Willow: Secure Aggregation with One-Shot Clients

James Bell-Clark Adrià Gascón Baiyu Li Mariana Raykova
Phillipp Schoppmann

Google

October 15, 2024

Abstract

A common drawback of secure vector summation protocols in the single-server model is that they impose at least one synchronization point between all clients contributing to the aggregation. This results in clients waiting on each other to advance through the rounds of the protocol, leading to large latency (or failures due to too many dropouts) even if the protocol is computationally efficient. In this paper we propose protocols in the single-server model where clients contributing data to the aggregation (i) send a single message to the server and (ii) can join aggregation sessions dynamically whenever they have resources, i.e., without the need for synchronizing their reporting time with any other clients. Our approach is based on a committee of parties that aid in the computation by running a setup phase before data collection starts, and a verification/decryption phase once it ends. Unlike existing committee-based protocols such as Flamingo (S&P 2023), the cost for committee members can be made sub-linear in the number of clients, and does not depend on the size of the input client vectors. Our experimental evaluation shows that our protocol, even while allowing dynamic client participation, is competitive with the state of the art protocols that do not have that feature in both computation and communication.

Contents

1 Introduction

Secure aggregation enables a server to learn an aggregate of the inputs of many users. It has wide application to private analytics and federated learning and has been studied in numerous papers [1, 2, 3, 4, 5, 6, 7, 8, 9]. One of the main disadvantages of many of the solutions is the fact that they require multiple synchronization points between clients. This is problematic when supporting large numbers of clients, or low client availability.

To see why, consider a setting where clients with appropriate data check in with the server at a rate of 10 clients per second. Gathering 10^4 clients will take over 15 minutes. By the time the last few clients appear, the first few ones might have dropped out. This is expected to happen with clients with unreliable connection. Therefore, synchronization among clients providing inputs is undesirable because (i) the latency of the protocol is then dominated by the "client gathering" phase, and (ii) clients are expected to be online for a long time.

One way to remove the need for synchronization among clients is to assume two non-colluding servers that can process the clients' contributions jointly. There are such constructions [10, 11, 12], which require both servers to receive communication proportional to all inputs and then do work that is also linear in the input size. Therefore, such solutions require finding parties that could both satisfy the non-collusion assumption, and also have the resources.

In contrast, in the single-server setting there is no solution that can obtain security and privacy relying only on the server. Even expensive primitives such as obfuscation and multi-input functional encryption would not directly achieve this since they still allows mix-and-match attacks across multiple contributions from different clients. Therefore, our protocols, similar to recent works [7, 9] relies on a small committee of parties that aid the server in the computation.

In this work, we present protocols that allow clients to contribute their inputs *at any time* with *a single message* to the server. Moreover, unlike prior works, our committee can be instantiated such that each committee member only does work *sub-linear* in the number of clients contributing inputs. Moreover, the committee's work is independent of the length of the vectors being aggregated, which makes our protocols well suited for large-scale applications. Our protocols also do not require (non-committee) clients to have fixed identities from the beginning, but rather allow *dynamic participation*.

At a high level, our protocols work by having the server homomorphically aggregate client contributions, without learning anything about the processed inputs. At the end of the protocol, it invokes the committee to obtain the decryption key for the final result. To protect against a malicious server, we additionally have the committee verify that each client is included in the aggregation at most once. We show that this verification work can be distributed efficiently, with both the committee size and the work of each committee member being sub-linear in the number of clients.

An important property of our protocols is the fact that both verification and decryption happen after all clients have sent their inputs. An alternative instantiation of our protocols could therefore be to implement these roles using a single second server. Unlike Prio and related two-server aggregation protocols [10, 11, 12], the work performed by the second server in this variant of our protocols is independent of the vector length.

1.1 Contributions

We propose protocols for secure aggregation of n private vectors $\mathbf{x_1}, \ldots, \mathbf{x_n}$ of length ℓ, held by n clients $\mathsf{C}_1, \ldots, \mathsf{C}_n$, respectively. In each aggregation, clients send a single message to the server, and do $O(\ell \log n)$ work. Our protocols also involve a decryptor role that can be implemented by a small committee D of parties that we call decryptors, each doing $O(|\mathsf{D}| + \log n)$ work independent of ℓ.

Our protocols withstand a *non-adaptive active* adversary simultaneously corrupting a minority of the decryptors, an arbitrary number of clients, and the server We rely on a committee V of parties called verifiers (which could overlap with D). Collectively, V's cost is $O(n)$, distributed within the committee so that each verifier does $O(n \log(n)/|\mathsf{V}|)$ work. We summarize our contributions next.

1. We formalize one-shot distributed vector summation in the real vs. ideal paradigm of secure computation by proposing a natural ideal-world functionality. We then show that realizing that functionality within

		Flamingo [7]	Acorn [8]	Ours
Client	**Comp.**	$c_d + \ell \log n$	$\ell \log n$	$\ell \log n$
	Comm.	$c_d + \ell \log n$	$\ell \log n$	$\ell \log n$
	One-Shot	✓	✗	✓
	Dynamic	✗	✗	✓
Decryptor	**Comp.**	$c_d^2 + n$	N/A	$c_d + \log n$
	Comm.	$c_d^2 + n \log n$		$c_d + \log n$
Server	**Comp.**	$c_d + n\ell \log n$	$n\ell \log n$	$n\ell \log n$
	Comm.	$n(c_d + \ell + \log n)$	$n\ell \log n$	$n\ell \log n$
Verifier	**Comp.**	N/A	N/A	n/c_v
	Comm.			n/c_v

Table 1: Comparison with the committee-based Flamingo and RLWE-based Acorn protocols. We report asymptotic costs with respect to n (number of clients), c_d (number of decryptor committee members), c_v (number of verifiers) and ℓ (input length), omitting dependencies on security parameters and input bit-width, and we drop the $O(.)$ notation for clarity. "One-shot" means that clients send a single message per aggregation, and "dynamic" means client can join the protocol at any point without needing a PKI (see Section 2).

our efficiency requirements (low communication for committee members) is impossible in the standard model. For this we exhibit a class of real-world attackers for which white box simulation (and therefore also black-box simulation) is impossible.

2. We present the first secure aggregation protocols with one-shot clients and dynamic participation in the single-server model, with committee members doing work independent of the input length ℓ. Moreover, using a committee of size $O(n^{1/\alpha})$, with $\alpha \geq 2$ results in all committe members doing work $o(n)$. Our protocols rely on the hardness of Ring Learning With Errors (RLWE) assumption. We also propose a more concretely efficient covertly secure variant of the verifier, i.e. where a misbehaving corrupted server gets caught by the protocol with constant probability. This variant is suitable for settings where a misbehaving server faces risk of reputation loss.
3. We implement our constructions and show that our protocol is practical. For example, for vectors of length 10^5, our protocol requires under 500KB of client upload and 227ms of server computation per client.

As part of our contribution, we develop a key and value homomorphic encryption scheme KAHE based on the Hint-RLWE assumption, as well as a threshold additively homomorphic encryption scheme AHE based on the standard RLWE assumption, which both may be of independent interest.

In Appendix B we review prior and concurrent works on secure aggregation. We compare the asymptotics of our protocol against the most relevant prior works in Table 1. For a practical comparison, we refer to Section 6.

2 Setting and Threat Model

In our setting, a server S aims to compute the sum of n vectors $\mathbf{x_i} \in \mathbb{F}^\ell$ held by a sample of a population of clients. The communication pattern has the server at the center of a star network. In our model, clients check-in with the server whenever some eligibility conditions are satisfied, e.g., when they have data to contribute and are in an idle state. Then, they get instructed to engage in the aggregation protocol. Therefore, we require that the clients that participate in a given aggregation are not determined up front, but decided/chosen dynamically as the protocol goes along. We call this property *dynamic client participation*, and in particular means that at the time a contribution is made, neither the client or server can be assumed to know who the other clients are.

The pool of clients may include devices with limited connectivity and computational resources. Therefore, our goal is *one-shot clients*, where a single message is enough for a client to contribute to an aggregation. As discussed above, this property eliminates the latency observed in practice due to the long tail of client response times.

Note that our protocol still requires clients to obtain the public key of the decryptor. This is needed even in Prio [10], or the insecure baseline with a single trusted server. We conjecture that the decryptor's public key can be re-used across multiple aggregations, as the secret key does not get revealed as part of the protocol, and replay attacks can be mitigated by tying contributions to one particular aggregation task by adding an aggregation ID to the statement proven in the verifiable encryption scheme (see Section 4.2). However, we do not optimize for this use case, and for the rest of the paper we assume the decryptor public key is tied to one particular aggregation.

2.1 Roles & Assumptions

Our protocols offer different instantiations. While the main instantiation is an aggregation in the single-server setting, where some trust is placed on the clients to whom computation is outsourced, it can also be realized in the two-server model. That is why we present our protocol in terms of different *roles*:

- Clients $\mathsf{C}_1, \ldots, \mathsf{C}_n$: These are the providers of data to be aggregated. There will be many of them some of whom may be corrupt. We would like them to do as little work as possible, including minimizing the amount of time they need to be online. Our protocols ensure that client's data remains private. We have no assumptions on the honesty of clients, and therefore an adversary might corrupt up to $n - 1$ clients. This allows active adversaries to mount sybil attacks, which we discuss later. We only require clients to know the decryptor's public key, but do not require a PKI between clients.
- Server S: The entity orchestrating the protocol, in the non-secure setting this is the party that client data is sent to. This party is capable of a significant amount of computation and communication. It is also the output recipient but it should not learn anything else about the input data (within the threat model that is considered).
- Decryptor D: The decryptor role can be instantiated as a committee of client-like parties, a small number of servers, or a single second server. To guarantee privacy, the decryptor is not allowed to collude with the server. When implemented by a committee of parties, this means that a majority of decryptors must remain honest.
- Verifier V: This role exists in the version of the protocol secure against an actively corrupted server. This party does not hold any state, and its purpose is to verify a *public* data structure generated by the Server. It is also assumed to not collude with the server, and similar to the decryptor it can be instantiated by a committee of clients, or a small set of trustworthy parties.

2.2 Failure & Threat Model

As discussed above, our security assumptions are that (i) the decryptor and the server do not collude and (ii) the verifier and the server do not collude.

Distributed Decryptor/Verifier. In the case where the role of the decryptor is distributed across $c_d > 1$ parties, which we call *decryptors*, our protocol assumes that no more than a minority fraction γ_d of the decryptors are corrupted. In the Flamingo work [7] the decryptor role is assigned by means of a trusted source of randomness, such as the one offered by Cloudfare [13]. Our protocols do not pose any constraints on how decryptors are selected, as long as the above assumption is satisfied. Moreover, there is no restriction on the value of c_d (beside being positive), and therefore the decryptor role could be implemented in the 3 parties, honest majority setting. Nevertheless, in our experiments we assume $c_d = 100$, to highlight that the (distributed) decryptor role is lightweight. In terms of robustness to dropouts, our protocols are robust to as many as $(1 - \gamma_d)c_d + 1$ committee members dropping out. In terms of correctness, we ensure

Aggregation Functionality $\mathcal{F}^{\mathsf{Agg}}$

Public Parameters:

- Input domain $\mathbb{F}^\ell$.
- Number of clients n.
- Minimum number of clients in the aggregation $\texttt{min\_n}$.

Parties:

- Server S.
- n clients $1, \ldots, n$, each holding private input $\mathbf{x_i} \in \mathbb{F}^\ell$.
- Trusted party T.
- Adversary $\mathcal{A}$ corrupting a subset of parties $\mathcal{C} \subseteq [n] \cup \{\mathsf{S}\}$.

Functionality:

1. T receives all client inputs, with corrupted clients' inputs chosen by $\mathcal{A}$.
2. If S is corrupted, $\mathcal{A}$ chooses $S \subseteq [n]$. Otherwise $S := [n]$.
3. $\mathcal{A}$ chooses $d \in \{\texttt{continue}, \texttt{abort}\}$ and sends (S, d) T.
4. If $|S| \geq \texttt{min\_n}$ and $d = \texttt{continue}$, T sends $\sum_{i \in S} \mathbf{x_i}$ to S, else T sends $\perp$ to S.

Figure 1: The aggregation functionality $\mathcal{F}^{\mathsf{Agg}}$ with non-selective abort.

security with abort, in the sense that corrupted committee member can cause the protocol to abort, but non-adaptively, i.e., without learning the result. Therefore, our protocol does not have guaranteed output delivery. This is also the case in other protocols such as the main protocols Flamingo [7], Bell et al. [3], and Acorn [8], although both Flamingo and Acorn present more costly extensions to that property. We do not see a fundamental limitation there, but in the present work we chose to focus on a lightweight decryptor and avoid costly primitives like verifiable secret sharing. Such an extension is left for further work.

Regarding distributed verifier, as mentioned above, the task of the verifier(s) boils down to checking a public data structure, in the same spirit of key transparency. Analogously to the decryptor role, our protocol assumes that no more than a minority fraction γ_v of the c_v verifiers are corrupted, and allow for a fraction of dropouts.

Passive/Active Security. All of the protocols we present in this paper are secure against a malicious / active adversary that may control any number of clients, the server, and a minority of decryptors and verifiers. We prove security in the standard ideal/real simulation paradigm. The simplest version of our protocol (Section 5) achieves full privacy against malicious servers, but does not make any correctness guarantees when the server is honest or semi-honest. In Appendix H.2, we discuss how to improve the guarantees by restricting the adversary's power to aborting the protocol non-adaptively.

Public Key Infrastructure. In our protocol, decryptors and verifiers need to establish secure channels among themselves, and apply cryptographic signatures. Therefore, as in previous works [1, 3, 7, 8], we rely on an external PKI, or a verifiable public key directory. For the latter option, Flamingo suggests a construction such as CONIKS [14] and its successors. An important difference with previous works, however, is that we only need a PKI among parties taking on the decryptor or verifier role, *not clients contributing data*.

2.3 Functionality

Functionality $\mathcal{F}^{\mathsf{Agg}}$ described in Figure 1 formalizes the security guarantee of our main protocol. We shall discuss three important aspects of the definition:

The min_n requirement. In our setting we have a large number of n clients that might *potentially* contribute to an aggregation in a dynamic way, i.e., without an established participation schedule or fixed client identities. At the same time, we want clients to be *one-shot*, i.e., only send a single message to the server. Our first observation is that any protocol with these properties must allow the server to ignore clients, as the server is the only party clients communicate with, and there is no way to distinguish between a client that never showed up, and one that the server chooses to ignore. Now if we allow for client dropouts, we want to allow protocols to abort if a malicious server ignores too many clients, such that no meaningful aggregation is happening. We stress that this is optional, and protocols may allow an arbitrary number of dropouts by setting min_n to 0. The protocols we describe in this work can be instantiated for any $0 \leq \texttt{min\_n} \leq n$, and the lower bound we present in Appendix F holds as long as the gap between n and min_n is not too large, i.e., the protocol tolerates at least a few dropouts.

Correctness. The adversary can choose to abort the functionality non-selectively (that is, without seeing the output first), and cannot influence the output beyond choosing corrupt clients' inputs. We note that in our protocols, an abort can only happen if the adversary actively corrupts a party that is not just a regular client, meaning that our protocols are not susceptible to denial of service attacks by regular clients. In Appendix I.2 we further describe an extension of our protocol that enables guaranteed output delivery.

Sybil attacks An important observation, which applies to any secure aggregation protocol, is that if the adversary has the ability to control a large majority of the clients, e.g. $n-1$, then the protections of secure aggregation alone are futile, as the adversary knows the input of all $n-1$ corrupted clients, and thus can recover the victim's input form the computed sum. We refer to this situation as a Sybil attack. An option to handle this situation is to enforce a large min_n in combination with some bound on the maximum number of corrupted clients. For settings where this cannot be ensured, we describe a variant of our protocol that provides differential privacy in Appendix I.1.

3 Technical Overview

Our protocols revolve around the high-level idea of *unbalanced MPC protocols*. In generic MPC protocols, it is often the case that all parties do amounts of work – either computation or communication – of the same order. For example, in Yao's garbled circuits, both parties do work proportional to the size of the computation. The same observation applies to other protocols in the two-server model that rely on outsourcing the computation to two non-colluding powerful servers, e.g. Prio [10], DPF-based aggregation [12]. In settings where a server S aims to process large amounts of data, balanced protocols like the ones mentioned above might be hard to instantiate in practice, as the non-colluding second party in the computation must have similar resources as the first server.

The above observation is particularly useful in the single-server model. There, the parties aiding the server are standard devices and, intuitively, they collectively play the role of the non-colluding server in the two-server model discussed above. Though it may also apply to a server from a second company whose services may be more expensive than the in-house solution.

Figure 2 describes the blueprint of our solution. At a high-level, we reduce summation of length ℓ vectors to summation of length $O(\lambda)$ keys, where λ is the security parameter. The clients encrypt their inputs using a key and message homomorphic symmetric encryption scheme KAHE. Similar to the secure aggregation protocol of Bell et al. [8], our KAHE is based on Ring Learning With Errors (RLWE). In particular, each client i encrypts their input $\mathbf{x_i}$ as $a_i = \mathsf{KAHE.Enc}_{\mathbf{k}_i}(\mathbf{x_i})$ under a fresh key $\mathbf{k}_i$ with *small* Gaussian coefficients[1]. The server homomorphically computes $\sum_i a_i \equiv \mathsf{KAHE.Enc}_{\sum_i \mathbf{k}_i}(\sum_i \mathbf{x_i})$. For the server to obtain $\sum_i \mathbf{k}_i$, we employ a second encryption scheme AHE that, in contrast to KAHE, is additively homomorphic only in the message, and asymmetric. The decryptor outputs an AHE public key pk to let clients encrypt their respective $\mathbf{k}_i$, and the server can homomorphically compute an encryption of $\sum_i \mathbf{k}_i$ for the decryptor to decrypt.

[1] Note that $\mathbf{k}_i$ in [8] are uniformly random over the quotient ring.

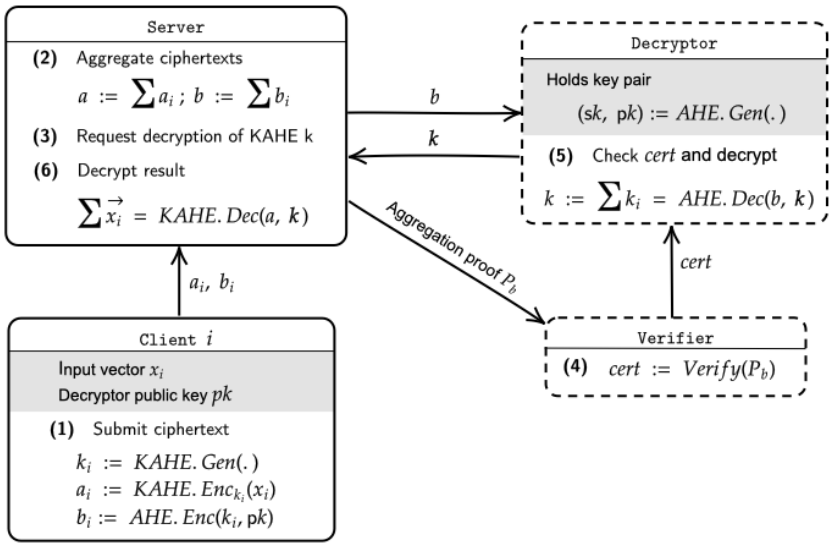

Figure 2: Blueprint of our protocol. Let KAHE be a symmetric encryption scheme with *both keys and messages* additive homomorphism. Let AHE be an asymmetric threshold AHE scheme. The decryptor publishes a public key pk of AHE. Clients send a pair of ciphertexts: (a) an encryption of their input under KAHE and (b) and encryption of the symmetric key used in (a), under AHE. The server adds ciphertexts of each kind as they are received, resulting in ciphertexts a, b encrypting the intended sum, and the symmetric key in the first ciphertext, respectively. The server then proves to the verifier that b encrypts the sum of n *distinct* keys, all coming from different clients. The verifier signs a hash of b, which the decryptor verifies before handing the decryption of b to the server.

While this is our high-level blueprint, significant challenges have to be overcome to make the entire protocol concretely efficient (or even a secure protocol!). In terms of efficiency, we need a threshold AHE scheme to implement the decryptor by an honest majority committee. Regarding security, it is unclear that revealing $\sum_i \mathbf{k}_i$ to the server still hides the keys $\mathbf{k}_i$ from the server. Finally, actively corrupted clients could try to choose their key $\mathbf{k}_i$ in a way that deviates from the prescribed protocol. This is not an issue per se, unless the adversary controlling those clients can also observe the server's transcript. We discuss these challenges and solutions in more detail next.

Threshold Additive Homomorphic Encryption (AHE). As we mentioned above, we require an AHE scheme that can be efficiently distributed within an honest majority committee. While the Paillier scheme [15] has the right homomorphic properties, it is hard to distribute. Conversely, the exponential ElGamal scheme – where inputs are encrypted in the exponent of an appropriately chosen group element – is inconvenient that decryption involves solving a discrete log. While this works for small inputs (see [6, 16] for examples), the exponential ElGamal is expensive for us to homomorphically compute $\sum_{i=1}^{n} \mathbf{k}_i$, which is a polynomial of 2^{10} to 2^{14} coefficients. We instead use an RLWE-based AHE scheme that can be regarded as an instance of the proposal by Bendlin and Damgård [17].

Our AHE instantiation has an efficient distributed key generation where each committee member generates its own pair of private-public key shares $(\mathsf{sk}_j, \mathsf{pk}_j)$. The AHE public key is then $\mathsf{pk} = \sum_j \mathsf{pk}_j$. Similarly, the AHE secret is additively shared among $\mathsf{sk} = \sum_j \mathsf{sk}_j$. While this constitutes a c-out-of-c sharing, we employ Shamir secret-sharing to get robustness to decryptors dropping out. Decryption is efficient in AHE, and it only takes one round: decryptors all receive the same ciphertext ct and compute the (polynomial) product of sk_j and ct. To hide information about sk_j from the server, partial decryptions are in fact of the form $\mathsf{sk}_j \cdot \mathsf{ct} + \mathbf{e}_{\mathtt{flood}}$, where $\mathbf{e}_{\mathtt{flood}}$ is a flooding noise with a variance that is exponentially large in the statistical

security parameter. This is a standard approach in lattice-based threshold encryption schemes [18] and, while solutions with smaller flooding noise have been recently suggested [19], it is not known yet whether they apply to RLWE. Note that the AHE key generation cost can be amortized across distinct aggregations facilitated by the same set of decryptors.

Leakage of $\sum_i \mathbf{k}_i$. In order to hide individual $\mathbf{k}_i$ and input from a corrupted server, the KAHE scheme must be resilient to the leakage of $\mathbf{k} := \sum_i \mathbf{k}_i$, which is given to the server for decrypting the aggregated KAHE ciphertext (Step (4) in Figure 2). Bell et al. [8] achieves such leakage-resilient security by relying on a Hint-RLWE assumption and assuming uniform distribution for $\mathbf{k}_i$ which results in increased parameters for the overall protocol. Instead, in this work we show that the leakage resilience property holds even when keys and errors both come from Gaussian distributions with small variance (only $2\times$ larger than that required for RLWE security in the standard setting, see Lemma 1). This result follows from an improved analysis of Hint-RLWE by Kim et al. [20], and leads to up to 50% less communication compared to uniform KAHE keys.

The "correlated ciphertext" attack. As mentioned above, subtle issues arise as soon as an adversary controlling a few clients can also observe the view of the (passively corrupted) server. Consider an attacker controlling a single client (say client $n-1$), that gets to observe all ciphertexts $b_1, \ldots, b_{n-2}$ sent by honest clients $1, \ldots, n-2$ to the server (recall that $b_i = \mathsf{AHE.Enc}(\mathbf{k}_i, \mathsf{pk})$). Then, assume that the corrupted client sets $b_{n-1} := -\sum_{i=1}^{n-2} b_i$. Then, when client n sends an honestly constructed b_n and the protocol progresses normally, the server reconstructs the KAHE key $\mathbf{k} = \sum_{i \neq n-1} \mathbf{k}_i + \mathbf{k}_{n-1} = \sum_{i \neq n-1} \mathbf{k}_i - \sum_{i=1}^{n-2} \mathbf{k}_i = \mathbf{k}_n$. Therefore $\mathbf{k}$ allows the server, and thus the adversary that observes its view, to recover client n's input. To address this issue clients are required to provide a Zero-Knowledge Proof of Knowledge (ZKPoK) of $\mathbf{k}_i$, along with their AHE encryption b_i. This ensures that each key is sampled independently of other client's keys.

While zero-knowledge can be expensive, three observations make it well suited for our protocol: first, the encryption operation in AHE (i.e., the relation for which clients provide a proof) is a simple linear function of the public key pk. This is because it corresponds to a "knowledge of (R)LWE secret", and the required polynomial multiplication can be written as matrix vector multiplication. Second, the witness $\mathbf{k}_i$ is a secret key of length that depends only on the security parameter, and not ℓ (let us anticipate that pk is a polynomial of at most 2^{12} coefficients modulo $q_2 < 2^{96}$ in all the applications we consider). Finally, the required zero knowledge proof is independent of $\mathbf{x_i}$ and therefore can be computed before the input is available. As in previous works [8, 21] we use Bulletproofs [22] in our evaluation, and rely on the approximate l_∞ proofs by Gentry et al. [21] for efficient proofs of knowledge of (R)LWE secret. For details on this, see Appendix E.

A similar issue happens with key generation when a corrupted decryptor can observe partial keys pk_j sent by honest decryptors as they are received by the server (this is called a *rogue key attack*). For this reasons we require the analogous proofs from decryptors as part of key generation and partial decryption. Also in this case we instantiate the ZKPoKs using the DL-based approaches from [8, 21].

