Willow: Secure Aggregation with One-Shot Clients

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

Google

June 11, 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 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 send a single message to the server in an asynchronous fashion, i.e., without the need for synchronizing their reporting time with any other clients. Our approach is based on a committee of parties, called decryptors, that aid in the computation. Decryptors run a setup phase before data collection starts, and a 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 data. Our experimental evaluation shows that our protocol, even while enabling asynchronous client contributions, is competitive with the state of the art protocols that do not have that feature in both computation and communication.

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 points 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 – so-called decryptors – 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. 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. When the server can be maliciously corrupted, we additionally have the committee verify that each client is included in the aggregation at most once, thus ensuring correctness (up to dropouts, which are unavoidable in the asynchronous / single-message setting without a PKI). 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. Decryptors do $O(|\mathsf{D}| + \log n)$ work independent of ℓ.

Our protocols withstand an adversary actively corrupting a minority of the decryptors, and an arbitrary number of clients. We provide protocol variants for both the case where the adversary *additionally* (a) passively corrupts the server and (b) actively corrupts the server. For the case with active corruption of the server we rely on a committee V of parties called verifiers (which could overlap with the set of decryptors). 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 below.

1. When instantiating the decryptor and verifier roles with a small commitee of parties, we get the first protocol with fully asynchronous clients that is practical in the single-server model, with committee members doing work independent of vector size. Moreover, in the maliciously secure variant, using a committee of size $O(n^{1/\alpha})$, with $\alpha \geq 2$ results in all committe members doing work $o(n)$.

2. We also propose a 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 reputation loss risk, and is more concretely efficient.

3. Our technical contributions include:

   (a) An *efficient* reduction from secure length-ℓ vector summation to secure aggregation of n RLWE secrets at the core of our protocol. To achieve this we leverage a recent result of *hint*-RLWE by Kim et al. [13].

   (b) Formalization of asynchronous summation in the real vs. ideal paradigm of secure computation. We show that a natural ideal-world formulation of our target functionality is not realizable in the standard model by protocols achieving our requirements (asynchronicity and committee work that is sublinear in ℓ) and also security with non-selective abort in the presence of a semihonest server. Therefore, our protocols are proven secure in the Random Oracle model. We also present a protocol variant secure in the standard model, but using a more technical functionality.

4. Experimental evaluation showing that the protocol is practical, i.e., constants in the asymptotics mentioned above are small after instantiating our cryptographic assumptions with standard parameters. More

concretely, our protocol matches and even improves on the communication overhead of the closest related work, Flamingo [7]. At the same time, its computation costs remain practical. For example, for vectors of length 10^5, our protocol requires under 500KB of client upload and 407ms of server computation per client.

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. However, this is a one-time download that can be re-used and amortized across multiple aggregations, and is needed even in Prio [10], or the insecure baseline with a single trusted server.

We also require *asynchronicity of client contributions*. This means that once the protocol setup has finished, clients contributing data should not have to wait for other clients to submit their (encrypted) input, i.e., the protocol does not impose synchronization points among clients. In Section 3 we review existing protocols in terms of dynamic client participation, one-shot clients, and asynchronicity.

Applications. Applications include both federated analytics tasks and learning tasks. The former corresponds to histogram computations, where we require protocols to handle n values of possibly in the billions, and collected over a long period of time, e.g. a week. Learning tasks involve aggregation of model updates, as required by federated learning. In that scenario the input length ℓ corresponds to size of the model update, and the required sum size n can be expected to be in the thousands.

2.1 Roles & Assumptions

Our protocols offer different instantiations. While the main instantiation is an asynchronous 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*, discussed next.

- 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 long-term 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 $c > 1$ parties, which we call *decryptors*, our protocol assumes that no more than a fraction $\gamma_d < 1/2$ 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 [14]. 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 (beside being positive), and therefore the decryptor role could be implemented in the 3 parties, honest majority setting. Nevertheless, in our experiments we assume $c = 100$, to highlight that the (distributed) decryptor role is lightweight. In terms of robustness to dropouts, our protocols are robust to as many as $(1 - \gamma_d)c + 1$ committee members dropping out. In terms of correctness, we ensure 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 fraction $\gamma_v < 1/2$ of the verifiers are corrupted, and allow for a fraction of dropouts. If decryptor and verifiers are the same set of parties, then $\gamma_d + \gamma_v < 1/2$.

Passive/Active Security. We consider two settings, and provide a protocol variant for each:

1. Passively corrupted server colluding with *actively* corrupted fraction of decryptors (recall that the verifier role is not needed for passively-corrupted server), all corrupted by the same adversary. Moreover, the adversary might also *actively* corrupt any number of clients.

