

Revisiting Malicious Private Aggregation: Formalization, Weaknesses, and Enhancements

Ananya Appan, Pranav Shriram Arunachalam, David Heath, and Ling Ren

University of Illinois at Urbana-Champaign
{psa3,aappan2,daheath,renling}@illinois.edu

Abstract. Private aggregation schemes enable aggregation of sensitive data across clients without leaking any individual client’s input. Their applications include privacy-preserving federated learning, private heavy hitters, and anonymous systems.

Many private aggregation schemes assume two non-colluding servers and aim to guarantee privacy even when one server is malicious, i.e., the malicious server learns an aggregation result that includes *all* honest client inputs. However, many private aggregation schemes in the literature fail to achieve privacy, which we believe is in large part due to a lack of formalism for the nuanced variations of privacy guarantees.

In this work, we present ideal functionalities for several variants of private aggregation. We also formalize a common paradigm underlying most prior private aggregation schemes. The formal treatment helps us identify security flaws or underspecifications in existing private aggregation schemes. We then present modular modifications or fill in critical details to help prior schemes in the share-aggregate paradigm securely realize the private aggregation functionalities we formally define, including in settings where clients may have unreliable networks.

1 Introduction

With the widespread adoption of cloud computing, applications that *crowdsource* data from clients have become commonplace. Ideally, these applications should operate on sensitive client data while keeping individual client data hidden. To protect client inputs, many prior works design such crowdsourcing applications with privacy-preserving protocols among n clients and a set of non-colluding servers. At the end of the protocol, the servers compute a function f of clients’ inputs, but learn nothing beyond that. In particular, the servers do not learn individual client inputs.

We refer to such protocols as *Private Aggregation* protocols. Private aggregation protocols have been proposed for various application scenarios, such as privacy-preserving collection of aggregate statistics [13,28], privacy-preserving federated learning [27,5], private heavy hitters [7,4], and anonymous broadcast [12,19,26,14,22,15]. They have also been deployed in practice by Apple [2], Google [1], and Mozilla [3].

Many private aggregation protocols in the literature, and those that we consider in this work, assume *two non-colluding servers*, and aim to guarantee some level of client privacy even when one server maliciously deviates from the protocol. Among these, most schemes [13,26,7,19,28] claim, often informally, to achieve the following guarantees:

- *Correctness*: When both servers are honest, they output an aggregation of *all* honest client inputs.
- *Privacy*: Even when one server is malicious, the malicious server learns nothing beyond an aggregation computed on all honest clients’ inputs.

A few works [27,24] explicitly target a weaker privacy notion: the malicious server may learn the aggregation of a “large enough” subset of (as opposed to all) honest clients’ inputs. We refer to this variant as *threshold private aggregation*.

Despite the development of numerous elegant techniques in a significant body of prior work, a rigorous definition of the private aggregation problem remained lacking. Most prior works also lack a formal network model and implicitly assume reliable and timely communication from all clients to both servers. In practice, however, some clients may have unreliable networks that can drop or delay messages. (Some prior works touched on this issue [13,28,24], but only briefly and informally.)

The lack of rigorous problem definitions and formal modeling has led to inconsistencies, security flaws, and underspecifications in the private aggregation literature, some of which were (implicitly or explicitly) identified in prior works [26,21]. In this work, we formalize and systematize the area of private aggregation. We apply our formal framework to identify and rectify inconsistencies in prior works, and to establish new feasibility and infeasibility results.

1.1 Our Contributions

Defining and systemizing prior protocols. We formalize, via ideal functionalities, various notions of private aggregation that prior works attempt to achieve. Almost all prior works [13,28,27,7,19,26,14] follow the same paradigm, which we call the *Share-Aggregate* paradigm: each client *encodes* and secret-shares its input to the two servers, and the servers validate and aggregate those shares. Analyzing protocols within the Share-Aggregate paradigm lets us pinpoint why prior works fail.

As an instructive example of the Share-Aggregate paradigm, consider a setting in which each client is allowed to submit a “vote” for one of n distinct choices, and the servers wish to aggregate a histogram of the clients’ votes. An efficient protocol might ask each client to encode its vote as a size- n one-hot vector with a single “1” and to additively secret-share it across the two servers. To prevent malicious clients from voting for more than one option, or voting for their preference more than once, the servers must (obviously) validate that the shares are indeed of a one-hot vector with a single “1”. Once validated, it is relatively straightforward for the servers to sum the vector shares and reconstruct the output.

Table 1. Summary of our results.

Protocol	Setup	Client Network	Possible?
VESS and PA	PKI	Synchronous	✓ (§3.3)
VESS	None	Synchronous	✗ (§3.4)
PA	PKI	Intermittent	✓ (§4)
Threshold PA	None	Intermittent	✓ (§5)

Revisiting existing PA schemes. We observe that prior works [13,26,7,19,28] that aim to achieve PA fail to do so because, while validating a client’s secret-shared input, they allow a malicious server to reject an honest client’s shares *without being detected*. This allows the adversary to remove arbitrary numbers of honest clients from the aggregation protocol, and learn the aggregation of a chosen *subset* of honest clients’ inputs.

Towards fixing prior works, we first identify the right ideal functionality that is required for the share phase to construct secure PA protocols. We call this functionality *Validated Encoded Secret Sharing* (VESS), which enables servers to verify that the received shares correspond to a *valid encoding* of the client’s input, *without* allowing a malicious server to reject honest clients’ shares.

We then give a modular fix to prior works by constructing a secure VESS, and thereby turning prior schemes [13,28,7,19,26] into secure PA solutions. Roughly, our VESS arranges that an honest client *can detect and identify* the malicious server when it attempts to reject the honest client’s shares. The client can then work with the honest server to ensure its input is included.

Our fix uses a three-party broadcast functionality [16] (among each client and the two servers), which typically requires digital signatures and a public-key infrastructure (PKI) [20]. We note that prior works in the Share-Aggregate paradigm rely solely on symmetric-key cryptography [13,28,7,19,26], and deliberately avoid public-key cryptography and PKI for efficiency. We show that these efforts were bound to fail. Specifically, we show that VESS, with the forms of client encodings considered in prior works, *implies a solution to the three-party Byzantine broadcast* problem [23], which cannot be solved without some trusted setup or digital signatures [23,20].

Achieving PA with unreliable networks. The previous PA schemes [13,28,7,19,26], even after our security fix, require all honest clients to have reliable networks. We refer to clients with unreliable networks as *intermittent clients*. Some prior works [13,28] claim to achieve PA with intermittent clients, but upon close inspection, their schemes only achieve threshold PA. Another prior work [24] implicitly hints that PA is impossible with intermittent clients. We show that this intuition is *not* correct. We present a natural adaptation of the PA functionality that ensures the aggregation includes the inputs of *all* honest clients with reliable networks. We then present a method to convert any PA scheme that does not allow intermittent clients into one that does.

Achieving threshold PA. Since PA comes with trade-offs in efficiency and assumptions, we next revisit *threshold PA*. Some works [13,27,24] have outlined how PA schemes in the Share-Aggregate paradigm can be modified into threshold PA, but their descriptions are missing critical details. For example, prior works do not specify the fraction of corrupt clients, but we show that it is important that only a minority of clients are corrupt. We describe the modifications in detail, state the required conditions, and prove that the modified protocol achieves threshold PA.

2 Towards Defining Private Aggregation

2.1 The Model

Setting. We focus on private aggregation protocols among two non-colluding servers and n clients. We use S_0 and S_1 to denote the two servers. We use $\mathcal{C} = \{1, 2, \dots, n\}$ to denote the set of clients. The i -th client has an input x_i .

We denote the aggregation function by f . f can take any number of inputs from a domain $\mathcal{I}$, and outputs a value in a range $\mathcal{R}$ (which could depend on the number of inputs), i.e., $f : \mathcal{I}^* \rightarrow \mathcal{R}$.

Threat Model. We consider a computationally-bounded, malicious adversary $\mathcal{A}$ that statically corrupts up to one server and any number of clients. We use $\mathcal{C}_c \subseteq \mathcal{C}$ to denote the set of corrupt clients and $\mathcal{C}_h = \mathcal{C} \setminus \mathcal{C}_c$ to denote the set of honest clients. The corrupted server and corrupted clients may collude and maliciously deviate from the prescribed protocol. We use λ to denote the security parameter.

Network Model. We assume that communication between the two servers is synchronous and reliable, meaning that messages between servers are guaranteed to arrive within a bounded delay. Client-to-server communication, on the other hand, can be either *synchronous* or *intermittent*, depending on the protocol. *Synchronous clients* have good network connections; messages sent by synchronous clients arrive at the servers within a bounded delay. *Intermittent clients* have unstable network connections; messages sent by intermittent clients may be delayed arbitrarily or dropped from the network by the adversary.

Next, we formally define private aggregation by presenting ideal functionalities [10,25]. The ideal functionalities capture what parties learn in an ideal world in which all computation is performed by a trusted party.

2.2 Strong Private Aggregation

We first define *Strong Private Aggregation* (Strong PA). This is a natural definition inspired by the standard definition of *secure two-party/multi-party computation with abort* (2PC/MPC) [29]. We note that strong PA is *stronger* than what prior work (and this work) targets. We define it primarily to contrast it with weaker PA definitions we introduce later in this paper.

Functionality $\mathcal{F}_{2PC}$
Setup: The functionality $\mathcal{F}_{2PC}$ interacts with two servers S_0, S_1 who agree on a function f. An ideal-world adversary $\mathcal{A}$ may corrupt up to one server. Input: Obtain X_0 from S_0 and X_1 from S_1. Compute: Set $y = f(X_0, X_1)$. Output: 1. If $\mathcal{A}$ corrupts a server, send y to $\mathcal{A}$, and receive “continue” or “abort” from $\mathcal{A}$. If $\mathcal{A}$ sends “abort”, set $y = \text{“abort”}$. 2. Send y to the honest servers.

Fig. 1. The standard functionality of secure two-party computation with abort.

Functionality $\mathcal{F}_{SPA}$
Setup: The Strong PA functionality $\mathcal{F}_{SPA}$ interacts with n clients in $\mathcal{C} = \{1, \dots, n\}$ and two servers S_0 and S_1. Parties agree on a function f. An ideal-world adversary $\mathcal{A}$ may corrupt up to one server and any set $\mathcal{C}_c$ of clients. $\mathcal{C}_h = \mathcal{C} \setminus \mathcal{C}_c$ denotes the set of honest clients. Input: 1. Obtain x_i from each client i. 2. Obtain $\mathcal{C}'_c \subseteq \mathcal{C}_c$ from the adversary. Let $\mathcal{C}' = \mathcal{C}'_c \cup \mathcal{C}_h$. Compute: Compute $y = f(\{x_i\}_{i \in \mathcal{C}'})$. Output: (same as $\mathcal{F}_{2PC}$) 1. If $\mathcal{A}$ corrupts a server, send y to $\mathcal{A}$, and receive “continue” or “abort” from $\mathcal{A}$. If $\mathcal{A}$ sends “abort”, set $y = \text{“abort”}$. 2. Send y to the honest servers.

Fig. 2. The strong private aggregation functionality.

Figure 2 specifies the Strong PA ideal functionality. Observe that strong PA is almost identical to the 2PC functionality (Figure 1). Just like 2PC, the strong PA functionality ensures that even if one server is malicious, the honest server must either output a correct result or abort. A correct result is one that aggregates all honest clients’ inputs (plus a subset of corrupt clients’ inputs). The corrupt server cannot learn anything beyond that result (privacy), and cannot make the honest server output anything other than that result (correctness). The only difference between 2PC and strong PA is whether the inputs come from the two servers themselves or the many clients.

This difference also highlights a key challenge to constructing strong PA. In particular, naïvely achieving strong PA using 2PC techniques is costly, as it seems necessary for each server to prove to the other server that it performed all computations correctly, i.e., consistently with client inputs. To our knowledge, no practical scheme targeting strong PA has been proposed.

Functionality $\mathcal{F}_{\text{PA}}$
Setup: The PA functionality $\mathcal{F}_{\text{PA}}$ interacts with n clients in $\mathcal{C} = \{1, \dots, n\}$ and two servers S_0 and S_1. Parties agree on functions g and Encode. An ideal-world adversary $\mathcal{A}$ may corrupt up to one server and any set C_c of clients. $C_h = \mathcal{C} \setminus C_c$ denotes the set of honest clients. Input: (same as $\mathcal{F}_{\text{SPA}}$) 1. Obtain x_i from each client i. 2. Obtain $C'_c \subseteq C_c$ from the adversary. Let $C' = C'_c \cup C_h$. Compute: 1. If $\mathcal{A}$ corrupts a server, then for each client $i \in C'$, obtain additive error e_i from $\mathcal{A}$ and set $z_i = \text{Encode}(x_i) + e_i$. Else, set $z_i = \text{Encode}(x_i)$ for all $i \in C'$. 2. Compute $y = g(\{z_i\}_{i \in C'})$. Output: (same as $\mathcal{F}_{\text{SPA}}$) 1. If $\mathcal{A}$ corrupts a server, send y to $\mathcal{A}$, and receive “continue” or “abort” from $\mathcal{A}$. If $\mathcal{A}$ sends “abort”, set $y = \text{“abort”}$. 2. Send y to the honest servers.

Fig. 3. The private aggregation functionality. The input and output parts are identical to those of SPA. Only the compute part differs from SPA.

Functionality $\mathcal{F}_{\text{TPA}}$
Setup: Same as $\mathcal{F}_{\text{PA}}$, except that there is an additional parameter n_q. Input: 1. Obtain x_i from each client i. 2. Obtain $C'_c \subseteq C_c$ from the adversary. 3. If $\mathcal{A}$ corrupts a server, obtain $C'_h \subseteq C_h$ from $\mathcal{A}$ such that $C'_h \geq n_q$, and set $C' = C'_c \cup C'_h$. Else, set $C' = C'_c \cup C_h$. Compute: Same as $\mathcal{F}_{\text{PA}}$ Output: Same as $\mathcal{F}_{\text{PA}}$

Fig. 4. The threshold private aggregation functionality.

2.3 Private Aggregation (PA)

To achieve practical efficiency, many prior works on private aggregation [13,26,7,19,28] only guarantee privacy and sacrifice correctness when a server is malicious. Further, they focus on aggregation functions that can be expressed as follows:

$$f(x_1, x_2, \dots, x_n) = g(\text{Encode}(x_1), \text{Encode}(x_2), \dots, \text{Encode}(x_n)),$$

where **Encode** is some (usually non-linear) encoding of a client input and g is some simple and efficient aggregation function (e.g., summation). For instance, [7] computes the histogram function by having **Encode**(x) output a one-hot vector with 1 at position x , and g as a simple summation of these one-hot vectors.