Malicious Server, and the role of the Verifier. As described in Figure 2, the role of the verifier is to prevent a malicious server to "copy/replay" contributions from honest clients. It can also ensure that the number of clients being aggregated is larger than a protocol parameter `min_n`. In a nutshell, the verifier's job is to check the ZKPoK associated with ciphertexts b_i before the server decrypts b. Moreover, the verifier checks that b is indeed the result of aggregating the b_i's. To do this we devise a tree data structure $\mathcal{T}$, akin to a Merkle tree, and similar to the aggregation tree used in the Honeycrisp work [23]. Each leaf of the (binary) tree contains a (constant-size) commitment to b_i and the corresponding ZK proof. Internal nodes contain commitments such that the commitment in a parent node commits to the sum of the committed value of its children. This is easy to achieve with additive commitment schemes like (vector) Pedersen commitment. Therefore, the root of a valid tree commits to b. An important observation is that nodes in $\mathcal{T}$ either constant (256-bit) commitments of proofs of size $O(\log(N_2))$ (less than 2KB). Therefore while the whole tree has size $O(n)$ the constant is very small, and crucially ciphertexts b_i (which are each hundreds of KB) do not need to

be part of $\mathcal{T}$ (this is in contrast with work [23]). Just like Merkle trees, our tree construction $\mathcal{T}$ is amenable to distributed verification: we provide two ways to distribute the verifier's role among many parties. The first one is based on committees and is fully secure (gives a cheating server negligible advantage). The second one is fully distributed (no need to form committes) and catches a cheating server with tunnable constant probability, e.g., 90%. This is appropriate for settings where the server faces some reputation loss risk when caught cheating. Also, let us remark that $\mathcal{T}$ does not contain private information, and can be made public for anyone to verify.

On Random Oracles. The high-level description in Figure 2 leaves out one important detail of our protocol: Instead of encrypting their input x_i directly under KAHE, clients secret-share it first into two parts, where one share is generated by a random oracle from a seed. They then encrypt one share under KAHE as before, and encrypt the seed used to generate the other separately under another public key where the secret key is shared between the decryptors. This serves two purposes. First, we need the random oracle in order to achieve simulation security. This follows from the following impossibility result, discussed below and proven in Appendix F.

Theorem 1 (Informal)**.** *Assume the existence of one way functions. Let λ be a security parameter, and let Π be a protocol with one-shot clients and dynamic participation implementing vector aggregation with $|\mathbb{F}| \geq 2$ and $\sum_{k=\mathtt{min\_n}}^{n} \binom{n}{k} > 2^{\lambda}$ (c.f. Functionality $\mathcal{F}^{\mathsf{Agg}}$, Fig.1). If after receiving the last client message the server engages in $o(l)$ communication, then there is no white-box (and thus no black-box) security proof in the standard model against an adversary actively corrupting the server* S.

Proof Sketch. The basic idea of the proof is to define a class of attacker with a random key to a collision-resistant hash function hardcoded in their definition. Such attacker(s) proceed by hashing their view up to the point where the last one-shot client has sent their input to determine which clients's contributions to ignore (let's call that the "first part of the view"). Then they complete the protocol normally and output their whole view, including the output received the be server. The important observation is that a valid view must satisfy the constraint that the set of ignored clients is uniquely determined by the first part of the view. In a setting where the client inputs have enough entropy, so does the output that the adversary will emit ($\Omega(\ell)$ in out argument). Therefore the second part of the view must have the same amount of entropy and thus size $\Omega(\ell)$ in expectation. We provide the full proof in Appendix F.

□

Second, the separate secret-share allows decryptors to delay revealing the output until all partial decryptions have been sent, which prevents selective aborts by an adversary that actively corrupts a decryptor while corrupting the server passively. We provide more details in Section 5.6 and Appendix H.2.

4 Main Cryptographic Primitives

In this section we introduce and instantiate two non-standard primitives used in our protocols: KAHE and AHE. More precisely, in our constructions, we require (i) a symmetric-key Key- *and* message-Additive Homomorphic Encryption (KAHE) and (ii) threshold asymmetric Additive Homomorphic Encryption (AHE). Moreover, we require (ii) to have verifiable key-generation, encryption, and distributed decryption through the use of Zero-Knowledge Proof of Knowledge for the underlying secrets. Next, we define the functionality that we need from these primitives and provide more comprehensive discussion of their properties and instantiations in Appendices C and D in the appendix. In the sequel, for any distribution D, we denote using $x \leftarrow D$ the process of sampling from D, and we denote using $x \leftarrow D(r)$ the process of sampling from D using randomness r. If X is a finite set, then by $x \leftarrow X$ we mean sampling at uniformly random from X. For any $\sigma > 0$, let D_σ be the discrete Gaussian distribution with parameter σ.

4.1 Key-Additive Homomorphic Encryption

We use a symmetric key encryption scheme $\mathsf{KAHE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ with additive key- and message-homomorphisms: Given any two ciphertexts $\mathbf{c}_1$ and $\mathbf{c}_2$ encrypting x_1 and x_2 under keys $\mathbf{k}_1$ and $\mathbf{k}_2$ respectively, $\mathbf{c}_1 + \mathbf{c}_2$ is a valid encryption of $x_1 + x_2$ under the key $\mathbf{k}_1 + \mathbf{k}_2$. We further need a leakage-resilient property presented in Definition 5 (see Appendix C), which guarantees that, given a number of ciphertexts encrypted under different KAHE keys, revealing the aggregate key only reveals the sum of the encrypted messages.

RLWE-based KAHE scheme. We instantiate KAHE based on the RLWE assumption. Let $N_1 > 0$ be a power of two, and let $R_{q_1} = \mathbb{Z}[X]/(q_1, X^{N_1} + 1)$ for integer $q_1 > 0$. Let $t_1 > 0$ be an integer coprime to q_1; the plaintext space in our KAHE scheme is $R_{t_1} = \mathbb{Z}[X]/(t_1, X^{N_1} + 1) \equiv \mathbb{Z}_{t_1}^{N_1}$. Let $\sigma_s, \sigma_e > 0$ be Gaussian parameters for the secret and error distributions. Then

- $\mathsf{KAHE.Setup}()$: Samples and returns $\mathsf{a} \leftarrow R_{q_1}$ as the public parameter which is implicit to the following algorithms.
- $\mathsf{KAHE.KeyGen}()$: Samples and returns $\mathbf{k} \leftarrow D_{\sigma_s}$.
- $\mathsf{KAHE.Enc}(\mathbf{x}, \mathbf{k})$: Samples $e \leftarrow D_{\sigma_e}$, and returns $\mathbf{c} = \mathsf{a} \cdot \mathbf{k} + t \cdot e + x \in R_{q_1}$.
- $\mathsf{KAHE.Dec}(\mathbf{c}, \mathbf{k})$: Returns $(\mathbf{c} - \mathsf{a} \cdot \mathbf{k}) \bmod t_1$.

We prove in Lemma 1 (see Appendix C) that this construction satisfies the desired leakage-resilience property.

4.2 Verifiable Threshold Additive Homomorphic Encryption

We use a public key threshold additive homomorphic encryption scheme with additive distributed key generation and decryption procedures. Such a scheme $\mathsf{AHE} = (\mathsf{Setup}, \mathsf{VerifiableKeyGen}, \mathsf{KeyAgg}, \mathsf{VerifiableEnc}, \mathsf{VerifiablePartialDec}, \mathsf{Rec}$ allows each party to generate independently a private key share and the corresponding public key share $(\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow \mathsf{KeyGen}(r_j)$. The final public key is obtained by aggregating the public key shares $\mathsf{pk} \leftarrow \mathsf{KeyAgg}(\{\mathsf{pk}_j\}_j)$. Each secret key share holder can partially decrypt a ciphertext and obtain $\mathsf{pd}_j \leftarrow \mathsf{PartialDec}(\mathsf{ct}, \mathsf{sk}_j)$. The underlying plaintext can be reconstructed from all partial decryptions $\mathsf{Recover}(\mathsf{ct}, \{\mathsf{pd}_j\}_j)$.

Additive Homomorphism. We require AHE to be additive homomorphic over plaintext: Given any two ciphertexts ct_1 and ct_2 encrypting m_1 and m_2 under pk, the sum $\mathsf{ct} = \mathsf{ct}_1 + \mathsf{ct}_2$ is a valid ciphertext of $m = m_1 + m_2$ under pk.

Verifiability. Additionally, we require this AHE scheme to be *verifiable*, meaning that it has publicly verifiable public key shares, fresh ciphertexts, and partial decryptions, through zero-knowledge proof of knowledge (ZKPoK). More specifically, we assume there exists a ZKPoK system $\Pi_R = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ for a relation R, where Gen generates a set of public parameters, and $\mathsf{Prove}(x, w)$ and $\mathsf{Verify}(x)$ are interactive PPT algorithms on statement x and witness w. We require the usual properties from this ZK system, in particular Π_R should be simulation extractable. We give detailed definitions in Appendix D.

RLWE-based verifiable AHE scheme.. We instantiate an AHE scheme using the standard RLWE assumption, and we augment it with ZKPoK systems Π_{KeyGen}, Π_{Enc}, and $\Pi_{\mathsf{PartialDec}}$. Let $N_2 > 0$ be a power of two, and let $R_{q_2} = \mathbb{Z}[X]/(q_2, X^{N_2} + 1)$ for an integer modulus $q_2 > 0$. Let $t_2 > 0$ be an integer such that the plaintext space is $\mathbb{Z}[X]/(t_2, X^{N_2} + 1) \equiv \mathbb{Z}_{t_2}^{N_2}$, and let $\Delta = \lfloor q_2/t_2 \rceil$ be a scaling factor. Let $\chi_s, \chi_e, \chi_{\mathsf{flood}}$ be distributions over R_{q_2}. Our AHE consists of the following algorithms:

- $\mathsf{AHE.Setup}() = (\mathsf{u}, \mathsf{zk}_{\mathsf{params}})$: Samples $\mathsf{u} \leftarrow R_{q_2}$, and generate the public parameters $\mathsf{zk}_{\mathsf{params}}$ of the ZK proof systems; they are implicit in the following algorithms.

- $\mathsf{AHE.VerifiableKeyGen}((r_1, r_2)) = (\mathsf{sk}_j, \mathsf{pk}_j, \pi_{\mathsf{KeyGen}})$: Samples $\mathsf{sk}_j \leftarrow \chi_s(r_1)$ and $e_j \leftarrow \chi_e(r_2)$, sets $\mathsf{pk}_j = -\mathsf{u} \cdot \mathsf{sk}_j + e_j$, and generates a ZK proof $\pi_{\mathsf{KeyGen}} \leftarrow \mathsf{Prove}(\mathsf{pk}_j, (\mathsf{sk}_j, e_j))$ for the following relation

$$R_{\mathsf{KeyGen}} = \{(\mathsf{pk}_j, \mathsf{sk}_j, e_j) \mid \mathsf{pk}_j = -\mathsf{u} \cdot \mathsf{sk}_j + e_j \bmod q_2, \|\mathsf{sk}_j\| \le B_s, \|e_j\| \le B_e\}, \tag{1}$$

  where B_s and B_e are parameters to be determined based on χ_s, χ_e and on the proof system.
- $\mathsf{AHE.KeyGenVerify}(\pi_{\mathsf{KeyGen}}, \mathsf{pk}_j)$: Returns true if π_{KeyGen} verifies relation R_{KeyGen} over pk_j.
- $\mathsf{AHE.KeyAgg}(\{\mathsf{pk}_j\}_{j=1}^m)$: Returns the public key $\mathsf{pk} = \sum_{j=1}^m \mathsf{pk}_j \in R_{q_2}$.
- $\mathsf{AHE.VerifiableEnc}(x, \mathsf{pk}; r_1, r_2, \tau) = (\mathsf{ct}^0, \mathsf{ct}^1, \pi_{\mathsf{Enc}})$: Samples $v \leftarrow \chi_s(r_1)$ and $e^0, e^1 \leftarrow \chi_e(r_2)$, and computes $\mathsf{ct}^0 = \mathsf{pk} \cdot v + e^0 + \Delta \cdot x \in R_{q_2}$ and $\mathsf{ct}^1 = \mathsf{u} \cdot v + e^1 \in R_{q_2}$. Also generates a ZK proof π_{Enc} for the following relation

$$R_{\mathsf{Enc},\tau} = \{(\mathsf{ct}^1, v, e^1, \tau) \mid \mathsf{ct}^1 = \mathsf{u} \cdot v + e^1 \bmod q_2, \|v\| \le B_s, \|e^1\| \le B_e\}, \tag{2}$$

  where B_s and B_e are parameters as in $\mathsf{VerifiableKeyGen}$, and τ is a random string used as a nonce. Note that τ is not to be hidden in π_{Enc}, and in fact the proof will include τ verbatimly such that the verifier can access it via $\pi_{\mathsf{Enc}}.\mathsf{nonce}$.
- $\mathsf{AHE.EncVerify}(\pi_{\mathsf{Enc}}, \mathsf{ct})$: Returns true if π_{Enc} verifies relation R_{Enc} over ct^1.
- $\mathsf{AHE.PartialDec}(\mathsf{ct}^1, \mathsf{sk}_j) = \mathsf{ct}^1 \cdot \mathsf{sk}_j + e_{\mathsf{flood}}$: To partially decrypt a ciphertext using its component ct^1, this algorithm samples $e_{\mathsf{flood}} \leftarrow \chi_{\mathsf{flood}}$, and returns $\mathsf{pd}_j = \mathsf{ct}^1 \cdot \mathsf{sk}_j + e_{\mathsf{flood}} \in R_{q_2}$.
- $\mathsf{AHE.VerifiablePartialDec}(\mathsf{ct}^1, \mathsf{sk}_j) = (\mathsf{pd}_j, \pi_{\mathsf{PartialDec}})$: Runs $\mathsf{PartialDec}(\mathsf{ct}^1, \mathsf{sk}_j)$ to obtain a partial decryption pd_j, and generates a ZK proof $\pi_{\mathsf{PartialDec}}$ for the following relation

$$R_{\mathsf{PartialDec}} = \{(\mathsf{pd}, \mathsf{ct}^1, \mathsf{sk}, e_{\mathsf{flood}} \mid \mathsf{pd} = \mathsf{ct}^1 \cdot \mathsf{sk} + e_{\mathsf{flood}} \bmod q_2, \|e_{\mathsf{flood}}\| \le B_{\mathsf{flood}}\}, \tag{3}$$

  where B_{flood} is a parameter to be determined based on χ_{flood} and the proof system.
- $\mathsf{AHE.PartialDecVerify}(\mathsf{pd}_j, \pi_{\mathsf{PartialDec}})$: Returns true if $\pi_{\mathsf{PartialDec}}$ verifies relation $R_{\mathsf{PartialDec}}$ over pd_j.
- $\mathsf{AHE.Recover}((\mathsf{ct}^0, \mathsf{ct}^1), \{\mathsf{pd}_j\}_{j=1}^m) = x$: Returns $\left\lfloor (\mathsf{ct}^0 + \sum_{j=1}^m \mathsf{pd}_j)/\Delta \right\rceil$.

Note that $\mathsf{AHE.PartialDec}$ is sufficient in certain variants of our protocols where ZKPoK of partial decryption relation is not needed for security. We include $\mathsf{AHE.PartialDec}$ and $\mathsf{AHE.VerifiablePartialDec}$ for completeness.

In Appendix D we prove the security properties described above, namely (i) that individual public key shares are independent and pseudorandom, (ii) that the scheme is IND-CPA secure, and that (iii) when the flooding noise parameters are chosen properly, given valid ciphertexts $\{\mathsf{ct}_i\}_{i=1}^k$, the partial decryptions on the sum of ct_i's do not leak secret information of honest decryptors against an active adversary. We also discuss some practical considerations.

5 Secure Aggregation with Dynamic Client Participation

We now present a protocol for the functionality $\mathcal{F}^{\mathsf{Agg}}$. Our protocol satisfies simulation-based security by operating in the RO model.

5.1 Cryptographic assumptions.

Our protocol uses the following cryptographic schemes: a verifiable m-out-of-m threshold AHE scheme AHE, an m-out-of-m threshold PKE scheme E with verifiable key generation, a verifiable secret-sharing scheme VSS, and a digital signature scheme Sign. Furthermore, we assume there is a PKI for decryptors and authenticated communication channels (relayed by the server) among decryptors. Note that a malicious server may not deliver messages among decryptors, but it cannot inspect them.

5.2 Decryptor Role: Robust DKG with dishonest relay.

Our protocol for the (distributed) decryptor role is given in Fig. 3. We use the roles defined in Section 2.1. We additionally introduce a coordinator role, denoted Coord, to represent the tasks performed by the server as part of key generation and decryption. For simplicity we describe the protocol assuming that decryptors have secure channels between them, and Coord relays encrypted messages among decryptors. This is the same setup as in previous works, e.g. [3, 7, 8], and its security follows from the assumption that there is a PKI for decryptors (see parties in Fig.3). Our summation protocol is secure as long as the number of corrupted decryptors is at most $2t - m - 1$, where m is the total number of decryptors and t is a public secret-sharing threshold (see "public parameters" in Fig. 3). Moreover, the protocol can withstand up to $t-1$ decryptors dropping out during the protocol execution, e.g. in between key generation and decryption. For example, when $m = 100$, assuming the number of corrupted decryptors is $32 < \lfloor m/3 \rfloor$, the protocol can withstand $m - t = m - 132/2 = 34$ dropouts.

Key Generation. In the key generation phase the decryptors generate parameters for an AHE scheme with distributed (with an additive m-out-of-m sharing) private key among all decryptors, as well as a public key scheme E. The underlying encryption schemes enable efficient key generation where each decryptor generates its own set of public and private key shares and the common encryption key is obtained by combining all individual public key shares, i.e. $\mathsf{pk} = \sum_j \mathsf{pk}_j$ and $\mathsf{pk}^{\mathsf{aux}} = \sum_j \mathsf{pk}_j^{\mathsf{aux}}$. Note that it is sufficient for E to be a secure threshold PKE scheme (and not necessarily additive homomorphic) with verifiable key shares, and therefore ElGamal is sufficient for this purpose, but of course another instance of AHE would also work.

Moreover, as discussed in Section 4, we need key generation to be verifiable, in the sense that each decryptor must provide a proof of knowledge of their secret keys $\mathsf{sk}_j, \mathsf{sk}_j^{\mathsf{aux}}$. This is a common approach to prevent *rogue key attacks*, where malicious decryptors collude with the server and generate public keys $\mathsf{pk}_j, \mathsf{pk}_j^{\mathsf{aux}}$ correlated with those provided by honest decryptors. More precisely, consider an adversary $\mathcal{A}$ passively corrupts the server, i.e., only observing its internal states, and also actively corrupts a decryptor d. $\mathcal{A}$ instructs d to delay its message until all other decryptors have sent their message m_j in Step 1. As $\mathcal{A}$ can observe such message, it then instructs d to submit $\mathsf{pk}_{\mathcal{A}} - \sum_{i \neq d} \mathsf{pk}_i$ as its public key pk_d, where $\mathsf{pk}_{\mathcal{A}}$ is a key for which the adversary know the private key. Note that the protocol will compute $\mathsf{pk} = \mathsf{pk}_{\mathcal{A}}$, and therefore the adversary can decrypt any message. This is exactly what verifiable key generation addresses: even if $\mathcal{A}$ can see public keys of honest decryptors, it cannot cancel the underlying sk share.

During key generation each decryptor threshold-secret shares its AHE and E key shares with the others to support dropouts among the decryptors during the decryption phase. This is done in Steps 1b and 1d. To be precise, each decryptor splits the corresponding secret keys $\mathsf{sk}_j, sk_j^{\mathsf{aux}}$ into $m-1$ shares using t-out-of-$(m-1)$ Shamir secret sharing, and sends one of the shares to each other decryptor *encrypted*, via the server. For this we rely on a PKI holding public keys of authenticated encryption scheme AuthE for members of $\mathcal{C}$. This means that the server/coordinator might not deliver some of these encrypted shares, but can't inspect them. In Figure 3 we simply say that the decryptor "secret-shares within $\mathcal{C}$" for simplicity.

Another important aspect to note is that, for efficiency reasons, in Steps 1b and 1d decryptors secret-share *small* pseudorandom seeds r_j, r_j^{aux} from which much larger keys $\mathsf{sk}_j, \mathsf{sk}_j^{\mathsf{aux}}$ are obtained. In practice, the former are 256 bits long, while the latter are > 1000 times larger (at least 40KB for our parameters of interest). This is because keys of the RLWE-based instantiation of AHE required in our protocol are polynomials of degree $N \sim 2^{12}$ with coefficients in $q \sim 2^{80}$. Therefore, our protocol does not require $O(Nq)$ communication, or secret sharing over such large domain.

Finally, in the second round of key generation decryptors verify each other's key generation and sign a set of partial public keys if the set is large enough. The signing is done using keys held by a PKI (see "parties" in Fig. 3). Note that the work of this second round, which essentially checks the work of the untrusted server in the end of the previous round, could be entirely performed by clients without incurring an extra round, resulting in a one-round key generation protocol. However, this would require *all* n clients to verify $O(m)$ proofs. Since the total number of clients is much larger than the number of decryptors, the total amount of work would be larger in that case. Nevertheless, that protocol variant might be appropriate in certain scenarios.

Protocol Π^M (Decryptor)

Parties:

- A committee $\mathcal{C}$ with members $1, \ldots, m$.
- Coordinator Coord forwarding messages.
- A verifier V that signs ciphertexts intended for decryption.
- PKI holding public signing keys of signature scheme Sig for V and members of $\mathcal{C}$.

Public Parameters: Timeout T, threshold t, parameters for schemes AHE, E and Sig.

Key Generation

Output: Public keys $\mathsf{pk}, \mathsf{pk}^{\mathsf{aux}}$ signed by a set $\mathsf{D} \subseteq \mathcal{C}$ of committee members such that $|\mathsf{D}| \geq t$.

Round 1: Share partial keys

1. Every committee member $j \in [m]$:
   (a) Computes $(\mathsf{sk}_j, \mathsf{pk}_j, \pi_j) := \mathsf{AHE.VerifiableKeyGen}(r_j)$ from randomness r_j.
   (b) Secret-shares r_j within $\mathcal{C}$ with threshold t.
   (c) Computes $(\mathsf{sk}_j^{\mathsf{aux}}, \mathsf{pk}_j^{\mathsf{aux}}, \pi_j^{\mathsf{aux}}) := \mathsf{E.VerifiableKeyGen}(r_j)$ from randomness r_j^{aux}.
   (d) Secret-shares r_j^{aux} within $\mathcal{C}$ with threshold t.
   (e) Sends $m_j := (\mathsf{pk}_j, \pi_j, \mathsf{pk}_j^{\mathsf{aux}}, \pi_j^{\mathsf{aux}})$ and $s_j = \mathsf{Sig.Sign}(m_j)$ to Coord.
2. Coord collects messages up to a timeout T. Let $\mathcal{C}_s \subseteq \mathcal{C}$ be the committee members that provide correct proofs and signatures. If $|\mathcal{C}_s| < t$, Coord aborts, otherwise Coord sets $\mathsf{D} := \mathcal{C}_s$, $\mathsf{pk} := \sum_{j \in \mathcal{C}_s} \mathsf{pk}_j$, and $\mathsf{pk}^{\mathsf{aux}} := \sum_{j \in \mathcal{C}_s} \mathsf{pk}_j^{\mathsf{aux}}$.

Round 2: Verify global keys $\mathsf{pk}, \mathsf{pk}^{\mathsf{aux}}$

3. Coord broadcasts $S = \{(m_j, s_j) | j \in \mathsf{D}\}$ within D.

// Decryptors independently check the server's work

4. Every decryptor $j \in \mathsf{D}$:
   (a) Checks $|S| \geq t$ and that all proofs and signatures in S are correct, and aborts otherwise.
   (b) Sends $\mathrm{sig}(ms_j) := \mathsf{Sig.Sign}(\sum_{x \in S} x_{1,1}, \sum_{x \in S} x_{1,2})$ to Coord.

// The server collects signatures from decryptors

5. Coord collects messages up to a timeout T, and sets S_{D} to be the resulting set of signatures. If all signatures in S_{D} are valid signatures of $(\mathsf{pk}, \mathsf{pk}^{\mathsf{aux}})$ and $|S_{\mathsf{D}}| \geq t$ then Coord sends $(\mathsf{D}, S_{\mathsf{D}}, \mathsf{pk}, \mathsf{pk}^{\mathsf{aux}})$ to D, and aborts otherwise.

Decryption

Input: Aggregated ciphertext component ct^1 and signature(s) s_{ct^1} (from verifier).

Output: Aggregated partial decryption of ciphertext $(\mathsf{ct}^0, \mathsf{ct}^1)$, and key $\mathsf{sk}_{\mathsf{S}}^{\mathsf{aux}}$.

Round 1: Collect partial decryptions

6. Coord receives $(\mathsf{ct}^1, s_{\mathsf{ct}^1})$ and broadcasts it within D.

// Decryptors provide a partial decryption, only if the ciphertext has been verified by V

7. Every $j \in \mathsf{D}$:
   (a) Checks that s_{ct^1} contains appropriate signature(s) from V, otherwise aborts.
   (b) Sends $\mathsf{pd}_j := \mathsf{AHE.PartialDec}(c^1, \mathsf{sk}_j)$ to Coord.
   (c) Sends shares received in Step 1d (for Coord to reconstruct $(r_k)_{k \in \mathsf{D}}$, and thus $\mathsf{sk}^{\mathsf{aux}}$).
8. Coord collects messages up to a timeout T. Let P be the set of decryptors that reply. If $|P| < t$, Coord aborts.
9. Otherwise Coord reconstructs $\mathsf{sk}_{\mathsf{S}}^{\mathsf{aux}}$ using the shares received in Step 7c. Coord checks that the reconstructed $\mathsf{sk}_{\mathsf{S}}^{\mathsf{aux}}$ is the secret key for $\mathsf{pk}^{\mathsf{aux}}$ received in Step 2, and aborts if not.

Round 2: Drop-out recovery

10. Coord sends P to every decryptor in P.
11. Every $j \in P$:
    (a) Aborts if $|P| < t$.
    (b) Sends shares received in Step 1b from each decryptor $k \notin P$, i.e. dropouts, to Coord.
12. Coord reconstructs $(r_k, \mathsf{pk}_k, \mathsf{sk}_k, \mathsf{pd}_k)$ for every dropout $k \notin P$. If pk_k doesn't match the one received in Step 1e Coord aborts.
13. Coord sends $\mathsf{pd} := \sum_{j \in \mathsf{D}} \mathsf{pd}_j$ and $\mathsf{sk}_{\mathsf{S}}^{\mathsf{aux}}$ to S.

Figure 3: Decryptor D by committee, with unreliable proxy Coord.

Decryption. Decryption proceeds in two rounds. In the first round the server provides a ciphertext component ct^1 for decryption, along with a signature s_{ct^1} from the verifier V. This assures the decryptors that ct^1 is well-formed, i.e., that it is the sum of a set of valid ciphertexts. We discuss the verifier role in detail in a subsequent section. In turn, the server receives from each decryptor j both (a) a partial decryption pd_j and (b) shares of secret keys $(r_k)_{k\neq j}$. Note that receiving (b) from t decryptors is enough to recover $\mathsf{sk}^{\mathsf{aux}}$. Also note that there is no guarantee that decryptors will send correct shares. This is not an issue against a malicious server, where we don't care about correctness. See Appendix H.2 for how to modify this part in case we do care.

