$.
   b. If $g_i(j) \neq f_i(j)$, then send detect message to all.
2. **Upon** receiving $(\texttt{YourPoint}, u_j)$ from P_j, add (j, u_j) to S_i.
3. **Upon** (a) $g_i(x) \in \mathsf{S}_i^2$; (b) S_i containing $(j, g_i(i))$ from at least $2t + 1$ distinct parties, then add $g_i(x)$ to S_i^3.

Sub-Protocol 5.5: Resolve Conflicts

1. **Upon** adding a polynomial $h_i(x)$ to S_i^3:
   a. Send to each party P_j the message $(\texttt{MyPoint}, h_i(j))$.
2. **Upon** $|\mathsf{S}_i^3| > 1$, send detect to all.
3. **Upon** receiving first $(\texttt{MyPoint}, u_j)$ from P_j, add (j, u_j) to R_i. (If a party sends more than one point, add to R_i only the first point it sent.)
4. **Upon** R_i containing messages from $2t+1$ distinct parties:
   a. Let $\mathsf{LD}_i := \mathsf{ListDecode}(R_i, t/7, t+1)$.
   b. If $|\mathsf{S}^3 \cup \mathsf{LD}_i| > 1$ then
      i. Find $i' = i + c \cdot n$ for the first integer c for which $|\{h(i') \mid h \in \mathsf{S}_i^3 \cup \mathsf{LD}_i\}| = |\mathsf{S}_i^3 \cup \mathsf{LD}_i|$, i.e., $h(i')$ is different for every polynomial in $\mathsf{S}_i^3 \cup \mathsf{LD}_i$.
      ii. Send to all parties $(\texttt{challenge}, i')$.
      iii. **Upon**
         A. receiving $2t+1$ responses $(\texttt{response}, v)$ from distinct parties, that all agree on v;
         B. $\left|\{h_i(x) \in \mathsf{S}_i^3 \mid h_i(i') = v\}\right| = 1$; and
         C. $g_i(x) = \perp$:
         Set $g_i(x)$ to be that unique polynomial $h_i(x)$.
   c. **Upon** seeing $(\texttt{challenge}, j')$ from j: if there exists $g_i(x) \in \mathsf{S}_i^3$ and you did not yet respond with $(\texttt{response}, g_i(j'))$ then send P_j $(\texttt{response}, g_i(j'))$.
   d. **Upon** (a) $|\mathsf{S}^3 \cup \mathsf{LD}_i| = 1$; (b) $\mathsf{S}_i^3 = \{h_i(x)\}$; (c) $g_i(x) = \perp$: set $g_i(x) = h_i(x)$.

Sub-Protocol 5.6: Totality and Abundance Phase

1. **Upon** $g_i(x) \neq \perp$, set `HaveOutput` to all.
2. **Upon** receiving `HaveOutput` from distinct $2t+1$ parties, send `done` to all.
3. **Upon** receiving `done` messages from distinct $t+1$ parties, and did not send `done` message yet, send `done` to all.
4. **Upon** receiving `done` messages from distinct $2t+1$ parties, set $\mathsf{output}_i = g_i(x)$ if $g_i(x) \neq \perp$ and set $\mathsf{output}_i = \mathsf{proceed}$ otherwise.

Sub-Protocol 5.7: Detection

1. **Upon** receiving $t+1$ detect message, send detect to all.
2. **Upon** receiving detect messages from $2t+1$ distinct parties, set $\mathsf{detect}_i = 1$.

In Section 5.2 we prove the following theorem:

▶ **Theorem 5.1.** *Protocol 5.1 is a* BOOST *protocol (as per Definition 3.1), tolerating at most t corrupted parties. The protocol is perfectly secure and requires $O(n^2 \log n)$ communication.*

The theorem is proven by explicitly specifying the properties each sub-protocol achieves. We provide a formal proof for those sub-protocols in the full version [2].