As another example, [13] computes the variance $f = E[x^2] - (E[x])^2$ of client inputs by using $\text{Encode}(x) = (x, x^2)$ and g to perform a *pairwise sum*.¹ In both examples, a nontrivial function f is computed with the help of appropriately chosen client encodings, while g simply sums up these encodings.

Figure 3 formalizes the Private Aggregation (PA) functionality $\mathcal{F}_{\text{PA}}$ that these prior works set out to achieve. While g in prior work is typically a linear function (most often just a simple sum), we allow g to be an arbitrary function to keep the $\mathcal{F}_{\text{PA}}$ functionality general-purpose.

$\mathcal{F}_{\text{PA}}$ differs from $\mathcal{F}_{\text{SPA}}$ by allowing a malicious server to introduce an additive error to each client’s encoded input before aggregation. This allows a malicious server to *change* the aggregation result to one that includes these errors, weakening correctness. This may also weaken privacy as the malicious server learns something other than g applied on encoded client inputs. When the function g is linear, as is the case in most prior works, the ability to add errors does not help the adversary learn anything beyond g applied to encoded client inputs, so *privacy* is not weakened.

Due to the relaxed guarantees, one must be careful when deploying a PA scheme. For instance, consider a private aggregation scheme used for clients to report malicious URLs [7]. A corrupt server could *decrease* the number of times a malicious URL is reported. Nevertheless, by targeting a weaker ideal functionality, PA admits techniques that are considerably simpler and more efficient. This makes PA appealing to practical applications, especially in scenarios where incorrect aggregation results may be detectable by out-of-band means.

2.4 Threshold Private Aggregation

Some prior works [27,24] set out to achieve what we call *Threshold Private Aggregation* (threshold PA). We define the threshold PA ideal functionality in Figure 4. This definition weakens $\mathcal{F}_{\text{PA}}$ by allowing a corrupt server to change the aggregation to include a large enough *subset* of honest client inputs (with additive errors on encoded client inputs, see the compute phase of Figure 3). The size of this honest client subset is at least n_q , where n_q is a parameter. PA can be understood as a special case of threshold PA where n_q is equal to the number of honest clients $|\mathcal{C}_h|$.

2.5 Constructions in Prior Works

Prior works [13,26,7,19,28] follow a generic paradigm which we call the *Share-aggregate* paradigm. This consists of two phases: a share phase during which the servers obtain secret-shares of an encoding of each client’s input, and an

¹ Note that this leaks the sum of the inputs and the sum of *squares* of the inputs to the servers. Prior work considers this acceptable leakage for this task. Note that we can get rid of this leakage without much additional cost by using a 2PC protocol in the aggregate phase (the only non-linear operation that the 2PC performs is a single multiplication).

Primitive ESS
Let $\text{Encode} : \mathcal{I} \rightarrow \mathbb{G}$ be an efficiently computable function with the following property: given a value z in the range of Encode, it is easy to find a value x such that $\text{Encode}(x) = z$. An encoded secret sharing scheme is a pair of algorithms $(\text{DistrEncode}, \text{Expand})$ with the following syntax: – DistrEncode is a randomized algorithm that takes as input a value $x \in \mathcal{I}$ and outputs a pair of keys k_0, k_1. – Expand is a deterministic algorithm that takes as input a key k_i and outputs a share $z_i \in \mathbb{G}$. Correctness: $\Pr \left[z_0 + z_1 = \text{Encode}(x) \mid \begin{array}{l} (k_0, k_1) \leftarrow \text{DistrEncode}(x), \\ z_0 = \text{Expand}(k_0), z_1 = \text{Expand}(k_1) \end{array} \right] \geq 1 - \text{negl}(\lambda)$ Security: For any pair of inputs $x, y \in \mathcal{I}$ and all $b \in \{0, 1\}$, the following ensembles are computationally indistinguishable $\{ k_b \mid (k_0, k_1) \leftarrow \text{DistrEncode}(x) \} \approx \{ k_b \mid (k_0, k_1) \leftarrow \text{DistrEncode}(y) \}$

Fig. 5. Definition of the Encoded Secret Sharing primitive

Functionality $\mathcal{F}_{\text{ZKSD}}$
Setup: A ZKSD functionality $\mathcal{F}_{\text{ZKSD}}$ interacts with a client, and two servers (S_0 and S_1). The parties agree on a predicate $\text{Valid} : \mathbb{G} \rightarrow \{0, 1\}$. An ideal-world adversary $\mathcal{A}$ may corrupt either the client or a single server. Input: 1. Receive shares z_0 and z_1 from the servers. 2. Receive z such that $\text{Valid}(z) = 1$ from the client. Compute: Set v to 1 if z equals $z_0 + z_1$ and to 0 otherwise. Output: 1. If $\mathcal{A}$ corrupts a server or the client, receive “invalid” or “continue” from $\mathcal{A}$. If $\mathcal{A}$ sends “invalid”, set $v = 0$. 2. Send v to the servers.

Fig. 6. The zero-knowledge proofs on secret-shared data functionality.

aggregation phase during which the servers compute an aggregation of these encoded inputs.

The share phase from prior works proceeds as follows: First, each client additively secret-shares its encoded input to the two servers. The client could do this by locally computing $\text{Encode}(x)$ and creating additive shares of it, and then simply sending the shares to the two servers. But there may be more efficient ways to do this, especially if $\text{Encode}(x)$ is much larger than x . For instance, if Encode outputs one-hot vectors, the client can use distributed point functions [9] to reduce communication. We capture both methods (and potentially others) with what we call *encoded secret sharing* (ESS) scheme. The client computes a pair of

Protocol $\Pi_{\text{insecure-Share}}$	
1.	The client computes $(k_0, k_1) \leftarrow \text{ESS.DistrEncode}(x)$. The client sends k_0 to S_0 and k_1 to S_1 . Each server S_i computes $z_i = \text{ESS.Expand}(k_i)$.
2.	The client and the servers invoke $\mathcal{F}_{\text{ZKSD}}$ with server inputs z_0 and z_1 and client input $z = \text{Encode}(x)$. If S_i gets 1 from $\mathcal{F}_{\text{ZKSD}}$, it outputs z_i ; otherwise, it discards the client shares.

Fig. 7. An insecure protocol, from prior works, for a client to share the encoding of its input.

short keys k_0, k_1 (using a function called `DistrEncode`) for its input x , and sends k_i to S_i . Each server S_i then expands k_i to an additive share x'_i of $\text{Encode}(x)$ (using a function called `Expand`). We define ESS formally in Figure 5.

A malicious client may secret share a value that is *not* a valid encoding (e.g., not a one-hot vector). Hence, the servers need to verify the shares they receive. They do this by invoking a Zero-knowledge proof on Secret-shared Data [13,6] (ZKSD) protocol, captured by the functionality $\mathcal{F}_{\text{ZKSD}}$ (defined in Figure 6), which allows honest servers to verify that the client indeed secret-shared an encoding of some value.

Specifically, $\mathcal{F}_{\text{ZKSD}}$ receives shares from the servers. Further, it receives an encoded input from the client, i.e., it obtains a value z from the client that satisfies a predicate $\text{Valid} : \mathbb{G} \rightarrow \{0, 1\}$; Valid returns 1 when applied to encoded values, i.e., $\text{Valid}(z) = 1$ if there exists some $x \in \mathcal{I}$ such that $\text{Encode}(x) = z$, and 0 otherwise.

When both servers are honest, $\mathcal{F}_{\text{ZKSD}}$ checks whether the servers' shares sum up to the encoded client input, and sends 1 to the servers if this holds. If a malicious client sends shares to servers that sum up to a non-encoding, $\mathcal{F}_{\text{ZKSD}}$ sends 0 to the servers, and the servers discard their shares for the client.

Figure 7 describes the share phase protocol executed by each client and the two servers. After all clients' shares are verified, the servers aggregate (remaining) client shares by invoking $\mathcal{F}_{2\text{PC}}$ (Figure 1).

3 Revisiting Private Aggregation

In this section, we first describe why protocols from prior works [13,26,7,19,28] fail to achieve PA. We then give a generic approach to achieve PA in the Share-aggregate paradigm. Using this approach, we fix the insecure protocols from prior works [13,26,7,19,28] to achieve PA.

3.1 Why Prior Works Fail to Achieve PA?

The security flaw. Recall the PA construction of prior protocols in Section 2.5. The ZKSD protocol gives no guarantee of *correctness when a server is corrupt*. Observe from the ZKSD functionality (Figure 6) that, even when the client is

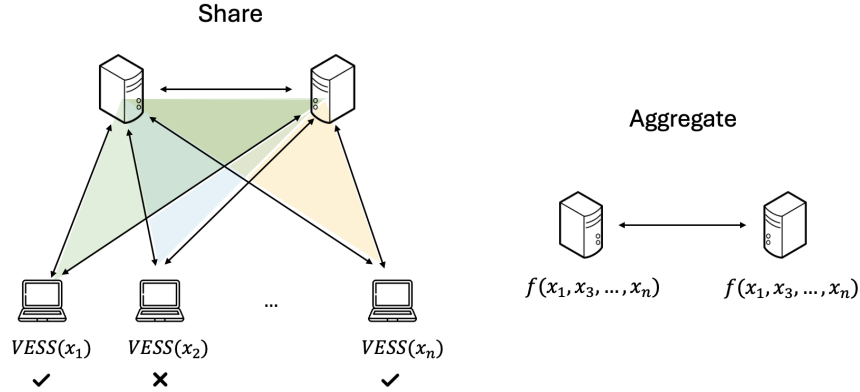


Fig. 8. Achieving PA in the Share-Aggregate Paradigm. The Share phase involves n instances of VESS, one for each client. During the Aggregate phase, servers aggregate shares obtained from VESS instances in which clients were honest.

honest, a corrupt server can participate with an arbitrary share and make the other server output 0; this discards an honest client’s share. When the servers aggregate the remaining client shares, the adversary can learn the aggregation of a *subset* of honest client inputs, thereby violating the privacy guarantee of PA as defined in Figure 3.

Many prior works [13,26,7,19,28,14] suffer from this problem. The “correctness” of the verification protocol used in Prio’s [13, §4.1] share phase holds only when all parties are honest; a malicious server can cause the other server to reject an honest client’s input. This same verification protocol was later formalized and given the name zero-knowledge proofs of secret-shared data (ZKSD) in [6]. Express [19, §4.1] states that they use a verification protocol that provides the same guarantees as that in Prio [13]. Whisper [28] also uses the same ZKSD definition formalized in [6].

Later works recognize this issue, but settle for weaker guarantees than PA. In Poplar [7, §2.3], the authors admit that the verification protocol they use allows a malicious server to “disqualify” honest clients, and present a differentially private mechanism to counter this. Spectrum [26] rightly observes that the verification protocols in Express [19] and Riposte [14] allow a malicious server to discard honest client inputs. Their fix [26, §4.3] allows a malicious server to learn individual inputs of honest clients, and catches the malicious server only after the privacy violation. Incidentally, in an attempt to tolerate intermittent clients, Prio [13, §7] briefly discusses a protocol that achieves threshold PA. While this protocol is correct, it downgrades security to threshold PA.

Protocol Π_{PA}
1. Each client $c \in \mathcal{C}$ invokes an instance of $\mathcal{F}_{\text{VSS}}$ with the two servers. If any instance of $\mathcal{F}_{\text{VSS}}$ outputs “blame S_i”, S_{1-i} aborts. 2. Let Z_0 (resp. Z_1) denote S_0’s (resp. S_1’s) list of additive shares obtained in step (1), i.e., from $\mathcal{F}_{\text{VSS}}$ instances that output shares. The servers S_0, S_1 invoke the $\mathcal{F}_{\text{ZPC}}$ functionality (Figure 1) to compute function $g(Z_0 + Z_1)$ and output the result.

Fig. 9. Achieving PA in the Share-Aggregate paradigm

Functionality $\mathcal{F}_{\text{VSS}}$
Setup: $\mathcal{F}_{\text{VSS}}$ interacts with a client and servers S_0, S_1. Parties agree on a deterministic encoding procedure $\text{Encode} : \mathcal{I} \rightarrow \mathbb{G}$. An ideal-world adversary $\mathcal{A}$ may corrupt the client and up to one server. Input: Obtain $x \in \mathcal{I}$ from the client. Compute: $z = \text{Encode}(x)$. Sample $z_0, z_1 \in \mathbb{G}$ such that $z_0 + z_1 = z$. Output: 1. If no party is corrupt, send z_i to each server S_i. 2. If $\mathcal{A}$ corrupts a client: – If $\mathcal{A}$ sends “abort client”, send “blame client” to the servers. – Else, replace z_0 by the choice of $\mathcal{A}$, and set $z_1 = z - z_0$. Send z_i to each server S_i. 3. If $\mathcal{A}$ corrupts a server S_j, – Replace z_j by the choice of $\mathcal{A}$, and set $z_{1-j} = z - z_j$. – Receive “continue” or “abort server” from $\mathcal{A}$. If “abort server” was received, send “blame S_j” to S_{1-j}. Else, send z_{1-j} to server S_{1-j}.

Fig. 10. The validated encoded secret sharing functionality.

3.2 Achieving PA using VESS

We next describe a generic approach to achieve PA in the Share-Aggregate paradigm (Section 2.5) by introducing a new functionality called VESS for the share phase.

Share phase. The goal of the share phase is to let the two servers obtain additive secret shares of each client’s encoded input. Instead of using Figure 7 as in prior works (Section 2.5), we define a stronger functionality called *validated encoded secret sharing* (VESS) (Figure 10) that provides the following key features.

When the client and both servers are honest, the servers obtain additive secret shares (over some finite group $\mathbb{G}$) of $\text{Encode}(x)$, where x is the client’s input. When the client is corrupt and both servers are honest, the servers either obtain shares of $\text{Encode}(x)$ or output “blame client”. If a server S_j is corrupt, it can cause the other (honest) server to output “blame S_j ”. But, unlike using ESS with ZKSD (Figure 7), a corrupt server S_j cannot cause the honest server to

output “blame client” when the client is honest. When the client and one server are corrupt, the guarantees of $\mathcal{F}_{\text{VSS}}$ are trivial.

During the share phase, the servers invoke one VESS instance per client. If a server S_i outputs “blame S_{1-i} ” in any instance, S_i aborts the PA protocol. If a particular VESS instance outputs “blame client”, the servers ignore that client in the aggregate phase. Once all VESS instances complete, the servers hold shares from each honestly-behaving client.