The second round is only needed for dropout recovery. Therefore, in the event of no dropouts in Step 7 this round is not needed and thus decryption in that case runs in a single round. For a reader familiar with the previous single-server malicious aggregation protocols the fact that dropout recovery takes only one round might come as a surprise. Previous protocols [1, 3, 7, 8] base on the DC-network idea (see Appendix B) also have each client secret-share private data to enable recovery in case they drop out before the end of the protocol. Just like in the dropout recovery round in Fig. 3, in those protocols the server reaches out to surviving clients to retrieve shares that allow to essentially rerun dropped out clients. Therefore, the server is supposed to broadcast the set $\bar{P}$ of dropped out clients/decryptors to surviving clients in P. Moreover, the protocol must ensure that $\bar{P}$ is not too large (in our case the protocol ensures that $|P| > t$). A challenge when handling a malicious server is that the server might give inconsistent views to clients, i.e., sends different sets of dropped out clients to different clients as opposed to broadcasting $\bar{P}$. The way this is addressed in all previous works discussed above is by having an additional "consistency check" round where decryptors share the set $\bar{P}$ they received from the server to make sure they all got the same set. We deviate from that approach by exploiting the way in that our protocol nests m-out-of-m and t-out-of-$(m-1)$ secret sharing: to recover the global key sk the adversary needs to recover *all* honest local keys sk_j, and so the total number of shares of honest keys needed by an adversary to learn sk is $s_{\mathsf{need}} := h(t-c)$, where h is the number of honest decryptors and c is the number of corrupted decryptors. Since honest decryptors refuse to send shares if they receive a set of dropped out clients larger than $m-t$ (as per Step 11a in Fig. 3), the total number of shares from honest decryptors is no more than $s_{\mathsf{get}} := h(m-t)$. The constraint that $c \leq 2t-m-1$ (which corresponds to our security parameters discussed above) implies that $s_{\mathsf{need}} = h(t-c) \geq h(-t+m+1) = h(m-t)+h > s_{\mathsf{get}}$. This ensures that a malicious server cannot recover sk by sending inconsistent sets of dropped out decryptors, and thus avoids the need for a consistency check.

Communication. Finally, let us discuss decryptors' communication cost: note that when our protocol is instantiated using RLWE-based AHE, decryptors receive just the second part ct^1 of an AHE ciphertext, i.e., one polynomial approximately resulting in $40KB$ of communication. An important aspect of our approach is that decryptor communication is essentially independent of input vectors length ℓ and number of clients. This is because decryptors only have to handle encryptions of KAHE keys.

5.3 Client & Server Roles: One-shot Publicly Verifiable Contributions.

The client and server roles of our protocol are described in Figures 5 and 4, respectively. We discuss the main aspects in the following.

Verifiable AHE encryption. In our protocol, each client samples a fresh KAHE key $\mathbf{k}$ and encrypts $\mathbf{k}$ itself using the AHE public key pk generated by the decryptors (as long as enough decryptors vouch for that key, see Step 2). Note that this is done using $\mathsf{AHE.VerifiableEnc}$, which generates a ZK proof π_{Enc} of knowledge of the randomness and error used in the encryption. The latter is necessary to prevent the server from deriving correlated AHE ciphertexts and using those to shift the aggregated KAHE key that will be decrypted by the decryptors (this is very similar to the rogue key attack attack described in the decryptor section).

PRG-based secret sharing. An important technical point is that the client does not simply KAHE-encrypt it input $\mathbf{x_i}$ as $\mathsf{KAHE.Enc}(\mathbf{k}, \mathbf{x_i})$. Instead, it first splits $\mathbf{x_i} \in \mathbb{F}^\ell$ into two shares $(\mathbf{y} \in \mathbb{F}^\ell, \mathsf{seed} \in \{0,1\}^\lambda)$ by means of a PRG modeled as a programmable random oracle (see Step 4 in Figure 5). To communicate seed to the server, the client encrypts it using $\mathsf{pk}^{\mathsf{aux}}$ and sends the resulting ciphertext c to the server. This approach is central to how our protocol overcomes the lower bound argument from the previous section to get simulation security in the random oracle model. We will now provide the underlying intuition (for details check the proof of Theorem 2). Assume the server is controlled by a real-world adversary $\mathcal{A}_R$. In the simulation proof, this interaction with the verifier and the server controlled by $\mathcal{A}_R$ can be observed by the simulator. This allows the simulator to extract from $\mathcal{A}_R$ which honest clients are desired to include in the sum, and learn their sum $\mathbf{s_h}$ by submitting the appropriate set S to the trusted party in the ideal world. However, the simulator now has to make sure that the execution involving $\mathcal{A}_R$ includes the right sum $\mathbf{s_h}$. Intuitively, this final step is what our lower bound (Theorem 1) shows to be impossible in the standard model and with small decryptor and verifier communication. In contrast, the simulator in the current setting can reprogram the PRG at seed for one of the honest clients included in the sum so that the sum reconstructed by the server (see Step 10 in Fig. 4) in the simulation has the right value $\mathbf{s_h}$. Let us remark that the reprogramming happens *before* the server (and therefore $\mathcal{A}_R$) get $\mathsf{sk}^{\mathsf{aux}}$ and thus can decrypt c to obtain seed. Therefore the probability of the adversary having queried the oracle at seed before reprogramming is negligible.

Indexed Zero-knowledge proofs. A crucial aspect of the interaction between client, server, and verifier in our protocol is the nonce τ sampled by clients in Step 7, and how it is used as part of AHE.VerifiableEnc. As discussed above, ZK proofs emitted by AHE.VerifiableEnc are parametrized by a nonce. This is achieved by including the nonce is the description of the relation being proven. When using Fiat-Shamir this means that the nonce value is part of the protocol transcript and therefore contributed to determining the challenges in the non-interactive proof. This ensures that the server cannot "replay" a given user's contribution. We use the notation $\pi.\mathtt{nonce}$ to denote the nonce associated with a ZK proof π. As shown in Figure 4, the server collects proofs and encryptions of seeds while aggregating AHE and KAHE ciphertexts (see Step 4c).

5.4 Verifier Role: Distributed Aggregation Verification.

The goal of the verifier is to make sure that the ciphertext that the server tried to decrypt is well formed. By that we informally mean that it includes "genuine" client contributions, and that each is included no more than once. Figure 6 shows a centralized version of the verifier that explicitly checks the above properties given a set of proofs. Note that we assume that ciphertexts and nonces associated with proofs are included in them (in fact this strictly the case as they're part of the relation being proven). The main properties of the verification stage are:

1. **Public verifiability.** The verification does not involve any private data, and can be executed publicly. If the Server is caught misbehaving, the verification process outputs a public verifiable proof incriminating the Server. This proof can't be forged to falsely accuse the Server.
2. **Efficiency.** The Verifier requires no interaction, and costs are independent of input length ℓ.
3. **Distributed verification.** The verifier's work can be distributed among several parties, to amortize costs.

In the rest of this section we show the last point above, namely how to distribute among several parties the verifier's task described in Figure 6. We start by defining a data structure, which we call aggregation tree, that allows to distribute and/or randomize the verifier's check.

Aggregation Tree. Our first observation to reduce concrete communication for the verifier is that the input it receives $(\mathtt{ct\_sum}, \mathtt{proofs} := (\pi_i)_i)$ can be reduced to a single AHE ciphertext ct_sum and each of the proofs $\pi \in \mathtt{proofs}$ can be of size size logarithmic over the size of an AHE ciphertext. This can be done by (i) using an appropriate proof system with logarithmic proof size and no trusted setup, and (ii) having

Protocol Π^M/Π^{Agg} (Server)

Parties:

- Clients $\mathsf{C}_1, \ldots, \mathsf{C}_n$.
- Decryptor(s) D.
- verifier(s) V.
- PKI holding public signing keys of signature scheme Sig for D and V.

Public Parameters:

- Parameters for schemes KAHE, AHE, E, PRG and Sig.
- Target number of clients of clients min_n.
- Input domain $\mathbb{F}^\ell$.

// Setup phase: Get key from D

1. Run key generation with D to receive $\mathrm{K} := (\mathsf{D}, S_{\mathsf{D}}, \mathsf{pk}, \mathsf{pk}^{\mathsf{aux}})$ (Figure 3).
2. Initialize encSum and encKey to zero.
3. Initialize encSeeds and proofs to emptyList.

// Aggregation phase: The server processes client connections until it receives min_n *valid* contributions.

4. **while** |proofs| $<$ min_n **do in paralell {**
   // Process request from client C_i
   (a) Send K to C_i
   (b) Receive (m, ct, π, c) from C_i (see Fig. 5)
   (c) If AHE.VerifyEnc(τ):
      i. encKey $+=$ ct
      ii. encSum $+= m$
      iii. encSeeds.$append(c)$
      iv. proofs.$append(\pi)$

   }

// Verification phase: generate proof for V to sign ciphertext

5. Parse encKey as $(\mathsf{ct}^0, \mathsf{ct}^1)$.
6. Send $(\mathsf{ct}^1, \mathsf{proofs})$ to V.
7. Receive signature s_{ct^1} from V (or set of signatures if V is distributed).
8. Verify that s_{ct^1} has valid signature(s) of ct^1 from V.

// Decryption phase

9. Run decryption with D (Figure 3).
   (a) Send ciphertext ct^1 and signature s_{ct^1} to D.
   (b) Receive $\mathbf{k}$ and $\mathsf{sk}^{\mathsf{aux}}$ from D.
10. Recover maskSum $= \sum_{c \in \mathsf{encSeeds}} \mathsf{PRG.Expand}(\mathsf{E.Dec}(\mathsf{sk}^{\mathsf{aux}}, c), \ell)$.
11. Output $\mathsf{KAHE.Dec}(m, \mathbf{k}) -$ maskSum

Figure 4: Server S. The server processes one-shot client contributions *in parallel* without need of synchronization.

the proof π be done with respect to a binding commitment to the underlying AHE ciphertext. Note that (ii) alone results in significant savings, as the size of ct^1 for a given client is a few KBs in practice while a Pedersen vector commitment is 16B, and the verifier receives n such ciphertexts. Therefore, below we assume that optimization and refer as π_i.ciphertext to the vector (Pedersen) *commitment* to the AHE ciphertext associated with proof π_i. We refer to Pedersen commitments to ease the presentation, but any additive homomorphic binding (not necessarily hiding) commitment scheme will do.

Given $\mathcal{L} := (\mathsf{ct\_sum}, (\pi_i)_i)$ We define $T(\mathcal{L})$ to be the binary tree with the proofs $(\pi_i)_i$ as leaves, *sorted* by nonce π_i.nonce, and internal nodes containing a single commitment corresponding to the (homomorphic) sum of the commitments in the two children, where for leaves such commitments are π_i.ciphertext Therefore,

Protocol Π^M/Π^{Agg} (Client)

Parties:

- Server S.
- PKI holding public signing keys of signature scheme Sig for decryptor(s) D.

Public Parameters: Parameters for schemes KAHE, AHE, E, PRG and Sig.

Input: $\mathbf{x} \in \mathbb{F}^{\ell}$.

1. Receive $(\mathsf{D}, S_{\mathsf{D}}, \mathsf{pk}, \mathsf{pk}^{\mathsf{aux}})$ from S (See Step 5 in Fig. 3).
2. Check that (i) $|S_{\mathsf{D}}| \geq t$ and (ii) all signatures in S_{D} are valid signatures of $(\mathsf{pk}, \mathsf{pk}^{\mathsf{aux}})$. Abort if (i, ii) do not hold.
3. Set $\mathbf{k} := \mathsf{KAHE.KeyGen}()$ and $\mathsf{seed} := \mathsf{PRG.KeyGen}()$.

// Compute masked input $\mathbf{y}$

4. $\mathbf{y} := \mathsf{PRG.Expand}(\mathsf{seed}, \ell) + \mathbf{x}$

// m is an symmetric key encryption of the masked input.

5. Set $m := \mathsf{KAHE.Enc}(\mathbf{y}, \mathbf{k})$
6. Sample $r \leftarrow \{0,1\}^{\lambda}$

// $(\mathsf{ct}^0, \mathsf{ct}^1)$ is an encryption of $\mathbf{k}_i$ under pk with randomness r

7. Sample nonce $\tau \leftarrow \{0,1\}^{128}$

// Client creates a proof of encryption parameterized by a random nonce τ of their choice

8. Set $(\mathsf{ct}^0, \mathsf{ct}^1, \pi) := \mathsf{AHE.VerifiableEnc}(\mathbf{k}, \mathsf{pk}, r, \tau)$
9. Set $c = \mathsf{E.Enc}(\mathsf{seed}, \mathsf{pk}^{\mathsf{aux}})$
10. Send $(m, (\mathsf{ct}^0, \mathsf{ct}^1), \pi, c)$ to S.

Figure 5: Client C. Note that computation required to be done online is independent of ℓ because m depends only on a locally sampled key.

Protocol Π^M/Π^{Agg} (Verifier)

Parties:

- Server S.
- PKI holding public signing keys of signature scheme Sig for verifier(s) V.

Public Parameters:

- Parameters for schemes AHE and Sig.
- Min. number of contributions min_n.

Input: Signing key sk_{V}.

1. Receive $(\mathsf{ct\_sum}, \mathsf{proofs} := (\pi_i)_i)$ from S (see Step 6 in Fig. 4).

// V checks that (a) there are enough proofs, (b) all proofs are valid, (c) their associated nonces are unique, and (d) their associated ciphertexts add to ct_sum.

2. Check that
   (a) $|\mathsf{proofs}| \geq \mathsf{min\_n}$
   (b) $\forall \pi \in \mathsf{proofs} : \mathsf{AHE.VerifyEnc}(\pi) = 1$ // Batched verification can be employed here
   (c) $(\pi.\mathsf{nonce})_{\pi \in \mathsf{proofs}}$ does not contain any duplicates.
   (d) $\sum_{\pi \in \mathsf{proofs}} \pi.\mathsf{ciphertext} = \mathsf{ct\_sum}$
3. If any of the checks fails, abort.
4. Otherwise set $s_{\mathsf{ct\_sum}} := \mathsf{Sig.Sign}(\mathsf{sk}_{\mathsf{V}}, \mathsf{ct\_sum})$, and send $s_{\mathsf{ct\_sum}}$ to S.

Figure 6: Verifier V.

$T(\mathcal{L})$ contains n proofs, and roughly $2n$ commitments. As in our implementation we use constant-size Pedersen commitments and logarithmic size proofs, the tree size is $O(n \log n)$ and thus independent of input length ℓ.

Implementing the Verifier. The verifier can be implemented in many ways: it could be a designated party, the set of clients themselves, a committee of clients, or a Trusted Execution Environment (via remote attestation, as confidentiality is not a concern here). The only assumption for this role is non-collusion with the Server, and therefore it can also be taken by the same party(s) implementing the decryptor. In fact, the server could make public the aggregation tree for "the public" to collectively check it, in the same spirit as distributed verification of key transparency logs [14].

We now present two variants of a distributed verifier, one with overwhelming deterrence for a cheating server, and one with constant (tunable) deterrence. By deterrence we mean the probability of a cheating server being caught. We assume that a pool of c_v clients are available to serve as a verifier, among which we trust $\gamma_v \geq 1/2$ do not collude with the adversary corrupting the server.

Verifier via committee. In this approach the c_v clients in the pool are grouped in c_v/k committee of size $k = O(\sigma + \log(c_v/k))$, ensuring that each committee contains at least a threshold t of honest clients except with negligible probability $2^{-\sigma}$, for statistical security σ. A randomness beacon could be used (as in Flamingo [7]) to make sure these committees are selected uniformly at random, and for the interval assignment that follows.

Then, the leaves of $T(\mathcal{L})$ are split into c_v/k contiguous intervals and each committee signs the root tree after verifying their interval. The decryptor requires at least t signatures from every subtree/interval to decrypt. Each committee includes enough honest parties to verify valid intervals, but not enough corrupted parties to verify an invalid interval. Moreover, by choosing a large enough threshold t we can offer robustness to a fraction of verifiers dropping out. We offer an evaluation of this approach in Section 6.

Verifier via random checks. In situations where a constant deterrence is enough, e.g., because the risk of reputational loss for the server is high, we can rely on the $\gamma_v \cdot c_v$ honest clients to check random intervals of leaves. Concretely, consider a verifier that selects, independently at random, s intervals of length w to be checked. If the server cheats at a given leaf, the probability of a particular check catching the lie is w/n. Therefore, the probability of the server cheating and getting away with it is $p = (1 - w/n)^s \leq e^{\frac{-ws}{n}}$. By having each verifier check $s/(\gamma_v c_v)$ random intervals, the honest majority assumption ensures a cheating server will get caught with probability $\epsilon = 1 - p$.

5.5 Security

We introduce our first security theorem next, which states the security of the protocol described in this section against an adversary that actively corrupts the server, a minority of decryptors, and any number of clients.

Theorem 2 (Malicious Server)**.** *Let n be the number of clients and let m be the number of decryptors. Assume* KAHE *is a KAHE scheme satisfying leakage-resilient security of Definition 5, and assume* AHE *is a* Verifiable *AHE scheme as in Section 4.2 that satisfies Lemmas 3, 4, and 5. Let Π^{Agg} be the protocol formed by Figures 3, 5, 4, 6.*

Then Π^{Agg} securely implements (with abort) functionality $\mathcal{F}^{\text{Agg}}$ (Fig. 1) in the RO model, against a static active adversary corrupting the server, any number of clients, and at most $2t - m - 1$ decryptors.

The Client role runs in 1 round with cost $O(\ell \log n)$. Decryptors have a 2-round setup, and 2-round decryption, running in $O(m + \log n)$. The server runs in $O(n\ell \log n)$.

The verifier role runs in $O(n)$. Moreover, in a distributed verifier with c verifiers grouped in c/k random committees, each verifier committee member runs in $O(nk/c + \log n)$.

We provide the proof in Appendix G. In the next section we will discuss the more complicated case when the server is honest or semi-honest.

5.6 Discussion

Until now we have focused on the case of security against an adversary that maliciously corrupts the server in addition to a subset of clients and decryptors. While this argument is enough to ensure privacy for clients even against this very powerful adversary, it does not provide any guarantees to the operator of the aggregation service who is running the server. In fact, the protocol described in this section does not provide any correctness guarantees, and if we wanted to prove simulation security for it we would have to allow the adversary to modify the output of the ideal functionality. In case the server is semi-honest / passively corrupted, we additionally have to account for the fact that the adversary can observe the sum of honest clients' inputs before choosing whether and how to manipulate the output.

While at first it may seem unintuitive to consider an adversary that corrupts some parties actively and others passively, we argue that this setting matches real-world deployments of secure aggregation well. As described in the introduction, clients in our deployments are usually consumer devices such as smartphones, and we want to allow decryptors to be instantiated by such client devices as well. Therefore, it seems natural to assume a real-world adversary has some clients and decryptors under its complete control. The server on the other hand is usually run by a large organization, which typically have much stricter controls for code running on production services [24], such as code signing, and physical security measures in data centers. These measures could provide integrity for the server's code, while a malicious insider may still be able to observe its memory. The server could also be running in a trusted execution environment such as AMD SEV or Intel TDX, which provide integrity through remote attestation, but generally leak data through various side channels [25]. To provide meaningful correctness guarantees to the server operator, we therefore must consider malicious insiders that can read the server's state without modifying its code, while at the same time being able to fully control a subset of clients and decryptors.

Formally speaking, proving security against an adversary with both semi-honest and malicious corruptions is more involved than just malicious corruptions, and semi-honest security does not necessarily imply malicious security. The reason is that in the indistinguishability game between the real and simulated views, we also have to consider the outputs of semihonest parties, while the output of malicious parties does not matter.

In Appendix H.1, we present a functionality that formalizes the capabilities of an adversary that passively corrupts the server in the protocol from Figures 3, 5, 4, 6. While we do this for illustrative purposes, it is worth remarking that that functionality does not offer meaningful guarantees in the semi-honest server case. In contrast, in Appendix H.2 we describe how to modify our protocol to remove these capabilities, and present a protocol that securely implements $\mathcal{F}^{\mathsf{Agg}}$ from Figure 1 in the presence of a server that is either actively corrupted, passively corrupted, or honest. We should remark that only the decryptor role requires (quite lightweight) changes.

6 Experiments

We implement our protocol in C++ and Rust. We use SHELL [26] for RLWE-based KAHE and AHE schemes, and we extend AHE with the Bulletproofs [27] implementation by de Valence et al. [22] to obtain verifiability. We extend the latter to support approximate and exact range proofs over committed vectors. While our protocol can be instantiated with any ZKPoK system for linear relations, we chose Bulletproofs for two reasons: (1) They allow for batched verification, allowing the server to efficiently verify large batches of client submissions as long as all clients are honest, and (2) they have comparably small size, thereby minimizing the communication overhead, in particular in our malicious protocol.

We use the lattice estimator [28] to estimate the hardness of RLWE problem used in KAHE and AHE, and set parameters to have at least 128 bits of computational security. For KAHE, we set the Gaussian parameters for the secret and error distributions to $\sigma_s = 4.5$ and $\sigma_e = 6.36$, respectively, and we cut off the tail at 6 times

n	c	ℓ	N_1	$\log q_1$	packing factor	N_2	$\log q_2$	**KAHE** comm.	**AHE** comm.	$\log(e_{\mathsf{flood}})$
10^3	10^2	10^3	2048	39	1	4096	68	5.00 KB	52.22 KB	55
10^5	10^2	10^3	2048	49	1	4096	74	6.12 KB	56.83 KB	58
10^7	10^2	10^3	4096	98	2	4096	80	6.12 KB	81.92 KB	61
10^3	10^2	10^5	8192	195	7	4096	77	348.22 KB	78.85 KB	55
10^5	10^2	10^5	8192	212	6	4096	86	441.68 KB	88.06 KB	58
10^7	10^2	10^5	8192	216	5	4096	96	540.00 KB	98.30 KB	61
10^3	10^2	10^7	16384	429	16	4096	95	33.52 MB	98.30 KB	55
10^5	10^2	10^7	16384	408	12	4096	87	42.60 MB	178.18 KB	58
10^7	10^2	10^7	16384	412	10	4096	96	51.50 MB	196.61 KB	61

Table 2: Concrete parameters used in our experiments for computational security $\lambda \geq 128$. We instantiate our KAHE using a ring $R_{q_1} = \mathbb{Z}_{q_1}[X]/(X^{N_1}+1)$, and AHE using a ring $R_{q_2} = \mathbb{Z}_{q_2}[X]/(X^{N_2}+1)$ and scaling factor Δ. We report the per-client communication costs on KAHE and AHE, and the l_∞ norm of the total flooding noise e_{flood}.

the standard deviation. Our KAHE scheme achieves the desired leakage-resilient security by Lemma 1. For our verifiable AHE, we set both χ_s and χ_e to be the discrete Gaussian distribution with standard deviation 3.2, which are standard choices for achieving semantic security in practice [29]. Furthermore, we set our flooding noise parameters to have 40 bits of statistical security according to the analysis in [30, Corollary 2], and we implement the constant-time discrete Gaussian sampler of Micciancio and Walter [31] for these distributions. For the proof systems Π_{KeyGen} and Π_{Enc}, we apply approximate Euclidean norm proofs to prove a slightly loose bound $B_s = B_e \approx 32\sqrt{N_2}$. When the proof $\Pi_{\mathsf{PartialDec}}$ is needed, we apply the exact norm proof using Lagrange three square theorem. See Appendix E for details of these proof systems.

More specifically, setting AHE parameters in our protocol requires several steps. First, we determine for each honest decryptor j, the l_∞ norm of the secret-dependent term $\sum_{i=1}^{n} e_i'' \cdot s_j + \sum_{i=1}^{n} r_i \cdot f_j$, which is bounded from above by $\beta = n\sqrt{N} \cdot (B_s \|s_j\|_\infty + B_e \|f_j\|_\infty)$ where B_s and B_e are the verifiable bounds proved in Π_{KeyGen} and Π_{Enc} by all clients. We then use this bound β to determine the parameters of the flooding noise e_{flood} as in Lemma 5; in experiments we see that $\|e_{\mathsf{flood}}\|_\infty$ is around 60 bits for up to 10^7 clients with binary input of length up to 10^7. To determine the size of the total error and thus to achieve optimal efficiency in our experiment settings, we numerically estimated the size on the aggregated honest error terms. Meanwhile, to minimize the KAHE ciphertext expansion ratio, we use up to 429 bits KAHE modulus q_1 to pack up to 62 input values in each KAHE ciphertext coefficient. Using such q_1 results in a large ring degree N_1 up to 2^{14} for security reasons; however, thanks to using small secret keys in our KAHE scheme, we managed to pack multiple KAHE secret coefficients in AHE ciphertext coefficients, and we use at most two AHE ciphertexts in all the input settings we consider. As a result our AHE ciphertext modulus q_2 is only 68 to 96 bits with a relatively small $N_2 = 2^{12}$, allowing an efficient AHE scheme implementation using up to two `uint64_t` RNS moduli. According to our benchmark, KAHE encryption takes at most 406ms, AHE encryption takes at most 8ms, and partial decryption takes up to 425ms mostly due to sampling large Gaussian flooding noises. For communication cost, a single AHE ciphertext is at most 91KB, and similar to KAHE, we can discard unused coefficients in the component ct^0. Table 2 lists the parameters we used in our experiments.

6.1 Microbenchmarks.

We conducted microbenchmarks on a Google Cloud `n2-standard-48` instance with Intel Ice Lake CPU supporting AVX512 SIMD instructions with 192GB memory. The benchmark results of our protocol are presented in Table 3. Encryption, homomorphic aggregation, and distributed decryption can all be done in the order of milliseconds. The largest per-client cost comes from the zero-knowledge proof generation (0.8s on the client) and verification (327ms per client on the Server and Verifier). We do not see this hindering the practicality of our protocol for two reasons: First, since the zero-knowledge proof from the client is with respect to a random secret, it could be both proven and verified in an offline phase, before the data is

ℓ	Client			Server				Decryptor				Verifier	
	ZKProve	Encrypt	Comm.	Aggregate (per client)	ZKverify (per client)	ZKVerify (setup)	ZKverify (decryption)	ZKProve (setup)	ZKProve (decryption)	Decrypt	Comm.	ZKverify (per client)	Comm. (per client)
10^3		2.3ms	57KB	0.0144ms									
10^5	< 829ms	8.2ms	427KB	0.0522ms	<227ms	<227ms	<635ms	<829ms	<1.9s	<53ms	105KB	<227ms	<1.5KB
10^7		296ms	34MB	0.1729ms									

Table 3: Microbenchmarks of concrete computation and communication costs for several inputs lenghts ℓ, with $n = 1000$ clients. For each client, we measure encryption, proof generation, and communication cost. For the server, we measure the aggregation running time (per client), and the running time of verification of (a) a client's proof π_{Enc}, (b) a decryptor's public key proof π_{KeyGen}, and (c) a partial decryption proof $\pi_{\mathsf{PartialDec}}$. For the decryptor we measure the proof generation costs of π_{KeyGen} and $\pi_{\mathsf{PartialDec}}$, at setup and at decryption. Finally, for the verifier we report the per-client running time and communication, assuming a single verifier. See Figure 8 for experiments with a distributed verifier.

even known to the client. Second, proof verification of each client can be performed independently and in parallel. This is because, unlike prior work (see next section), our clients don't have to block on each other. In practice, the wall-clock time of our protocol will therefore not be dominated by the total computation cost.