▷ Claim 5.2. Sub-protocol 5.2 (Exchange phase) satisfies the following properties:

1. **Validity:** If all honest start with the same input $F(x)$, then no honest send detect message and eventually all honest add $F(x)$ to S_i^1.
2. **Locally small:** Each honest party P_i adds no more than 3 polynomials to S_i^1.
3. **Boost:** If there exists a set of at least $t+1$ honest parties that start with the same polynomial $F(x)$, then each honest party eventually adds $F(x)$ to its S_i^1.
4. **Detect:** If there is no set of $t+1$ honest parties with the same polynomial, then all honest parties eventually detect.

▷ Claim 5.3. Sub-Protocol 5.3 (Filter phase) satisfies the following properties:
1. **Validity:** if all honest parties have $F(x)$ in their input, then no honest party detects, and each honest party eventually adds $F(x)$ to S_i^2. Moreover, no other polynomial is ever added to S_i^2.
2. **Detect or correct**: If there exists a set of $t+1$ honest parties that start with the same input $F(x)$, then each honest party eventually has $F(x) \in \mathsf{S}_i^2$. Moreover, honest parties that do not hold $F(x)$ as input will detect.

Now the problem is that we may get two polynomials in S_i^2. The above protocol guarantees that any $\neq F(x)$ input party will detect, but they are the minority so we may just get just $|B| < t+1$ detects from this.

The next step makes sure that if all honest eventually have $F(x) \in \mathsf{S}_i^2$ and some honest adds $G(x) \neq F(x)$ to S_i^3 with $G(i) \neq F(i)$ then at least $t+1-|B|$ additional parties that started with $F(x)$ detect. For a total desired $t+1$ honest detections.

▷ Claim 5.4. Sub-protocol 5.4 (Detect Simple Conflicts) satisfies the following properties:
1. **Validity:** If all honest eventually have $\mathsf{S}_i^2 = \{F(x)\}$ then no honest detects and eventually all honest have $\mathsf{S}_i^3 = \{F(x)\}$.
2. **Detect or correct:** Assume there exists a set of $t+1$ honest parties that hold the same $F(x)$ as input. Then; (a) all honest parties eventually add $F(x)$ to S_i^3, (b) if some honest party i adds $G(x) \neq F(x)$ to S_i^3 with $G(i) \neq F(i)$, then all honest parties eventually detect.

So now the remaining problem is if all honest that add $G(x) \neq F(x)$ to S_i^3 have $G(i) = F(i)$.

The next protocol handles this case. First, it makes sure that the parties with $G(i) = F(i)$ will see $F(x)$ in in a new List Decoding, then it resolves this by having them choose a new index where $G(i') \neq F(i')$.

▷ Claim 5.5. Sub-protocol 5.5 (Resolve conflicts) satisfies the following properties:
1. **Validity:** If all honest parties eventually have $\mathsf{S}_i^3 = \{F(x)\}$, then all eventually set $g_i(x) = F(x)$.
2. **Detect or correct:** If all honest eventually have $F(x) \in \mathsf{S}_i^3$, all honest will eventually have set $g_i(x) = F(x)$, or $t+1$ honest parties eventually detect.

▷ Claim 5.6. Sub-Protocol 5.6 (Totality and Abundance Phase) The protocol satisfies the following properties:
1. **Totality:** If an honest set output then all will eventually set output.
2. **Abundance:** If an honest set output then at least $t+1$ must have $g_i(x) \neq \perp$.
3. **Validity:** If all honest set $g_i(x) \neq \perp$, then all will set output output_i.

▷ Claim 5.7. Sub-protocol 5.7 (Detection phase) satisfies the following conditions:
1. If parties set $\mathsf{detect}_i = 1$, then at least one honest party initiated a detect message throughout the protocol.
2. If $t+1$ honest parties initiate detect message, then all honest parties eventually set $\mathsf{detect}_i = 1$.