Aggregate phase. During the aggregate phase, the servers compute the aggregation function g by invoking an instance of 2PC (Figure 1). Each server participates with the (list of) *shares* of encoded honestly-behaving client inputs obtained in the share phase, denoted Z_0 and Z_1 . Since these are additive shares, to compute g , the servers use the 2PC protocol to compute $g(Z_0 + Z_1)$. We remark that when g is a simple summation, the aggregate phase can be particularly efficient; for completeness, Appendix B.2 recalls the standard techniques for achieving maliciously-secure 2PC addition (via a single commitment).

Putting things together. We describe the full private aggregation protocol that combines our share and aggregate phases in Figure 9. In Appendix B.1, we prove that this protocol achieves $\mathcal{F}_{\text{PA}}$. Our simulation-based proof is a relatively standard stitching of the two ideal functionalities $\mathcal{F}_{\text{VSS}}$ and $\mathcal{F}_{\text{2PC}}$.

3.3 A secure VESS Construction

In this section, we construct a VESS protocol Π_{VSS} that securely realizes $\mathcal{F}_{\text{VSS}}$ and can be used to construct a PA protocol following the approach in Figure 9.

Building Blocks. We construct Π_{VSS} by using any encoded secret sharing scheme ESS (Figure 5) and a ZKSD protocol Π_{ZKSD} from prior works, while performing additional steps to ensure that a malicious server cannot reject an honest client’s input. The ZKSD protocols in all prior works [13,26,7,19,28] have the following natural property (refer to [6]).

Property 1. In Π_{ZKSD} , the servers have shared randomness denoted by ρ and no other randomness.

With this property, observe that messages sent during Π_{ZKSD} depend only on party inputs, client randomness, and the shared server randomness ρ . Then, if a client knows the servers’ inputs and obtains the servers’ shared randomness ρ , it can simulate the entire interaction in Π_{ZKSD} . We will leverage this observation in our VESS construction.

Our protocol Π_{VSS} relies on another standard primitive called *Byzantine Broadcast* [16], which we define next.

Definition 1 (Byzantine Broadcast). *The Byzantine Broadcast functionality $\mathcal{F}_{\text{BB}}$ interacts with a sender that has an input m and a set of receivers. An ideal-world adversary $\mathcal{A}$ may corrupt the sender and any number of the receivers. The functionality acts as follows:*

Protocol Π_{Vess}	
Setup:	
– The client has an input $x \in \mathcal{I}$. – The protocol uses i) an encoded secret-sharing scheme ESS (Figure 5), and ii) a ZKSD protocol Π_{ZKSD} that securely realizes $\mathcal{F}_{\text{ZKSD}}$ (Figure 6) and has Property 1. – The servers agree on common randomness ρ for Π_{ZKSD}.	
1. The client secret shares its encoded input $z = \text{Encode}(x)$ using ESS as follows: – The client computes $(k_0, k_1) \leftarrow \text{ESS.DistrEncode}(x)$ and broadcasts $\text{ES} = [\text{Enc}_{pk_0}(k_0), \text{Enc}_{pk_1}(k_1)]$. – Each server S_i computes $k_i = \text{Dec}_{sk_i}(\text{ES}[i])$ and $z_i = \text{ESS.Expand}(k_i)$. 2. The client and servers execute Π_{ZKSD} with inputs $\text{Encode}(x)$, z_0, and z_1 respectively. – If S_i obtains 1 from ZKSD, it outputs z_i and finishes. – Else, S_i records (1) the transcript T_i of messages it <i>sent</i> to S_{1-i}, and (2) the transcript T_{1-i} of messages it <i>received</i> from S_{1-i} during Π_{ZKSD}. 3. Server S_0 broadcasts (T_0, T_1, ρ). 4. Server S_1 receives (T'_0, T'_1, ρ') from the broadcast of S_0 and verifies this as follows: – If (T'_0, T'_1, ρ') equals (T_0, T_1, ρ), S_1 broadcasts “continue”. Otherwise, S_1 outputs “blame S_0” and finishes. – If S_1 did not broadcast “continue”, server S_0 outputs “blame S_1” and finishes. 5. The client identifies the corrupt server and broadcasts a complaint as follows: – If server S_1 did not broadcast “continue” in step (4), the client aborts. Otherwise, the client obtains (T_0, T_1, ρ) from the broadcast of S_0 in step (3). Using $\rho, z_0, z_1, z = \text{Encode}(x)$ the client computes the transcript of messages $\mathcal{M}$ during Π_{ZKSD}. – Each message in $\mathcal{M}$ that was not sent by the client has a sender S_k and must match the corresponding message in T_k (if the recipient is the other server) or be received by the client. The client identifies the malicious server S_j that sent the first <i>wrong</i> message and broadcasts “complaint j”. 6. The servers output as follows: – If the client did not broadcast “complaint j” in the previous step, both servers output “blame client” and finish. – Otherwise, • S_j sends k_j along with $\pi \leftarrow \text{NIZK.Prove}((pk_j, \text{ES}[j], k_j); sk_j)$ to S_{1-j}. • Suppose that S_{1-j} receives π from S_j. If $\text{NIZK.Verify}((pk, c, m), \pi) \rightarrow 0$, S_{1-j} outputs “blame S_j”. Otherwise, S_{1-j} computes $x = \text{ESS.Expand}(k_0) + \text{ESS.Expand}(k_1)$. If x is not in the range of Encode, send “blame client” to S_j, and both servers output “blame client”. Otherwise, each server S_i outputs its share x_i.	

Fig. 11. Our secure VESS protocol

- Receive input m from the sender.
- Send m to each receiver.

Essentially, the Byzantine broadcast functionality forces a sender to broadcast its input to the receivers. Even if the sender is corrupt, all receivers output the same value.

Our protocol also uses *Public-key Encryption*. Each server S_i runs the setup algorithm, $\text{Gen}(1^\lambda)$, of a public-key encryption scheme to obtain secret-key/public-key pair (sk_i, pk_i) ; let (Enc, Dec) denote the encryption and decryption algorithms. We assume each party knows each server’s public key. Further, we assume that there is a one-to-one mapping $\text{SkToPk} : sk \rightarrow pk$ between the secret key and public key; this assumption is satisfied by several encryption schemes such as the El-Gamal [18] encryption scheme.

Our protocol also uses a standard *Non-interactive Zero-knowledge Argument* (NIZK) to prove that a message m is a correct decryption of a ciphertext c encrypted under a secret key sk with corresponding public key pk . Specifically, for a public statement (pk, c, m) , we require a NIZK that allows a *prover* with a *witness* sk to prove to the *verifier* that the following relation $\mathcal{R}$ between the statement and the witness is satisfied:

$$\mathcal{R}((pk, c, m); sk) = 1 \iff pk = \text{SkToPk}(sk) \wedge m = \text{Dec}(sk, c).$$

For a public-key encryption scheme such as El-Gamal, the zero-knowledge argument boils down to a proof of equal discrete logarithms, which can be done efficiently using a simple sigma protocol [11]. We will use NIZK to denote this zero-knowledge argument, and NIZK.Prove and NIZK.Verify to denote its prove and verify algorithms. For completeness, we define non-interactive zero-knowledge arguments in Appendix C.

We next describe how we use these primitives to construct our protocol.

Intuition underlying Π_{VESS} . The client first computes additive secret shares of its encoded input using ESS.DistrEncode , encrypts each share using each server’s public key, and broadcasts the encrypted shares. Each server then decrypts its share and uses ESS.Expand to obtain an additive secret share of the client’s encoded input.

Then, the client and two servers execute Π_{ZKSD} to verify the shares. If the servers obtain 1 as the result of Π_{ZKSD} , they simply output the shares they obtained and finish. Otherwise, the servers learn that either a server or the client is corrupt (refer to Figure 6 to see this). The key step in our protocol is that the three parties then interact to identify the culprit. If the client is honest, it *identifies* the malicious server S_j by rerunning the Π_{ZKSD} protocol in its head, with the help of Property 1. Once the malicious server is identified, the client broadcasts “complaint j ”. However, a malicious client may be attempting to frame an honest server after sharing an invalid encoding. To defend itself, S_j reveals its share to server S_{1-j} , along with a NIZK proof that the revealed share is a valid decryption of its broadcast encrypted share. Then, S_{1-j} either (1) has

both shares broadcasted by the client or (2) knows that S_j is corrupt, as it failed to send a correct share. In the latter case, it simply outputs “blame S_{1-j} ”. In the former case, it can ensure that either the servers output shares of a valid encoding or correctly output “blame client”.

This idea fixes the vulnerability of the insecure share phase protocol (Figure 7) used in prior works, as an honest server is guaranteed to either output a share of a valid encoding of the client’s input or appropriately assign blame to a malicious party.

Construction. We next describe our protocol Π_{VSS} in detail; Figure 11 lists the formal procedure.

The client, with an input x , first uses ESS to secret share $\text{Encode}(x)$ among the servers. Specifically, the client computes keys $(k_0, k_1) \leftarrow \text{ESS.DistrEncode}(x)$, and then broadcasts $\text{ES} = [\text{Enc}_{pk_0}(k_0), \text{Enc}_{pk_1}(k_1)]$. Each server S_i decrypts $\text{ES}[i]$ to obtain k_i and expands its key to obtain its share $z_i = \text{ESS.Expand}(k_i)$.

The parties then invoke the ZKSD protocol Π_{ZKSD} to verify that the shares z_0 and z_1 are indeed additive secret-shares of $\text{Encode}(x)$; in this protocol, the client participates with input $\text{Encode}(x)$ and the servers participate with shares z_0 and z_1 . If S_i obtains 1 as the result of Π_{ZKSD} , S_i simply outputs its share z_i and finishes. Otherwise, an honest client must identify which server is malicious.

To help an honest client identify the malicious server, each server S_i records (1) the shared server randomness ρ used in Π_{ZKSD} , and (2) the transcript of messages T_{1-i} that it *received* from S_{1-i} during Π_{ZKSD} , and (3) the transcript of messages T_i that it *sent* to S_{1-i} during Π_{ZKSD} . Then, server S_0 broadcasts (T_0, T_1, ρ) to all parties, and server S_1 verifies that the broadcasted values match its transcripts (sent and received) and local randomness for Π_{ZKSD} .

If this verification succeeds, S_1 broadcasts “continue”; otherwise, S_1 outputs “blame S_0 ” and finishes. It is also possible that an honest S_0 broadcasts the correct shared randomness and transcripts of messages, but a malicious S_1 does not broadcast “continue”; in this case, S_0 outputs “blame S_1 ” and finishes. If S_1 broadcasts “continue”, then the client can now trust the partial transcript (T_0, T_1) and the randomness ρ , because these values were either sent by *or* verified by an honest server.

Notice that the client now holds all quantities that determine a correct run of Π_{ZKSD} , namely, all party inputs and all randomness. The client locally simulates a correct execution of Π_{ZKSD} , the message transcript of which we denote by $\mathcal{M}$. Note that this simulation is possible due to Property 1. Each message in $\mathcal{M}$ must be present (1) in T_0 , if the message was sent from S_0 to S_1 , (2) in T_1 , if the message was sent from S_1 to S_0 , or (3) known to the client, if the message was sent or received by the client. But since Π_{ZKSD} outputs 0, some server must have deviated from the protocol and sent an incorrect message that is not in $\mathcal{M}$. If the *first* incorrect message is sent by server S_j , the client identifies S_j as the malicious server. Then, the client broadcasts “complaint j ”.

However, a malicious client can complain against an honest server S_j . To prove that it executed the protocol faithfully, an honest S_j sends k_j and

a proof π that decryption of $\text{ES}[j]$ matches k_j ; specifically, S_j sends $\pi \leftarrow \text{NIZK.Prove}((pk_j, \{k_j\}_{S_j}, k_j); sk_j)$ to S_{1-j} .

With all the information at hand, server S_{1-j} ensures that each honest server either outputs a share of a valid encoding or correctly blames a malicious party. Server S_{1-j} checks if π is correct, i.e., checks if $\text{NIZK.Verify}(pk, c, m, \pi)$ outputs 1. If it outputs 0, S_j outputs “blame S_{1-j} ” as an honest S_{1-j} wouldn’t send an incorrect proof. Otherwise (i.e., the proof verification passes), server S_{1-j} has both keys k_0 and k_1 that the client sent, and can use these to check if $x = \text{ESS.Expand}(k_0) + \text{ESS.Expand}(k_1)$ is a valid encoding. If x is invalid, S_{1-j} sends “blame client” to S_j and both output “blame client”. Otherwise, each server outputs its share $x_i = \text{Expand}(k_i)$.

Remark. Note that, in step 5 of Π_{Vess} , after identifying the malicious server, the client could simply send its input to the other honest server. The honest server can then locally verify that this input is correct and secret-share it.

While this protocol is formally secure, in practice, it is undesirable to ask the client to reveal its input in clear to any server. Often, a malicious server may not truly be malicious: for instance, it could simply crash, and the above protocol leaks the client’s input to the other server. Our protocol ensures that if some server learns an honest client’s input, then it is because the other (malicious) server decides to reveal its share of the client’s input.

Correctness proofs and efficiency analysis. Using our new VESS protocol Π_{Vess} , we can upgrade prior works [13,26,7,19,28,19,14] to achieve PA by replacing their insecure protocols in the share phase with our VESS protocol. We believe that our VESS protocol is still practically efficient when compared to the insecure baselines. When *all* parties are honest, the only additional overhead incurred is due to the need for the client to broadcast encryptions of each server’s share ($\text{ES} = [\text{Enc}_{pk_0}(k_0), \text{Enc}_{pk_1}(k_1)]$) among the 2 servers. When one party is corrupt, the honest parties in our protocol exchange additional messages and perform additional computation (steps (3) - (6) of Figure 11).

Lemma 1. *If all parties are honest in Π_{Vess} and client has input x , S_0 outputs z_0 and S_1 outputs z_1 such that $z_0 + z_1 = z = \text{Encode}(x)$.*

Proof. If all parties are honest, in step (1), each server S_i decrypts $\text{ES}[i]$ and computes an additive secret share $z_i = \text{ESS.Expand}(k_i)$ of $\text{Encode}(x)$. In step (2), since all parties are honest, each server S_i retains its share z_i obtained in step (1) and outputs it.

Lemma 2. *If the client is malicious and both servers are honest, either both servers output “blame client”, or S_0 outputs z_0 and S_1 outputs z_1 such that $z_0 + z_1 = \text{Encode}(x)$ for some $x \in \mathcal{I}$.*

Proof. Servers output in step (2), (4), or (6) of Π_{Vess} . We show that the lemma holds in any case.

Case 1: Servers output during step (2). In this step, the servers output only if they output 1 during Π_{ZKSD} , so the lemma holds when both servers are honest (refer to Figure 6).