2. Actively corrupted server colluding with actively corrupted fraction of decryptors and verifiers. In this model the server is fully malicious, and also controls a minority of the verifiers and decryptors.

In both cases, we prove our protocols are secure in the simulation paradigm [15, 16], which we recall in Appendix D.

Functionalities. Our protocols are secure with (non-selective) abort. This means that while an adversary controlling some of the decryptors can force the server to abort an execution, it cannot do that adaptively after observing the result. This is the same guarantee offered by the main protocols in previous works [7, 8]. Moreover, it is not hard to upgrade our protocols to guarantee output delivery to the server by using Verifiable Secret Sharing (VSS), but this comes with significant (but realistic) communication overhead.

Aggregation Functionality $\mathcal{F}_{\text{agg}}$

Setup: n clients identified by $1, 2, \ldots, n$ holding private inputs $\mathbf{x_1}, \ldots, \mathbf{x_n} \in \mathbb{F}^{\ell}$. Adversary $\mathcal{A}$ controlling a subset $\mathcal{C} \subseteq [n]$ of the clients, and possibly corrupting the server S.

1. Functionality $\mathcal{F}_{\text{agg}}$ receives all honest clients' inputs.
2. If S is *actively* corrupted, $\mathcal{A}$ chooses a subset $S \subseteq [n] \setminus \mathcal{C}$ to drop. Otherwise $S = [n] \setminus \mathcal{C}$.
3. $\mathcal{A}$ chooses inputs of corrupted clients $I_{\mathcal{A}} := (\mathbf{y_i})_{i \in \mathcal{C}}$.
4. $\mathcal{A}$ sends $(I_{\mathcal{A}}, d)$ to $\mathcal{F}_{\text{agg}}$, with $d \in \{\texttt{continue, abort}\}$.
5. If $d = \texttt{continue}$ then $\mathcal{F}_{\text{agg}}$ sends $\sum_{i \in \mathcal{C}} \mathbf{y_i} + \sum_{i \in S} \mathbf{x_i}$ to S, otherwise $\mathcal{F}_{\text{agg}}$ sends $\perp$ to S.

Figure 1: The summation functionality implemented by our protocols. The adversary can abort the protocol (security with abort) non-selectively, i.e. without seeing the result. A malicious server can exclude honest clients from the sum.

The functionality achieved by our protocols is presented in Figure 1. Note that, as discussed above, the adversary can decide to abort the protocol (Step 4), but this decision must happen before observing the result (non-selective abort).

In the case of an adversary that passively corrupts the server and actively corrupts clients and decryptors, we show in Section 6 and Appendix B that this functionality cannot be implemented in the standard model by a protocol satisfying our performance constraints. We also present a functionality that *can* be realized, while preserving non-selective abort.

Public Key Infrastructure. In the variant of our protocol secure against actively-corrupted server, decryptors and verifiers need to establish secure channels among themselves, and apply cryptographic signatures and 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 [17] 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*. This makes the PKI assumption much more manageable in practice. In particular, in settings where the verifier is a single party (two-server model) or consists of a small number of parties, this assumption is trivial.

3 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 [18].

Solutions based on Pairwise Masking. An important family of protocols for single-server secure aggregation follow a dining cryptographers based approach [19], 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 in an asynchronous fashion. 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 asynchronicity, i.e., 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 require a synchronization point. In Figure 2 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 achieve asynchronous client contribution is to employ additively homomorphic encryption. However, the simple approach, where clients directly encrypt $\mathbf{x_i}$ under a threshold AHE and the server homomorphically aggregates, 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. Moreover, HE-based approaches in the presence of a malicious server must be enhanced with verifiability to prevent the server from decrypting individual client's contribution. Willow falls in this category, and our approach consists of using RLWE-based key and message homomorphism combined with a threshold AHE scheme to address the communication issues, and non-interactive Zero-Knowledge to achieve verifiability without giving up on one-shot clients.

While we refer the reader to a recent survey [18] for details on HE-based protocols, there are two recent works worth discussing: SASH [20] 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. We are also aware of concurrent work by Karthikeyan and Polychroniadou on "One-Shot Private Aggregation" [21], which we

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

Figure 2: Comparison with the committee-based Flamingo and RLWE-based Acorn protocols. We report asymptotic costs with respect to n (number of clients), c (number of committee members), 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).

plan on looking into and discussing here in a future revision.

Solutions based on Secret-Sharing. Another line of work including FastSecAgg [22] 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 shown in Acorn [8] are compatible with Willow, as the underlying KAHE scheme is the same.