5.2 Putting it All Together: Proving Theorem 5.1

▶ **Theorem 5.8** (Theorem 5.1, restated)**.** *Protocol 5.1 is a* BOOST *protocol (as per Definition 3.1), tolerating at most t corrupted parties. The protocol is perfectly secure and requires $O(n^2 \log n)$ communication.*

Proof. We show that each of the properties must hold.

Validity. We show that if all honest parties start with the same input $F(x)$, then honest parties do not detect, and at least $t+1$ honest parties output $F(x)$ and the rest output detect.

1. From the validity condition of Claim 5.2, no honest party detects in that subprotocol, and eventually all honest parties add $F(x)$ to S_i^1.
2. From the validity condition of Claim 5.3, no honest party detects, and each party eventually have $\mathsf{S}_i^2 = \{F(x)\}$.
3. From the validity condition of Claim 5.4, all honest parties eventually have $\mathsf{S}_i^3 = \{F(x)\}$ and do not trigger detect.
4. From the validity condition of Claim 5.5, all honest parties eventually set $g_i(x) = F(x)$.
5. From the validity condition of Claim 5.6, when all honest parties set $g_i(x) \neq \perp$, then all set output output_i.
6. If one honest party sets output output_i, then from the Abundance condition of Claim 5.6, at least $t+1$ honest parties must have $g_i(x) \neq \perp$. The only value that this could be is $F(x)$. The rest (which might not have $g_i(x) \neq \perp$) set their output to proceed.

Set output. If an honest party sets its output, then all honest parties eventually set output – this follows directly from the abundance condition in Claim 5.6.

Detect. If there is no set of $t+1$ honest parties with the same input, then all honest parties eventually detect. This follows directly from the detect condition in Claim 5.2.

Detect or correct. We show that if at least $t+1$ honest parties hold the same polynomial $F(x)$ as input, then (a) either eventually all honest parties detect; or (b) $t+1$ honest parties set their output to $F(x)$ and the remaining set output to proceed.

We show that if there is no detect, then $t+1$ honest parties set their output to $F(x)$, and the remaining set to proceed:

1. From the boost condition in Claim 5.2, all honest parties eventually add $F(x)$ to S_i^1.
2. From the detect or correct condition of Claim 5.3, each honest party eventually has $F(x) \in \mathsf{S}_i^2$. Moreover, honest parties that do not hold $F(x)$ as input trigger detect. However, we assume that those are less than t parties.
3. From the detect or correct condition of Claim 5.4, then all honest parties eventually add $F(x)$ to S_i^3; moreover, if some honest party adds $G(x) \neq F(x)$ to S_i^3 with $G(i) \neq F(i)$, then all honest parties detect. Therefore, since we assume that there is no detect, the only polynomials $G(x)$ that could be added to S_i^3 must satisfy $G(i) = F(i)$.
4. From the detect or liveness property of Claim 5.5, if all honest parties eventually have $F(x) \in \mathsf{S}_i^3$, then all honest parties eventually set $g_i(x)$. Moreover, from detect or correct, if some party sets $g_i(x)$ to be $G(x) \neq F(x)$, then all honest parties must detect. Assuming no detect, we must conclude that all honest parties set $g_i(x) = F(x)$.
5. From the validity condition of Claim 5.6, when all honest parties set $g_i(x) \neq \perp$, then all set output output_i.
6. If one honest party sets output output_i, then from the Abundance condition of Claim 5.6, at least $t+1$ honest parties must have $g_i(x) \neq \perp$. We showed that if this value is not $F(x)$, then we must eventually detect. Moreover, the rest (which might not have $g_i(x) \neq \perp$ yet) set their output to proceed.