Case 2: Servers output during step (4). This case does not happen when both servers are honest since honest servers compute the same transcripts.

Case 3: Servers output during step (6). Both servers output “blame client” if the client did not broadcast a “complaint” message. Otherwise, suppose the client broadcasts “complaint j for some server S_j . Since both servers are honest, due to the completeness of NIZK, S_j sends k_j along with a valid decryption of $\text{ES}[j]$ to S_{1-j} , and S_{1-j} uses k_0 and k_1 to compute x . If x is invalid, then the servers output “blame client”. Otherwise, the servers output the shares that they obtained from the client during step (1).

Lemma 3. *If an honest client broadcasts “complaint j ” during step (5) of Π_{VSS} , then S_j is corrupt.*

Proof. If both servers are honest, the servers output and finish in step (1) (Lemma 1), and do not execute step (5). Therefore, at least one server must be corrupt.

The client executes step (5) only if it receives correct transcripts T_0, T_1 , and shared randomness ρ of Π_{ZKSD} , approved by both servers. Then, the client can simulate the transcript of messages $\mathcal{M}$ and identify the malicious server S_j correctly (from Property 1).

Theorem 1. *In the $\mathcal{F}_{\text{BB}}$ -hybrid world, protocol Π_{VSS} (Figure 11) securely realizes the ideal functionality $\mathcal{F}_{\text{VSS}}$ (Figure 10).*

Proof. We construct simulators in the ideal world that have black-box access to the adversary, play the role of $\mathcal{F}_{\text{BB}}$ towards the adversary, and interact with the ideal functionality $\mathcal{F}_{\text{VSS}}$, such that the output distribution in the ideal world matches² the hybrid world.

Note that when one client and one server are corrupt, all guarantees of $\mathcal{F}_{\text{VSS}}$ ’s hold trivially, so simulation is almost trivial. The only non-triviality is that the simulation must be possible without knowing the honest server’s private secret key, but this is possible due to the zero-knowledge property of NIZK (refer to Definition 2 of Appendix C).

We consider the two remaining cases in detail:

The client is corrupt: In this case, the simulator $\mathcal{S}_c$ simulates two servers that interact with the adversary program. The simulated servers simply execute the protocol as given in Figure 11. Following this, $\mathcal{S}_c$ obtains shares z_0 and z_1 of the adversary’s encoded input $z = z_0 + z_1$ from the simulated servers or obtains “blame client” from both servers (following Lemma 2).

If the simulated servers obtain “blame client”, $\mathcal{S}_c$ sends “abort client” to $\mathcal{F}_{\text{VSS}}$. Else, $\mathcal{S}_c$ computes a value x such that $\text{Encode}(x) = z$ (this can be done efficiently according to Figure 5) and sends x to $\mathcal{F}_{\text{VSS}}$.

² The ensembles could be statistically indistinguishable or computationally indistinguishable, depending on the nature of the underlying protocols Π_{ESS} and Π_{ZKSD} .

There is one non-triviality in showing this simulation is indistinguishable from the real execution. If the client program is malicious, and if it causes Π_{ZKSD} to reject, then we cannot use the fact that Π_{ZKSD} realizes $\mathcal{F}_{\text{ZKSD}}$ to argue any type of meaningful security. Indeed, in the reject case, the client sees all server inputs and randomness (since S_0 broadcasts transcripts during step (3)), so we cannot replace the call to Π_{ZKSD} by a call to $\mathcal{F}_{\text{ZKSD}}$ in a hybrid step. However, this is still fine, since Π_{ZKSD} only takes the shares z_0, z_1 , which the client already knows, as inputs from the server, so—conditioned on the event that the servers reject the ZKSD proof—we do not need any security from Π_{ZKSD} . On the other hand, if the malicious client allows Π_{ZKSD} to succeed, then the client does not see server randomness, so we *can* in a hybrid argument replace the call to Π_{ZKSD} by a call to $\mathcal{F}_{\text{ZKSD}}$, and then use the guarantees of $\mathcal{F}_{\text{ZKSD}}$ to show indistinguishability of real from ideal.

A server is corrupt: In this case, the simulator $\mathcal{S}_s$ is more involved; we present this in Figure 17 of Appendix C. The critical steps are steps (5) and (6) of Figure 11. To simulate the view of the adversary in step (5), we require that an honest server is never blamed by the client. We show this is indeed the case in Lemma 3. During step (6), we require that an honest client is never blamed by an honest server. This is ensured by the soundness of the NIZK scheme (Definition 2 of Appendix C)).

3.4 Does VESS require Signatures and PKI?

Our VESS protocol in Figure 11 uses Byzantine broadcast (Definition 1). It is well known that broadcast protocols among three parties with one possibly malicious party require extra assumptions, such as digital signatures and a PKI [20]. We now justify our use of broadcast by showing that VESS, for a large class of **Encode** functions, implies a solution to three-party broadcast. In other words, VESS, in general, requires broadcast. Specifically, we consider VESS for **Encode** functions that satisfy the following property, which is satisfied by **Encode** functions in all prior works:

Property 2. The range V of **Encode** can be partitioned into two sets V_0 and V_1 such that:

1. For any $j \in \{0, 1\}$, one can efficiently sample a uniformly random value from the set $\{y_j : \text{Encode}(y_j) \in V_j\}$;
2. For any $\delta \in \mathbb{G}$ and uniformly random $v \in V_b$ ($b \in \{0, 1\}$), either (i) $v + \delta \in V_b$ holds with overwhelming probability, or (ii) $v + \delta \notin V$ holds with non-negligible probability (at least $1/\text{poly}(\lambda)$ where λ is a security parameter).

To understand this property, consider the commonly used **Encode** function that encodes inputs as one-hot vectors. We can define V_0 to be the set of one-hot vectors whose non-zero position is in the *first half* of the vector, and V_1 to be the set of one-hot vectors whose non-zero position is in the *second half* of the vector. It is easy to see that, when any nonzero δ is added to a uniformly random

Protocol Π_{BB}	
Setup:	
– Let $k = \text{poly}(\lambda)$, where λ is the security parameter. – The sender S has an input $m \in \{0, 1\}$. – The parties interact with a VESS functionality $\mathcal{F}_{\text{VESS}}$ with an encoding function Encode. The parties agree on a partition V_0 and V_1 of the range of Encode as per Property 2.	
1. S, R_0 and R_1 invoke $2k$ instances of $\mathcal{F}_{\text{VESS}}$, where S acts as the client, and R_0 and R_1 as the servers; for each instance, S uniformly samples a value from the set $\{y_m : \text{Encode}(y_m) \in V_m\}$ and uses it as its input. S also sends m to R_0 and R_1. 2. For each receiver R_i, if some instance does not output a share: – If the instance outputs “blame S”, output a default bit (say 1). – If the instance outputs “blame R_{1-i}”, output m received from S. 3. The receivers partition the $\mathcal{F}_{\text{VESS}}$ instances as follows: – Receiver R_0 chooses a random partition P_0 and P_1 of the set $T = \{1, 2, \dots, 2k\}$ s.t. each P_i has k elements. R_0 sends P_0 and P_1 to R_1. – If R_0 doesn’t send the partitions, R_1 outputs the value m given by the sender. 4. The receivers compute the reconstructed results of $\mathcal{F}_{\text{VESS}}$ instances as follows: – Each instance of $\mathcal{F}_{\text{VESS}}$ scheme outputs a share to each receiver. Let the shares obtained by R_i be denoted by the set $\{y_i^j\}_{j \in T}$. Each receiver R_i sends shares $\{y_i^j\}_{j \in P_{1-i}}$ to R_{1-i}. – If some receiver R_i does not receive shares $\{y_{1-i}^j\}_{j \in P_i}$ from R_{1-i}, output the value m sent by the sender. – Otherwise, each receiver R_i computes the set of values $\{y^j = y_0^j + y_1^j\}_{j \in P_i}$. 5. The receivers (try to) obtain the broadcast output as follows: – If $y^j \notin V_0$ and $y^j \notin V_1$, for any $j \in P_i$, R_i outputs m received from S. – Else, if at least $3k/4$ of $\{y^j\}_{j \in P_i}$ belong to V_b (for any $b \in \{0, 1\}$), then R_i sets $b_i = b$. – Else, R_i sets $b_i = \perp$ and sends “None” to R_{1-i}. 6. Each receiver R_i outputs as follows: – If R_i sent and received “None”, output 1. – Else if, R_i received “None”, send b_i to R_{1-i} and output b_i. – Else, output b_i	
Any receiver R_i that has not yet output, outputs the value sent by receiver R_{1-i} .	

Fig. 12. A three-party Byzantine broadcast protocol from VESS.

one-hot vector in V_0 (resp. V_1), the resulting sum is not a one-hot vector with non-negligible probability (adding $\delta = 0$ has no effect).

We remark that restricting to VESS for **Encode** functions that satisfy the given property is not due to our inability to prove a more general result. Rather, VESS protocols with certain simple **Encode** functions *can* be constructed *without* using broadcast. For instance, when **Encode** is the identity function (and the domain of **Encode** matches its codomain), the client simply additively secret-shares its input among the servers, and each server outputs the received share (or outputs 0 if no share is received). It is straightforward to see that this protocol securely realizes $\mathcal{F}_{\text{VESS}}$ and does not require a Byzantine broadcast.

We now show that VESS with **Encode** functions that satisfy Property 2 can be used to construct (with no additional assumptions) a Byzantine Broadcast protocol among three parties, where one party may be malicious. In particular, we construct a protocol among a sender S with an input bit m and two receivers R_0 and R_1 , such that the following two properties hold with overwhelming probability:

1. *Correctness.* If S is honest, any honest receiver outputs m .
2. *Consistency.* If both receivers are honest, they output the same value.

A strawman protocol We first describe a strawman protocol that does not work, but helps build intuition. When S has input m , it participates in $\mathcal{F}_{\text{VESS}}$ as the client with a random input from the set $\{y_m : \text{Encode}(y_m) \in V_m\}$ (the sender can efficiently sample such an input due to Property 2), and the receivers participate as servers. If the receivers obtain “blame client” as the output of $\mathcal{F}_{\text{VESS}}$, they output a default bit, say 1. Otherwise, the servers obtain additive secret shares. They exchange shares, add both shares, and reconstruct a value. If the reconstructed value is in V_0 , both receivers output 0, else both receivers output 1.

This approach guarantees consistency when both receivers are honest. The sender participates with an input in $\{y_m : \text{Encode}(y_m) \in V_m\}$, and the receivers reconstruct a value in V_m and output m .

But the approach fails to guarantee correctness if a receiver is corrupt. During reconstruction, a corrupt receiver could perform a “rushing” attack: it could wait to receive the other honest receiver’s share before sending a manipulated share that causes the honest receiver to reconstruct a value in V_{1-m} . This causes the honest receiver to output the bit $1 - m$, while the sender’s input is m .

Three-party Byzantine broadcast protocol using VESS We next describe the Byzantine broadcast protocol. Instead of invoking a single instance of VESS, the parties invoke $2k$ instances of VESS (we will specify k later). In each instance, the sender participates with a *uniformly random* input from the set $\{y_m : \text{Encode}(y_m) \in V_m\}$. The sender also sends m directly to both R_0 and R_1 .

The receivers may not obtain shares from each VESS instance. If the receivers obtain “blame S ” from any VESS instance, the receivers output the default bit 1, as this event means that the sender is malicious. Furthermore, if any VESS

instance outputs “blame R_i ”, then the receiver R_{1-i} outputs the bit m it received from the sender, since the other receiver is malicious. Otherwise, the receivers obtain shares from each VESS instance.

From here, the high-level idea is that the receivers will treat the shared values as “votes” towards the true output; a shared value from V_0 is a vote towards 0, and a shared value from V_1 a vote towards 1. To prevent a corrupt receiver from performing a “rushing” attack, receivers open half of the votes towards R_0 and half towards R_1 . Note that a malicious receiver might be able to flip some votes by corrupting its shares, but Property 2 ensures it cannot flip too many without reconstructing an invalid value and getting caught (except with negligible probability). This allows the receivers to tally the votes, check that there are sufficient votes from V_m , and output m .

In more detail, receiver R_0 *randomly* partitions the $2k$ VESS instances into sets P_0 and P_1 , each of size k . R_0 sends P_0 and P_1 to R_1 . If R_1 does not receive them from R_0 , it outputs the value m it received from the sender (only a malicious R_0 would not send partitions). Otherwise, each R_i sends its output shares of VESS instances in P_{1-i} to R_{1-i} . Then, R_i computes the reconstructed encodings of VESS instances in P_i , by adding both output shares of each instance in P_i . If R_i does not receive shares from R_{1-i} , it outputs the value m it received from the sender (as R_{1-i} is malicious).

Each receiver R_i uses these reconstructed encodings to compute the output of the broadcast protocol. If some reconstructed value is neither in V_0 nor in V_1 , R_i outputs the m it received from the sender. This is because when both receivers are honest, the reconstructed output of any VESS instance cannot be outside the range of **Encode**. If at least $3k/4$ reconstructed results are in V_b (for some bit b), R_i sets its output $b_i = b$ but does not yet finish as it may need to help the other receiver output. Otherwise, R_i is not able to obtain an output yet, it sends “None” to the other receiver R_{1-i} (to request R_{1-i} ’s help to obtain the output).

Finally, the receivers output as follows. If both receivers sent “None” to each other, they output a default bit 1 (the sender is malicious in this case). If only one receiver, say R_j , sent “None”, R_{1-j} sends b_{1-j} to R_j and both receivers output b_{1-j} . If no receiver sent “None”, each receiver R_i simply outputs b_i .

Proof sketch. We first consider correctness. If both receivers (and the sender) are honest, correctness is straightforward. Now, without loss of generality, suppose R_1 is corrupt. R_1 can send wrong shares to R_0 for the VESS instances in P_0 . From Property 2, if R_1 gets lucky, some VESS instance in P_0 reconstructs to a value in V_{1-m} (where m is the input of the sender). However, the probability that R_1 flips $k/4$ reconstructed results *without resulting in an invalid value* is $p^{k/4}$ where $p = (1 - \frac{1}{\text{poly}(\lambda)})$ (as each flip is independent). This probability is negligible if we set k appropriately to $k = \text{poly}(\lambda)$, so the malicious receiver cannot stop the honest receiver from outputting the sender’s input.

To violate consistency, a corrupt sender needs to set up inputs to VESS instances such that at least $3k/4$ instances in P_0 reconstruct to values in V_b (for some bit b), and at least $3k/4$ instances in P_1 reconstruct to values in V_{1-b} .