We also note that our implementation largely builds on an existing implementation of Bulletproofs [22], which lacks several optimizations. For example, it assumes that the relation being proven is represented as a quadratic constraint, meaning that both sides of the final inner product remain private. However, our proofs (see Appendix E) only require linear constraints. As observed by Gentry et al. [21], exploiting this can save up to half of the Bulletproof prover time. A second optimization left for future work is batching the proof verification, which as pointed out by [27, Section 6.3] greatly reduces the server cost.

6.2 Comparison with Prior Work.

Before comparing against prior work, we emphasize that our protocol is the first to allow single-server aggregation with dynamic clients where committee work is sub-linear in the number of clients. As we argue in Section 3, this is a major qualitative difference for large-scale deployments, as it allows scaling to millions of clients while tolerating large dropout rates. as such any quantitative comparison with prior work is inherently inaccurate.

The prior works closest to ours are LERNA (Li et al. [9]) and Flamingo (Ma et al. [7]). While they do not support dynamic participation, they also utilize a committee to help in the aggregation.

First, let us consider server computation. The Flamingo protocol requires about 2.5s for 1000 clients [7, Figure 7], or 2.5ms per client. LERNA on the other hand aggregates 20000 client contributions in 5s, or 250μs per client of server time. In comparison, our protocol requires 227ms of computation time per client (See Table 3). While this is almost two orders of magnitude more than the two related works, we believe the scalability of our protocol outweighs its cost for most real-world applications. Moreover, as clients in our protocol are completely independent, our per-client running times are representative of our protocol's throughput, which is not the case for Flamingo and LERNA, where clients have to wait for each other. In terms of monetary cost our protocol is still practical: Using the Google Cloud spot price of 0.4 US cents per vCPU-hour at the time of writing [32], an aggregation of a million clients with vectors of length 10^7 costs less than 50 cents.

In Figure 7, we compare the client communication cost of our protocol against Flamingo. We exclude LERNA here, since its client communication is at least 2GB [9, Table 3]. It can be seen that as the vector length increases, the upload size of our protocol approaches that of Flamingo. Both protocols require about 10MB for contributing a vector of 10^7 16-bit numbers.

Finally, we compare the communication cost of committee members. Here our protocol allows some flexibility as to how the verifier role is implemented (see Section 5.4). We consider both a fully malicious server, which requires the verifiers to check all aggregated contributions, and a covert server with that will

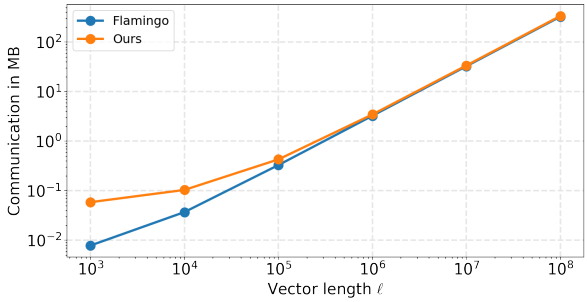

Figure 7: Client communication costs of our protocol and Flamingo for various vector lengths, input range $t = 2^{16}$, statistical security parameter $\sigma = 40$, and 10% dropouts.

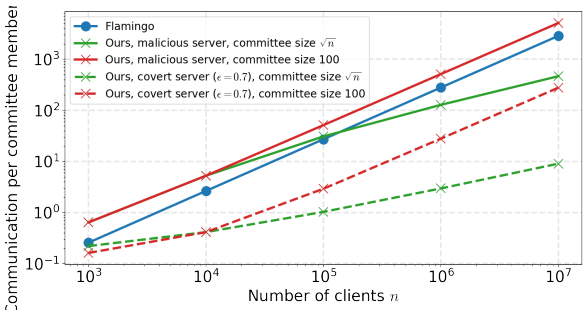

Figure 8: Committee communication cost of different variants of our protocol and Flamingo. This includes both Decryptor and Verifier cost.

get caught with a probability of 70% when cheating. In both settings, we either set the number of verifiers to 100, or scale it with $\sqrt{n}$ as the number of clients increases. We fix the number of decryptors to 100.

When the number of committee members is fixed, both our protocol and Flamingo require work from each committee member that is linear in the number of clients n. This is to be expected, since in Flamingo the committee has to decrypt all client contributions, while in our protocol it needs to verify them (or a constant fraction of them in the covert case). However, a main advantage of our protocol is that it benefits from scaling the committee up. With a committee size of $\sqrt{n}$, our protocol outperforms Flamingo even in the malicious case as soon as n exceeds 10^5.

7 Conclusion

Our work greatly increases the practicality of secure aggregation, by removing synchronization points between clients while at the same time only requiring lightweight computations and small communication overhead from helper parties. By further avoiding the need for a PKI between clients, our work is well suited for real-world deployments with dynamic client participation. Since the bottleneck of our construction is the verification of zero-knowledge proofs on the server, we believe that progress in that area will directly translate into improved efficiency for our protocol. Beyond the single-server setting, our protocol can be instantiated with two non-colluding servers with asymmetric resource requirements. Given that this asymmetry is often present when deploying protocols between real-world parties, we see other protocols with this property as an interesting target for future research.

References

[1] Kallista Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *ACM SIGSAC Conf. on Comp. and Comm. Security*, pages 1175–1191. ACM, 2017.

[2] Peter Kairouz, H. Brendan McMahan, et al. Advances and open problems in federated learning. *CoRR*, abs/1912.04977, 2019. URL `http://arxiv.org/abs/1912.04977`.

[3] James Henry Bell, Kallista A Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly) logarithmic overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1253–1269, 2020.

[4] Jinhyun So, Corey J Nolet, Chien-Sheng Yang, Songze Li, Qian Yu, Ramy E Ali, Basak Guler, and Salman Avestimehr. Lightsecagg: a lightweight and versatile design for secure aggregation in federated learning. *Proceedings of Machine Learning and Systems*, 4:694–720, 2022.

[5] Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. Eiffel: Ensuring integrity for federated learning. In *CCS*, pages 2535–2549. ACM, 2022.

[6] Hidde Lycklama, Lukas Burkhalter, Alexander Viand, Nicolas Küchler, and Anwar Hithnawi. Rofl: Robustness of secure federated learning. In *SP*, pages 453–476. IEEE, 2023.

[7] Yiping Ma, Jess Woods, Sebastian Angel, Antigoni Polychroniadou, and Tal Rabin. Flamingo: Multi-round single-server secure aggregation with applications to private federated learning. In *SP*, pages 477–496. IEEE, 2023.

[8] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. ACORN: input validation for secure aggregation. In *USENIX Security Symposium*, pages 4805–4822. USENIX Association, 2023.

[9] Hanjun Li, Huijia Lin, Antigoni Polychroniadou, and Stefano Tessaro. LERNA: secure single-server aggregation via key-homomorphic masking. In *ASIACRYPT (1)*, volume 14438 of *Lecture Notes in Computer Science*, pages 302–334. Springer, 2023.

[10] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, pages 259–282. USENIX Association, 2017.

[11] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. In *SCN*, volume 13409 of *Lecture Notes in Computer Science*, pages 516–539. Springer, 2022.

[12] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[13] Cloudflare. League of Entropy, 2024. URL `https://www.cloudflare.com/leagueofentropy/`.

[14] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: bringing key transparency to end users. In *USENIX Security Symposium*, pages 383–398. USENIX Association, 2015.

[15] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany. doi: 10.1007/3-540-48910-X_16.

[16] Leonid Reyzin, Adam D. Smith, and Sophia Yakoubov. Turning HATE into LOVE: compact homomorphic ad hoc threshold encryption for scalable MPC. In *CSCML*, volume 12716 of *Lecture Notes in Computer Science*, pages 361–378. Springer, 2021.

[17] Rikke Bendlin and Ivan Damgård. Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems. In Daniele Micciancio, editor, *TCC 2010: 7th Theory of Cryptography Conference*, volume 5978 of *Lecture Notes in Computer Science*, pages 201–218, Zurich, Switzerland, February 9–11, 2010. Springer, Heidelberg, Germany. doi: 10.1007/978-3-642-11799-2_13.

[18] Katharina Boudgoust and Peter Scholl. Simple threshold (fully homomorphic) encryption from LWE with polynomial modulus. In *ASIACRYPT (1)*, volume 14438 of *Lecture Notes in Computer Science*, pages 371–404. Springer, 2023.

[19] Daniele Micciancio and Adam Suhl. Simulation-secure threshold PKE from LWE with polynomial modulus. *IACR Cryptol. ePrint Arch.*, page 1728, 2023. URL https://eprint.iacr.org/2023/1728.

[20] Duhyeong Kim, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. Toward practical lattice-based proof of knowledge from hint-mlwe. In *CRYPTO (5)*, volume 14085 of *Lecture Notes in Computer Science*, pages 549–580. Springer, 2023.

[21] Craig Gentry, Shai Halevi, and Vadim Lyubashevsky. Practical non-interactive publicly verifiable secret sharing with thousands of parties. In *EUROCRYPT (1)*, volume 13275 of *Lecture Notes in Computer Science*, pages 458–487. Springer, 2022.

[22] Henry de Valence, Cathie Yun, and Oleg Andreev. Bulletproofs, 2018. URL https://github.com/zkcrypto/bulletproofs.

[23] Edo Roth, Daniel Noble, Brett Hemenway Falk, and Andreas Haeberlen. Honeycrisp: large-scale differentially private aggregation without a trusted core. In *SOSP*, pages 196–210. ACM, 2019.

[24] Google Cloud. Binary authorization for borg, 2024. URL https://cloud.google.com/docs/security/binary-authorization-for-borg.

[25] Mengyuan Li, Yuheng Yang, Guoxing Chen, Mengjia Yan, and Yinqian Zhang. Sok: Understanding design choices and pitfalls of trusted execution environments. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1600–1616, 2024.

[26] SHELL authors. Simple homomorphic encryption library with lattices (SHELL), 2021. URL https://github.com/google/shell-encryption.

[27] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society, 2018.

[28] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015.

[29] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[30] Baiyu Li, Daniele Micciancio, Mark Schultz, and Jessica Sorrell. Securing approximate homomorphic encryption using differential privacy. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part I*, volume 13507 of *Lecture Notes in Computer Science*, pages 560–589, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany. doi: 10.1007/978-3-031-15802-5_20.

[31] Daniele Micciancio and Michael Walter. Gaussian sampling over the integers: Efficient, generic, constant-time. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 455–485, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany. doi: 10.1007/978-3-319-63715-0_16.

[32] Google Cloud. Spot VMs pricing, 2024. URL https://cloud.google.com/spot-vms/pricing.

[33] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer International Publishing, 2017.

[34] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, pages 62–73. ACM, 1993.

[35] Mohamad Mansouri, Melek Önen, Wafa Ben Jaballah, and Mauro Conti. Sok: Secure aggregation based on cryptographic schemes for federated learning. *Proc. Priv. Enhancing Technol.*, 2023(1):140–157, 2023.

[36] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptol.*, 1(1):65–75, 1988.

[37] Marek Jawurek and Florian Kerschbaum. Fault-tolerant privacy-preserving statistics. In *Privacy Enhancing Technologies*, volume 7384 of *Lecture Notes in Computer Science*, pages 221–238. Springer, 2012.

[38] Zizhen Liu, Si Chen, Jing Ye, Junfeng Fan, Huawei Li, and Xiaowei Li. SASH: efficient secure aggregation based on SHPRG for federated learning. In *UAI*, volume 180 of *Proceedings of Machine Learning Research*, pages 1243–1252. PMLR, 2022.

[39] Guilhem Castagnos and Fabien Laguillaumie. Linearly homomorphic encryption from ddh. In *CT-RSA*, volume 9048 of *Lecture Notes in Computer Science*, pages 487–505. Springer, 2015.

[40] Lennart Braun, Ivan Damgård, and Claudio Orlandi. Secure multiparty computation from threshold encryption based on class groups. In *CRYPTO (1)*, volume 14081 of *Lecture Notes in Computer Science*, pages 613–645. Springer, 2023.

[41] Lennart Braun, Guilhem Castagnos, Ivan Damgård, Fabien Laguillaumie, Kelsey Melissaris, Claudio Orlandi, and Ida Tucker. An improved threshold homomorphic cryptosystem based on class groups. In *SCN (2)*, volume 14974 of *Lecture Notes in Computer Science*, pages 24–46. Springer, 2024.

[42] Aniket Kate, Easwar Vivek Mangipudi, Pratyay Mukherjee, Hamza Saleem, and Sri Aravinda Krishnan Thyagarajan. Non-interactive VSS using class groups and application to DKG. *IACR Cryptol. ePrint Arch.*, page 451, 2023.

[43] Harish Karthikeyan and Antigoni Polychroniadou. OPA: One-shot private aggregation with single client interaction and its applications to federated learning. Cryptology ePrint Archive, Paper 2024/723, 2024. URL `https://eprint.iacr.org/2024/723`. `https://eprint.iacr.org/2024/723`.

[44] Swanand Kadhe, Nived Rajaraman, Onur Ozan Koyluoglu, and Kannan Ramchandran. Fastsecagg: Scalable secure aggregation for privacy-preserving federated learning. *CoRR*, abs/2009.11248, 2020.

[45] Duhyeong Kim, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. Toward practical lattice-based proof of knowledge from hint-MLWE. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part V*, volume 14085 of *Lecture Notes in Computer Science*, pages 549–580, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Heidelberg, Germany. doi: 10.1007/978-3-031-38554-4_18.

[46] Thomas Espitau, Guilhem Niot, and Thomas Prest. Flood and submerse: Distributed key generation and robust threshold signature from lattices. In *CRYPTO (7)*, volume 14926 of *Lecture Notes in Computer Science*, pages 425–458. Springer, 2024.

[47] Ngoc Khanh Nguyen. *Lattice-Based Zero-Knowledge Proofs Under a Few Dozen Kilobytes*. PhD thesis, ETH Zurich, Zürich, Switzerland, 2022. URL `https://hdl.handle.net/20.500.11850/574844`.

[48] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[49] Peter Kairouz, Ziyu Liu, and Thomas Steinke. The distributed discrete gaussian mechanism for federated learning with secure aggregation. In *International Conference on Machine Learning*, pages 5201–5212. PMLR, 2021.

[50] Amos Beimel, Kobbi Nissim, and Eran Omri. Distributed private data analysis: Simultaneously solving how and what. In *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 451–468. Springer, 2008.

A Security definitions

Our security proofs are in the ideal vs. real paradigm. We follow closely the definitions in Lindell's tutorial [33], but simplify some of them to match our setting, e.g., only the Server obtains an output, and only clients have input.

As expected, we define the ideal world execution by means of a so-called functionality, denoted $\mathcal{F}_I^{\mathcal{A}}$, consisting of a computation between the honest and corrupted parties (operated by an ideal-world adversary $\mathcal{A}_I$), mediated by a trusted party. The ideal world defines the standard for security achieved by the protocol. By $\text{IDEAL}_{\mathcal{F}_I^{\mathcal{A}}}\big((\mathbf{x_i})_i, \lambda\big)$ we denote the joint distribution of (i) the output of (semi-)honest parties and (ii) the output of the adversary $\mathcal{A}_I$ running functionality $\mathcal{F}_I^{\mathcal{A}}$ with inputs $\big(\mathbf{x_i}\big)_i$, while controlling the corrupted parties. Finally, λ denotes the security parameter, which we might omit for simplicity. Note that in our setting only the server has output.

In the ideal vs. real paradigm a concrete protocol π is secure if its leakage to an attacker $\mathcal{A}_R$ corrupting some of the parties in the protocol can be obtained by an attacker running in the ideal world $\mathcal{A}_I$. This proves that whatever leakage can be obtained in the protocol, can also be obtained in an ideal world that is secure by definition. We use $\pi_R^{\mathcal{A}}$ to denote an execution of protocol π in the context of $\mathcal{A}_R$. By $\text{REAL}_{\pi_R^{\mathcal{A}}}\big((\mathbf{x_i})_i, \lambda\big)$ we denote the joint distribution of the output of (semi-)honest parties and the adversary $\mathcal{A}_R$ after running the protocol where $\mathcal{A}_R$ corrupts some of the the parties.

Corruption model. We assume that the adversary corrupts parties statically, i.e. once before the protocol starts. Moreover, we consider both passive/semi-honest and active/malicious corruption. In the former, the adversary might observe the internal state of corrupted parties, but they must behave as prescribed by π. In the latter they might behave arbitrarily. Concretely, we consider two corruption models, and provide protocols for both. In both cases the adversary corrupts the server and other parties, i.e., decryptors, clients and verifiers, simultaneously. In the first setting the server is corrupted passively, while the rest of the parties are corrupted actively. This models the situation where the attacker can launch some parties fully under their control, while being able to only observe the execution of the server (e.g., because malicious modifications of Server code would be deemed suspicious). In the second case the adversary fully controls also the server.

Definition 1. *We say that a protocol* π securely computes *functionality* $\mathcal{F}$ *if, for every real-world adversary* $\mathcal{A}_R$*, if there exists a probabilistic polynomial time* Sim *so that, for all inputs* $(\mathbf{x_i})_i$*,*

$$\text{IDEAL}_{\mathcal{F}^{\mathsf{Sim}}}\big((\mathbf{x_i})_i, \lambda\big) \equiv \text{REAL}_{\pi^{\mathcal{A}_R}}\big((\mathbf{x_i})_i, \lambda\big)$$

where $\equiv$ *denotes computational indistinguishability with respect to security parameter* λ*, over the randomness of* $\mathcal{F}$ *and* π*.*

In accordance to the previous definition, one way to prove that a protocol π is secure, is to exhibit a probabilistic polynomial time simulator Sim which takes a description of the adversary (either as a black-box, a circuit or code) as input before producing the above ideal view. With this approach the simulator has black-box access to $\mathcal{A}_R$, and can set its randomness, input, and auxiliary input, so we can consider $\mathcal{A}_R$ to have those fixed/hardcoded, and therefore it is a deterministic algorithm with no input (see Remark 6.5 in Lindell's tutorial for a discussion on this point [33]). One concrete way of thinking about black-box access is that Sim can issue a `next-action(event)` query on $\mathcal{A}_R$, and obtain the next action party each of the corrupted parties takes, given an `event`, e.g. an incoming message from honest parties. Actions correspond to (i) aborting, (ii) sending a message to another (possibly corrupted) party, and (iii) termination possibly producing an output.

Definition 2. *We say that a proof that a protocol π satisfies definition 1 is a* Black-Box *security proof if it provides a PPT algorithm that can simulate (in the sense of fulfilling the computational indistinguishability in Definition 1) any adversary it is provided as a black-box. We say the proof is a* White-Box *security proof if it provides a PPT algorithm that can simulate any adversary whose code or circuit description it is provided with.*

We note that the vast majority of security proofs are black-box.

A.1 Definitions in the Single-Server Setting

For clarity, we specialize the above definitions to our setting.

(Semi)Honest Server case. We first consider the case where the server is either honest, or passively corrupted, while the adversary also actively corrupts a fraction of the clients and decryptors.

In this case, we define the ideal view as the joint distribution of output for the server, and output of the real-world attacker, after an interaction with the ideal functionality. Including the output of the server in the ideal world distribution is an important aspect of modelling security in the real vs. ideal model that captures a correctness requirement: the real world adversary shouldn't be able to cause the server to receive an incorrect output. Here, by incorrect we mean an output different from the one prescribed by the functionality. Note that in our setting only the server has output, and only clients have inputs.

Definition 3 ((Semi)honest Server, Ideal View)**.** *Consider a setting with n clients holding private inputs $(\mathbf{x_i})_{i\in n}$, and a Server S_I that is the intended recipient of the sum of all clients' inputs. Let $\mathcal{F}_I^{\mathcal{A}}\big((\mathbf{x_i})_{i\in n}\big)$ be a functionality that interacts with an ideal-world adversary $\mathcal{A}_I$, resulting in the server S_I receiving output $\mathtt{output}_{\mathsf{S}_I}$. Let $\mathtt{output}_{\mathcal{A}_I}$ be $\mathcal{A}_I$'s output at the end of the interaction. We define the ideal world view of functionality $\mathcal{F}^{\mathcal{A}}$ as*

$$\mathrm{IDEAL}_{\mathcal{F}^{\mathcal{A}_I}}\big((\mathbf{x_i})_i, \lambda\big) = (\mathtt{output}_{\mathsf{S}_I}, \mathtt{output}_{\mathcal{A}_I}).$$

That is, the joint *distribution of server and adversary outputs when interacting with the ideal functionality $\mathcal{F}_I^{\mathcal{A}}$, where λ denotes a security parameter.*

We define a real execution accordingly, as the joint distribution of the output obtained by the server, and the output of the adversary, after an execution of a protocol π.

Definition 4 ((Semi)honest server, Real View)**.** *Consider a setting with n clients holding private inputs $(\mathbf{x_i})_{i\in n}$, a server S_R, and c decryptors. Let $\mathcal{A}_R$ be an static adversary either* passively *corrupting the server and* actively *corrupting a fraction of the clients and decryptors, or only* actively *corrupting a fraction of the clients and decryptors. Let π be a randomized protocol, resulting in the server S receiving output $\mathtt{output}_{\mathsf{S}_R}$. Let $\mathtt{output}_{\mathcal{A}_R}$ be $\mathcal{A}_R$'s output at the end of the execution. We define the real-world execution of π interacting with adversary $\mathcal{A}_R$, as*

$$\mathrm{REAL}_{\pi^{\mathcal{A}_R}}\big((\mathbf{x_i})_i, \lambda\big) = (\mathtt{output}_{\mathsf{S}}, \mathtt{output}_{\mathcal{A}_R}).$$

That is, the joint *distribution of server and adversary outputs when running protocol π, where λ denotes a security parameter.*

Malicious Server case. In the case where the server also actively corrupts the server, along with a fraction of the clients and decryptors correctness can't be expected, as the adversary can instruct the server to output a value of their choice. Accordingly, we define the ideal world view of functionality $\mathcal{F}^{\mathcal{A}}$ as

$$\mathrm{IDEAL}_{\mathcal{F}^{\mathcal{A}_I}}\big((\mathbf{x_i})_i, \lambda\big) = \mathtt{output}_{\mathcal{A}_I}.$$

and the real-world execution of π interacting with adversary $\mathcal{A}_R$, as

$$\mathrm{REAL}_{\pi^{\mathcal{A}_R}}\big((\mathbf{x_i})_i, \lambda\big) = \mathtt{output}_{\mathcal{A}_R}.$$

A.2 Random Oracle Model

Like previous works [1, 3], our maliciously secure protocol is proven secure in the Random Oracle Model (RO) [34].

A random oracle can be regarded as a public randomize funcionality $\mathcal{F}_{\text{RO}}$ that, on input (x, ℓ) outputs a random string of length ℓ such that

1. $\mathcal{F}_{\text{RO}}(x, \ell)$ is a independently sampled uniformly random length ℓ string.
2. Repeated queries on the same point points, i.e. $\mathcal{F}_{\text{RO}}(x, \ell)$ output the same value.

In our proofs for malicious security, we assume parties are equipped with access to a common random oracle $\mathcal{F}_{\text{RO}}$. All parties can query the oracle during the execution. Moreover, calls to the the expanding pseudorandom generator $\mathsf{PRG.Expand}(\mathsf{seed}, \ell)$ in the protocols are replaced by calls to $\mathcal{F}_{\text{RO}}(\mathsf{seed}, \ell)$.

B Related Work

In this section we discuss previous work in secure aggregation in the single-server setting, giving particular attention to protocols with one-shot clients and the requirement for synchronization across clients. For a survey, see [35].

Solutions based on Pairwise Masking. An important family of protocols for single-server secure aggregation follow a dining cryptographers based approach [36], enhanced with robustness to dropouts. These include Bonawitz et al. [1] and subsequent improvements [3, 8]. The basic structure of these protocols is that clients mask their input before they report it to the server with both (i) pairwise-masks, i.e., shares of zero vectors computed with some of the other clients – their so-called neighbors – and (ii) self-masks, i.e., a pseudorandom vector. Crucially, in a setup phase clients secret-share with each other key material to recover such masks. The server aggregates all received masked inputs, which result in a masked sum. In a subsequent recovery phase, the server request shares to recover self-masks of clients that reported their masked input, and pairwise-marks of dropouts. There are two important assumptions in these works, which we lift in Willow. First, for malicious security either the server is assumed to be semi-honest during key distribution, or a PKI holding keys *for all clients* is in place. Moreover, these protocols involve several rounds among clients, each of which constitutes a synchronization point. Remarkably, the setup phase where clients share key material to be able to recover masks in the recovery phase constitutes a synchronization point that is inherent to this family of protocols. We will come back to this point later.

A recent work operating in this paradigm worth discussing is Flamingo [7]. While the works mentioned above tackle an aggregation task in isolation, Flamingo reduces the overall round trip complexity for sequences of T sums, which arise naturally in applications of secure aggregation to federated learning. To achieve this, Flamingo relies on an honest majority committee, just like Willow. Moreover, clients contributing data to an aggregation send a single message. However, Flamingo makes two important assumptions to achieve this, which are not required in Willow: (a) a PKI is available for all clients participating in the aggregation, (b) the subset of clients participating in round i are set by the protocol (possibly via a random beacon) and a significant fraction of them are expected to be online for aggregation when round i takes place. While these assumptions might be acceptable in some cases, as discussed in Section 2 they are not realistic in our setting. In fact, assumption (b) is highlighted by the authors of Flamingo as a limitation, and exploring "the case of handling clients that dynamically join the training session" is left as an open problem in their work.