Efficiency. From Claim 5.2, each party has at most 3 polynomials in S_i^1. This bounds all the potential points that a party might send, and by inspection, each party sends at most $O(n)$ points on $\mathbb{F}$ to every other party. We conclude that all honest parties send/receive $O(n^2 \log n)$ bits. In case of $L \in \Omega(n \log n)$, the protocol requires encoding the input as polynomials, and requires $O(nL + n^2 \log n)$ communication. ◀

References

1 Ittai Abraham and Gilad Asharov. Gradecast in synchrony and reliable broadcast in asynchrony with optimal resilience, efficiency, and unconditional security. In *PODC '22: ACM Symposium on Principles of Distributed Computing, 2022*, pages 392–398. ACM, 2022. doi:10.1145/3519270.3538451.

2 Ittai Abraham and Gilad Asharov. ABEL: Perfect asynchronous byzantine extension from list-decoding. Cryptology ePrint Archive, Paper 2025/1488, 2025. URL: https://eprint.iacr.org/2025/1488.

3 Ittai Abraham, Gilad Asharov, and Anirudh Chandramouli. Simple is COOL: graded dispersal and its applications for byzantine fault tolerance. In Raghu Meka, editor, *16th Innovations in Theoretical Computer Science Conference, ITCS 2025, January 7-10, 2025, Columbia University, New York, NY, USA*, volume 325 of *LIPIcs*, pages 1:1–1:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ITCS.2025.1.

4 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*, volume 13747 of *Lecture Notes in Computer Science*, pages 384–414. Springer, 2022. doi:10.1007/978-3-031-22318-1_14.

5 Ittai Abraham, Naama Ben-David, and Sravya Yandamuri. Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 381–391, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3519270.3538426.

6 Ittai Abraham, T.-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. *Distributed Comput.*, 36(1):3–28, 2023. doi:10.1007/s00446-022-00428-8.

7 Jinyuan Chen. Optimal error-free multi-valued byzantine agreement. In *DISC 2021*, volume 209 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.17.

8 Jinyuan Chen. Ociormvba: Near-optimal error-free asynchronous mvba, 2024. doi:10.48550/arXiv.2501.00214.

9 B. A. Coan and R. Turpin. Extending binary byzantine agreement to multivalued byzantine agreement. Technical report, Massachusetts Institute of Technology, USA, 1984.

10 Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous data dissemination and its applications. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721. ACM, 2021. doi:10.1145/3460120.3484808.

11 Sourav Das, Zhuolun Xiang, and Ling Ren. Balanced quadratic reliable broadcast and improved asynchronous verifiable information dispersal. *IACR Cryptol. ePrint Arch.*, page 52, 2022. URL: https://eprint.iacr.org/2022/052.

12 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 1982*, pages 132–140. ACM, 1982. doi:10.1145/800220.806690.

13 Mose Mizrahi Erbes and Roger Wattenhofer. Extending asynchronous byzantine agreement with crusader agreement, 2025. doi:10.48550/arXiv.2502.02320.

14 Hanwen Feng, Zhenliang Lu, Tiancheng Mai, and Qiang Tang. Faster hash-based multi-valued validated asynchronous byzantine agreement. Cryptology ePrint Archive, Paper 2024/479, 2024. to apper in DSN 25. URL: https://eprint.iacr.org/2024/479.

15 Matthias Fitzi and Martin Hirt. Optimally efficient multi-valued byzantine agreement. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 163–168, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1146381.1146407.

16 Venkatesan Guruswami. Algorithmic results in list decoding. *Found. Trends Theor. Comput. Sci.*, 2(2):107–195, January 2007. doi:10.1561/0400000007.

17 Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, pages 129–138, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3382734.3405707.

18 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with t < n/3, o(n2) messages, and o(1) expected time. *J. ACM*, 62(4), September 2015. doi:10.1145/2785953.

19 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang. Improved Extension Protocols for Byzantine Broadcast and Agreement. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2020.28.

20 Madhu Sudan. Decoding of reed solomon codes beyond the error-correction bound. *J. Complex.*, 13(1):180–193, March 1997. doi:10.1006/jcom.1997.0439.