Functionality $\mathcal{F}_{\text{IPA}}$
Setup: Same as $\mathcal{F}_{\text{PA}}$. In addition, $\mathcal{C}_{h_s}$ denotes the set of synchronous honest clients and $\mathcal{C}_{h_i} = \mathcal{C}_h \setminus \mathcal{C}_{h_s}$ denotes the set of intermittent honest clients. Input: 1. Obtain x_i from each client i . 2. Obtain $\mathcal{C}'_c \subseteq \mathcal{C}_c \cup \mathcal{C}_{h_i}$ from the adversary. Let $\mathcal{C}' = \mathcal{C}'_c \cup \mathcal{C}_{h_s}$. Compute: Same as $\mathcal{F}_{\text{PA}}$. Output: Same as $\mathcal{F}_{\text{PA}}$.

Fig. 13. The private aggregation for intermittent clients functionality.

But P_0 and P_1 form a *random partition* of the VESS instances (unknown to the sender). We show that a corrupt sender cannot succeed except with negligible probability using a Chernoff-style bound on the Hypergeometric distribution (see Appendix D).

We describe the full protocol in detail in Figure 12 and prove that it solves Byzantine broadcast in Appendix D.

Theorem 2. $\mathcal{F}_{\text{VESS}}$ with an Encode function satisfying Property 2 can be used to construct a Byzantine Broadcast protocol among three parties that tolerates one malicious party.

4 Private Aggregation for Clients with Unreliable Networks

Even after applying our fix in Section 3.3, prior protocols [13,26,7,19,28,14] can only achieve PA when all honest clients are synchronous and have a reliable network (i.e., the messages sent by honest clients are delivered within bounded time). Some prior works [13,28] consider clients with unreliable networks (henceforth called intermittent clients): messages from and to such clients may be arbitrarily delayed or dropped by a network adversary; but these works only achieve threshold PA.

In this section, we describe how any PA protocol in the Share-Aggregate paradigm (Section 3.2) designed for synchronous clients (such as the one constructed in Section 3.3) can be compiled into one that additionally handles intermittent clients.

4.1 Defining PA for Intermittent Clients

When clients are intermittent, it is *impossible* to achieve the notion of PA defined in Figure 3, which requires *all* honest clients' inputs to be included in the aggregation result when both servers are honest. Observe that when the two honest servers do not hear from a client at all, they cannot tell if the client is corrupt or intermittent. The servers cannot wait for the client indefinitely (because the

client may be malicious and never send its input), but proceeding without this client may miss the input of this (possibly honest but intermittent) client.

In Figure 13, we present a definition that naturally generalizes PA to one that considers intermittent clients. We highlight the key difference from PA in yellow. Essentially, this definition guarantees that the aggregate result includes the inputs of *all honest synchronous* clients when both servers are honest (unlike PA, which guarantees to include the inputs of *all honest* clients). Accordingly, when a server is corrupt, it can learn nothing beyond the aggregation of all honest synchronous clients’ encoded inputs (rather than all honest clients’ encoded inputs in PA) with additive errors to each encoded input chosen by the corrupt server (just like PA).

4.2 Achieving PA for Intermittent Clients

In this section, we construct a compiler that compiles a PA protocol $\Pi_{\text{PA}}^{\text{S}}$ in the Share-Aggregate paradigm designed for *synchronous* clients into a protocol $\Pi_{\text{PA}}^{\text{I}}$ that achieves PA *even for intermittent clients*. Essentially, our compiler compiles the *share* phase into one that handles intermittent clients. The aggregate phase can be executed as in the original protocol $\Pi_{\text{PA}}^{\text{S}}$. Recall that we assume the servers have a synchronous communication link between them.

Challenges with intermittent clients. Consider a message sent from one party to another during the share phase of $\Pi_{\text{PA}}^{\text{S}}$, where either the sender or the receiver *is a client*. In synchrony, the sender knows that the receiver receives this message in time, and the VESS protocol used in $\Pi_{\text{PA}}^{\text{S}}$ is designed under this assumption. However, when clients are intermittent, the sender cannot be sure that the receiver receives this message.

High-level ideas. Our compiler bridges this gap. For each message sent during the share phase of $\Pi_{\text{PA}}^{\text{S}}$, our compiler makes the receiver send an *acknowledgment* back to the sender. An intermittent client may not send/receive an acknowledgment in time, but we leverage the fact that the link *between servers* is always synchronous to guarantee that either (1) the sender receives an acknowledgment or (2) all parties, including the client itself, learn that the client is intermittent.

First, consider the case when the client is the *sender*. The client sends its message not only directly to the receiver, but also *via the other server*, which we call the *witness server*. Since the witness server may be corrupt, messages are *encrypted* using the receiver’s public keys and *signed* by the sender to ensure that the witness server cannot learn from or tamper with the message.

On receiving the message, the receiver sends an acknowledgment not only directly to the client but also via the witness server. If the (honest) witness server forwarded the client message to the receiver server but does not receive an acknowledgment, it knows the receiver server is malicious and aborts. If an honest client does not receive an acknowledgment within some bounded time, it must be that neither server received its message on time. In this case, the client learns it is intermittent and “withdraws” from the protocol.

A similar strategy works when the client is the *receiver*. The sender server sends its (encrypted and signed) message via the witness server, in addition to sending it directly to the client. If the (honest) witness server does not receive the message from the sender server, it aborts; otherwise, it forwards the message to the client. On receiving the message, whether directly or via forwarding, the client sends an acknowledgment to both servers (and the witness server forwards it).

This design ensures that, for each client, the execution of $\Pi_{\text{PA}}^{\text{I}}$ proceeds like that of $\Pi_{\text{PA}}^{\text{S}}$ until it “withdraws” from the protocol. Since synchronous clients never withdraw from the protocol, their inputs are included in the final aggregate result. For intermittent clients that may withdraw from the protocol, privacy is preserved.

Details of the compiler. Figure 14 presents our compiler that compiles a share-aggregate PA protocol $\Pi_{\text{PA}}^{\text{S}}$ for synchronous clients into one $\Pi_{\text{PA}}^{\text{I}}$ for intermittent clients.

Our compiler uses *digital signatures* and *public-key encryption*. We use $\{m\}_p$ and $\langle m \rangle_p$ to denote a message m encrypted under the party p ’s public key and signed by party p , respectively.

We first describe how the share phase is compiled. Each round in the share phase is compiled into four rounds. We have already discussed how messages to and from a client are handled with the help of the witness server. Server-to-server messages and local computations require no special treatment other than that they are moved to the appropriate round: server-to-server messages sent in round r of $\Pi_{\text{PA}}^{\text{S}}$ are now sent in round $4r$ of $\Pi_{\text{PA}}^{\text{I}}$, and local computations at the end of round r of $\Pi_{\text{PA}}^{\text{S}}$ are now carried out by the parties at the end of round $4r + 3$ of $\Pi_{\text{PA}}^{\text{I}}$.

For each client $c \in \mathcal{C}$, the servers maintain flag_c , which indicates whether or not c has withdrawn from the protocol. It is important that honest servers agree on which clients withdraw. For messages from a client c , the message is routed via both servers, so servers agree on the value of flag_c . For messages to a client c , the servers forward client acknowledgment (if any) to each other, so they agree on the value of flag_c . The aggregate phase is then executed by the servers for all clients $c \in \mathcal{C}$ with valid inputs (as determined by the VESS instances) and $\text{flag}_c = 1$.

In Appendix E, we formally prove that $\Pi_{\text{PA}}^{\text{S}} \rightarrow \Pi_{\text{PA}}^{\text{I}}$ yields a protocol that achieves $\mathcal{F}_{\text{IPA}}$.

Theorem 3. *A real-world execution of the protocol output by $\Pi_{\text{PA}}^{\text{S}} \rightarrow \Pi_{\text{PA}}^{\text{I}}$ (Figure 14) is indistinguishable from an ideal-world execution of the ideal functionality $\mathcal{F}_{\text{IPA}}$ (Figure 13).*

5 Achieving Threshold Private Aggregation

In this section, we describe a protocol Π_{TPA} using which prior works [13,26,7,19,28,27] in the Share-Aggregate paradigm can achieve threshold

Protocol $\Pi_{\text{PA}}^{\text{S}} \rightarrow \Pi_{\text{PA}}^{\text{I}}$	
Setup:	
– The parties use a protocol $\Pi_{\text{PA}}^{\text{S}}$ in the share-aggregate paradigm for synchronous clients. – Each party (both clients and servers) executes the setup algorithm of digital signatures and public-key encryption to obtain a verification key and a secret key. The verification key (of digital signatures) and public key (of public-key encryption) of each party are known to all other parties. – Each server initializes a flag flag_c to 1 for each $c \in \mathcal{C}$, i.e., all clients are initially participants.	
Compiling the share phase: Each round $r \in [R]$ in the share phase of $\Pi_{\text{PA}}^{\text{S}}$ is compiled as follows into (4 rounds of) $\Pi_{\text{PA}}^{\text{I}}$:	
1. For each message m sent from a client c with $\text{flag}_c = 1$ to server S_i during round r of $\Pi_{\text{PA}}^{\text{S}}$: (a) c sends (r, m) to S_i and $\langle r, c, \{m\}_{S_i} \rangle_c$ to S_{1-i}. (b) S_{1-i}, on receiving $\langle r, c, \{m\}_{S_i} \rangle_c$ from c, forwards this to S_i. (c) S_i, on receiving $\langle r, c, \{m\}_{S_i} \rangle_c$ from S_{1-i} or on receiving (r, m) from c, sends $\langle \text{ack}, r, c \rangle_{S_i}$ to c and S_{1-i}. Otherwise S_i sets flag_c to 0. (d) – If S_{1-i} forwarded $\langle r, c, \{m\}_{S_i} \rangle_c$ in step (b) but does not receive $\langle \text{ack}, r, c \rangle_{S_i}$ in step (c), then S_{1-i} aborts. – If S_{1-i} receives $\langle \text{ack}, r, c \rangle_{S_i}$, it forwards this to c. Else, it sets flag_c to 0. – If c does not receive $\langle \text{ack}, r, c \rangle_{S_i}$, it aborts. 2. For each message m sent from a server S_i to client c with $\text{flag}_c = 1$ during round r of $\Pi_{\text{PA}}^{\text{S}}$: (a) S_i sends (r, m) to c and $\langle r, c, \{m\}_c \rangle_{S_i}$ to S_{1-i}. (b) S_{1-i} aborts if it does not receive $\langle r, c, \{m\}_c \rangle_{S_i}$ from S_i. Otherwise, it forwards this to c. (c) c, on receiving (r, m) from S_i or $\langle r, c, \{m\}_c \rangle_{S_i}$ from S_{1-i}, sends $\langle \text{ack}, r, c \rangle_c$ to S_i and S_{1-i}. Otherwise, it aborts. (d) If S_0 receives $\langle \text{ack}, r, c \rangle_c$ from c, then it forwards this to S_1 (and vice versa). If S_0 (resp. S_1) does not receive $\langle \text{ack}, r, c \rangle_c$ at the end of this round, then it sets flag_c to 0. 3. For each message (r, m) sent from S_0 to S_1 (resp. S_1 to S_0) during round r of $\Pi_{\text{PA}}^{\text{S}}$, S_0 (resp. S_1) sends (r, m) during round $4r$ of $\Pi_{\text{PA}}^{\text{I}}$. 4. Each party performs the computation scheduled for the end of round r of $\Pi_{\text{PA}}^{\text{S}}$ at the end of round $4r + 3$ of $\Pi_{\text{PA}}^{\text{I}}$.	
Compiling the aggregate phase: Execute the aggregate phase of $\Pi_{\text{PA}}^{\text{S}}$ for all clients $\{c \mid \text{flag}_c = 1\}$ with valid encoded inputs.	

Fig. 14. Compiler from any PA protocol in the SA paradigm for synchronous clients to one for intermittent clients.

Protocol Π_{TPA}	
1.	Each client $c \in \mathcal{C}$ computes $(k_0, k_1) \leftarrow \text{ESS.DistrEncode}(x_c)$, and sends k_0 to S_0 and k_1 to S_1 . Each server S_i computes $z_i^c = \text{ESS.Expand}(k_i)$ for each $c \in \mathcal{C}$.
2.	For each client $c \in \mathcal{C}$, start an instance of $\mathcal{F}_{\text{ZKSD}}$ where c inputs $z^c = \text{Encode}(x_c)$ and each server S_i inputs z_i^c . Each server S_i computes a list Z_i of additive shares corresponding to instances of $\mathcal{F}_{\text{ZKSD}}$ that output “1”.
3.	If $ Z_i < \tau$, S_i aborts. Otherwise, S_i participates in the $\mathcal{F}_{\text{2PC}}$ functionality (Figure 1) with input Z_i (to compute $g(Z_0 + Z_1)$) and outputs the result.

Fig. 15. A threshold PA protocol using parameter $\tau = \frac{n+n_q}{2}$.

PA (Figure 4). While we have already shown that these works can be upgraded to achieve the stronger notion of PA, that fix requires PKI and digital signatures to instantiate Byzantine broadcast and incurs nontrivial additional cost when some parties are malicious. Here, we show that threshold PA can be achieved *without* those extra assumptions or costs. We first describe the protocol that works in synchrony and show how this can be extended to handle intermittent clients.

Our protocol Π_{TPA} follows the strategy outlined in prior works [13,27]. Additionally, we show that the protocol requires a minority of corrupt clients, a critical condition that prior works failed to specify.

The threshold PA protocol Figure 15 gives the threshold PA protocol Π_{TPA} . The protocol uses a **ESS** scheme (Figure 5), a **ZKSD** functionality (Figure 6), and a **2PC** functionality (Figure 1).

Each client uses the encoded secret sharing scheme **ESS** to secret-share its encoded input, and the client and the two servers invoke $\mathcal{F}_{\text{ZKSD}}$ to check if the secret-shared value is a valid encoding; these two steps are identical to the insecure share protocol (Figure 7) in prior works, which we have described at length in Section 2.5. If a server outputs 0 during $\mathcal{F}_{\text{ZKSD}}$, it discards the client shares.

Then, each server invokes $\mathcal{F}_{\text{2PC}}$ using the remaining client shares instance *only* if at least τ shares are obtained (meaning that in at least τ instances of $\mathcal{F}_{\text{ZKSD}}$, the server did not output 0 and obtained an additive share of the client’s encoded input).