Modifications to Flamingo to drop these assumptions are conceivable, but come at the expense of introducing a synchronization point between clients. The reason is that the protocol would have to wait for sufficiently many clients to show up in a given round, to only then assign neighbors to clients and start negotiating pairwise masks. This is related to the claim above that techniques based on DC networks inherently preclude dynamic client participation. In Table 1 we provide an asymptotic comparison of our protocols with Acorn [8] and Flamingo. Acorn is the state of the art pairwise masking based solution as far as we

know. There, the pairwise masking technique is implemented via an (almost) key homomorphic PRF based on RLWE. Note that, while achieving one-shot clients, Willow's asymptotic costs are better than Flamingo and Acorn.

Solutions based on HE. A natural approach to allow dynamic client participation is to employ additively homomorphic encryption. While we refer the reader to a recent survey [35] for details on HE-based protocols, there are a couple of works worth discussing here.

Jawurek and Kerschbaum [37] explore the HE-based approach using Paillier encryption, where clients directly encrypt $\mathbf{x_i}$ and the server homomorphically aggregates. However, this approach has two main drawbacks: ciphertext expansion, and more importantly, decryptor communication. In our target applications, having the decryptors cost grow as $O(\ell)$ is impractical, as we want to keep that role as light-weight as possible.

Two more recent works overcome the large $O(\ell)$ decryptor cost for long vectors: SASH [38] and LERNA [9]. Both these works share a core idea with Willow: By employing (almost) key homomorphic PRFs (in both cases based on the Learning With Rounding assumption) SASH and LERNA reduce the problem of aggregation of long vectors to aggregation of short keys. In SASH, each client i sends encryptions of their input under a key k_i, and then keys k_i are aggregated by using the protocol from Bell et al. [3] discussed above. Therefore SASH clients are not one-shot. LERNA uses a similar idea in a setting with an honest majority committee analogous to the one of Willow and Flamingo, but resulting in one-shot clients. As in SASH, client i sends key homomorphic encryptions of their input under a key k_i (and a session tag). Additionally, k_i is shared with the committee. Note that clients can do this in one round. As the server homomorphically aggregates contributions from a set S of clients, it requests from the committee a sharing of the corresponding sum of keys (appropriately transformed so that keys are reusable). The communication costs of LERNA in the sharing stage are quite significant: authors report 2GB of communication per client, and 4.4GB per committee member. This cost can be amortized across many aggregations *involving the same clients*, and therefore LERNA is well suited for such applications. However, this does not transfer to our setting with dynamic client participation.

An alternative to (R)LWE-based HE schemes is provided by class groups, formalized by Castagnos and Laguillaumie [39] in the CL framework. These are interesting for our use case because of they enable comparably small ciphertexts, and are compatible with straight-forward zero-knowledge proofs like sigma protocols. At the same time, class-group based encryption schemes allow for efficient decryption even with large plaintext space, and have efficient threshold schemes [40, 41, 42]. However, given that small ciphertexts are less relevant in the federated learning setting we focus on (where plaintexts are long vectors), we build our protocols from the more well-studied hint-RLWE assumption. Still, we do see a CL-based instantiation of Willow as interesting future work.

Concurrent Work. In recent concurrent work, Karthikeyan and Polychroniadou [43] present a protocol called OPA that also allows for one-shot clients and dynamic participation. Like our work, they rely on a committee of clients to aid in the computation. At the core of their construction is a "key-homomorphic PRF" primitive, which they instantiate from class groups [39] and the Learning With Rounding assumption. A key difference between our protocol and OPA is the fact that they rely on secret sharing, which inherently requires each client to communicate with each committee member (via the server). For that reason, committee members in OPA incur $\Omega(n)$ communication, where n is the number of clients, and client communication similarly scales with the number of committee members [43, Table 4]. In contrast, our committee can be instantiated with $o(n)$ communication per member (by setting the number of verifiers to $n/o(n)$), and client communication is independent of the committee size. See Table 1.

Solutions based on Secret-Sharing. Another line of work including FastSecAgg [44] and the work of So et al. [4] relies mostly on secret sharing, i.e. robust coding techniques: clients secret share their input vector with a committee of clients (or every other client) and shares are aggregated and returned to the server for

reconstruction. As discussed by Ma et al. [7], the main limitations with this approach is communication. Both FastSecAgg and the protocol of So et al. assume *both* the clients and the server are semi-honest.

Input validation. Another line of work in Single-Server Secure Aggregation is concerned with enhancing protocols with input validation [5, 6, 8]. We do not consider this aspect in this paper, but we believe that the techniques from Acorn [8] are compatible with ours, as their KAHE scheme is the same.

C Key-Additive Homomorphic Encryption (KAHE)

We will use a symmetric key KAHE scheme consisting of the following algorithms

- $\mathsf{KAHE.Setup}()$ which returns a public parameter. We assume this public parameter is implicit in all the following algorithms.
- $\mathsf{KAHE.KeyGen}()$ which returns a key $\mathbf{k}$.
- $\mathsf{KAHE.Enc}(x, \mathbf{k})$ which encrypts a value x with key $\mathbf{k}$, returning a ciphertext $\mathbf{c}$.
- $\mathsf{KAHE.Dec}(\mathbf{c}, \mathbf{k})$ which decrypts a ciphertext $\mathbf{c}$ under the key $\mathbf{k}$ and returns the underlying plaintext.

Key and Message Additive Homomorphism. Given any two ciphertexts $\mathbf{c}_1$ and $\mathbf{c}_2$ encrypting x_1 and x_2 under keys $\mathbf{k}_1$ and $\mathbf{k}_2$ respectively, we require that $\mathbf{c}_1 + \mathbf{c}_2$ is a valid encryption of $x_1 + x_2$ under the key $\mathbf{k}_1 + \mathbf{k}_2$.

Leakage-resilient security. Loooking ahead, each client will encrypt its input under its own KAHE secret key, and the server will learn the sum of all clients' secret keys to decrypt the aggregated KAHE ciphertexts. In order to protect individual input under such leakage of the sum of secret keys, we require that the KAHE scheme must satisfy the following leakage-resilient security.

Definition 5. *For any $n > 0$, we say that a KAHE scheme $\mathcal{E}$ is* n-semantic secure under leakage of sum of secret keys *if there exists an e cient simulator* Sim *such that, for any sequence of input $x_1, \ldots, x_n$, the following distribution*

$$D_0 = \left\{ (\mathsf{a}, \sum_{i=1}^{n} \mathbf{k}_i, \mathbf{c}_1, \ldots, \mathbf{c}_n) \;\middle|\; \begin{array}{r} \mathsf{a} \leftarrow \mathsf{Setup}(), \\ \forall i \in [n]. \mathbf{k}_i \leftarrow \mathsf{KeyGen}(), \\ \mathbf{c}_i \leftarrow \mathsf{Enc}(x_i, \mathbf{k}_i) \end{array} \right\},$$

is computationally indistinguishable from

$$D_1 = \{(\mathsf{a}, \mathsf{Sim}(\mathsf{a}, \sum_{i=1}^{n} x_i)) \mid \mathsf{a} \leftarrow \mathsf{Setup}()\}.$$

RLWE-based KAHE scheme. We instantiate a KAHE scheme based on RLWE assumption. Let N_1 be a power of two, and let $R_{q_1} = \mathbb{Z}[X]/(q_1, X^{N_1} + 1)$ be a quotient ring for some modulus $q_1 > 0$. Let $t_1 > 0$ be an integer coprime to q_1; the plaintext space in our KAHE scheme is $R_{t_1} = \mathbb{Z}[X]/(t_1, X^{N_1} + 1) \equiv \mathbb{Z}_{t_1}^{N_1}$. For any $\sigma > 0$, let D_σ be the distribution over degree-N_1 polynomials such that the coefficients are independent discrete Gaussians with parameter σ. Let $\sigma_s, \sigma_e > 0$ be Gaussian parameters for the secret and error distributions. Then

- $\mathsf{KAHE.Setup}() = \mathsf{a}$: Samples and returns $\mathsf{a} \leftarrow R_{q_1}$ as the public parameter which is implicit in all of the following algorithms.
- $\mathsf{KAHE.KeyGen}() = \mathbf{k}$: Sample $\mathbf{k} \leftarrow D_{\sigma_s}$.
- $\mathsf{KAHE.Enc}(x, \mathbf{k}) = \mathsf{a} \cdot \mathbf{k} + t_1 \cdot e + x \in R_{q_1}$: The encryption algorithm samples an error term $e \leftarrow D_{\sigma_e}$, and returns a ciphertext $\mathbf{c} = \mathsf{a} \cdot \mathbf{k} + t \cdot e + x$.

- $\mathsf{KAHE.Dec}(\mathbf{c}, \mathbf{k}) = (\mathbf{c} - \mathsf{a} \cdot \mathbf{k}) \bmod t_1$: The decryption algorithm computes $m - \mathsf{a} \cdot \mathbf{k}$ and then reduce modulo t_1 to remove error.

Note that, to support long messages that span multiple polynomials in the plaintext space, we can naturally extend this scheme using multiple public random polynomials $\mathsf{a}_1, \mathsf{a}_2, \ldots$

The following lemma shows that, by using Gaussian secrets and errors whose widths are slightly larger than those used in standalone schemes, our RLWE-based KAHE scheme satisfies the leakage resilient security of Definition 5.

Lemma 1. *For any $n, \sigma > 0$, assume $\mathsf{RLWE}_{N_1,q_1,D_\sigma,D_\sigma}$ is κ-bit hard for up to n samples. Let $\sigma_s = \sqrt{2}\sigma$ and $\sigma_e = 2\sigma$, and let D_{σ_s} and D_{σ_e} be the secret and error distributions of the RLWE-based KAHE scheme. Then the RLWE-based KAHE scheme is n-semantic secure under leakage of sum of secret keys as in Definition 5.*

Our security proof relies on the following Hint-RLWE assumption, which is known to be equivalent to standard RLWE assumption [45].

Definition 6 (HintRLWE)**.** *Let N be a power of two, and let $m > 0$ be an integer. Let R be a cyclotimic ring of degree N, and let R_q be its residue ring modulo $q > 0$. The* Hint-RLWE *problem is to e ciently distinguish the following two distributions:*

$$\{(\mathbf{a}, \mathbf{a} \cdot s + \mathbf{e}, s + r, \mathbf{e} + \mathbf{f}) : \mathbf{a} \leftarrow R_q^m, s, r \leftarrow \chi_s, \mathbf{e}, \mathbf{f} \leftarrow \chi_e^m\},$$

and

$$\{(\mathbf{a}, \mathbf{u}, s + r, \mathbf{e} + \mathbf{f}) : \mathbf{a} \leftarrow R_q^m, \mathbf{u} \leftarrow R_q^m, s, r \leftarrow \chi_s, \mathbf{e}, \mathbf{f} \leftarrow \chi_e^m\}$$

Lemma 2 (HintRLWE Hardness [45, Theorem 1])**.** *For $\sigma > 0$, let $\sigma_1 = 2\sigma$. If $\sigma \geq \sqrt{2} \cdot \eta_\epsilon(\mathbb{Z}^N)$, then there exists an e cient reduction from RLWE over R_q with noise distribution D_σ to HintRLWE over R_q with $\chi_s = \chi_e = D_{\sigma_1}$.*

Now we are ready to prove Lemma 1.

Proof of Lemma 1. Fix input $\{x_i\}_{i=1}^n$. Assume HintRLWE with secret and error distributions χ_s and χ_e is hard in R_q such that it is as hard as RLWE (c.f. Lemma 2). We want to prove that the real distribution

$$D_0 = \left\{ (a, z = \sum_{i=1}^n k_i, w = \sum_{i=1}^n (e_i + f_i), y_1, \ldots, y_n) \;\middle|\; \begin{array}{r} a \leftarrow R_q; \\ \forall i \in [n]. k_i \leftarrow \chi_s, e_i, f_i \leftarrow \chi_e; \\ y_i = a \cdot k_i + e_i + f_i + \Delta \cdot x_i \end{array} \right\},$$

is indistinguishable from the simulated distribution

$$D_1 = \left\{ \begin{array}{l} (a, z = \sum_{i=1}^n k_i, w = \sum_{i=1}^n (e_i + f_i), u_1, \ldots, u_{n-1}, \\ \quad a \cdot z + w - \sum_{i=1}^{n-1} u_i + \Delta \cdot \sum_{i=1}^n x_i) \end{array} \;\middle|\; \begin{array}{r} a \leftarrow R_q; \\ \forall i \in [n]. u_i \leftarrow R_q, e_i, f_i \leftarrow \chi_e \end{array} \right\}.$$

We prove $D_0 \approx D_1$ using the following hybrids. When not explicitly defined, we will follow the convention that k_i's are sampled from χ_s, e_i's and f_i's are sampled from χ_e, and a and u_i's are uniformly sampled from R_q, all independently. Differences between consecutive hybrids are highlighted.

- $\mathsf{Hyb}^{(0)} = (a, z = \sum_{i=1}^n k_i, w = \sum_{i=1}^n (e_i + f_i), y_1 = a \cdot k_1 + e_1 + f_1 + \Delta \cdot x_1, \ldots, y_n = a \cdot k_n + e_n + f_n + \Delta \cdot x_n)$. This is exactly D_0.
- As a warm up, we first switch y_1 to an uniform element, and adjust y_n such that the sum $\sum_{i=1}^n y_i$ is consistent:

$$\mathsf{Hyb}^{(1)} = \left(\begin{array}{l} a, z = \sum_{i=1}^n k_i, w = \sum_{i=1}^n (e_i + f_i), y_1 = u_1, y_2 = a \cdot k_2 + e_2 + f_2 + \Delta \cdot x_2, \ldots, \\ \quad y_n = a \cdot k_n + e_n + f_n - u_1 + a \cdot k_1 + e_1 + f_1 + \Delta \cdot (x_n + x_1) \end{array} \right).$$

First, notice that u_1 is uniformly random over R_q. So, we can move $f_1 + \Delta \cdot x_1$ to y_1, and thus $\mathsf{Hyb}^{(1)}$ is equivalent to the following distribution:

$$\left(\begin{array}{l} a, z = \sum_{i=1}^{n} k_i, w = \sum_{i=1}^{n} (e_i + f_i), y_1 = u_1 + f_1 + \Delta \cdot x_1, \\ y_2 = a \cdot k_2 + e_2 + f_2 + \Delta \cdot x_2, \ldots, y_n = a \cdot k_n + e_n + f_n - u_1 + a \cdot k_1 + e_1 + \Delta \cdot x_n \end{array} \right). \quad (4)$$

Assume $\mathcal{A}$ is a distinguisher of $\mathsf{Hyb}^{(0)}$ and $\mathsf{Hyb}^{(1)}$. We build an adversary $\mathcal{B}$ for the $\mathsf{HintRLWE}$ problem to distinguish between $(a, b = a \cdot k + e, k + r, e + f)$ and $(a, b = u, k + r, e + f)$.

$\mathcal{B}(a, b, \rho = k + r, \epsilon = e + f)$:
sample $k_2, \ldots, k_{n-1} \leftarrow \chi_s$; $e_2, \ldots, e_n, f_1, \ldots, f_{n-1} \leftarrow \chi_e$;
let $z = \sum_{i=2}^{n-1} k_i + \rho$;
$w = \epsilon + \sum_{i=2}^{n} e_i + \sum_{i=1}^{n-1} f_i$;
$y_1 = b + f_1 + \Delta \cdot x_1$;
$y_i = a \cdot k_i + e_i + f_i + \Delta \cdot x_i$ for $i = 2 \ldots n-1$;
$y_n = a \cdot \rho + \epsilon + e_n - b + \Delta \cdot x_n$;
return $\mathcal{A}(a, z, w, y_1, \ldots, , y_n)$.

Conceptually, k is used as k_1, r is used as k_n, e is used as e_1, and f is used as f_n. More specifically, if $\mathcal{B}$ is given a HintRLWE sample with $b = a \cdot k + e$, then

$$y_1 = a \cdot k + e + f_1 + \Delta \cdot x_1, y_n = a \cdot r + e_n + f + \Delta \cdot x_n.$$

So $\mathcal{A}$ is given exactly $\mathsf{Hyb}^{(0)}$.

If $\mathcal{B}$ is given a uniform $b = u$, then

$$y_1 = u + f_1 + \Delta \cdot x_1, y_n = a \cdot (k + r) + e + f + e_n - u + \Delta \cdot x_n.$$

So $\mathcal{A}$ is given a sample of Eq 4 which is equivalent to $\mathsf{Hyb}^{(1)}$.

- We can generalize $\mathsf{Hyb}^{(1)}$ to all $1 \le j \le n$: In $\mathsf{Hyb}^{(j)}$, we set $y_1, \ldots, y_j$ to be uniform elements, and adjust y_n such that the sum $\sum_{i=1}^{n} y_i = a \cdot z + w + \sum_{i=1}^{n} x_i$:

$$\mathsf{Hyb}^{(j)} = \left(\begin{array}{l} a, z = \sum_{i=1}^{n} k_i, w = \sum_{i=1}^{n} (e_i + f_i), y_1 = u_1, \ldots, y_{j-1} = u_{j-1}, \\ y_j = u_j, y_{j+1} = a \cdot k_{j+1} + e_{j+1} + f_{j+1} + \Delta \cdot x_{j+1}, \ldots, \\ y_n = a \cdot (k_n + \sum_{i=1}^{j} k_i) + e_n + f_n + \sum_{i=1}^{j} (e_i + f_i) - \sum_{i=1}^{j} u_i + \Delta \cdot (x_n + \sum_{i=1}^{j} x_i) \end{array} \right).$$

This is equivalent to (by moving $f_j + \Delta \cdot x_j$ from y_n to y_j)

$$\left(\begin{array}{l} a, z = \sum_{i=1}^{n} k_i, w = \sum_{i=1}^{n} (e_i + f_i), y_1 = u_1, \ldots, y_{j-1} = u_{j-1}, \\ y_j = u_j + f_j + \Delta \cdot x_j, y_{j+1} = a \cdot k_{j+1} + e_{j+1} + f_{j+1} + \Delta \cdot x_{j+1}, \ldots, \\ y_n = a \cdot (k_n + \sum_{i=1}^{j} k_i) + e_n + f_n + \sum_{i=1}^{j} e_i + \sum_{i=1}^{j-1} f_i - \sum_{i=1}^{j} u_i + \Delta \cdot (x_n + \sum_{i=1}^{j-1} x_i) \end{array} \right). \quad (5)$$

For completeness, the previous hybrid is

$$\mathsf{Hyb}^{(j-1)} = \left(\begin{array}{l} a, z = \sum_{i=1}^{n} k_i, w = \sum_{i=1}^{n} (e_i + f_i), y_1 = u_1, \ldots, y_{j-1} = u_{j-1}, \\ y_j = a \cdot k_j + e_j + f_j + \Delta \cdot x_j, \ldots, \\ y_n = a \cdot (k_n + \sum_{i=1}^{j-1} k_i) + e_n + f_n + \sum_{i=1}^{j-1} (e_i + f_i) - \sum_{i=1}^{j-1} u_i + \Delta \cdot (x_n + \sum_{i=1}^{j-1} x_i) \end{array} \right).$$

Now, we build an adversary $\mathcal{B}$ for the $\mathsf{HintRLWE}$ problem given $\mathcal{A}$ a distinguisher of $\mathsf{Hyb}^{(j-1)}$ and $\mathsf{Hyb}^{(j)}$.

$\mathcal{B}(a, b, \rho = k + r, \epsilon = e + f)$: // Use k as k_j, r as k_n, e as e_j, f as f_n
sample $u_1, \ldots, u_{j-1} \leftarrow R_q$;
sample $k_1, \ldots, k_{j-1}, k_{j+1}, \ldots, k_{n-1} \leftarrow \chi_s$;
sample $e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_n, f_1, \ldots, f_{n-1} \leftarrow \chi_e$;
let $z = \sum_{i=1}^{j-1} k_i + \sum_{i=j+1}^{n-1} k_i + \rho$;
$w = \epsilon + \sum_{i=1}^{j-1} e_i + \sum_{i=j+1}^{n} e_i + \sum_{i=1}^{n-1} f_i$;
$y_1 = u_1, \ldots, y_{j-1} = u_{j-1}$;
$y_j = b + f_j + \Delta \cdot x_j$;
$y_i = a \cdot k_i + e_i + f_i + \Delta \cdot x_i$ for $i = j + 1 \ldots n - 1$;
$y_n = a \cdot (\rho + \sum_{i=1}^{j-1} k_i) + \epsilon + e_n + \sum_{i=1}^{j-1}(e_i + f_i) - b - \sum_{i=1}^{j-1} u_i + \Delta \cdot (x_n + \sum_{i=1}^{j-1} x_i)$;
return $\mathcal{A}(a, z, w, y_1, \ldots, , y_n)$.

If $b = a \cdot k + e$ and $\mathcal{B}$ is given a $\mathsf{HintRLWE}$ sample, then

$$y_j = a \cdot k + e + f_j + \Delta \cdot x_j,$$
$$y_n = a \cdot (r + \sum_{i=1}^{j-1} k_i) + f + e_n + \sum_{i=1}^{j-1}(e_i + f_i) - \sum_{i=1}^{j-1} u_i + \Delta \cdot (x_n + \sum_{i=1}^{j-1} x_i).$$

So this is as in $\mathsf{Hyb}^{(j-1)}$.

If $b = u$ is uniform, then

$$y_j = u + f_j + \Delta \cdot x_j,$$
$$y_n = a \cdot (k + r + \sum_{i=1}^{j-1} k_i) + e + f + e_n + \sum_{i=1}^{j-1}(e_i + f_i) - u - \sum_{i=1}^{j-1} u_i + \Delta \cdot (x_n + \sum_{i=1}^{j-1} x_i).$$

This is as in $\mathsf{Hyb}^{(j)}$. So $\mathsf{Hyb}^{(j-1)}$ and $\mathsf{Hyb}^{(j)}$ are indistinguishable assuming $\mathsf{HintRLWE}$.

Finally, notice that

$$\mathsf{Hyb}^{(n-1)} = \left(\begin{array}{l} a, z = \sum_{i=1}^{n} k_i, w = \sum_{i=1}^{n}(e_i + f_i), y_1 = u_1, \ldots, \\ y_{n-1} = u_{n-1}, \\ y_n = a \cdot \sum_{i=1}^{n} k_i + \sum_{i=1}^{n}(e_i + f_i) - \sum_{i=1}^{n-1} u_i + \Delta \cdot (\sum_{i=1}^{n} x_i) \end{array} \right)$$
$$= D_1.$$

So, we now conclude that D_0 is indistinguishable from D_1. □

D Verifiable Additive Homomorphic Encryption

In this section we discuss details of our AHE instantiation, and present and prove its necessary properties.

Verifiability. We require the AHE scheme to be *verifiable*, meaning that it has publicly verifiable public key shares, fresh ciphertexts, and partial decryptions, through zero-knowledge proof of knowledge (ZKPoK). More specifically, we assume there exists a ZKPoK system $\Pi_R = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ for a relation R, where Gen generates a common reference string, and $\mathsf{Prove}(x, w)$ and $\mathsf{Verify}(x)$ are interactive PPT algorithms on statement x and witness w. We require the usual properties from this ZK system, which we define in Appendix D.

We write $(tr, b) \leftarrow \langle \mathsf{Prove}(x, w), \mathsf{Verify}(x) \rangle$ to denote the interaction of Prove and Verify that produces the transcript tr and a decision bit b.

As usual, we require Π_R to satisfy the following properties:

- Perfect completeness: For all $(x, w) \in R$, we should have with probability 1 that $\langle \mathsf{Prove}(x, w), \mathsf{Verify}(x) \rangle = 1$.
- Knowledge soundness with negligible knowledge error ε: For any PPT $\mathcal{P}^*$, there exists a PPT extractor $\mathsf{Extract}$ with oracle access to $\mathcal{P}^*$ such that, for all statement x, if $\mathcal{P}^*$ convinces the verifier on x with probability $\epsilon > \varepsilon$, then $\mathcal{E}$ runs in expected polynomial time and outputs w with probability at least $(\epsilon - \varepsilon)/\mathsf{poly}(\lambda)$ such that $(x, w) \in R$.
- Public coin: All verifier messages sent to the prover are chosen uniformly at random and independent of prover's messages.
- Special honest-verifier zero-knowledge: There exists a PPT simulator Sim such that for all adversaries $\mathcal{A}_1, \mathcal{A}_2$,

$$\begin{aligned} & \Pr\{(x, w) \in R \wedge \mathcal{A}_1(tr) = 1 \mid (x, w, \rho) \leftarrow \mathcal{A}_2(1^\lambda), tr \leftarrow \langle \mathsf{Prove}(x, w), \mathsf{Verify}(x; \rho) \rangle\} \\ = \quad & \Pr\{(x, w) \in R \wedge \mathcal{A}_1(tr) = 1 \mid (x, w, \rho) \leftarrow \mathcal{A}_2(1^\lambda), tr \leftarrow \mathsf{Sim}(x, \rho)\}, \end{aligned}$$

  where ρ is the public randomness used by the verifier.

We further assume that Π_R is non-interactive via Fiat-Shamir transformation, and we write $\pi \leftarrow \mathsf{Prove}(x, w; \mathsf{aux})$ for the proof generated by the non-interactive prover, where aux is an auxiliary input string that is used to bound the proof for Fiat-Shamir, and correspondingly we write $b \leftarrow \mathsf{Verify}(x, \pi; \mathsf{aux})$ where b is either 1 (accept) or 0 (reject). For efficiency, the particular ZKPoK we use can also be verified given only a commitment c_x to the statement x, and by abusing of notations, we also call this verification algorithm as $\mathsf{Verify}(c_x, \pi)$[2].

D.1 RLWE-based verifiable AHE scheme.

In this section we proof the required properties from our scheme introduced in Section 4.2. For clarity, we reproduce its description here.