A Omitted Proofs for Section 2

▶ **Theorem A.1** (Theorem 2.6, restated). *Protocol 2.3 is a reliable agreement protocol tolerating $t < n/3$ corrupted parties. It requires $O(nL + n^2 \log N)$ communication for an input of size L.*

Proof. We show that each one of the properties hold:

- **Termination:** A party terminates only after the dissemination terminates. As guaranteed by dissemination, if one party terminates, then all honest parties eventually terminate.
- **Validity:** If all honest parties enter the protocol with the same input $F(x)$, then dispersal is guaranteed to terminate, and at least $t + 1$ honest parties terminate with the same output $F(x)$, and the rest terminate with $\perp$. The validity of dissemination then guarantees that all honest parties terminate and output $F(x)$.
- **Agreement:** Assume an honest party terminates the protocol with output $F'(x)$. This implies that dispersal must have terminated. Dispersal guarantees weak agreement – if an honest party terminates, then at least $t + 1$ honest parties terminates with the same polynomial $F''(x)$, and the rest might terminate with either $F''(x)$ or $\perp$. The parties then run dissemination, which then guarantees that all honest parties terminate with the output $F''(x)$. It thus hold that $F''(x) = F'(x)$, and that all honest parties must terminate with $F'(x)$. ◀

B Omitted Proofs for Section 3

▶ **Theorem B.1** (Theorem 3.2, restated). *Protocol 3.1 is a multivalued byzantine agreement protocol, tolerating t corrupted parties.*

Proof. We show that each one of the properties hold.

Validity. If all honest parties start with the same input $F(x)$, then:

- The validity property of BOOST (Definition 3.1) guarantees that no honest party triggers $\mathsf{detect}_i = 1$; moreover, all honest parties must eventually set $\mathsf{output}_i = F(x)$.

- The validity property of data dissemination then guarantees that all parties eventually set $f_i^{(2)}(x) = F(x)$.
- The validity reliable agreement guarantees that all honest parties must receive $F(x)$ as output.
- We get that no honest party enters 0 to the Byznatine Agreement, as no honest party ever set $\mathsf{BOOST.detect}_i = 1$. All honest parties start the BA with the same input 1. The validity property of BA then guarantees that all honest parties must output 1.
- All honest parties output the output of the second dissemination, i.e., $F(x)$.

Termination. To show termination, we show that the parties must eventually enter the BA. To show that the parties must eventually enter the BA, we have three cases to consider:

1. If all honest parties hold the same input $F(x)$, then as guaranteed from the validity property, all terminate, and in particular also enter the BA.
2. If there is no set of $t+1$ honest parties with the same input $F(x)$, then the detect property of BOOST guarantees that all honest parties eventually activate detect. Therefore, all honest parties must eventually enter the BA (some might enter earlier with different input).
3. If there is a set of $t+1$ honest parties with the same input $F(x)$, then the detect or correct property of BOOST guarantees that either:
   a. **Detect:** Eventually, all honest parties activate their detect output tape. In that case, parties have input to the BA.
   b. **Correct:** At least $t+1$ honest parties set their output tape to $F(x)$ and the remaining to proceed. The validity of dissemination (Definition 2.3) property then guarantees that all honest parties must terminate with output $F(x)$. The protocol then proceeds to reliable agreement, which now its validity guarantees that all output $F(x)$. We conclude that all honest parties eventually can enter the BA with 1.