Theorem 4. *The protocol Π_{TPA} (Figure 15) achieves $\mathcal{F}_{\text{TPA}}$ in the $\mathcal{F}_{\text{2PC}}$ -hybrid world when the number of corrupt clients $n_c \leq (n - n_q)/2$.*

Proof. (sketch) If servers do not abort, then the servers obtain shares for at least $\frac{n+n_q}{2} = \frac{n-n_q}{2} + n_q \geq n_c + n_q$ encoded inputs, which contains at least n_q honest inputs. Further, when both servers are honest, each Z_i contains at least $n - n_c \geq n - \frac{n-n_q}{2} = \frac{n+n_q}{2} = \tau$ shares of honest clients, and no server aborts. The rest of the proof is similar to that of Theorem 6.

A bound on the number of corrupt clients We proved Π_{TPA} secure under the condition that the number of corrupt clients n_c is at most $(n - n_q)/2$ (Theorem 4). We now show this condition is *necessary* for Π_{TPA} for *any* threshold τ . Consider the following two scenarios:

Scenario 1: Both servers are honest. In this scenario, n_c corrupt clients secret-share “invalid” values (i.e., values that are not encodings). By the guarantees of ZKSD (Figure 7), these client inputs are discarded. Since the protocol must not abort when both servers are honest, τ must be at most $n - n_c$.

Scenario 2: One server is malicious, and the other server is honest. In this scenario, suppose n_c corrupt clients execute the protocol faithfully. But the malicious server causes n_c honest clients’ inputs to be discarded (this is possible with ZKSD, refer to Figure 6). Let $\mathcal{C}'$ be the set of remaining $n - n_c$ clients. Since $n - n_c \geq \tau$ (from Scenario 1), the servers invoke $\mathcal{F}_{\text{2PC}}$ and output $g(\{z_i\}_{i \in \mathcal{C}'})$. But $\mathcal{F}_{\text{TPA}}$ requires $\mathcal{C}'$ to contain at least n_q honest clients’ inputs. This requires $|\mathcal{C}'| - n_c = n - 2n_c \geq n_q$. This implies that $n_c \leq (n - n_q)/2$.

Handling Intermittent Clients. The ideal functionality for threshold private aggregation for intermittent clients is the same as the threshold private aggregation functionality in Figure 4. The protocol in Figure 15 can also be naturally extended to handle intermittent clients, provided the ZKSD protocol used in Step (2) is secure (i.e., hides the client input) when the client is intermittent. ZKSD protocols in prior works satisfy this condition.

We now describe the extension. If a server S_i does not receive a message from a client c that it expects in Steps (1) and (2) within some bounded time, it means that c is intermittent or malicious. S_i informs S_{1-i} about this, and both servers exclude c from Z_i and Z_{1-i} . Step (3) remains the same. It is easy to see that the analysis in Theorem 4 applies to this protocol when the bound on corrupt clients is naturally extended. Specifically, if n_{h_i} is the number of honest and intermittent clients, from Scenario 1, $\tau \leq n - n_c - n_{h_i}$ (note that messages sent by intermittent clients may be dropped). From Scenario 2, even when $n_c + n_{h_i}$ honest clients’ inputs are excluded, the remaining set $\mathcal{C}'$ should contain at least n_q honest clients, implying that $|\mathcal{C}'| - n_c = n - n_{h_i} - 2n_c \geq n_q$. In this case, it is necessary that $n_c \leq (n - n_q - n_{h_i})/2$.

References

1. Exposure notification privacy-preserving analytics (enpa) white paper. https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf
2. Learning iconic scenes with differential privacy. <https://machinelearning.apple.com/research/scenes-differential-privacy>
3. Next steps in privacy-preserving telemetry with prio. <https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/>
4. Asharov, G., Hamada, K., Ikarashi, D., Kikuchi, R., Nof, A., Pinkas, B., Takahashi, K., Tomida, J.: Efficient secure three-party sorting with applications to data anal-

- ysis and heavy hitters. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 125–138 (2022)
5. Bell, J.H., Bonawitz, K.A., Gascón, A., Lepoint, T., Raykova, M.: Secure single-server aggregation with (poly) logarithmic overhead. In: Proceedings of the 2020 ACM SIGSAC conference on computer and communications security. pp. 1253–1269 (2020)
 6. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Zero-knowledge proofs on secret-shared data via fully linear pcps. In: Annual international cryptology conference. pp. 67–97. Springer (2019)
 7. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Lightweight techniques for private heavy hitters. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 762–776. IEEE (2021)
 8. Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., Ishai, Y.: Arithmetic sketching. In: Annual International Cryptology Conference. pp. 171–202. Springer (2023)
 9. Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 337–367. Springer (2015)
 10. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001)
 11. Chaum, D., Pedersen, T.P.: Wallet databases with observers. In: Annual international cryptology conference. pp. 89–105. Springer (1992)
 12. Chaum, D.L.: Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM* **24**(2), 84–90 (1981)
 13. Corrigan-Gibbs, H., Boneh, D.: Prio: Private, robust, and scalable computation of aggregate statistics. In: 14th USENIX symposium on networked systems design and implementation (NSDI 17). pp. 259–282 (2017)
 14. Corrigan-Gibbs, H., Boneh, D., Mazières, D.: Riposte: An anonymous messaging system handling millions of users. In: 2015 IEEE Symposium on Security and Privacy. pp. 321–338. IEEE (2015)
 15. Corrigan-Gibbs, H., Ford, B.: Dissent: accountable anonymous group messaging. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 340–350 (2010)
 16. Dolev, D., Strong, H.R.: Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing* **12**(4), 656–666 (1983)
 17. Dutta, M., Ganesh, C., Patranabis, S., Singh, N.: Compute, but verify: Efficient multiparty computation over authenticated inputs. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 133–166. Springer (2024)
 18. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory* **31**(4), 469–472 (1985)
 19. Eskandarian, S., Corrigan-Gibbs, H., Zaharia, M., Boneh, D.: Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 1775–1792 (2021)
 20. Fischer, M.J., Lynch, N.A., Merritt, M.: Easy impossibility proofs for distributed consensus problems. *Distributed Computing* **1**(1), 26–39 (1986)
 21. Hazay, C., Venkitasubramaniam, M., Weiss, M.: Your reputation’s safe with me: Framing-free distributed zero-knowledge proofs. In: Theory of Cryptography Conference. pp. 34–64. Springer (2023)

22. Kwon, A., Corrigan-Gibbs, H., Devadas, S., Ford, B.: Atom: Horizontally scaling strong anonymity. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 406–422 (2017)
23. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems **4**(3), 382–401 (1982)
24. Lehmkuhl, R., Corrigan-Gibbs, H., Dauterman, E., Wu, D.J.: Heli: Heavy-light private aggregation. Cryptology ePrint Archive (2026)
25. Lindell, Y.: How to simulate it—a tutorial on the simulation proof technique. Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich pp. 277–346 (2017)
26. Newman, Z., Servan-Schreiber, S., Devadas, S.: Spectrum: High-bandwidth anonymous broadcast. In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22). pp. 229–248 (2022)
27. Rathee, M., Shen, C., Wagh, S., Popa, R.A.: Elsa: Secure aggregation for federated learning with malicious actors. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 1961–1979. IEEE (2023)
28. Rathee, M., Zhang, Y., Corrigan-Gibbs, H., Popa, R.A.: Private analytics via streaming, sketching, and silently verifiable proofs. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 3072–3090. IEEE (2024)
29. Wang, X., Ranellucci, S., Katz, J.: Authenticated garbling and efficient maliciously secure two-party computation. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. pp. 21–37 (2017)
30. Wikipedia contributors: Hypergeometric distribution (2026), https://en.wikipedia.org/wiki/Hypergeometric_distribution

A Additional Related Work

We discuss prior works on Zero-knowledge proofs on Secret-shared Data (ZKSD). ZKSD was first introduced for performing special-purpose verifications on secret-shared data [13], and was later extended to enable more general-purpose verifications [6,8] and “silent” batch verification [28]. These initial works did not provide *strong completeness*, which at a high level means that a malicious server cannot make honest servers reject the proof of an honest client. Using such a ZKSD protocol directly to construct VESS leads to the security flaws we discussed in Section 3.1. ZKSD schemes with *strong completeness* have been studied only recently [21,17]. These recent works can be used to construct VESS protocols, but they are designed for settings with multiple servers and a corrupt minority, and thus cannot be used in our two-server setting.

B The Share-Aggregate Paradigm

B.1 Achieving PA using $\mathcal{F}_{\text{VESS}}$ and $\mathcal{F}_{2\text{PC}}$

In this section we formally show that, using $\mathcal{F}_{\text{VESS}}$ and $\mathcal{F}_{2\text{PC}}$, the protocol in Figure 9 achieves the guarantees of PA.

Theorem 5. *Protocol Π_{PA} presented in Figure 9 achieves $\mathcal{F}_{\text{PA}}$ in the $(\mathcal{F}_{\text{VESS}}, \mathcal{F}_{2\text{PC}})$ -hybrid world.*

Protocol Π_{Sum}
Setup: Each server S_i participates with an input $y_i \in \mathbb{G}$ and wants to compute output $y = y_0 + y_1$. - Each S_i locally computes the following, $(c_i, d_i) = \text{Com}(y_i)$, and sends c_i to S_{1-i}. If c_{1-i} is not received from S_{1-i}, S_i aborts. - Each S_i sends d_i to S_{1-i}. If d_{1-i} is not received from S_{1-i}, S_i aborts. Else, S_i computes $y_{1-i} = \text{Open}(c_{1-i}, d_{1-i})$. If y_{1-i} is $\perp$, S_i aborts, and otherwise outputs $y_i + y_{1-i}$.

Fig. 16. A 2PC protocol that sums a vector of inputs

Proof. We describe a simulator $\mathcal{S}_{\text{PA}}$ that interacts with $\mathcal{F}_{\text{PA}}$ and $\mathcal{A}$ to simulate the view in the real world. To produce a convincing protocol execution, the simulator will play the roles of $\mathcal{F}_{\text{Vess}}$ and $\mathcal{F}_{2\text{PC}}$. Consider two cases.

No server is corrupt: In this case, $\mathcal{S}_{\text{PA}}$ runs an instance of $\mathcal{F}_{\text{Vess}}$ for each adversarial client. $\mathcal{S}_{\text{PA}}$ records the set $\mathcal{C}'_c$ of clients for whom $\mathcal{A}$ participates in their $\mathcal{F}_{\text{Vess}}$ with an input (without sending “abort client”). $\mathcal{S}_{\text{PA}}$ records these inputs, and sends $\mathcal{C}'_c$ along with these inputs to $\mathcal{F}_{\text{PA}}$. This matches the output in the real world since (1) by the guarantees of $\mathcal{F}_{\text{Vess}}$, the inputs of all honest clients and the clients in $\mathcal{C}'_c$ are sent to $\mathcal{F}_{2\text{PC}}$ and (2) $\mathcal{F}_{2\text{PC}}$ guarantees that the output in the real world equals the output of $\mathcal{F}_{\text{PA}}$.

Server S_i is corrupt: To simulate the view of S_i , $\mathcal{S}_{\text{PA}}$ runs n instances of $\mathcal{F}_{\text{Vess}}$, one for *each* client (including honest ones). For each honest client $c \in \mathcal{C}_h$, $\mathcal{S}_{\text{PA}}$ sends some fixed input $x \in \mathcal{I}$ to the corresponding instance of $\mathcal{F}_{\text{Vess}}$. For each malicious client, as done in the previous case, records the set $\mathcal{C}'_c$ of clients for whom $\mathcal{A}$ participates in their $\mathcal{F}_{\text{Vess}}$ with an input (without sending “abort client”). $\mathcal{S}_{\text{PA}}$ sends $\mathcal{C}'_c$ along with the recorded (adversarial) inputs to $\mathcal{F}_{\text{PA}}$.

If $\mathcal{A}$ sends “abort server” for any instance of $\mathcal{F}_{\text{Vess}}$, $\mathcal{S}_{\text{PA}}$ sends “abort” to $\mathcal{F}_{\text{PA}}$. Otherwise, for each client $c \in \mathcal{C}_h \cup \mathcal{C}'_c$, $\mathcal{F}_{\text{Vess}}$ outputs a share z_i^c for S_i , and $\mathcal{S}_{\text{PA}}$ records these.

$\mathcal{S}_{\text{PA}}$ also runs an instance of $\mathcal{F}_{2\text{PC}}$. For each $c \in \mathcal{C}_h \cup \mathcal{C}'_c$, $\mathcal{S}_{\text{PA}}$ receives $Z_i[c]$ from $\mathcal{A}$, sends it to $\mathcal{F}_{2\text{PC}}$, and computes the error $e_c = Z_i[c] - z_i^c$. $\mathcal{S}_{\text{PA}}$ sends these recorded errors to $\mathcal{F}_{\text{PA}}$ to obtain an output y , and sends y to $\mathcal{A}$ as the response from $\mathcal{F}_{2\text{PC}}$. If $\mathcal{A}$ sends “abort” to $\mathcal{F}_{2\text{PC}}$, $\mathcal{S}_{\text{PA}}$ sends “abort” to $\mathcal{F}_{\text{PA}}$. Otherwise, $\mathcal{S}_{\text{PA}}$ sends “continue” to $\mathcal{F}_{\text{PA}}$.

Crucially, this simulated view matches the real world since $\mathcal{F}_{\text{Vess}}$ ensures that the view of S_i is independent of the client’s input and, as a result, the output of $\mathcal{F}_{\text{PA}}$ (computed using the supplied errors) matches the output of $\mathcal{F}_{2\text{PC}}$ in the real world.

B.2 Instantiating $\mathcal{F}_{2\text{PC}}$ for the Sum Function

The 2PC protocol to compute the sum of server inputs uses a *commitment* protocol. We (informally) define the commitment functionality $\mathcal{F}_{\text{comm}}$ as follows.

$\mathcal{F}_{\text{comm}}$ has two procedures: **Com** and **Open**. In the **Com** procedure, $\mathcal{F}_{\text{comm}}$ takes in a value y and returns a commitment c and a randomness d . In the **Open** procedure, $\mathcal{F}_{\text{comm}}$ inputs a commitment c and randomness d , and returns an *opening* y if c, d is a valid commitment-randomness pair (obtained using the **Com** procedure on y); otherwise, it returns $\perp$.

Figure 16 presents a protocol that achieves $\mathcal{F}_{2\text{PC}}$ for the function that sums up the inputs of the two servers. During the protocol, each server S_i participates with a group element y_i , and the servers output the *sum* $y_0 + y_1$. The protocol is simple: the servers compute commitments on their inputs, exchange commitments, open these commitments by exchanging their randomness, and then sum up the values they obtained by opening the commitments. If a malicious server fails to send a commitment or correct randomness, the honest server aborts.