We instantiate an AHE scheme using the standard RLWE assumption, and we augment it with ZKPoK systems Π_{KeyGen}, Π_{Enc}, and $\Pi_{\mathsf{PartialDec}}$. Let $N_2 > 0$ be a power of two, and let $R_{q_2} = \mathbb{Z}[X]/(q_2, X^{N_2} + 1)$ for an integer modulus $q_2 > 0$. Let $t_2 > 0$ be an integer such that the plaintext space is $\mathbb{Z}[X]/(t_2, X^{N_2} + 1) \equiv \mathbb{Z}_{t_2}^{N_2}$, and let $\Delta = \lfloor q_2/t_2 \rceil$ be a scaling factor. Let $\chi_s, \chi_e, \chi_{\mathsf{flood}}$ be distributions over R_{q_2}. Our AHE consists of the following algorithms:

- $\mathsf{AHE.Setup}() = (\mathsf{u}, \mathsf{zk}_{\mathsf{params}})$: Samples $\mathsf{u} \leftarrow R_{q_2}$, and generate the public parameters $\mathsf{zk}_{\mathsf{params}}$ of the ZK proof systems; they are implicit in the following algorithms.
- $\mathsf{AHE.VerifiableKeyGen}((r_1, r_2)) = (\mathsf{sk}_j, \mathsf{pk}_j, \pi_{\mathsf{KeyGen}})$: Samples $\mathsf{sk}_j \leftarrow \chi_s(r_1)$ and $e_j \leftarrow \chi_e(r_2)$, sets $\mathsf{pk}_j = -\mathsf{u} \cdot \mathsf{sk}_j + e_j$, and generates a ZK proof $\pi_{\mathsf{KeyGen}} \leftarrow \mathsf{Prove}(\mathsf{pk}_j, (\mathsf{sk}_j, e_j))$ for the following relation

$$R_{\mathsf{KeyGen}} = \{(\mathsf{pk}_j, \mathsf{sk}_j, e_j) \mid \mathsf{pk}_j = -\mathsf{u} \cdot \mathsf{sk}_j + e_j \bmod q_2, \|\mathsf{sk}_j\| \le B_s, \|e_j\| \le B_e\}, \tag{6}$$

  where B_s and B_e are parameters to be determined based on χ_s, χ_e and on the proof system.
- $\mathsf{AHE.KeyGenVerify}(\pi_{\mathsf{KeyGen}}, \mathsf{pk}_j)$: Returns true if π_{KeyGen} verifies relation R_{KeyGen} over pk_j.
- $\mathsf{AHE.KeyAgg}(\{\mathsf{pk}_j\}_{j=1}^m)$: Returns the public key $\mathsf{pk} = \sum_{j=1}^m \mathsf{pk}_j \in R_{q_2}$.
- $\mathsf{AHE.VerifiableEnc}(x, \mathsf{pk}; r_1, r_2, \tau) = (\mathsf{ct}^0, \mathsf{ct}^1, \pi_{\mathsf{Enc}})$: Samples $v \leftarrow \chi_s(r_1)$ and $e^0, e^1 \leftarrow \chi_e(r_2)$, and computes $\mathsf{ct}^0 = \mathsf{pk} \cdot v + e^0 + \Delta \cdot x \in R_{q_2}$ and $\mathsf{ct}^1 = \mathsf{u} \cdot v + e^1 \in R_{q_2}$. Also generates a ZK proof π_{Enc} for the following relation

$$R_{\mathsf{Enc},\tau} = \{(\mathsf{ct}^1, v, e^1, \tau) \mid \mathsf{ct}^1 = \mathsf{u} \cdot v + e^1 \bmod q_2, \|v\| \le B_s, \|e^1\| \le B_e\}, \tag{7}$$

  where B_s and B_e are parameters as in $\mathsf{VerifiableKeyGen}$, and τ is a random string used as a nonce. Note that τ is not to be hidden in π_{Enc}, and in fact the proof will include τ verbatimly such that the verifier can access it via $\pi_{\mathsf{Enc}}.\mathsf{nonce}$.

[2] Technically we consider a commit-and-prove system, where the relation is $R^{\mathsf{com}} = \{((x, c), (w, r)) \mid (x, w) \in R, c = \mathsf{commit}(x; r)\}$.

- $\mathsf{AHE.EncVerify}(\pi_{\mathsf{Enc}}, \mathsf{ct})$: Returns true if π_{Enc} verifies relation R_{Enc} over ct^1.
- $\mathsf{AHE.PartialDec}(\mathsf{ct}^1, \mathsf{sk}_j) = \mathsf{ct}^1 \cdot \mathsf{sk}_j + e_{\mathsf{flood}}$: To partially decrypt a ciphertext using its component ct^1, this algorithm samples $e_{\mathsf{flood}} \leftarrow \chi_{\mathsf{flood}}$, and returns $\mathsf{pd}_j = \mathsf{ct}^1 \cdot \mathsf{sk}_j + e_{\mathsf{flood}} \in R_{q_2}$.
- $\mathsf{AHE.VerifiablePartialDec}(\mathsf{ct}^1, \mathsf{sk}_j) = (\mathsf{pd}_j, \pi_{\mathsf{PartialDec}})$: Runs $\mathsf{PartialDec}(\mathsf{ct}^1, \mathsf{sk}_j)$ to obtain a partial decryption pd_j, and generates a ZK proof $\pi_{\mathsf{PartialDec}}$ for the following relation

$$R_{\mathsf{PartialDec}} = \{(\mathsf{pd}, \mathsf{ct}^1, \mathsf{sk}, e_{\mathsf{flood}} \mid \mathsf{pd} = \mathsf{ct}^1 \cdot \mathsf{sk} + e_{\mathsf{flood}} \bmod q_2, \|e_{\mathsf{flood}}\| \leq B_{\mathsf{flood}}\}, \tag{8}$$

  where B_{flood} is a parameter to be determined based on χ_{flood} and the proof system.
- $\mathsf{AHE.PartialDecVerify}(\mathsf{pd}_j, \pi_{\mathsf{PartialDec}})$: Returns true if $\pi_{\mathsf{PartialDec}}$ verifies relation $R_{\mathsf{PartialDec}}$ over pd_j.
- $\mathsf{AHE.Recover}((\mathsf{ct}^0, \mathsf{ct}^1), \{\mathsf{pd}_j\}_{j=1}^m) = x$: Returns $\left\lfloor (\mathsf{ct}^0 + \sum_{j=1}^m \mathsf{pd}_j)/\Delta \right\rceil$.

We now consider security properties of our verifiable AHE scheme described above. First, it is easy to see that individual public key shares are independent and pseudorandom.

Lemma 3. *Assume* $\mathsf{RLWE}_{N_2,q_2,\chi_s,\chi_e}$ *is hard. Then, for any* $m > 0$*, the distribution* $\{(\mathsf{pk}_j)_{j=1}^m \mid (\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow \mathsf{AHE.VerifiableKeyGen}()\}$ *is pseudorandom.*

Next we show that, by assuming standard hardness results about RLWE , we achieve semantic security when considering $(\mathsf{VerifiableKeyGen}, \mathsf{VerifiableEnc})$ as a threshold PKE scheme.

Lemma 4. *For any* $m, n > 0$*, for any subset* $\mathcal{H} \subseteq [m]$*, let* $\mathcal{E}$ *be our verifiable AHE scheme based on* $\mathsf{RLWE}_{N_2,q_2,\chi_s,\chi_e}$ *with secret and error distributions* χ_s *and* χ_e*. Assume* $\mathsf{RLWE}_{N_2,q_2,\chi_s,\chi_e}$ *is* κ*-bit hard. Then* $(\mathcal{E}.\mathsf{Setup}, \mathcal{E}.\mathsf{VerifiableKeyGen}, \mathcal{E}.\mathsf{VerifiableEnc})$ *is* κ*-bit IND-CPA secure.*

Our next lemma establishes that, when the flooding noise parameters are chosen properly, given valid ciphertexts $\{\mathsf{ct}_i\}_{i=1}^k$, the partial decryptions on the sum of ct_i's do not leak secret information of honest decryptors against an active adversary.

Lemma 5. *For any* $m, n > 0$*, for any subset* $\mathcal{H} \subseteq [m]$*, let* $\mathcal{E}$ *be our verifiable AHE scheme based on* $\mathsf{RLWE}_{N_2,q_2,\chi_s,\chi_e}$ *with secret and error distributions* χ_s *and* χ_e*. Let* B_s *and* B_e *be the verifiable bounds in* Π_{Enc}*, and let* b_s *and* b_e *be high probability bounds on* $\|\mathsf{sk}_j\|$ *and* $\|e_j\|$ *for* $\mathsf{sk}_j \leftarrow \chi_s$ *and* $e_j \leftarrow \chi_e$*, respectively. Let* $\kappa, \lambda > 0$ *be the bit computational and statistical security parameters, and let* $\sigma_{\mathsf{flood}} = \sqrt{24N_2}2^{\lambda/2} \cdot k(B_e b_s + B_s b_e)$*. Assume* $\mathsf{RLWE}_{N_2,q_2,\chi_s,\chi_e}$ *is* κ*-bit hard. For any e cient adversary* $\mathcal{A}$ *who outputs ciphertext-proof pairs* $(\mathsf{ct}_1, \pi_1), \ldots, (\mathsf{ct}_k, \pi_k)$ *such that* $\mathsf{AHE.EncVerify}(\pi_i, \mathsf{ct}_i) = 1$ *and* $\mathsf{Extract}^{\mathcal{A}}(\mathsf{ct}_i) = (v_i, e_i)$ *for all* $1 \leq i \leq k$*, there exists an e cient simulator* Sim *such that, for any* $0 < k \leq n$ *and , the following distribution*

$$\left\{ \begin{array}{l} (\{\mathsf{sk}_j\}_{j\in\mathcal{H}}, \mathsf{ct}, \{\mathsf{pk}_j, \mathsf{PartialDec}(\mathsf{ct}, \mathsf{sk}_j)\}_{j\in\mathcal{H}}) : \\ \forall j \in \mathcal{H}. r_j \leftarrow \{0,1\}^\kappa, (\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow \mathsf{KeyGen}(r_j), \mathsf{ct} = \sum_{i=1}^k \mathsf{ct}_i \end{array} \right\}$$

is (κ, λ)*-bit indistinguishable from*

$$\left\{ \begin{array}{l} (\{\mathsf{sk}_j\}_{j\in\mathcal{H}}, \mathsf{ct}, \mathsf{Sim}(\sum_{i=1}^k v_i, \sum_{i\in\mathcal{H}} \mathsf{pk}_j) : \\ \forall j \in \mathcal{H}. r_j \leftarrow \{0,1\}^\kappa, (\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow \mathsf{KeyGen}(r_j), \mathsf{ct} = \sum_{i=1}^k \mathsf{ct}_i \end{array} \right\}.$$

Practical considerations. In KAHE we use small Gaussian secrets and errors that are secure according to Lemma 1. As a result we can use small parameters to instantiate AHE for aggregating the KAHE secret keys.

The native plaintexts in our KAHE scheme are polynomials of degree in the range of 2^{10} to 2^{14}, with a coefficient modulus t_1 up to 400-bit. To achieve close to optimal ciphertext expansion, we pack multiple entries of the input vectors $\mathbf{x}$ on a polynomial coefficient, i.e., encoding $\mathbf{x} \in [t]^\ell$ as $\mathbf{Gx} \in [t_1]^L$, where $\mathbf{G} = \mathbf{I} \otimes (1, B, B^2, \ldots)^T$ for $B = nt$ and $L = \lceil \frac{\ell}{\lceil \log_B t_1 - 1 \rceil} \rceil$, to fit the sum of n input vectors. When L is not a multiple of N_1 and hence $\mathbf{Gx}$ does not fully occupy all coefficients of plaintext polynomials, we simply drop the unused ciphertext coefficients. Note that such truncation still permits decryption as we only use additive homomorphism. On the other hand, the public parameters in both schemes are uniformly random polynomials and can be transmitted to the clients using PRG seeds.

E Zero-Knowledge Proof of Knowledge (ZKPoK)

Next we discuss how we instantiate the three ZKPoK constructions needed by our verifiable AHE scheme in Section 4.2.

Succinct Proof of RLWE-based AHE via linear constraints. In the context of our RLWE-based instantiation of verifiable AHE (Section 4.2), the three required ZKPoKs boil down to proving inner product constraints on vectors corresponding to polynomial coefficients. This approach has been used in previous works [8, 21] to achieve concrete efficiency by combining lattice-base cryptography for encryption with DL-based zero-knowledge for succintness. We also follow this approach.

Recall that we use a power-of-two cyclotomic ring $R = \mathbb{Z}[X]/(X^N + 1)$ in our RLWE-based AHE scheme. For any polynomial $a(X) \in R$ with coefficient vector $\mathbf{a} = (a_0, \ldots, a_{N-1})$, the *negacyclic matrix* representation of a is

$$\mathsf{negacyclic}(a) = \begin{pmatrix} a_0 & -a_{N-1} & \cdots & -a_1 \\ a_1 & a_0 & \cdots & -a_2 \\ \vdots & \vdots & \cdots & \vdots \\ a_{N-1} & a_{N-2} & \cdots & a_0 \end{pmatrix} \in \mathbb{Z}^{N \times N}.$$

For any $a, b \in R$ with coefficient vectors $\mathbf{a}$ and $\mathbf{b}$, $\mathsf{negacyclic}(a) \cdot \mathbf{b}$ is the coefficient vector of the polynomial product $a \cdot b \in R$. So, we can rewrite an RLWE sample $(a, a \cdot s + e) \in R_q^2$ as $(\mathbf{A}, \mathbf{As} + \mathbf{e})$, where $\mathbf{A} = \mathsf{negacyclic}(a) \in \mathbb{Z}_q^{N \times N}$ and $\mathbf{s}, \mathbf{e} \in \mathbb{Z}_q^N$ are the coefficient vectors of s and e, respectively. This allows us to make claim about linear relations over polynomials by simply proving linear relations in $\mathbb{Z}_q^N$.

In our RLWE-based verifiable AHE scheme, both Proof of plaintext knowledge Π_{Enc} and Proof of key generation Π_{KeyGen} boil down to the following. Let $(a, a \cdot r + e) \in R_q^2$ be an RLWE sample. For public polynomials a, c, the prover shows knowledge of *small* polynomials r and e such that the relation $c = ar + e$ holds in R_q. By *small* here we mean that $\|\mathbf{r}\|_\infty$ and $\|\mathbf{e}\|_\infty$ are bounded by public parameters B_s and B_e, respectively. However, since $\mathbb{Z}_q$ is typically different from the underlying mathematical structure used to implement the proof system, we capture both Π_{KeyGen} and Π_{Enc} using the following relation over the naturals. Formally, both R_{KeyGen} and R_{Enc} are captured by the relation $R_{N,q,B_s,B_e,\mathbf{c},\mathbf{A}}$ modulo P, parameterized by public values $N, q, B_s, B_e, \mathbf{c}$ and $\mathbf{A}$. Let us remark that we do not restrict the norm used in the relation, and in practice we can use either l_∞ or l_2 norms.

Definition 7 (Knowledge of short secret)**.** *Let* $N, q, B_s, B_e > 0$ *be integers, and let* $\mathbf{A} := \mathsf{negacyclic}(a) \in \mathbb{Z}^{N \times N}$ *and* $\mathbf{c} \in \mathbb{Z}^N$ *be the randomness and ciphertext corresponding to an RLWE sample* $c := (a, a \cdot r + e) \in R_q^2$*. We define the* knowledge of short secret *relation as* $R_{N,q,B_s,B_e,\mathbf{c},\mathbf{A}} := \big\{(\mathbf{r}, \mathbf{e}, \mathbf{w}) \in \mathbb{Z}^{3N} \mid \mathbf{c} = \mathbf{Ar} + \mathbf{e} - \mathbf{w}q \wedge ||\mathbf{r}|| \leq B_s \wedge ||\mathbf{e}|| < B_e\big\}$.

In the following, we will instantiate our three ZKPoKs using the Bulletproofs framework [27] for the above relation that have size sub-linear in N. We stress, however, that there are many ways to implement a ZKPoK system for the above relation, and we leave the application-specific optimization of this step to future work.

Note that, while Definition 7 is stated over the naturals, Bulletproofs is defined over elliptic curve groups of a finite order P and natively supports inner product relations modulo P. So, instead of proving $R_{N,q,B_s,B_e,\mathbf{c},\mathbf{A}}$ directly, our ZKPoK proves the following three constraints over $\mathbb{Z}_P$:

1. $\mathbf{c} = \mathbf{Ar} + \mathbf{e} - \mathbf{w}q \pmod P$
2. $||\mathbf{r}|| \leq B_s \pmod P$ and $||\mathbf{e}|| \leq B_e \pmod P$
3. $||\mathbf{w}||_\infty < P/2 - (qNB_s + B_e) \pmod P$

Note that the third constraint ensures that the first and second constraints hold over the naturals even if we prove them modulo P, so long as $B_s, B_e \ll P/3$.

For Π_{KeyGen} and Π_{Enc}, the bounds B_s and B_e are used in our security proof to determine the size of flooding noise. So there is a tradeoff between the efficiency of AHE operations (having B_s and B_e as small

as possible to use small parameters) and the efficiency of our ZKPoK implementations (having loose B_s and B_e to possibly improve prover and verifier costs).

For $\Pi_{\mathsf{PartialDec}}$, in which case $\mathbf{e}$ has large coefficients as it corresponds to the flooding noise, we can, in the same way as Bell et al. [8], combine inner product Bulletproofs [27] and the optimized range proofs of Gentry et al. [21]. This is what we use for AHE.VerifiablePartialDec, and we refer the reader to [8, Section 5.3.2] for the details. More specifically, to convince the verifier that $\|\mathbf{e}\|_\infty \leq B_e$, it suffices to prove that $B_e \cdot \mathbf{1} - \mathbf{e}$ has nonnegative entries, which reduces to finding integer vectors α, β, γ such that

$$4(B_e \cdot \mathbf{1} - \mathbf{e}) \circ (B_e \cdot \mathbf{1} - \mathbf{e}) + 1 = \alpha \circ \alpha + \beta \circ \beta + \gamma \circ \gamma,$$

where α, β, γ exist due to the Lagrange Three Square theorem. Notice that the above equation must hold over the integers; thus the prover additionally applies an approximate proof to show that

$$\|(4\mathbf{e}|\alpha|\beta|\gamma)\|_\infty < \sqrt{P}/4,$$

which guarantees that the equation above does not wraparound modulo P and hence can be proven using Bulletproofs.

An improved construction for small error vectors. In Π_{KeyGen} and Π_{Enc}, the error term $\mathbf{e}$ has small coefficients, and hence we can do better than Bell et al. [8] by using approximate range proofs.

Lemma 6 (Approximate proof of small l_∞ norm ([21], Lemma 3.5))**.** *Let $\mathbf{x} \in \mathbb{F}^\ell$ be a vector, and let $b, \gamma \in \mathbb{N}$ such that $||\mathbf{x}||_\infty \leq b/\gamma$ with $\gamma > 2500\sqrt{\ell}$. There is a ZK proof system to show $||\mathbf{x}||_\infty \leq b$ where the prover sends (a) a ZK proof π of an inner product constraint of the form $\langle \mathbf{x}|\mathbf{y}, \mathbf{b}\rangle = c$, for public $\mathbf{b}, c$, of length $\ell + 128$ and (b) a vector $\mathbf{z} \in [b]^{128}$. The verifier (i) checks π and (ii) checks that $||\mathbf{z}||_\infty \leq b/2$.*

To get better efficiency of the proof systems Π_{KeyGen} and Π_{Enc}, we set the secret distribution χ_s to uniform ternary $\{0, \pm 1\}^N$, and the error distribution χ_e to centered binomial distribution with variance 8 which has support $[-16, 16]^N$. These distributions are among the standard choices for homomorphic encryption [], and they have small supports that imply tighter verifiable bounds B_s and B_e using the approximate l_∞ proof. More specifically, Lemma 6 allows us to prove $\|\mathbf{r}\|_\infty \leq 2500\sqrt{N}$ for $\mathbf{r} \leftarrow \chi_s$, and $\|\mathbf{e}\|_\infty \leq 16 \cdot 2500\sqrt{N}$ for $\mathbf{e} \leftarrow \chi_e$. Also, for all the applications we consider (with number of clients going up to a billion, and vector lengths up to 10 million) N is bounded by 2^{12}, and q is bounded by 2^{80} (see Table 2). Finally, recall that $\mathbf{r}$ is a ternary vector.

Recall that we need the multiplicative gap between these two quantities to be at least $2500\sqrt{N}$, as per Lemma 6. This is easy to verify for (3), given the concrete values of N, q and P discussed above. The following lemma states this reduction to inner product constraints, which then can be offloaded to Bulletproofs.

Lemma 7 (Proof of knowledge of small secret)**.** *Let $\mathbf{A} := \mathsf{negacyclic}(a) \in \mathbb{Z}_q^{N \times N}$ and $\mathbf{r}, \mathbf{e} \in \mathbb{Z}_q^N$ be the matrix and vectors corresponding to an RLWE sample $(a, ar + e) \in R_q^2$ such that $||\mathbf{r}||_\infty \leq 1$ and $||\mathbf{e}||_\infty \leq 16$. Let $N, q, B_s, B_e \in \mathbb{N}$ be such that $B_s > 2500\sqrt{N}$, $B_e > 40000\sqrt{N}$, and $qNB_s + B_e + 2500q\sqrt{N}(N+2) < P/2$. Then, there is a proof system for $R_{N,q,B_s,B_e,\mathbf{c},\mathbf{A}}$ with proof size $O(\log N)$ and prover and verifier costs of $O(N^2)$. Moreover, verification of k proofs can be batched.*

Proof. We first define an inner product constraint that holds iff (1) holds mod P, except for a small probability N/P. Let $\mathbf{M} = \mathsf{negacyclic}(a)$ with rows $\{\mathbf{M}_i\}_{i \in [N]}$. Note that (1) can be written as a conjunction of constraints $\bigwedge_{i \in [N]} \langle \mathbf{r}, \mathbf{M}_i\rangle + \langle \mathbf{e}, 0^{i-1}10^{N-i}\rangle + \langle \mathbf{w}, 0^{i-1}(-q)0^{N-i}\rangle = \langle \mathbf{c}, 0^{i-1}10^{N-i}\rangle$ which by taking random linear combinations as in Gentry et al. [21] can be encoded as a single linear constraint $\langle (\mathbf{r}|\mathbf{e}|\mathbf{w}|\mathbf{c}), \mathbf{b}\rangle = c$, with public $\mathbf{b}, c$. This reduction requires $O(N^2)$ scalar products. Note that $\mathbf{c}$ appears on the left-hand side here, which allows the proof to be verified with only a commitment to the ciphertext $\mathbf{c}$. This allows us to significantly reduce the communication overhead of the verifier role (see Section 5.4).

Next, we show that constraints (2,3) can be reduced to an inner product constraint. First note that the bounds B_s, B_e in (2) satisfies the condition of Lemma 6, as $2500\sqrt{N} < B_s$ and $40000\sqrt{N} = 2500 \cdot 16\sqrt{N} < B_e$, and therefore they can be proven using a constraint $\langle \mathbf{r}|\mathbf{e}|\mathbf{y}|\mathbf{y}', \mathbf{b}'\rangle = c'$, with public $\mathbf{b}', c'$, along with vectors

$\mathbf{z}$ with norm bounded by $B_s/2$ and $\mathbf{z}'$ with norm bounded by $B_e/2$. Finally, we can again apply Lemma 6 to handle (3), given that $24000qN^3 < P$. Note that $\|\mathbf{w}q\|_\infty \leq \|\mathbf{Ar}\|_\infty + \|\mathbf{e}\|_\infty$, and thus honest $\mathbf{w}$ should have norm bounded by $N+2$. By Lemma 6, the prover can prove $\|\mathbf{w}\|_\infty \leq 2500\sqrt{N}(N+2)$ using an inner product $\langle \mathbf{w}|\mathbf{y}'', \mathbf{b}''\rangle = c''$ of length 128. Then we have $\|\mathbf{w}\|_\infty \leq P/2 - (qNB_s + B_e)$ and so (3) holds.

We can then apply another linear combination with random challenges to merge all three above constraints into a final inner product constraint of the form $\langle (\mathbf{r}|\mathbf{e}|\mathbf{w}|\mathbf{y}|\mathbf{y}'|\mathbf{y}''|\mathbf{c}), \hat{\mathbf{b}}\rangle = \hat{c}$, with public $\hat{\mathbf{b}}, \hat{c}$. Then the proof consists on proving the validity of this one constraint of length $4N + 3 \cdot 128$, along with the fact that $||\mathbf{z}|| < B_s/2$ $||\mathbf{z}'|| < B_e/2$, and $||\mathbf{z}''|| < 2500\sqrt{N}(N+2)/2$ for vectors $\mathbf{z} \in [B_s]^{128}$, $\mathbf{z}' \in [B_e]^{128}$, and $\mathbf{z}'' \in [2500\sqrt{N}(N+2)]^{128}$, respectively. By offloading this proof to Bulletproofs we get the costs in the statement of the Lemma. □

Alternative construction for small error vectors. Instead of using approximate l_∞ norm proofs with ternary secret and binomial error distributions as mentioned above, we can sample secret and errors from a small Gaussian distribution and prove their smallness using approximate Euclidean norm proofs. This alternative approach has the advantage that the approximate bounds can be much tighter than in the approximate l_∞ norm proofs, especially when they are used to determine the size of polynomial products. At the same time, sampling from Gaussian distribution is slower than the uniform ternary and binomial distributions from the previous subsection.

We adopt the approximate Euclidean norm proof from [46]. Let $\mathbf{x} \leftarrow D_\sigma^N$ be a Gaussian vector with expected Euclidean norm $\sigma\sqrt{N}$, and the prover wants to convince the verifier $\|\mathbf{x}\|_2 \leq \gamma\sigma\sqrt{N}$ for some gap factor $\gamma \geq 1$. As in the approximate l_∞ norm proof, the challenge first chooses a random projection $\mathbf{R} \in \mathsf{Bin}_1^{256\times N}$ and sends R to the prover. The prover then commits to $\mathbf{v} = \mathbf{Rx} \in \mathbb{Z}_q^{256}$, and prove in zero-knowledge that $\mathbf{Rx} + \mathbf{y} = \mathbf{z}$ and $\|\mathbf{z}\|_2$ is small, where $\mathbf{y} \leftarrow D_{\sigma_y}^{256}$ is a random masking vector. By Lemma [47, Lemma 3.2.4], we have $\|\mathbf{Rx}+\mathbf{y}\|_2 \leq \sqrt{337}\cdot\|\mathbf{x}\|_2 + \|\mathbf{y}\|_2$ with probability at least $1-2^{-128}$. On the other hand, Lemma [47, Lemma 3.2.5] tells us that for any $\mathbf{x} \in [\pm P/2]^N$ and $\mathbf{y} \in [\pm P/2]^{256}$, we have $\|\mathbf{Rx}+\mathbf{y} \bmod P\| < \sqrt{26}/2 \cdot B_e$ with probability at most 2^{-128}. Thus, by setting $\sigma_y = \sqrt{337}\cdot\sigma$, the verifier can be convinced that $\|\mathbf{x}\|_2 \leq 2.8\sqrt{337/26}\cdot\sigma\sqrt{N}$ by checking $\|\mathbf{z}\|_2 \leq 1.4\cdot\sqrt{337}\sigma$. At the same time, $\mathbf{z}$ does not leak information about $\mathbf{x}$ due to Hint-RLWE assumption, and hence the proof is zero-knowledge. We summarize this approximate Euclidean norm proof in the following lemma.