Asynchronous FL. Since Willow allows dynamic client participation, it is compatible with Asynchronous FL, where updates coming from clients are considered by the server to update the current model (at round i) even if they were computed from the model at rounds $j < i - 1$ (see [2] for details). While Willow is compatible with Asynchronous training, it does not preclude (cohort-less) synchronous FL.

4 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 root of trust in a trusted execution environment or a server from a second company whose services may be more expensive than the in-house solution.

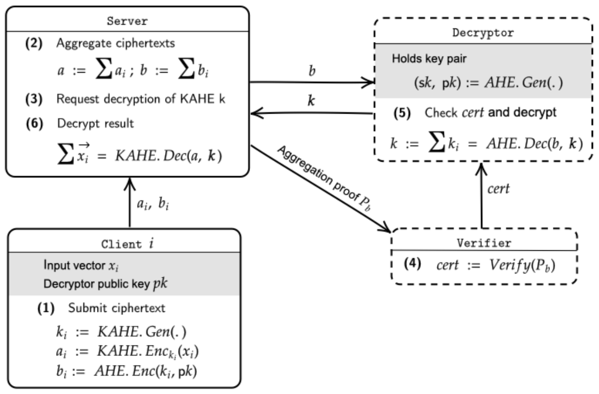
Server
(2) Aggregate ciphertexts
$a := \sum a_i$; $b := \sum b_i$
(3) Request decryption of KAHE k
(6) Decrypt result
$\sum \vec{x_i} = KAHE.Dec(a, k)$
b
k
Decryptor
Holds key pair
$(sk, pk) := AHE.Gen(.)$
(5) Check cert and decrypt
$k := \sum k_i = AHE.Dec(b, k)$
Aggregation proof P_b
cert
a_i, b_i
Client i
Input vector x_i
Decryptor public key pk
(1) Submit ciphertext
$k_i := KAHE.Gen(.)$
$a_i := KAHE.Enc_{k_i}(x_i)$
$b_i := AHE.Enc(k_i, pk)$
Verifier
(4) $cert := Verify(P_b)$

Figure 3: 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. In the actively secure variant, the server 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.

4.1 Our Approach: High-level Overview

Figure 3 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.

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 [23] 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, 24] 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 a RLWE-based AHE scheme that can be regarded as an instance of the proposal by Bendlin and Damgård [25].

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

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 $t+1$ 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 $\mathsf{sk}_j \cdot \mathsf{ct} + \mathbf{e}_{\texttt{flood}}$, where $\mathbf{e}_{\texttt{flood}}$ 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 [26] and, while solutions with smaller flooding noise have been recently suggested [27], they do not 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 3). 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. [13], 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 $q_2 < 2^{80}$ 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, 28] we use Bulletproofs [29] in our evaluation, and rely on the approximate l_∞ proofs by Gentry et al. [28] for efficient proofs of knowledge of (R)LWE secret. For details on this, see Appendix A.3.

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. For this reasons we require the analogous proofs from decryptors as part of key generation and partial decryption. Also in this case we rely on DL-based approaches from [8, 28].

Malicious Server, and the role of the Verifier. As described in Figure 3, the role of the verifier is to ensure that an actively corrupted server cannot send a small subset of the clients' inputs for decryption, instead of the total sum. Moreover, the verifier prevents a malicious server to "copy/replay" contributions from honest clients. 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 [30]. 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 [30]). 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.

Differentially private summation. Note that the verifier ensures that every client that is included in the sum, i.e., that is not ignored by a malicious server, is included just once, and also checks how many clients are included. This does not prevent a malicious server from launching a Sybil attack, as we do not place any assumptions on the identity of asynchronous contributors (as mentioned above we do not assume a PKI for them). While our protocols allow messages to be authenticated, we propose a mitigation based on Differential Privacy in Appendix C.1.

Simulation-based security. Our proofs are in the real vs. ideal paradigm with non-selective abort. Detailed definitions are provided in Appendix D. The functionality achieved by our protocol is given in Figure 1. We show the security of our constructions in the Random Oracle Model (ROM). Moreover, we show an impossibility result stating that the functionality of Figure 1 cannot be achieved in the standard model by a protocol with asynchronous clients and sublinear postprocessing of client messages, i.e., a decryptor running sublinear in ℓ. This is stated in Theorem 4. However, we show a modified functionality (Figure 11) that retains security with non-selective abort in the standard model.

5 Main Cryptographic Primitives

Notation. For any distribution D, we denote using $x \leftarrow D$ the process of sampling from D. If X is a finite set, then by $x \leftarrow X$ we mean sampling at uniformly random from X.