We note that most prior private aggregation protocols do not use commitments. As a result, an adversarial server can learn the output y before sending its share. This allows an adversarial server to set the output of the other server to be any *arbitrary* value of its choice. Π_{sum} acts as a plug-in replacement for the aggregation phase in prior works to prevent the described attack.

Theorem 6. *Protocol Π_{sum} presented in Figure 16 securely realizes $\mathcal{F}_{2\text{PC}}$ in the $(\mathcal{F}_{\text{comm}})$ -hybrid world.*

Proof. (sketch) We describe a simulator $\mathcal{S}_{\text{sum}}$ that interacts with $\mathcal{F}_{2\text{PC}}$ and $\mathcal{A}$ to simulate the view in the real world. The $\mathcal{F}_{2\text{PC}}$ functionality has a function f that simply sums up the inputs of the two servers. To produce a convincing protocol execution, the simulator will play the role of functionalities $\mathcal{F}_{\text{comm}}$.

The simulator obtains the input of the malicious server y_m as it plays the role of $\mathcal{F}_{\text{comm}}$. Then, it sends y_m to $\mathcal{F}_{2\text{PC}}$, and obtains the result y . The simulator computes $y_{1-m} = y - y_m$, and computes commitments $c_m, d_m = \text{Com}(y_m)$ and $c_{1-m}, d_{1-m} = \text{Com}(y_{1-m})$. The simulator sends c_{1-m} to the adversary. If the adversary does not send any commitment, the simulator sends “abort” to $\mathcal{F}_{2\text{PC}}$. Otherwise, the simulator sends d_{1-m} to the adversary. If the adversary does not send d_m , the simulator sends “abort” to $\mathcal{F}_{2\text{PC}}$. Otherwise, it sends “continue” to $\mathcal{F}_{2\text{PC}}$.

C Simulator for VESS

We first give the formal definition of Non-interactive zero-knowledge argument (NIZK):

Definition 2 (Non-interactive zero-knowledge argument). *Let $\mathcal{R}$ be an NP relation with language*

$$\mathcal{L}_{\mathcal{R}} = \{ x : \exists w \text{ s.t. } \mathcal{R}(x; w) = 1 \},$$

where x is a public statement and w a private witness. A non-interactive zero-knowledge argument (NIZK) for $\mathcal{R}$ is a pair of probabilistic polynomial time

(PPT) algorithms

$$\begin{aligned} \text{Prove}(x, w) &\rightarrow \Pi, \\ \text{Verify}(x, \Pi) &\rightarrow \{0, 1\}, \end{aligned}$$

satisfying the following:

- **Completeness.** For every (x, w) with $\mathcal{R}(x; w) = 1$,

$$\Pr[\text{Verify}(x, \Pi) = 1 : \Pi \leftarrow \text{Prove}(x, w)] \geq 1 - \text{negl}(\lambda).$$

- **Soundness.** For any PPT adversary $\mathcal{A}$,

$$\Pr[x \notin \mathcal{L}_{\mathcal{R}} \wedge \text{Verify}(x, \Pi) = 1 : (x, \Pi) \leftarrow \mathcal{A}(\lambda)] \leq \text{negl}(\lambda).$$

- **Zero knowledge.** There exists a PPT simulator Sim such that for every (x, w) with $\mathcal{R}(x; w) = 1$, the ensembles

$$\left\{ \Pi : \Pi \leftarrow \text{Prove}(x, w) \right\}_{\lambda} \stackrel{c}{\approx} \left\{ \Pi' : \Pi' \leftarrow \text{Sim}(x) \right\}_{\lambda}$$

are computationally indistinguishable. In particular, since Sim receives no witness, the proof reveals nothing about w beyond the truth of $x \in \mathcal{L}_{\mathcal{R}}$.

In Section 3.3, we describe our VESS protocol Π_{VESS} . Figure 17 presents a simulator that simulates the views of parties in the real world when some server is corrupt. Note that, in step 1, due to the security property of ESS (Figure 5), the distribution of k_{1-h} is indistinguishable from what the malicious S_{1-h} sees in the real world. The critical step is steps (5) and (6) of Figure 11. To simulate the view of the adversary in this step, we require that, during step (5), an honest server is never blamed by the client. We show this is indeed the case in Lemma 3. During step (6), we require that an honest client is never blamed by an honest server. This is ensured by the soundness of the NIZK scheme.

D Proof that VESS Implies Three-party Broadcast

Lemma 4. Consider any uniformly random even partition (P_0, P_1) of a vector V of size $2k$ that consists of 0s and 1s. If V has z 0s where $3k/4 \leq z \leq 5k/4$, then the probability that P_0 or P_1 has at least $3k/4$ 0s is negligible when $k = \text{poly}(\lambda)$.

Proof. Without loss of generality, we prove that this is true for P_0 . Since P_0 is a uniformly random half of S , P_0 follows the $\text{Hypergeometric}(2k, z, k)$ distribution³. The following [30] holds for $X \sim \text{Hypergeometric}(2k, z, k)$ and for any $0 < t < z/2k$:

$$\Pr[X \geq \frac{z}{2} + tk] \leq e^{-2t^2k}$$

³ $\text{Hypergeometric}(N, K, n)$ is a discrete probability distribution that describes the probability of s successes in n draws, without replacement, from a distribution of size N that contains K successful objects.

Simulator $\mathcal{S}_s$	
Setup:	
–	Let $\mathcal{A}$ be the adversary program. $\mathcal{S}_s$ writes the common randomness ρ , that is shared between the two servers, on $\mathcal{A}$'s random tape.
1.	Let S_h be the honest server and c be the honest client.
–	To simulate the client computing keys using its input, the simulator simply samples a random value $r \leftarrow \mathcal{I}$, computes $k_0, k_1 \leftarrow \text{ESS.DistrEncode}(r)$, and sends $\text{ES} = [\text{Enc}_{pk_0}(k_0), \text{Enc}_{pk_1}(k_1)]$ to the adversary program on behalf of $\mathcal{F}_{\text{BB}}$.
–	Send $\text{ESS.Expand}(k_{1-h})$ as the share of the malicious server to $\mathcal{F}_{\text{VSS}}$.
2.	– Invoke the simulator of Π_{ZKSD} with input $\text{ESS.Expand}(k_{1-h})$ to interact with $\mathcal{A}$ and simulate S_h and C . Record the messages S_h sends to S_{1-h} as T_h and the messages S_h receives from S_{1-h} as T_{1-h} .
–	If the ZKSD simulator sends “continue” to $\mathcal{F}_{\text{ZKSD}}$, send “continue” to the functionality $\mathcal{F}_{\text{VSS}}$.
3.	If the ZKSD simulator sends “invalid” (as opposed to “continue”) to $\mathcal{F}_{\text{ZKSD}}$, then:
–	If S_h is S_0 , send (T_0, T_1, ρ) to S_{1-h} on behalf of $\mathcal{F}_{\text{BB}}$ (since $\mathcal{S}_s$ emulates $\mathcal{F}_{\text{BB}}$).
–	If S_h is S_1 , receive (T'_0, T'_1, ρ') that S_{1-h} sends to $\mathcal{F}_{\text{BB}}$. Check if (T'_0, T'_1, ρ') match (T_0, T_1, ρ) . If they match, send “continue” to $\mathcal{F}_{\text{BB}}$. Otherwise, send “abort server” to $\mathcal{F}_{\text{VSS}}$ and finish.
4.	If S_h is S_0 , and if S_1 did not send “continue” to $\mathcal{F}_{\text{BB}}$, send “abort server” to $\mathcal{F}_{\text{VSS}}$ and finish.
5.	If $\mathcal{F}_{\text{BB}}$ outputs “continue”, either because $\mathcal{S}_s$ sent “continue” to $\mathcal{F}_{\text{BB}}$ (and S_1 is honest) or S_{1-h} sent “continue” to $\mathcal{F}_{\text{BB}}$ (and S_1 is corrupt), record the messages S_{1-h} sends to S_h . If S_{1-h} does not send k_{1-h} and Π such that $\text{NIZK.Verify}(pk, c, m, \pi) \rightarrow 1$, send “blame S_{1-h} ” to $\mathcal{F}_{\text{VSS}}$. Otherwise, send “continue” to $\mathcal{F}_{\text{VSS}}$.

Fig. 17. Simulator for a corrupt server in Π_{VSS} .

Now, we bound $\Pr[X \geq 3k/4]$ from above.

$$\begin{aligned} \Pr[X \geq 3k/4] &= \Pr[X \geq \frac{5k}{8} + \frac{k}{8}] \\ &\leq \Pr[X \geq \frac{z}{2} + \frac{k}{8}] \quad (\text{since } z \leq 5k/4) \end{aligned}$$

By setting $t = \frac{1}{8} \leq \frac{3}{8} \leq \frac{z}{2k}$, we can upper bound $\Pr[X \geq 3k/4]$:

$$\Pr[X \geq 3k/4] \leq e^{-\frac{k}{32}}$$

When $k = \text{poly}(\lambda)$ the above probability is negligible.

Theorem 2. Any protocol Π_{VSS} that implements $\mathcal{F}_{\text{VSS}}$ can be used to achieve a Byzantine Broadcast protocol among three parties that tolerates one byzantine fault.

Proof. In Figure 12 of Section 3.4, we present a protocol Π_{BB} that is built using any VESS protocol Π_{VESS} . Here, we prove that Π_{BB} is a byzantine broadcast protocol.

Proving correctness: First, suppose that S and receiver R_i are honest. Consider the following four exhaustive cases.

(1) *Some instance of Π_{VESS} outputs “blame R_{1-i} ”.* In this case, R_i outputs m received from S .

(2) *R_0 is malicious and does not send partitions to R_1 .* In this case, R_1 outputs m received from S .

(3) *R_{1-i} does not provide shares of outputs in P_i to R_i .* In this case, R_i outputs m received from S .

(4) *All instances of Π_{VESS} output shares, and R_i obtains shares for instances in P_i .* R_i receives n shares from R_{1-i} to compute the set $\{y^j\}_{j \in P_i}$. If any of these are invalid, i.e. $\text{Decode}(y^j) \notin \mathcal{I}_{R_i}$, R_i outputs m received from S . Otherwise, by Property 2, at least $3k/4$ values are from V_m , and R_i sets $b_i = m$ and outputs m .

In all the above cases, correctness holds.

Proving consistency: Consistency is trivial when one receiver is corrupt. Consider the case that both receivers are honest and S is corrupt. Consider the following exhaustive cases.

(1) *Some instance of Π_{VESS} outputs “blame S ”.* In this case, both receivers output the default bit 1.

(2) *No instance of Π_{VESS} outputs “blame S ”.* In this case, Π_{VESS} ensures that all reconstructed inputs are valid, and each R_i computes b_i by checking if $3k/4$ values in $\{y^j\}_{j \in P_i}$ belong to V_b . Consider the following cases.

- *Some receiver R_i sets $b_i = \perp$.* If R_{1-i} also sets $b_{1-i} = \perp$, then both receivers send “None” messages, and output the default bit 1. Otherwise, R_{1-i} sets b_{1-i} to some bit b , and R_i receives this from R_{1-i} and outputs b .
- *No receiver R_i sets $b_i = \perp$.* We show that it is not possible for R_i to set $b_i = b$ and R_{1-i} to set $b_{1-i} = 1 - b$ (except with negligible probability). For contradiction, suppose that this can occur with non-negligible probability. Then, the set of inputs that S participates with must include at least $3k/4$ inputs from V_0 and $3k/4$ inputs from V_1 , meaning that the number of inputs from V_0 (and V_1) lies in the range $[5k/4, 3k/4]$. But since P_0 and P_1 are uniformly random even partitions of $\{1, 2, \dots, 2k\}$, from Lemma 4, the probability that $\{y^j\}_{j \in P_i}$ contains at least $3k/4$ values from V_b is negligible.

In all the above cases, consistency holds. Hence, Π_{BB} is a Byzantine Broadcast protocol.

E Simulating PA for Intermittent Clients

In Section 4, we present a compiler (Figure 14) that compiles any PA protocol $\Pi_{\text{PA}}^{\text{S}}$ designed for synchronous clients into a protocol $\Pi_{\text{PA}}^{\text{I}}$ that handles intermittent clients. In this section, we formally prove that the protocol output by our compiler achieves the notion of PA for intermittent clients defined in Figure 13.

Lemma 5. *If S_0 and S_1 are honest, they agree on flag_c for each client $c \in \mathcal{C}$.*

Proof. While executing step 1.c, observe that an honest S_i sets flag_c to 0 only if it doesn't send $\langle \text{ack}, r, c \rangle_{S_i}$ to S_{1-i} , and an honest S_{1-i} sets flag_c to 0 only if it doesn't receive $\langle \text{ack}, r, c \rangle_{S_i}$ from S_i . Hence, both servers agree on whether they see $\langle \text{ack}, r, c \rangle_{S_i}$, and agree on the value of flag_c computed during this step. During step 2.c, if an honest server receives $\langle \text{ack}, r, c \rangle_c$ from c , it forwards this to the other server. Both servers agree on whether they see $\langle \text{ack}, r, c \rangle_c$ during this step, and compute the same value for flag_c .

Lemma 6. *A server aborts during the share phase of $\Pi_{\text{PA}}^{\text{I}}$ only if the other server is corrupt.*

Proof. First, suppose that a server S_{1-i} aborts while executing step (1) of $\Pi_{\text{PA}}^{\text{I}}$ for some round r of $\Pi_{\text{PA}}^{\text{S}}$. Then, although S_i was supposed to send $\langle \text{ack}, r, c \rangle_{S_i}$ during step 1.c of $\Pi_{\text{PA}}^{\text{I}}$, it did not. So, S_i is corrupt.

Next, suppose that a server S_{1-i} aborts during step (2) of $\Pi_{\text{PA}}^{\text{I}}$ for some round r of $\Pi_{\text{PA}}^{\text{S}}$. Since two honest servers agree on the value of flag_c , for each client $c \in \mathcal{C}$. Then, S_i was supposed to send $\langle r, c, \{m\}_c \rangle_{S_i}$ during step 2.a of $\Pi_{\text{PA}}^{\text{I}}$, but it did not. So, S_i is corrupt.

Lemma 7. *If a client c is honest and synchronous and no server aborts, then during the share phase (1) no honest server sets flag_c to 0 and (2) c does not abort.*

Proof. Servers set flag_c to 0 either during step (1) or step (2) of $\Pi_{\text{PA}}^{\text{I}}$ for some round r of $\Pi_{\text{PA}}^{\text{S}}$. Consider an execution of step (1) when c sends a message m for round r to S_i . Since c is synchronous, S_i and S_{1-i} receive m and $\langle r, c, \{m\}_{S_i} \rangle_c$ from c at the end of step 1.a, and S_i does not set flag_c to 0. Since S_{1-i} does not abort, it receives $\langle \text{ack}, r, c \rangle_{S_i}$ from S_i at the end of step 1.c and does not set flag_c to 0. c receives $\langle \text{ack}, r, c \rangle_{S_i}$ either from S_i or from S_{1-i} by the end of step 1.d, and does not abort.