Agreement. We show that all honest parties must have the same output. In particular, this follows from BA: all honest parties must receive the exact same output from BA. There are two cases:

- If the output is 0, then all honest parties terminate and output $\perp$.
- If the output is 1, then there exists an honest party that inputs 1 to the BA. A party inputs 1 to the BA only if its reliable agreement terminated. If an honest party terminates the reliable agreement, then all honest parties would eventually terminate the reliable agreement with some output $F'(x)$, then eventually all honest parties terminate with output $F'(x)$. ◀

C Omitted Proofs for Section 4

▶ **Theorem C.1** (Theorem 4.1, restated)**.** *Protocol 4.1 is a* BOOST *protocol (as per Definition 3.1) where validity and set output hold unconditionally (with probability 1), and the Detect or correct and detect properties hold with probability* $1 - \frac{n^3}{|\mathbb{F}|}$.

Proof. We show that the protocol satisfies all those properties.

Validity. If all honest parties enter with the same polynomial $F(x)$, then no matter what challenge each party chooses, the parties will agree with each other. Each party P_i might add to DA_i only corrupted parties, and thus $|\mathsf{DA}_i| \leq t$. Each party never sends `support` message on a value that is not on F, and each party sees $2t+1$ support messages to points on F. Therefore, each party P_i sends $(\mathsf{YourPoint}, F(C_i[k]))$ to each P_k. Each P_k receives

$2t+1$ YourPoint messages with the same value, and therefore sends $(\mathsf{MyPoint}, r_k, F(r_k))$ to all. Each party will eventually see $2t+1$ points on $F(x)$, and the unique decoding is exactly $F(x)$. Therefore, each party eventually sets $g_i(x) = F(x)$, and send `HaveOutput` message.

However, parties might set output before setting $g_i(x) = F(x)$. A party sets its output when receiving $2t+1$ `done` messages. This implies that $t+1$ honest parties sent a `done` message. A party sends a `done` message if it received $t+1$ `done` messages, or if it received $2t+1$ `HaveOutput` messages. In the latter case, $t+1$ sent `HaveOutput` messages; In the former case, this implies that at least one honest party initiated a `done` message – which again implies that there are $t+1$ honest parties that sent `HaveOutput` message. We showed that all honest parties eventually set $g_i(x) = F(x)$, send `HaveOutput` message, and therefore when a party sets its output, there are at least $t+1$ honest parties that already set $g_i(x) = F(x)$, and the rest would set their output to proceed.

We also claim that no honest party send detect message. An honest party P_j might send such a message in the following cases:

1. $|\mathsf{DA}_j| \geq t+1$. However, since all honest parties agree with their exchange messages, a party might add to DA_j only corrupted parties, and therefore $|\mathsf{DA}_j| \leq t$.
2. A party sends a detect message if it receives $t+1$ messages $(\texttt{support}, r_j, v_j)$ such that $C_i[j] \neq r_j$ or $v_j \neq f_i(r_j)$. However, an honest party P_i sends `support` only after seeing at least $t+1$ `Exchange` messages with the same (r_i, u_i). Since all honest parties have the same input, the only value that can reach $t+1$ cardinality is a value on F. Therefore, the only $(\texttt{support}, r_i, v_i)$ for which $C_j[i] \neq r_i$, or $v_i \neq f_j(r_i)$ are messages from corrupted parties, and therefore it can receive at most t of those.
3. An honest party might send detect upon receiving $t+1$ messages $(\mathsf{YourPoint}, u)$ with the same u, but $u \neq f_j(r_j)$. As previously, honest parties send YourPoint messages only on values on F, and therefore no other value can reach cardinality $t+1$.
4. Once again, parties add i to DA_j if they receive $(\mathsf{MyPoint}, r_i, u_i)$ for which $f_j(r_i) \neq u_i$. However, since all honest parties send MyPoint only on values on F, DA_j never reaches cardinality $t+1$.

We conclude that no honest party ever sends detect message.

Set output. An honest party sets its output only after receiving `done` messages from $2t+1$ distinct parties. This implies that $t+1$ honest parties must send `done` messages. Those parties send that message to all parties, and therefore each honest party must eventually receive $t+1$ `done` messages, and thus also send `done` messages. As a result, each honest party must eventually receive `done` messages from $2t+1$ distinct parties, and set its output.