Primitives. In out constructions, we require (i) symmetric key *and* plaintext Additive Homomorphic Encryption (KAHE) and (ii) threshold asymmetric Additive Homomorphic Encryption (AHE). Moreover, for (ii) we require zero-knowledge proofs for (a) knowledge of plaintext, (b) knowledge of secret key, and (c) partial decryptions.

Next, we define the functionality that we need from these primitives and provide more comprehensive discussion of their properties and instantiations in Appendix A.

5.1 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, we require that $\mathbf{c}_1 + \mathbf{c}_2$ is a valid encryption of $x_1 + x_2$ under the key $\mathbf{k}_1 + \mathbf{k}_2$. We further need a leakage-resilient property presented in Definition 1, which guarantees that, given a number of ciphertexts encrypted under different KAHE keys, revealing the aggregate key only reveal the sum of the encrypted messages.

RLWE-based KAHE scheme. We instantiate KAHE based on RLWE assumption. Let N_1 be a power of two, and let $R_{q_1} = \mathbb{Z}[X]/(q_1, X^{N_1}+1)$ for integer $q_1 > 0$. Let $t_1 > 0$ be an integer coprime to q_1; the plaintext space in our KAHE scheme is $R_{t_1} = \mathbb{Z}[X]/(t_1, X^{N_1}+1) \equiv \mathbb{Z}_{t_1}^{N_1}$. For any $\sigma > 0$, let D_σ be the distribution over degree-$N_1 - 1$ 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 $\mathsf{a} \leftarrow R_{q_1}$ as the public parameter which is implicit in the following algorithms.
- $\mathsf{KAHE.KeyGen}() = \mathbf{k}$: Samples and returns $\mathbf{k} \leftarrow D_{\sigma_s}$.
- $\mathsf{KAHE.Enc}(\mathbf{x}, \mathbf{k}) = \mathsf{a} \cdot \mathbf{k} + t_1 \cdot e + \mathbf{x} \in R_{q_1}$: Samples $e \leftarrow D_{\sigma_e}$, 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 $\mathbf{c} - \mathsf{a} \cdot \mathbf{k}$ and then reduce modulo t_1.

We prove in Lemma 1 that this construction satisfies the desired leakage-resilience property.

5.2 Threshold Additive Homomorphic Encryption

We use a public key threshold additive homomorphic encryption scheme with additive distributed key generation and decryption procedures. Such scheme $\mathsf{AHE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{KeyAgg}, \mathsf{Enc}, \mathsf{PartialDec}, \mathsf{Recover})$ allows each party to generate independently its 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 share holder of the secret key can partially decrypt a ciphertext $\mathsf{pd} \leftarrow \mathsf{PartialDec}(\mathsf{ct}, \mathsf{sk}_j)$ and the final decryption 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.

RLWE-based AHE scheme. We use the following RLWE-based AHE scheme. Let N_2 be a power of two, and let $R_{q_2} = \mathbb{Z}[X]/(q_2, X^{N_2}+1)$ be a quotient ring 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}. For any distribution χ, we denote using $s \leftarrow \chi(r)$ the process of sampling from χ using randomness r. We sometimes omit the explicit the randomness parameter r to indicate that it is uniformly sampled. Our AHE consists of the following algorithms:

- $\mathsf{AHE.Setup}() = \mathsf{u}$: Samples $\mathsf{u} \leftarrow R_{q_2}$ as the public parameter which is implicit in the following algorithms.
- $\mathsf{AHE.KeyGen}((r_1, r_2)) = (\mathsf{sk}_j, \mathsf{pk}_j)$: Samples $\mathsf{sk}_j \leftarrow \chi_s(r_1)$ and $e \leftarrow \chi_e(r_2)$, and sets $\mathsf{pk}_j = -\mathsf{u} \cdot \mathsf{sk}_j + e$.
- $\mathsf{AHE.KeyAgg}(\{\mathsf{pk}_j\}_{j=1}^m) = \mathsf{pk}$: Returns the aggregated public key $\mathsf{pk} = \sum_{j=1}^m \mathsf{pk}_j \in R_{q_2}$.
- $\mathsf{AHE.Enc}(x, \mathsf{pk}; r) = (\mathsf{ct}^0, \mathsf{ct}^1)$: Parses $(r_1, r_2) = r$, 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 $\mathsf{ct}^1 = \mathsf{u} \cdot v + e^1 \in R_{q_2}$.
- $\mathsf{AHE.PartialDec}(\mathsf{ct}^1, \mathsf{sk}_j) = \mathsf{ct}^1 \cdot \mathsf{sk}_j + e_{\mathsf{flood}}$: To mask a ciphertext component ct^1, this algorithm samples $e_{\mathsf{flood}} \leftarrow \chi_{\mathsf{flood}}$, and returns $\mathsf{ct}^1 \cdot \mathsf{sk}_j + e_{\mathsf{flood}}$.
- $\mathsf{AHE.Recover}(\mathsf{ct}, \{\mathsf{pd}_j\}_{j=1}^m) = x$: Parses $(\mathsf{ct}^0, \mathsf{ct}^1) = \mathsf{ct}$, and returns $\left\lfloor (\mathsf{ct}^0 + \sum_{j=1}^m \mathsf{pd}_j)/\Delta \right\rceil$.