Next, consider an execution of step (2) when S_i sends a message m to c . Since S_{1-i} does not abort, it receives $\langle r, c, \{m\}_c \rangle_{S_i}$ from S_i by the end of step 2.a, and forwards this to c . Since c receives $\langle r, c, \{m\}_c \rangle_{S_i}$ from either of the servers by the end of step 2.b, it does not abort, and sends $\langle \text{ack}, r, c \rangle_c$ to both the servers. Since c is synchronous, the servers receive this by the end of step 2.d, and do not set flag_c to 0.

Lemma 8. *If S_0 and S_1 have flag_c set to 1 for a client c at the end of round $4r + 3$ of $\Pi_{\text{PA}}^{\text{I}}$, then, for client c , then for each round $0 \leq r' \leq r$, (1) S_0 and S_1 receive a message from c and (2) c receives messages from S_0 and S_1 . Messages sent by honest parties are computed according to a (synchronous) run of the share phase of $\Pi_{\text{PA}}^{\text{S}}$ until round r .*

Proof. Each message m that c has to send to any server S_i (either honest or corrupt) during some round r of $\Pi_{\text{PA}}^{\text{S}}$ is received by S_i by the end of round $4r + 3$ of $\Pi_{\text{PA}}^{\text{I}}$. This is because, by the end of round $4r + 3$ of step (1) (step 1.d),

- *Case 1: S_i is honest.* Since flag_c is 1 for S_i , S_i has sent $\langle \text{ack}, r, c \rangle_{S_i}$, indicating that S_i receives a value which, by the unforgeability of digital signatures, must be $(r, c, \{m\}_{S_i})$.
- *Case 2: S_i is corrupt.* Since flag_c is 1 for S_{1-i} (which is honest), S_i has sent $\langle \text{ack}, r, c \rangle_{S_i}$, indicating that S_i received $(r, c, \{m\}_{S_i})$.

Similarly, each message that any server S_i (either honest or corrupt) has to send to c during some round r of Π_{PA}^S is received by c by the end of round $4r + 3$ of Π_{PA}^I . By the end of round $4r + 3$ of step 2.d

- *Case 1: S_i is honest.* Since flag_c is 1 for S_i , S_i receives $\langle \text{ack}, r, c \rangle_c$ from c , indicating that c receives a value which, by the unforgeability of digital signatures, is $(r, c, \{m\}_c)$.
- *Case 2: S_i is corrupt.* Since flag_c is 1 for S_{1-i} (which is honest), S_{1-i} receives $\langle \text{ack}, r, c \rangle_c$ from c , indicating that c receives $(r, c, \{m\}_c)$. $\square$

Theorem 8. *A real-world execution of the protocol output by $\Pi_{\text{PA}}^S \rightarrow \Pi_{\text{PA}}^I$ (Figure 14) is indistinguishable from an ideal-world execution of the ideal functionality $\mathcal{F}_{\text{IPA}}$ (Figure 13).*

Proof. Figures 18 and 19 present a simulator $\mathcal{S}_{\Pi_{\text{PA}}^S \rightarrow \Pi_{\text{PA}}^I}$ that interacts with $\mathcal{F}_{\text{PA}}$ to simulate views that match parties' real-world execution of the share phase of Π_{PA}^I . At a high level, $\mathcal{S}_{\Pi_{\text{PA}}^S \rightarrow \Pi_{\text{PA}}^I}$ invokes the simulator $\mathcal{S}_{\text{PA}}^S$ for Π_{PA}^S to simulate messages that must be sent from each client c to each server S_i (if S_i is corrupt), or from S_i to c (if c is corrupt). It does this simply by following the description of the compiler $\mathcal{S}_{\Pi_{\text{PA}}^S \rightarrow \Pi_{\text{PA}}^I}$ to simulate the views of (each) client c and the servers S_i and S_{1-i} , while using adversarial input to decide whether messages of asynchronous clients are delivered or not.

Importantly, $\mathcal{S}_{\Pi_{\text{PA}}^S \rightarrow \Pi_{\text{PA}}^I}$ also helps decide the set $\mathcal{C}'$ (clients whose inputs are included in the aggregation) to be sent to $\mathcal{F}_{\text{PA}}$ by simulating whether the honest server(s) sets flag_c to 1 or 0. This allows us to use $\mathcal{S}_{\text{PA}}^S$ to simulate the aggregation phase; we can use $\mathcal{S}_{\text{PA}}^S$ to extract the input of each corrupt client c with flag_c set to 1, and send these inputs (along with the set $\mathcal{C}'_c$ of corrupt parties) to $\mathcal{F}_{\text{PA}}$.

We next show that this simulates the view in the real world by considering two cases.

- *No server aborts.* By Lemmas 5 and 6, in the real world, both servers complete the share phase and agree on the set of client inputs to be included in the aggregation function. By Lemma 7, the inputs of all honest and synchronous clients are included in the aggregation, as flag_c is set to 1 for all these clients. By Lemma 8, in both the real and ideal world, the run of the share phase for clients with $\text{flag}_c = 1$ at the end of Π_{PA}^I is like that of Π_{PA}^S . By the guarantees of Π_{PA}^S , the servers' output matches that of $\mathcal{F}_{\text{PA}}$ when $\mathcal{C}'$ is the set of adversarial / intermittent clients with flag_c set to 1.
- *Some server S_i causes the protocol to abort.* In the real world, this occurs when S_i causes Π_{PA}^S to abort, or when S_i does not send $\langle \text{ack}, r, c \rangle_{S_i}$ or a cipher-text to S_{1-i} during step (1) or (2). In both cases, $\mathcal{S}_{\Pi_{\text{PA}}^S \rightarrow \Pi_{\text{PA}}^I}$ sends

Simulator $\mathcal{S}_{\Pi_{\text{PA}}^{\text{S}} \rightarrow \Pi_{\text{PA}}^{\text{I}}}$	
<i>Setup:</i> For each honest server S_i (resp. honest client c), invoke a PKI setup to compute a secret-key public-key pair (sk_i, pk_i) (resp. (sk_c, pk_c)). Send pk_i (resp. pk_c) to $\mathcal{A}$. Set flag_c to 1 for each client $c \in \mathcal{C}$. <i>Simulation of step(1).</i> Invoke $\mathcal{S}_{\text{PA}}^{\text{S}}$ to compute message m to be sent from c to S_i during round r of $\Pi_{\text{PA}}^{\text{S}}$. Then,	
(a)	If c is honest, (i) if S_i is corrupt, to simulate its view, send (r, m) to $\mathcal{A}$ (ii) if S_{1-i} is corrupt, to simulate its view, send $\langle r, c, \{m\}_{S_i} \rangle_c$ to $\mathcal{A}$.
(b)	If S_{1-i} is honest and S_i is corrupt, then to simulate S_i's view, – If c is honest and synchronous, send $\langle r, c, \{m\}_{S_i} \rangle_c$ to $\mathcal{A}$. – If c is honest and intermittent, if $\mathcal{A}$ allows $\langle r, c, \{m\}_{S_i} \rangle_c$ to be received by the start of this round, send $\langle r, c, \{m\}_{S_i} \rangle_c$ to $\mathcal{A}$. – If c is corrupt, if some cipher-text is received from $\mathcal{A}$, send this to $\mathcal{A}$. If none of the above cases hold, set flag_c to 1 for S_{1-i}.
(c)	If S_i is honest, – If c is corrupt, then to simulate its view, send $\langle \text{ack}, r, c \rangle_{S_i}$ to $\mathcal{A}$ if either of the following hold. • $\mathcal{A}$ sent (r, m) to S_i on behalf of c. • S_{1-i} is honest, and $\mathcal{A}$ sent a cipher-text of the form $\langle r, c, \{m\}_{S_i} \rangle_c$ to S_{1-i} by the end of round $4r$ • S_{1-i} is corrupt, and $\mathcal{A}$ sent a cipher-text of the form $\langle r, c, \{m\}_{S_i} \rangle_c$ to S_i. – If S_{1-i} is corrupt, then to simulate its view, send $\langle \text{ack}, r, c \rangle_{S_i}$ to $\mathcal{A}$ if either of the following hold. • c is honest and synchronous. • c is honest and intermittent, and if either (1) $\mathcal{A}$ allows (r, m) to be received from c, or (2) $\mathcal{A}$ sends $\langle r, c, \{m\}_{S_i} \rangle_c$ on behalf of S_{1-i}. • c is corrupt and if either (1) $\mathcal{A}$ sends (r, c, m) on behalf of c, or (2) $\mathcal{A}$ sends $\langle r, c, \{m\}_{S_i} \rangle_c$ on behalf of S_{1-i}. If $\langle \text{ack}, r, c \rangle_{S_i}$ was sent on behalf of S_i, set flag_c to 1 on behalf of S_i.
(d)	If S_{1-i} is honest and if c is corrupt, then, to simulate c's view, – If S_i is honest, send $\langle \text{ack}, r, c \rangle_{S_i}$ to $\mathcal{A}$ if $\langle r, c, \{m\}_{S_i} \rangle_c$ was sent by $\mathcal{A}$ to S_{1-i} by the end of round $4r$. – If S_i is corrupt, then, if a cipher-text was received from $\mathcal{A}$ by the end of round $4r$, (1) if no $\langle \text{ack}, r, c \rangle_{S_i}$ was sent by $\mathcal{A}$ on behalf of S_i, send “abort” to $\mathcal{F}_{\text{PA}}$ and output what $\mathcal{A}$ outputs (2) else, send $\langle \text{ack}, r, c \rangle_{S_i}$ to $\mathcal{A}$.

Fig. 18. Simulator for the compiler from synchronous PA to one that handles intermittent clients.

“abort” to $\mathcal{F}_{\text{PA}}$ in the ideal world; in the former case, this should be the output of $\mathcal{S}_{\text{PA}}^{\text{S}}$ and in the latter case, $\mathcal{S}_{\Pi_{\text{PA}}^{\text{S}} \rightarrow \Pi_{\text{PA}}^{\text{I}}}$ keeps track of what S_i is supposed to send during steps (1) and (2) and sends “abort” to $\mathcal{F}_{\text{PA}}$ if this does not happen. $\square$

Simulator $\mathcal{S}_{\Pi_{\text{MFA}} \rightarrow \Pi_{\text{AMFA}}}^{\text{S}}$ (continued)	
<i>Simulation of step(2).</i> Invoke $\mathcal{S}_{\text{PA}}^{\text{S}}$ to compute message m to be sent from S_i to c during round r of $\Pi_{\text{PA}}^{\text{S}}$. Then,	
(a)	If S_i is honest (i) if c is corrupt, to simulate its view, send (r, m) to $\mathcal{A}$ (ii) if S_{1-i} is corrupt, to simulate its view, send $\langle r, c, \{m\}_c \rangle_{S_i}$ to $\mathcal{A}$.
(b)	If S_{1-i} is honest, – If c is corrupt, to simulate its view, send $\langle r, c, \{m\}_c \rangle_{S_i}$ to $\mathcal{A}$ if (1) S_i is honest or (2) S_i is corrupt and $\mathcal{A}$ sends $\langle r, c, \{m\}_c \rangle_{S_i}$ on behalf of S_i. – If c is honest and S_i is corrupt, if $\mathcal{A}$ does not send $\langle r, c, \{m\}_c \rangle_{S_i}$, send “abort” to $\mathcal{F}_{\text{PA}}$.
(c)	If c is honest and synchronous, to simulate corrupt S_i / S_{1-i}’s view, send $\langle \text{ack}, r, c \rangle_c$ to $\mathcal{A}$. If c is honest and intermittent, to simulate corrupt S_i / S_{1-i}’s view, send $\langle \text{ack}, r, c \rangle_c$ to $\mathcal{A}$ if either – S_i is honest and $\mathcal{A}$ allows c to receive (r, m). – S_{1-i} is honest and $\mathcal{A}$ allows c to receive $\langle r, c, \{m\}_c \rangle_{S_i}$.
(d)	If c is honest and synchronous, then, – If S_i is honest and S_{1-i} is corrupt, then to simulate S_{1-i}’s view, send $\langle \text{ack}, r, c \rangle_c$ to $\mathcal{A}$. – If S_{1-i} is honest and S_i is corrupt, then to simulate S_i’s view, send $\langle \text{ack}, r, c \rangle_c$ to $\mathcal{A}$. If c is honest and intermittent, then, – If S_i is honest and S_{1-i} is corrupt, then to simulate S_{1-i}’s view, if $\mathcal{A}$ allows S_i to receive $\langle \text{ack}, r, c \rangle_c$, send $\langle \text{ack}, r, c \rangle_c$ to $\mathcal{A}$. – If S_{1-i} is honest and S_i is corrupt, then to simulate S_i’s view, if $\mathcal{A}$ allows S_{1-i} to receive $\langle \text{ack}, r, c \rangle_c$, send $\langle \text{ack}, r, c \rangle_c$ to $\mathcal{A}$. Set flag_c to 1 on behalf of honest S_i / S_{1-i} if S_i / S_{1-i} sent $\langle \text{ack}, r, c \rangle_c$.
<i>Simulating step (3)</i> If S_i is corrupt, use $\mathcal{S}_{\text{PA}}^{\text{S}}$ to simulate m to be sent to S_i during round r of $\Pi_{\text{PA}}^{\text{S}}$. Send this during round $4r$ of $\Pi_{\text{PA}}^{\text{I}}$.	
<i>Simulating the aggregate phase.</i> Let $\mathcal{C}'_{\text{c}} \cup$ and $\mathcal{C}'_{h_i}$ be the sets of adversarial and intermittent clients with flag_c set to 1. Use $\mathcal{S}_{\text{PA}}^{\text{S}}$ to compute x_c for each adversarial $c \in \mathcal{C}'_{\text{c}}$. Send $\mathcal{C}' = \mathcal{C}'_{\text{c}} \cup \mathcal{C}'_{h_i}$ and $\{x_c\}_{c \in \mathcal{C}'}$ to $\mathcal{F}_{\text{PA}}$. Send this to $\mathcal{A}$ and use $\mathcal{S}_{\text{PA}}^{\text{S}}$ to compute the output y to be sent to $\mathcal{F}_{\text{PA}}$. Send y to $\mathcal{F}_{\text{PA}}$.	

Fig. 19. Simulator for the compiler from synchronous PA to one that handles intermittent clients (continued).