Lemma 8 (Approximate Proof of Euclidean Norm [46])**.** *Let $\mathbf{x} \sim D_s^\ell$ be a vector of ℓ independent discrete Gaussian entries of parameter $\sigma > 0$, and let $\gamma = 2.8\sqrt{337/26}$. There exists a ZKPoK system for a prover to show $\|\mathbf{x}\|_2 \leq \gamma\sigma\sqrt{\ell}$, where the prover sends (a) a ZKPoK proof π of an inner product constraint of the form $\langle \mathbf{x}|\mathbf{y}, \mathbf{b}\rangle = c$, for public $\mathbf{b}, c$, of length $\ell + 256$, and (b) a vector $\mathbf{z} \in [\pm P]^{256}$. The verifier (i) checks π and (ii) checks that $\|vecz\|_2 \leq 1.4\cdot\sqrt{337}\sigma$.*

When adopting this approximate Euclidean norm proof to build a ZKPoK for $R_{N,q,B_s,B_e,\mathbf{c},\mathbf{A}}$, we first construct linear constraints $\bigwedge_{i\in[N]}\langle \mathbf{r}, \mathbf{M}_i\rangle + \langle \mathbf{e}, 0^{i-1}10^{N-i}\rangle + \langle \mathbf{w}, 0^{i-1}(-q)0^{N-i}\rangle = \langle \mathbf{c}, 0^{i-1}10^{N-i}\rangle$, in the same way as in the previous subsection. Then we construct 512 linear constraints of the form $\langle \mathbf{r}|\mathbf{e}|\mathbf{y}|\mathbf{y}', \mathbf{b}'\rangle = c'$ corresponding the randomized projection $\mathbf{Rr}+\mathbf{y} = \mathbf{z}$ and $\mathbf{R}'\mathbf{e}+\mathbf{y}' = \mathbf{z}'$, for binomial $\mathbf{R}, \mathbf{R}' \in \{0, \pm 1\}^{256\times N}$, discrete Gaussian $\mathbf{y}, \mathbf{y}' \leftarrow D_{\sigma_y}^{256}$, and public $\mathbf{b}', c'$. To show that $\mathbf{w}$ has bounded l_∞ norm, we follow the same approach as in the previous subsection, resulting in linear constraints of the form $\langle \mathbf{w}|\mathbf{y}'', \mathbf{b}''\rangle = c''$, for $\mathbf{y}'' \in [\pm P/2]^{128}$ and public $\mathbf{b}'', c''$. Finally, we can merge all these constraints into an inner product of length $3N + 640$ using random challenges. As before, the prover and the verifier running times are $O(N^2)$ due to merging N linear constraints, and the proof size is $O(\log N)$.

F Lower Bound in the Standard Model

An important motivation for our work is to enable a lightweight role in secure aggregation protocol that can be delegated to an honest majority committee or some other type of non-colluding entity. Since we're aggregating long length ℓ vectors, a minimum requirement for what it means to be lighweight is that the

committe should be doing work at least sublinear in ℓ. Next we show that this requirement is incompatible with one-shot clients in the standard model with simulation-based security. More generally, the lower bound applies to any protocol where the adversary may decide to drop clients, and generally speaking says that the amount of communication after the point were that decision can be made must be linear in l.

Note that the lower bound, presented in the following theorem, assumes that the gap between n and $\mathtt{min\_n}$ is large enough and, more precisely, that the number of ways in that the adversary can choose the set of clients included in the aggregation is exponential in a security parameter. This is not a hard requirement in practice: in the proof λ relates to a collision-resistant hash function. Thus, setting $\lambda = 256$, concrete parameters as small $n = 500$ and $\mathtt{min\_n} \leq 440$ satisfy the constraints, while in practice we expect n to be much larger than $\mathtt{min\_n}$.

Theorem 1. *Assume the existence of one way functions. Let λ be a security parameter, and let Π be a protocol implementing vector aggregation with $|\mathbb{F}| \geq 2$ and $\sum_{k=\mathtt{min\_n}}^{n} \binom{n}{k} > 2^{\lambda}$ (c.f. Functionality $\mathcal{F}^{\mathsf{Agg}}$, Fig.1). Suppose that after the last point at which an adversarial server can choose which clients to drop (e.g. after the last client contribution if clients' submissions are made independently) the server engages in $o(l)$ communication. Then there is no white-box (and thus no black-box) security proof in the standard model against an adversary actively corrupting the server S.*

Proof. To formally prove this claim, we show that, for any such protocol, there exists a family of real-world adversarys $\mathcal{A}_s^R$ (parametrized by $s \in \{0,1\}^{\lambda}$ and implicitly the security parameter λ) which can't all be successfully simulated by the same simulator. We will give a corresponding family of distinguishers (parametrised by s) and an honest party input distribution such that, for any white-box simulator Sim, if the parameter $s \in \{0,1\}^{\lambda}$ is chosen uniformly at random then with all but negligible probability the corresponding distinguisher will be able to correctly tell $\mathrm{IDEAL}_{\mathcal{F}^{\mathsf{Sim}}}\big((\mathbf{x_i})_i, \lambda\big)$ and $\mathrm{REAL}_{\Pi^{\mathcal{A}_s^R}}\big((\mathbf{x_i})_i, \lambda\big)$ apart. Here $\mathcal{F}^{\mathsf{Sim}}$ denotes the functionality of Figure 1 with Sim playing the role of the adversary $\mathcal{A}$.

Assume w.l.o.g. that all honest clients submit all expected messages in an execution of π, i.e., no client drops out. Let $g := n - \mathtt{min\_n}$ be the gap between the available number of clients and the required minimum. Let $G := \bigcup_{k=0}^{g} \binom{[n]}{n-k} = \bigcup_{k=\mathtt{min\_n}}^{n} \binom{[n]}{k}$ be the collection of all possible subsets of honest clients that the server could ignore whilst still getting an output, as per functionality $\mathcal{F}^{\mathsf{Agg}}$.

Let $\{H_s\}_{s\in\{0,1\}^{\lambda}}$ be a family of collision resistant hash functions with co-domain G, whose existence is guaranteed by the existence of one way functions and the fact that G has size exponential in λ. The adversary $\mathcal{A}_s^R$ will only use H_s and this is the only influence of s on $\mathcal{A}_s^R$.

$\mathcal{A}_s^R$ proceeds by running the server honestly and collecting its transcript/view. Let $\mathcal{V}_1$ be the view of the server until the last point it can choose to lie about who has dropped out without including input. $\mathcal{A}_s^R$ then computes a set $I \in G$ as $I := H_s(\mathcal{V})$. $\mathcal{A}_s^R$ then instructs the server to behave honestly, other than ignoring messages from clients in I, to produce a server view $\mathcal{V} := \mathcal{V}_1 || \mathcal{V}_2$, where $\mathcal{V}_1$ denotes the view up to the point of choosing I. Also note that the server's output, which we denote $\mathtt{output} \in \mathbb{F}^{\ell}$, can be obtained from $\mathcal{V}$. Finally, $\mathcal{A}_s^R$ outputs $\mathcal{V}$, i.e. its whole view. Note that since H_s is collision resistant, the set I of dropped out clients in $\mathcal{V}_{I,1}$ is uniquely determined by $\mathcal{V}$ or, in other words, I constitutes a commitment to $\mathcal{V}$, and therefore also $\mathcal{V}_{I,1}$.

Our input distribution is to simply give the ith client an independent uniformly random binary vector $\mathbf{x_i} \in \mathbb{F}^{\ell}$. Our distinguisher will merely check that (i) the $\mathcal{V}$ and choice of I in the view indeed satisfy $I = H_s(\mathcal{V})$ (ii) the server's output computed from $\mathcal{V}$ would match the sum of inputs of the clients in I.

The main idea of the proof is that when Sim sends S to T in step 2, S in fact constitutes a commitment to the first part of the view that Sim will output, which should match $\mathcal{V}_1$. Sim must then complete that view to match the view outputted by $\mathcal{A}_s^R$, and in particular the sum $\mathtt{output} = \sum_{i\in[n]\setminus I} \mathbf{x_i}$. Note that since the second part of the view is $o(\ell)$, it cannot encode the output as the output has $\Omega(\ell)$ entropy. Thus any Sim will fail at this task with constant probability. In turn, a distinguisher that checks consistency of the view (checks i, ii above) will succeed with constant probability and distinguishing Sim's output from $\mathcal{A}_s^R$'s.

More formally, assume a successful simulator Sim, and let S be the set sent by the Sim in step 2 in the ideal world. Note that $S \geq \mathtt{min\_n}$ must hold for Sim to get information about honest inputs. Let $\mathcal{V}^{\mathsf{Sim}}$ be

the output of the simulator, i.e., $\text{IDEAL}_{\mathcal{F}^{\mathsf{Sim}}}\big((\mathbf{x_i})_i, \lambda\big)$. Recall that whenever checks (i), (ii) do not pass, D claims that the view does not come from a real-world simulation, otherwise it claims real-world.

Note that it must be the case that, with overwhelming probability, that $S = I$ because otherwise the simulator doesn't have information about at least one client's input. If that is the case, the Sim fails at producing the right output because every client's input has enough entropy in it.

Moreover, with all but negligible probability Sim can find at most one $\mathcal{V}_1^{\mathsf{Sim}}$ that hashes to S due to the collision resistance property of H_s. Since S was chosen independent of inputs, even after conditioning on $\mathcal{V}_1^{\mathsf{Sim}}$, the output still has $\Omega(\ell)$ bits of entropy in it. Therefore, the server needs to receive $\Omega(\ell)$ entropy after seeing the client 's messages. This implies that $|\mathcal{V}_2^{\mathsf{Sim}}|$ must have ℓ bits of entropy in it, which means its size is $\Omega(\ell)$ (in expectation), a contradiction.

Note that the last part of the argument goes through so long as the subset of clients the adversary chooses to sum over (the clients not in I) has, with non-negligible probability over that choice, at least $\Omega(\ell)$ entropy in the sum of their inputs, i.e. the random variable $\sum_{i\in[n]\setminus I} \mathbf{x_i}$ has at least $\Omega(\ell)$ entropy. □

Let us remark that this result holds for the setting of semi-honest server colluding with an actively corrupted client, with a similar proof. The idea there is that while the adversary can't drop clients, the corrupted client can set its input in a way that depends on the adversary's view, in a way that is analogous to how the adversary chooses which clients to dropout in the above proof.

G Proof of Malicious Server Protocol

Proof of Theorem 2. We reiterate that this theorem is only dealing with security against an adversary with control of the server. The honest server case is covered by Theorem 3. Let $\mathcal{H}$, $\mathcal{C}$ be the sets of honest and corrupted clients, and let $\mathcal{H}_D$ and $\mathcal{C}_D$ be the sets of honest and corrupted decryptors. Without loss of generality, we assume that $1 \in \mathcal{H}$ (Note that if $\mathcal{H}$ were empty the result is trivial as no honest party has input or output). Assume $\mathcal{A}$ is any real world adversary to the protocol. We build the following simulator Sim in the (verifier, RO)-hybrid model with a programmable RO, i.e. the simulator provides an ideal verifier oracle and a RO to the adversary.

Key Generation phase. Sim interacts with $\mathcal{A}$ simulating all honest decryptors by following the protocol.

Data collection. Sim interacts with $\mathcal{A}$ running simulated honest clients normally, with input $\mathbf{0}$. $\mathcal{A}$ then interacts with the verifier V. By observing this interaction (as we are in a verifier-hybrid model) Sim learns the set of honest clients S included in the result. By submitting S to the trusted party in step 3 of the ideal functionality Sim learns the sum $s_{\mathcal{H}}$ of honest inputs included in the result. Then Sim reprograms the random oracle at the points used by one of the honest clients in S to make their input match $s_{\mathcal{H}}$. This is to ensure that the sum of honest clients in the simulation with $\mathcal{A}$ matches the actual sum.

Decryption. Sim interacts with $\mathcal{A}$ simulating all honest decryptors as prescribed by the protocol.

Output. Finally Sim answers any remaining random oracle queries from $\mathcal{A}$ and outputs whatever adversary outputs. This concludes the description of Sim.

It remains to show that the real world execution is indistinguishable from the ideal world. We first explain this intuitively and then give a formal hybrid argument after that.

The only difference between the real and ideal worlds is that in the real world the honest clients use x_i as input whereas in the ideal world they use 0 and the random oracle is reprogrammed at seed_1 to make the sums match. We now argue that the two worlds are indistinguishable.

Firstly, reprogramming an oracle can be a problem if the adversary is able to query it before and after reprogramming at the reprogrammed point. In our case we avoid this as the seed_1 is encrypted with $\mathsf{pk}^{\mathsf{aux}}$ and the corresponding secret key is shared out with threshold t which is greater than the number of malicious

decryptors and no honest decryptor will provide any information about those shares until the decryption phase.

Next, consider which messages in the view are changed by this change in input/RO. The directly changed messages are those m from the clients. However, those are encrypted by the semantically secure KAHE scheme which by Lemma 1 only leaks the sum of inputs (which is the same in both cases) when the sum of the corresponding keys is revealed.

Each of those keys is encrypted with $\mathsf{AHE.VerifiableEnc}()$. These messages on their own leak nothing as Lemma 4 establishes they are semantically secure. However, that still leaves the possibility that they might leak something in conjunction with other messages that depend on sk.

The messages with information about $\mathsf{sk} = \sum_j \mathsf{sk}_j$ are (i) the pk_j and π_j (ii) the shares of the sk_j that are sent out to other parties and some of which are shared with the server in step 11b (iii) the partial decryptions. We address each of these in turn.

Firstly, $\{\mathsf{pk}_j\}_{j\in\mathcal{H}_D}$ are pseudorandom and hence they can be released by Lemma 3 and the π_j are zero-knowledge so provide no extra information if client j were already assumed to be honest.

Secondly, we consider the shares of the sk_j that the $\mathcal{A}$ can recover. $\mathcal{A}$ controls at most $2t-m-1$ decryptors who will hand over at most $2t-m-1$ shares sent out by each of the honest decryptors (as that is all they are sent). Each of the honest decryptors will abort if told more than $m-t$ decryptors have dropped out, i.e. $|P| < t$, and thus will provide at most $m-t$ shares each. Thus the average number of shares originating from an honest party that $\mathcal{A}$ recovers is at most $2t-m-1+(m-t) = t-1$, therefore for at least one honest decryptors secret key contribution sk_j will not be recovered. This is enough to guarantee that even conditional on all the recovered shares the sum of honest contributions to the secret key is still uniformly distributed. The sum of the honest contributions to sk is equivalent to sk for the adversary as they were required to prove that they new the malicious sk_j using proofs checked by the verifier. Therefore these shares don't break the semantic security.

Finally, we consider the partial decryptions. Let v_i be the encryption randomness used to generate the AHE ciphertext in step 8 by client i, matching the notation in Lemma 5. That lemma tells us that the partial decryptions can be simulated using only the public keys and the sum of the v_i contributing to ct. Because for each malicious client i the adversary provided a proof of knowledge of v_i, these partial decryptions provide no more knowledge to the adversary than the sum of v_i over honest clients i included in ct. This allows the adversary to decrypt the sum of the honest client contributions, which of course was the same in the ideal world and real world so is fine, but not anything else as the (honest) v_i are all uniformly random so their sum reveals nothing about any other subset of them.

We now formalize the above argument in the following sequence of hybrids to show that the real world is indistinguishable from the ideal world.

- $\mathsf{Hyb}^{(0)}$: This is the real world execution. In particular, the adversary's view contains
  - $(m_j, s_j), \mathrm{sig}(ms_j)$ from all honest decryptors $j \in \mathcal{H}_D$;
  - $\{\mathsf{share}_{k,j}\}_k, \{\mathsf{share}^{\mathsf{aux}}_{k,j}\}_k$ from all honest decryptors $j \in \mathcal{H}_D$ which are shares of r_j and r_j^{aux}, respectively;
  - s_{ct^1} from the verifier;
  - pd_j from all honest decryptors $j \in \mathcal{H}_D$.

  The adversary's output (on behalf of corrupted parties) contains

  - $(\mathsf{D}, S_{\mathsf{D}}, \mathsf{pk}, \mathsf{pk}^{\mathsf{aux}})$ to all honest decryptors j;
  - $(\mathsf{ct}^1, \mathtt{proofs})$ to the verifier;
  - $(\mathsf{ct}^1, s_{\mathsf{ct}^1})$ to all honest decryptors j.

  In particular, we have that:

  - For all $j \in \mathcal{H}_D$, $\mathsf{pk}_j = -\mathsf{u}\cdot\mathsf{sk}_j + e_j$ for some $e_j \sim \chi_e$.
  - If the decryptor does not abort at Step 4, then for all $k \in \mathcal{C}_D$, we have $\mathsf{pk}_k = -\mathsf{u}\cdot\mathsf{sk}_k + e_k$ for some sk_k and e_k satisfying the bound B_s and B_e.

– If the decryptor does not abort at Step 7, then $\mathsf{ct}^1 = \sum_{i\in\mathcal{H}} \mathsf{ct}^1_i + \sum_{k\in\mathcal{C}} \mathsf{ct}^1_k$, and $\mathsf{ct}^1_k = v_k \cdot \mathsf{u} + e''_k$ for all $k \in \mathcal{C}$ with v_k and e''_k satisfying the bounds B_s and B_e. Let $v = \sum_{i\in\mathcal{H}} v_i + \sum_{k\in\mathcal{C}} v_k$ and $e'' = \sum_{i\in\mathcal{H}} e''_i + \sum_{k\in\mathcal{C}} e''_k$.

- $\mathsf{Hyb}^{(1)}$: If the honest decryptors abort in Step 7, then this is the same as $\mathsf{Hyb}^{(0)}$. Assume they do not abort in the following. Then we compute all honest decryptor j's partial decryption pd_j using pk_j and the term v used in ct^1. Specifically, for all $j \in \mathcal{H}$ we set $\mathsf{pd}_j = -v \cdot \mathsf{pk}_j + e'''_j$. We now argue that this is indistinguishable from $\mathsf{Hyb}^{(0)}$. Note that in $\mathsf{Hyb}^{(0)}$ we have

$$\begin{aligned}\mathsf{pd}_j &= \mathsf{sk}_j \cdot (\mathsf{u} \cdot v + e'') + e'''_j \\ &= v \cdot (\mathsf{sk}_j \cdot \mathsf{u} - e_j) + v \cdot e_j + \mathsf{sk}_j \cdot e'' + e'''_j \\ &= -v \cdot \mathsf{pk}_j + v \cdot e_j + \mathsf{sk}_j \cdot e'' + e'''_j.\end{aligned}$$

  By Lemma 5, pd_j in the above expression (as in $\mathsf{Hyb}^{(0)}$) is statistically close to $\mathsf{pd}_j = -v \cdot \mathsf{pk}_j + e'''_j$ as in $\mathsf{Hyb}^{(1)}$. So the two hybrids are indistinguishable.
- $\mathsf{Hyb}^{(2)}$: If the honest decryptors abort in Step 4, then this is the same as $\mathsf{Hyb}^{(1)}$. Assume they do not abort in the following. Then we compute pk_j for all $j \in \mathcal{H}_D$ without using individual sk_j's. Specifically, for all $j \in \mathcal{H}_D$ we replace pk_j with truly random element conditioned on their sum $\sum_{j\in\mathcal{H}_D} \mathsf{pk}_j$ being unchanged, and we simulate the proof π_j of jth public key share on pk_j. That is, in this hybrid we compute $\mathsf{pk}_{\mathcal{H}_D} = -\mathsf{u} \cdot \sum_{j\in\mathcal{H}_D} \mathsf{sk}_j + \sum_{j\in\mathcal{H}_D} e_j$, and we compute $\{\mathsf{pk}_j\}_{j\in\mathcal{H}_D} \leftarrow \mathsf{SecretShare}(\mathsf{pk}_{\mathcal{H}_D})$. By Lemma 3, this hybrid is indistinguishable from $\mathsf{Hyb}^{(1)}$.
- $\mathsf{Hyb}^{(3)}$: If the honest decryptors abort in Step 4, then this is the same as $\mathsf{Hyb}^{(2)}$. Assume they do not abort in the following. Then we replace honest partial decryptions pd_j's with random values whose sum is unchanged. Specifically, we first compute $\mathsf{pd}_{\mathcal{H}_D} = -v \cdot \mathsf{pk}_{\mathcal{H}_D} + \sum_{j\in\mathcal{H}_D} e'''_j$, and then we compute $\{\mathsf{pd}_j\}_{j\in\mathcal{H}_D} = \mathsf{SecretShare}(\mathsf{pd}_{\mathcal{H}_D})$. Note that, for all honest $j \in \mathcal{H}_D$, in $\mathsf{Hyb}^{(2)}$ their partial decryptions pd_j are random conditioned on $\sum_{j\in\mathcal{H}_D} \mathsf{pd}_j = \mathsf{pd}_{\mathcal{H}_D}$. So $\{\mathsf{pd}_j\}_{j\in\mathcal{H}_D}$ in this hybrid is indistinguishable that in $\mathsf{Hyb}^{(2)}$.
- $\mathsf{Hyb}^{(4)}$: If the honest decryptors abort in Step 4, then this is the same as $\mathsf{Hyb}^{(2)}$. Assume they do not abort in the following. In this hybrid we do not run honest decryptors, and instead we directly sample $(\mathsf{sk}_j, \mathsf{pk}_j, \pi_j) \leftarrow \mathsf{AHE.VerifiableKeyGen}()$ for all $j \in \mathcal{H}_D$. In addition, we generate shares of r_j and r_j^{aux} for all $j \in \mathcal{H}_D$ as in the honest decryptors, and handle decryptor aborts and client aborts directly in the hybrid.

  So $\mathsf{Hyb}^{(4)}$ is identical to $\mathsf{Hyb}^{(3)}$.
- $\mathsf{Hyb}^{(5)}$: In this hybrid, we set all surviving clients' input to 0, and we program the RO accordingly. Specifically, assume client $1 \in S$, and we set $\mathsf{RO}(\mathsf{seed}_1) = s_{\mathcal{H}}$ be the sum of inputs of clients in S. Additionally, for all $i \in S \setminus \{1\}$, we set $\mathsf{RO}(\mathsf{seed}_i) = 0$. This effectively replaces input x_i with 0 for all honest clients $i \in S$ without changing the adversary's view. Since all seed_i in $\mathsf{Hyb}^{(4)}$ are sampled uniformly at random from an exponentially large domain, and since PRG is modeled as a random oracle, the probability of collisions is negligible, and $\mathsf{Hyb}^{(5)}$ is indistinguishable from $\mathsf{Hyb}^{(4)}$.
- $\mathsf{Hyb}^{(6)}$: In this hybrid we replace pk_j and pd_j for all honest decryptors j with their real distributions. Specifically, we compute

$$\mathsf{pk}_j = -\mathsf{u} \cdot sk_j + e_j,$$

  and

$$\mathsf{pd}_j = \mathsf{sk}_j \cdot (\mathsf{u} \cdot v + e'') + e'''_j.$$

  By Lemma 3 and Lemma 5, this hybrid is indistinguishable from $\mathsf{Hyb}^{(5)}$.

Note that $\mathsf{Hyb}^{(6)}$ is exactly as the ideal world, where the Sim simulates honest clients in S with input $\mathbf{0}$, and it simulates honest decryptors $j \in \mathcal{H}_D$ as in the Decryptor protocol. We now formally conclude that the real and the ideal worlds are indistinguishable. □

Aggregation Functionality $\mathcal{F}^M$

Public Parameters:

- Input domain $\mathbb{F}^\ell$.
- Number of clients n.
- Minimum number of clients in the aggregation $\texttt{min\_n}$.

Parties:

- Server S.
- n clients $1, \ldots, n$, each holding private input $\mathbf{x_i} \in \mathbb{F}^\ell$.
- Trusted party T.
- Adversary $\mathcal{A}$ corrupting a subset of parties $\mathcal{C} \subseteq [n] \cup \{\mathsf{S}\}$.

Functionality:

1. T receives all client inputs, with corrupted clients' inputs chosen by $\mathcal{A}$.
2. If S is corrupted, $\mathcal{A}$ chooses $S \subseteq [n]$. Otherwise $S := [n]$.
3. $\mathcal{A}$ sends S to T.
4. If $|S| < \texttt{min\_n}$ then T aborts.
5. If S is corrupted then T sends $\sum_{i \in S} \mathbf{x_i}$ to $\mathcal{A}$.
6. $\mathcal{A}$ chooses offset $o_\mathcal{A} \in \mathbb{F}^\ell$ and $d \in \{\texttt{continue}, \texttt{abort}\}$ and sends $(o_\mathcal{A}, d)$ to T.
7. If $d = \texttt{continue}$ then T sends $o_\mathcal{A} + \sum_{i \in S} \mathbf{x_i}$ to S, otherwise it aborts.

Figure 9: Manipulable aggregation. An adversary corrupting the server *even just semi-honestly* can manipulate the output *adaptively*, and therefore this functionality offers no correctness guarantees in that setting.

H The Semi-Honest Server Case

H.1 Manipulable Secure Aggregation

Figure 9 shows the functionality implemented by protocol Π^M (Section 5) for the case where the server is passively corrupted or honest.

H.2 Implementing $\mathcal{F}^{\mathsf{Agg}}$ Securely Against a Semi-Honest Server

The protocol Π^M in Section 5 focused on privacy, while not providing any correctness guarantees but instead allowing the adversary to change the output arbitrarily (see Figure 9). In this section, we introduce a functionality (Figure 1) that guarantees the output given to the server is the correct sum of client inputs . Note that a maliciously corrupted server can still choose to output a different value, which we do not aim to prevent. We also still allow the adversary to affect the output by corrupting clients. As mentioned in Appendix B, the techniques from Acorn [8] can be used to ensure that client inputs are within a valid range, and our protocol is compatible with these.

In the remainder of this section, we discuss extensions to our protocol to securely realize the functionality in Figure 1. Let us begin by identifying attacks an adversary controlling the server and a subset of decryptors could perform against Π^M (Fig. 3). First, observe that in Step 7b in Fig. 3, decryptors send partial decryptions $\mathsf{pd}_j := \mathsf{AHE.PartialDec}(c^1, \mathsf{sk}_j)$ to the coordinator (i.e., the server). Nothing prevents them from sending an invalid partial decryption in this step, which would result in an offset added to the output. We prevent this attack by requiring decryptors to prove in zero knowledge that they performed partial decryption correctly, which we formalize as a function $\mathsf{AHE.VerifiablePartialDec}(c^1, \mathsf{sk}_j)$ that returns a proof in addition to the partial decryption.