Bad event. Before proceeding to show the detect or correct property that and the detect property, we first define a bad event. let $\mathsf{Bad}_{i,j}$ denote the event in which P_i and P_j hold input $f_i(x) \neq f_j(x)$, respectively, but P_i chooses r_i such that $f_i(r_i) = f_j(r_i)$. In that case, P_i might not recognize that P_j does not hold the same polynomial. We can bound $\Pr[\mathsf{Bad}_{i,j} = 1] \leq \frac{d}{|\mathbb{F}|}$. This is because the two polynomials agree on d points, and P_i chooses its challenge uniformly at random. Moreover, let Bad denote the event in which there exists some i, j for which $\mathsf{Bad}_{i,j}$ occurs. We have that

$$\Pr[\mathsf{Bad} = 1] \leq \sum_{i,j} \Pr[\mathsf{Bad}_{i,j} = 1] \leq \frac{n^2 d}{|\mathbb{F}|} \leq \frac{n^3}{|\mathbb{F}|} .$$

We note that here we assume that the adversary must choose the input of all honest parties before the protocol starts. To handle adaptive inputs, a variant with additional rounds are needed (see [1]). Since our goal with statistical security is as a warm-up to the perfect security case, we defer this variation to a full version. We proceed with the analysis assuming that Bad does not occur.

Detect or correct. We assume that $t+1$ honest parties start with the same input $F(x)$. We show that, unless there is a detection, at least $t+1$ honest parties set their output to $F(x)$, and the rest set their output to proceed.

There are two cases to consider:

1. Suppose some honest party P_i receives $2t+1$ messages $(\texttt{support}, r_i, u_i)$ such that P_i previously sent $(\texttt{challenge}, r_i)$ and $u_i \neq F(r_i)$. Then, we show that all honest parties eventually detect.

   Specifically, if an honest party sees $2t+1$ such messages, then $t+1$ honest parties sent $(\texttt{support}, r_i, v_i)$ to all. All honest parties would see those messages, specifically also the parties that hold $F(x)$. Due to Step 2e, they would see $t+1$ values v_i for which $v_i \neq F(r_j)$, and thus would send detect to all. Since $t+1$ parties hold $F(x)$, $t+1$ honest parties send detect, which leads to all parties detect.
2. No honest parties gets $2t+1$ support on a value $F'(r_i) \neq F(r_i)$. Clearly, all honest parties must receive $2t+1$ messages $(\texttt{support}, r_i, F(r_i))$: Each honest party sends $(\texttt{challenge}, r_i)$ to all parties; it would receive from the parties in F the same value $(\texttt{Exchange}, F(r_i))$, and thus send $(\texttt{support}, r_i, F(r_i))$ to all. The $t+1$ honest parties holding F would eventually receive $2t+1$ messages $(\texttt{support}, r_i, F(i))$ (with difference indices i), and therefore all would send to each P_i $(\mathsf{YourPoint}, F(r_i))$. According to our assumption, no other value receives $2t+1$, therefore, all messages honest parties send are on F. Each party would receive $t+1$ $(\mathsf{YourPoint}, F(r_i))$ with the same value, and therefore would send $(\mathsf{MyPoint}, r_i, F(r_i))$ to all. The only polynomial that can be decoded is $F(x)$.

Detect. We show that if there is no set of $t+1$ honest parties that hold the same polynomial, and assuming that Bad does not occur, then all honest parties set $\mathsf{detect}_i = 1$. Since there is no common input for $t+1$ honest parties, each honest party must hold a different input than at least $t+1$ parties. Assuming that Bad does not occur, each party P_i adds at least $t+1$ indices to its DA_i set, and eventually send detect message. All honest parties eventually send detect, and activate $\mathsf{detect}_i = 1$. ◀

D Omitted Proofs for Section 5

For the proofs of section 5, please refer to the full version of this paper [2].