A second attack that a coalition of malicious decryptors could perform is creating invalid secret-shares in Steps 1b or 1d of Fig. 3. When these malicious decryptors now drop out, the server wrongly reconstructs

their state from the invalid shares, which results again in an offset to the output. We solve this by having the server check whether the reconstructed state generates the same public keys that were shared in the key generation phase. This means that any invalid shares result in a protocol abort instead of an output manipulation.

The last attack we have to defend against are selective aborts, where an adversary aborts the protocol after learning the result, but before the server learns it. For example, in Step 7c of Fig. 3, if $t-1$ honest decryptors send their shares first, and the t-th decryptor is malicious, it can force the server to abort by dropping out while having shared invalid seeds in Steps 1b and 1d. Alternatively, a decryptor could force an abort by sending invalid shares in Step 7c, again after the adversary has observed the intended result. We prevent these two attacks by (i) moving the reconstruction of $(\mathsf{sk}_k^{\mathsf{aux}})_{k\in\mathsf{D}}$ to Round 2 of the decryption phase, and (ii) employing *verifiable* secret sharing in Steps 1b and 1d. The former ensures that decryptors have to decide whether to send their partial decryption before the adversary can learn the output, while the latter allows the server to exclude invalid secret-shares in the dropout recovery phase, thus preventing decryptors to force an abort at that point.

One remaining opportunity for selective aborts appears when the number of dropouts is so large that less than t honest decryptors survive. In that case, protocol termination hinges on malicious decryptors, who again can abort the protocol after the adversary observes the result. To avoid giving the adversary this power of selective abort, we *always* abort the protocol if the number of dropouts exceeds a parameter d, where $0 < d \leq m - t - c$. Together with the constraint that $c < 2t - m - 1$, this means that our protocol can withstand at most $(m-1)/3$ corruptions.

We present the modified decryptor role in Figure 10, with changes needed for correctness highlighted in blue.

Theorem 3 (Semi-Honest Server)**.** *Let $n, m > 0$. Assume* KAHE *is a KAHE scheme satisfying leakage-resilient security of Definition 5, and assume* AHE *is a* Verifiable *AHE scheme satisfying Lemmas 3, 4, and 5. Let Π^{Agg} be the protocol with n clients and m decryptors formed by Figures 10, 5, 4, 6.*

Then Π^{Agg} securely implements (with abort) functionality $\mathcal{F}^{\mathsf{Agg}}$ (Fig. 1) in the RO model, against a static active adversary corrupting any number of clients, at most $t+d-m$ decryptors and possibly either maliciously or semi-honestly the server.

The Client role runs in 1 round with cost $O(\ell \log n)$. Decryptors have a 2-round setup, and 2-round decryption, running in $O(m + \log n)$. The server runs in $O(n\ell \log n)$.

The verifier role runs in $O(n)$. Moreover, in a distributed verifier with c verifiers grouped in c/k random committees, each verifier committee member runs in $O(nk/c + \log n)$.

Proof. For the cases of a maliciously corrupted or an honest server the proof is much the same as in the proofs that $\Pi_{\mathcal{F}^M}$ implements $\mathcal{F}^M$.

We note that adding verification to the secret sharing and partial decryptions and increasing the required number of non-dropouts from t to $m-d$, can only provide the adversary with new ways to cause aborts which they are allowed to do at will anyway.

Meanwhile the functionalities are equivalent in most cases. If the server is malicious then no honest party receives output so it doesn't matter whether the adversary can add an offset to it. If the server is honest then the adversary won't receive the sum of honest party inputs halfway through the functionality and so any value the adversary wants to offset by could have been included in a malicious clients input (assuming there is one) at the beginning. Thus the only case in which the functionalities aren't equivalent are if the adversary controls only decryptors.

Consider the case in which the adversary only controls decryptors. In $\Pi_{\mathcal{F}}$ every message that the decryptors send is verified by the honest server to have been honestly constructed (except for the distribution of the randomness of the keys but they are checked to be validly constructed keys). Thus the decryptors get no opportunity to alter the output of the server, as required, and the protocol is secure.

We now move on to the main part of this proof: considering the case in which the server is semi-honest. The simulator works as follows.

Key Generation phase. Sim runs simulations of all honest decryptors and the semi-honest server, following the protocol exactly. A simulation of the adversary (having the messages to the semi-honest server copied to it throughout) provides the messages from malicious decryptors. Then, for each malicious decryptors j, Sim extracts the secret keys sk_j and $\mathsf{sk}_j^{\mathsf{aux}}$ from the proofs π_j and π_j^{aux} provided with the verifiable key generation and uses this to reconstruct the secret keys sk and $\mathsf{sk}^{\mathsf{aux}}$ that are being chosen.

Data Collection phase. Sim runs simulations of all honest clients, behaving completely honestly but with inputs of 0 in place of their true inputs. When the adversary makes submissions on behave of malicious clients i, Sim uses $\mathsf{sk}^{\mathsf{aux}}$ to recover the encrypted seed and sk to recover their KAHE keys k. It then decrypts m with k to get y and y with seed to get x, i.e. the implicit malicious input. The simulator then submits the malicious inputs to the ideal functionality. Assuming there are enough client contributions (if not the semi-honest server will decide to abort and the adversary's output can then be passed to the distinguisher) the ideal functionality replies with the sum of all contributions. The Sim now selects an honest client included in S, assume client 1, and reprograms $RO(\mathsf{seed}_i)$ to be smaller than it was by the sum of honest clients in S's inputs.

Decryption and Output phase. Finally the Sim runs all the simulated parties to the end of the protocol honestly, it then outputs whatever the adversary outputs.

We now wish to see that the result of this simulation (including the server's output) is indistinguishable from the output of the adversary and server's output in the real world. As in the other proof we do this first informally and then present a hybrid proof below.

We suppose that the random tapes of all parties in the simulations in the ideal world are the same as those of the corresponding parties in the real world, this is fine because they are indeed drawn from the same (uniform) distribution. We further assume that the values of the random oracle (before any reprogramming) between those two worlds differ so that the resulting submissions from the honest clients (in the real world and simulated within the ideal world) are identical i.e. they differ by the honest parties input values, this is fine so long as no two honest parties use the same seed which is true with all but negligible probability. We start by showing that the outputs of the server in the two worlds is the same and then we will show that the changes in the adversaries view are indistinguishable.

The output of the ideal functionality is the sum of the true honest inputs and the malicious inputs that the simulator found implicit in the adversaries submissions. It remains to show that this is also the value that is output by the simulated semi-honest server and the real semi-honest server. The real server takes in the inputs from the honest clients and the inputs from the malicious clients (matching those in the ideal world). We just need to show that the Decryption and Output phase computes the correct result (or aborts independently of the honest inputs). This is guaranteed by the correctness of the protocol when everyone is honest and the fact that every message from the malicious parties (i.e. from the malicious decryptors) in that phase is verified by the server and it can't force a abort after having $\mathsf{sk}^{\mathsf{aux}}$ (and thus the honest seeds) revealed (unless it had already shared out bad information on $\mathsf{sk}^{\mathsf{aux}}$ in which case it can't prevent the abort).

Meanwhile the server in the simulation is computing exactly the same function on exactly the same inputs interacting with exactly the same adversary. The only difference is that the values of the random oracle at the seeds that the server recovers at the end are different but the differences cancel when the server adds them together at the end.

It remains to show that the adversaries' views are indistinguishable. Note that the adversary can't query the random oracle on the honest seeds until they are reveal which (as explained in the proof of the malicious server theorem) is not until right at the end. The view are in fact identical except for the values that the RO takes on the honest seeds when the adversary gets them at the end. They are still uniformly distributed so they are not distinguishable in themselves. However, that doesn't rule out the possibility that they could be dsitinguished in combination with the encryptions of them sent earlier i.e. the m from honest clients.

However, those are encrypted by the semantically secure KAHE scheme which by Lemma 1 only leaks the sum of inputs (which is the same in both cases) when the sum of the corresponding keys is revealed. That last sentence appears in the proof of Theorem 2 and the rest of this proof can be completed identically to the rest of that proof.

The above argument is formalized in the following sequence of hybrids.

- $\mathsf{Hyb}^{(0)}$: This is the real world execution. In particular, the adversary's view contains
  - $(m_j, s_j), \mathrm{sig}(ms_j)$ from all honest decryptors $j \in \mathcal{H}_D$;
  - $\{\mathsf{share}_{k,j}\}_k, \{\mathsf{share}^{\mathsf{aux}}_{k,j}\}_k$ VSS shares from all honest decryptors $j \in \mathcal{H}_D$ which are shares of r_j and r^{aux}_j, respectively;
  - s_{ct^1} from the verifier;
  - $(\mathsf{pd}_j, \pi^{\mathsf{pd}}_j)$ from all honest decryptors $j \in \mathcal{H}_D$.

  The adversary's output (on behalf of corrupted parties) contains
  - $(\mathsf{ct}^1, s_{\mathsf{ct}^1})$ to all honest decryptors j.

  The server's output contains
  - $\{(m_j, s_j)\}_{j\in\mathsf{D}}$ to all decryptors $j \in \mathsf{D}$;
  - $(\mathsf{D}, S_{\mathsf{D}}, \mathsf{pk}, \mathsf{pk}^{\mathsf{aux}})$ to all decryptors $j \in \mathsf{D}$;
  - $(\mathsf{ct}^1, \mathtt{proofs})$ to the verifier;

  In particular, we have that:
  - For all $j \in \mathcal{H}_D$, $\mathsf{pk}_j = -\mathsf{u} \cdot \mathsf{sk}_j + e_j$ for some $e_j \sim \chi_e$.
  - If the decryptor does not abort at Step 4, then for all $k \in \mathcal{C}_D$, we have $\mathsf{pk}_k = -\mathsf{u} \cdot \mathsf{sk}_k + e_k$ for some sk_k and e_k satisfying the bound B_s and B_e.
  - If the decryptor does not abort at Step 7, then $\mathsf{ct}^1 = \sum_{i\in\mathcal{H}} \mathsf{ct}^1_i + \sum_{k\in\mathcal{C}} \mathsf{ct}^1_k$, and $\mathsf{ct}^1_k = v_k \cdot \mathsf{u} + e''_k$ for all $k \in \mathcal{C}$ with v_k and e''_k satisfying the bounds B_s and B_e. Let $v = \sum_{i\in\mathcal{H}} v_i + \sum_{k\in\mathcal{C}} v_k$ and $e'' = \sum_{i\in\mathcal{H}} e''_i + \sum_{k\in\mathcal{C}} e''_k$.
- $\mathsf{Hyb}^{(1)}$: If the honest decryptors abort in Step 7, then this is the same as $\mathsf{Hyb}^{(0)}$. Assume they do not abort in the following. Then we compute all honest decryptor j's partial decryption pd_j using pk_j and the sum of all v_j's including those extracted from π_j for $j \in \mathcal{C}_D$. Specifically, for all $j \in \mathcal{H}$ we set $\mathsf{pd}_j = -v \cdot \mathsf{pk}_j + e'''_j$. We now argue that this is indistinguishable from $\mathsf{Hyb}^{(0)}$. Note that in $\mathsf{Hyb}^{(0)}$ we have

$$\begin{aligned}\mathsf{pd}_j &= \mathsf{sk}_j \cdot (\mathsf{u} \cdot v + e'') + e'''_j \\ &= v \cdot (\mathsf{sk}_j \cdot \mathsf{u} - e_j) + v \cdot e_j + \mathsf{sk}_j \cdot e'' + e'''_j \\ &= -v \cdot \mathsf{pk}_j + v \cdot e_j + \mathsf{sk}_j \cdot e'' + e'''_j.\end{aligned}$$

  By Lemma 5, pd_j in the above expression (as in $\mathsf{Hyb}^{(0)}$) is statistically close to $\mathsf{pd}_j = -v \cdot \mathsf{pk}_j + e'''_j$ as in $\mathsf{Hyb}^{(1)}$. In addition, this hybrid invokes the ZK simulator for the proof system $\Pi_{\mathsf{PartialDec}}$ to simulate $\pi^{\mathsf{PartialDec}}_j$ on input pd_j, for all $j \in \mathcal{H}_D$; these simulated proofs are indistinguishable from the proofs in $\mathsf{Hyb}^{(0)}$. So the two hybrids are indistinguishable.
- $\mathsf{Hyb}^{(2)}$: If the honest decryptors abort in Step 4, then this is the same as $\mathsf{Hyb}^{(1)}$. Assume they do not abort in the following. Then we compute pk_j for all $j \in \mathcal{H}_D$ without using individual sk_j's. Specifically, for all $j \in \mathcal{H}_D$ we replace pk_j with truly random element conditioned on their sum $\sum_{j\in\mathcal{H}_D} \mathsf{pk}_j$ being unchanged, and we simulate the proof π_j of jth public key share on pk_j. That is, in this hybrid we compute $\mathsf{pk}_{\mathcal{H}_D} = -\mathsf{u} \cdot \sum_{j\in\mathcal{H}_D} \mathsf{sk}_j + \sum_{j\in\mathcal{H}_D} e_j$, and we compute $\{\mathsf{pk}_j\}_{j\in\mathcal{H}_D} \leftarrow \mathsf{SecretShare}(\mathsf{pk}_{\mathcal{H}_D})$. By Lemma 3, this hybrid is indistinguishable from $\mathsf{Hyb}^{(1)}$.
- $\mathsf{Hyb}^{(3)}$: If the honest decryptors abort in Step 4, then this is the same as $\mathsf{Hyb}^{(2)}$. Assume they do not abort in the following. Then we replace honest partial decryptions pd_j's with random values whose sum is unchanged. Specifically, we first compute $\mathsf{pd}_{\mathcal{H}_D} = -v \cdot \mathsf{pk}_{\mathcal{H}_D} + \sum_{j\in\mathcal{H}_D} e'''_j$, and then we compute $\{\mathsf{pd}_j\}_{j\in\mathcal{H}_D} = \mathsf{SecretShare}(\mathsf{pd}_{\mathcal{H}_D})$. Note that, for all honest $j \in \mathcal{H}_D$, in $\mathsf{Hyb}^{(2)}$ their partial decryptions pd_j are random conditioned on $\sum_{j\in\mathcal{H}_D} \mathsf{pd}_j = \mathsf{pd}_{\mathcal{H}_D}$. So $\{\mathsf{pd}_j\}_{j\in\mathcal{H}_D}$ in this hybrid is indistinguishable that in $\mathsf{Hyb}^{(2)}$.

- $\mathsf{Hyb}^{(4)}$: If the honest decryptors abort in Step 4, then this is the same as $\mathsf{Hyb}^{(2)}$. Assume they do not abort in the following. In this hybrid we do not run honest decryptors, and instead we directly sample $(\mathsf{sk}_j, \mathsf{pk}_j, \pi_j) \leftarrow \mathsf{AHE.VerifiableKeyGen}()$ for all $j \in \mathcal{H}_D$. In addition, we generate shares of r_j and r_j^{aux} for all $j \in \mathcal{H}_D$ as in the honest decryptors, and handle decryptor aborts and client aborts directly in the hybrid.

  So $\mathsf{Hyb}^{(4)}$ is identical to $\mathsf{Hyb}^{(3)}$.
- $\mathsf{Hyb}^{(5)}$: In this hybrid, we set all surviving clients' input to 0, and we program the RO accordingly. Specifically, assume client $1 \in S$, and we set $\mathsf{RO}(\mathsf{seed}_1) = s_{\mathcal{H}}$ be the sum of inputs of clients in S. Additionally, for all $i \in S \setminus \{1\}$, we set $\mathsf{RO}(\mathsf{seed}_i) = 0$. This effectively replaces input x_i with 0 for all honest clients $i \in S$ without changing the adversary's view. Since all seed_i in $\mathsf{Hyb}^{(4)}$ are sampled uniformly at random from an exponentially large domain, and since PRG is modeled as a random oracle, the probability of collisions is negligible, and $\mathsf{Hyb}^{(5)}$ is indistinguishable from $\mathsf{Hyb}^{(4)}$.
- $\mathsf{Hyb}^{(6)}$: In this hybrid we replace pk_j and pd_j for all honest decryptors j with their real distributions. Specifically, we compute
$$\mathsf{pk}_j = -\mathsf{u} \cdot sk_j + e_j,$$
and
$$\mathsf{pd}_j = \mathsf{sk}_j \cdot (\mathsf{u} \cdot v + e'') + e_j'''.$$
By Lemma 3 and Lemma 5, this hybrid is indistinguishable from $\mathsf{Hyb}^{(5)}$.

Note that $\mathsf{Hyb}^{(6)}$ is exactly as the ideal world, where the Sim simulates honest clients in S with input $\mathbf{0}$, and it simulates honest decryptors $j \in \mathcal{H}_D$ as in the Decryptor protocol. We now formally conclude that the real and the ideal worlds are indistinguishable.

□

I Protocol Extensions

In this section we describe two extensions to our protocol. First, we show how to provide differential privacy for the output, by having the verifier additionally act as a client that inputs noise, while simultaneously checking it was not dropped out. We then discuss how to achieve guaranteed output delivery by slightly increasing decryptor communication.

I.1 Differential Privacy

As discussed in Section 2.3, our functionality allows specifying a minimum number of contributions `min_n` before an aggregation can be performed. While this may offer some protections to individual clients, it becomes moot if the adversary can control all clients except for a target client, and spawn additional clients at will (i.e., a Sybil attack). As we want to avoid requiring an external PKI for clients, this type attack cannot be avoided in general.

Differential Privacy (DP) [48] provides a way to guarantee to individual clients that the output of a function does not reveal much about their input, by means of adding noise from an appropriate distribution to it. A straight-forward way to add noise in our protocol is to view it as another client input. However, we have to make sure that the server cannot ignore that "special" client's contribution. One way to achieve this is to have committee members collectively act as this special client C^*: each individual committe member can act as a separate client, adding a share of the noise, with the noise parameter tuned such that the sum of all committe member noise shares achieves the required privacy level. Exploiting infinite divisibility properties of random variables for this purpose is a well studied problem (see [49] and references therein). Note that since we require an honest majority among committe members, the noise added will be strictly less than twice the noise required. The fact that committee members can sign their contributions in our model is sufficient to ensure that the server must include C^*, as the verifier can easily be modified to additionally

check that sufficiently many signed proofs are present in the message it receives from the server (see step 2c in Figure 6). This approach is compatible with the distributed verifier implementation discussed in Section 5.4. A drawback of having committee members act as clients is that their communication now grows linear in ℓ. We leave the problem of reducing this cost, possibly in an amortized way, for future work.

An alternative approach is to let all clients add noise directly to their inputs. This ensure protection in the form of a *local* differential privacy guarantee. However achieving a significant level of privacy in the local model necessarily $\Omega(\sqrt{n})$ error [50], where n the number of clients in the sum. Limiting the capabilities of the adversary by assuming a bound on the number of corrupt clients (as in previous work [1, 3, 8]) allows to interpolate between the local model and the central model. For example, if we assume at most $n/3$ clients are corrupt, setting `min_n` to $2n/3$ guarantees that any successful aggregation will have at least $n/3$ honest clients, and so we can calibrate the noise added by each client accordingly.

I.2 Guaranteed Output Delivery

Finally, let us consider an extension that further strengthens the security guarantees of our protocol at the cost of increased communication. Recall that Functionality $\mathcal{F}^{\mathsf{Agg}}$ in Figure 1 allows the adversary to abort the computation before the serve receives the result. A natural question may be: can we remove this capability, and guarantee that the server receives an output, provided it is not maliciously corrupted and there are enough honest committee members? If we assume that at most d decryptors drop out, the only steps in Π^{Agg} in Figure 10 that lead to an abort are Steps 12 and 13. We can ensure that no dropouts happen there if we make sure to exclude malicious decryptors that send invalid shares in the key generation phase. To that end, each decryptor checks VSS shares they receive, and forwards the corresponding commitment to the server if the check passes. The server then checks that the commitments created by a given client are all equal, and match the commitment received from the verifiable generation. The public keys pk and $\mathsf{pk}^{\mathsf{aux}}$ are then only computed from the shares that pass all these checks. Note that this requires decryptors to share the actual secret keys via VSS, as opposed to a PRNG seed for them. We chose to stick to non-selective abort in this paper, and leave a more detailed analysis of guaranteed output delivery to future work.

Protocol Π^{Agg} (Decryptor)

Parties:

- A committee $\mathcal{C}$ with members $1, \ldots, m$.
- Coordinator Coord forwarding messages.
- A verifier V that signs ciphertexts intended for decryption.
- PKI holding public signing keys of signature scheme Sig for V and members of $\mathcal{C}$.

Public Parameters: Timeout T, threshold t, maximum number of dropouts d, parameters for schemes AHE, E, VSS and Sig.

Key Generation

Output: Public keys $\mathsf{pk}, \mathsf{pk}^{\mathsf{aux}}$ signed by a set $\mathsf{D} \subseteq \mathcal{C}$ of committee members such that $|\mathsf{D}| \geq t$.

Round 1: Share partial keys

1. Every committee member $j \in [m]$:
   (a) Computes $(\mathsf{sk}_j, \mathsf{pk}_j, \pi_j) := \mathsf{AHE.VerifiableKeyGen}(r_j)$ from randomness r_j.
   (b) Verifiably secret-shares r_j within $\mathcal{C}$ with threshold t.
   (c) Computes $(\mathsf{sk}_j^{\mathsf{aux}}, \mathsf{pk}_j^{\mathsf{aux}}, \pi_j^{\mathsf{aux}}) := \mathsf{E.VerifiableKeyGen}(r_j)$ from randomness r_j^{aux}.
   (d) Verifiably secret-shares r_j^{aux} within $\mathcal{C}$ with threshold t.
   (e) Sends $m_j := (\mathsf{pk}_j, \pi_j, \mathsf{pk}_j^{\mathsf{aux}}, \pi_j^{\mathsf{aux}})$ and $s_j = \mathsf{Sig.Sign}(m_j)$ to Coord.
2. Coord collects messages up to a timeout T. Let $\mathcal{C}_s \subseteq \mathcal{C}$ be the committee members that provide correct proofs and signatures. If $|\mathcal{C}_s| < t$, Coord aborts, otherwise Coord sets $\mathsf{D} := \mathcal{C}_s$, $\mathsf{pk} := \sum_{j \in \mathcal{C}_s} \mathsf{pk}_j$, and $\mathsf{pk}^{\mathsf{aux}} := \sum_{j \in \mathcal{C}_s} \mathsf{pk}_j^{\mathsf{aux}}$.

Round 2: Verify global keys $\mathsf{pk}, \mathsf{pk}^{\mathsf{aux}}$

3. Coord broadcasts $S = \{(m_j, s_j) | j \in \mathsf{D}\}$ within D.

// Decryptors independently check the server's work

4. Every decryptor $j \in \mathsf{D}$:
   (a) Checks $|S| \geq m - d$ and that all proofs and signatures in S are correct, and aborts otherwise.
   (b) Sends $\mathsf{sig}(ms_j) := \mathsf{Sig.Sign}(\sum_{x \in S} x_{1,1}, \sum_{x \in S} x_{1,2})$ to Coord.

// The server collects signatures from decryptors

5. Coord collects messages up to a timeout T, and sets S_{D} to be the resulting set of signatures. If all signatures in S_{D} are valid signatures of $(\mathsf{pk}, \mathsf{pk}^{\mathsf{aux}})$ and $|S_{\mathsf{D}}| \geq m - d$ then Coord sends $(\mathsf{D}, S_{\mathsf{D}}, \mathsf{pk}, \mathsf{pk}^{\mathsf{aux}})$ to D, and aborts otherwise.

Decryption

Input: Aggregated ciphertext component ct^1 and signature(s) s_{ct^1} (from verifier).

Output: Aggregated partial decryption of ciphertext $(\mathsf{ct}^0, \mathsf{ct}^1)$, and key $\mathsf{sk}_{\mathsf{S}}^{\mathsf{aux}}$.

Round 1: Collect partial decryptions

6. Coord receives $(\mathsf{ct}^1, s_{\mathsf{ct}^1})$ and broadcasts it within D.

// Decryptors provide a partial decryption, only if the ciphertext has been verified by V

7. Every $j \in \mathsf{D}$:
   (a) Checks that s_{ct^1} contains appropriate signature(s) from V, otherwise aborts.
   (b) Sends $(\mathsf{pd}_j, \pi_j^{\mathsf{pd}}) := \mathsf{AHE.VerifiablePartialDec}(c^1, \mathsf{sk}_j)$ to Coord.
   (c) Do nothing. // Moved to Round 2, Step 11c.
8. Coord collects messages up to a timeout T. Let P be the set of decryptors that reply. If $|P| < m - d$, Coord aborts.
9. Do nothing. // Moved to Round 2, Step 12.

Round 2: Drop-out recovery

10. Coord sends P to every decryptor in P.
11. Every $j \in P$:
    (a) Aborts if $|P| < m - d$.
    (b) Sends shares received in Step 1b from each decryptor $k \notin P$, i.e. dropouts, to Coord.
    (c) Sends shares received in Step 1d from every decryptor (for Coord to reconstruct $(\mathsf{sk}_k^{\mathsf{aux}})_{k \in \mathsf{D}}$, and thus $\mathsf{sk}^{\mathsf{aux}}$).
12. Coord checks the shares $\mathsf{sk}_k^{\mathsf{aux}}$ against the VSS commitments and if fewer than $m - d$ survive the check aborts. Otherwise it reconstructs $\mathsf{sk}_{\mathsf{S}}^{\mathsf{aux}}$ using the good shares. Coord checks that the reconstructed $\mathsf{sk}_{\mathsf{S}}^{\mathsf{aux}}$ is the secret key for $\mathsf{pk}^{\mathsf{aux}}$ received in Step 2, and aborts if not.
13. Coord reconstructs $(r_k, \mathsf{pk}_k, \mathsf{sk}_k, \mathsf{pd}_k)$ for every dropout $k \notin P$. If pk_k doesn't match the one received in Step 1e Coord aborts.
14. Coord sends $\mathsf{pd} := \sum_{j \in \mathsf{D}} \mathsf{pd}_j$ and $\mathsf{sk}_{\mathsf{S}}^{\mathsf{aux}}$ to S.

Figure 10: Decryptor D by committee, with non-selective abort.