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.

5.3 Zero-Knowledge Proofs of Knowledge

In our constructions we use three types of ZKPoK:

Proof of plaintext knowledge: The goal of this proof is to guarantee that a particular ciphertext was generated by a party who knows the encrypted message (not by homomorphic computation on other ciphertexts). For the encryption scheme we described above, this boils down to giving a ZKPoK of the randomness v used to generate the second half ct^1 of a given ciphertext using $\mathsf{AHE.Enc}$. The algorithms for the proof are: $\mathsf{Prove}_{\mathsf{AHE.Enc}}(\mathsf{ct}, x, \mathsf{pk}, r, \mathsf{aux})$ returns a proof p. $\mathsf{Verify}_{\mathsf{AHE.Enc}}(\mathsf{ct}, \mathsf{pk}, p, \mathsf{aux})$ return True with overwhelming probability iff the prover knows x such that for some r, $\mathsf{ct} = \mathsf{AHE.Enc}(x, \mathsf{pk}; r, \mathsf{aux})$. Here, aux is an arbitrary string that the proof is bound to using Fiat-Shamir. For efficiency, the particular ZKPoK we use can also be verified given only a commitment c_{ct^1} to the ciphertext component ct^1. By abuse of notation, we also call this verification algorithm as $\mathsf{Verify}_{\mathsf{AHE.Enc}}(c_{\mathsf{ct}^1}, \mathsf{pk}, p, \mathsf{aux})$.

Proof of key generation: This proof guarantees that the party who provides a public key share knows the corresponding secret key share. As in the previous case, in our particular encryption scheme it is enough to prove knowledge of the randomness used in $\mathsf{AHE.KeyGen}$. We use the notation $\mathsf{Prove}_{\mathsf{AHE.KeyGen}}(\mathsf{pk}, \mathsf{sk}, r)$ for the prover and $\mathsf{Verify}_{\mathsf{AHE.KeyGen}}(\mathsf{pk}, p)$ for the verifier.

Proof of partial decryption: This proof guarantees that the partial decryption is generated by knowing a corresponding secret key. We give a ZKPoK of a secret sk used in partial decryption that generates the output. Analogously to the previous case, we use notation $\mathsf{Prove}_{\mathsf{AHE.PartialDec}}(\mathsf{pd}, \mathsf{sk})$ for the prover and $\mathsf{Verify}_{\mathsf{AHE.PartialDec}}(\mathsf{pd}, p)$ for the verifier.

We present constructions for the above proofs in Appendix A.3. Our main observation is that for our RLWE based instantiation of AHE, all of these proofs can be reduced to proving linear relations over vector commitments. As in previous work [8, 28], we instantiate the proof using Bulletproofs [31] based on curve25519, along with Pedersen vector commitments. Besides having very compact proofs with logarithmic size in the number of constraints, Bulletproofs also allow batched verification to improve server performance for a large number of clients. However, other choices of proof systems are possible here, and the right choice will likely depend on the exact application. We leave a detailed analysis of suitable proof systems for future work.

6 Our Protocol: Semi-honest Security

In this section we present the version of our protocol that is secure against a semi-honest server who may control a fraction of clients (respectively decryptors when we have distributed decryptors) and these clients could be fully malicious. As a first step we present a protocol for the decryptor role, assuming it is implemented distributed by a committee of m decryptors.

6.1 Decryptor role

Our protocol for the (distributed) decryptor role is given in Figure 4. We use the roles defined in Section 2.1. For simplicity we describe the protocol assuming the decryptors have secure channels between them, as the server can relay encrypted messages among decryptors. This is the same setup as in previous works, e.g. [3, 7, 8].

Setup: A committee of m members, identified by indices in $[m]$, with secure authenticated channels among themselves, and a coordinator Coord. Each decryptor $j \in \mathsf{D}$ performs all steps below except those done by Coord.

Parameters: Public parameters of AHE, and threshold t.

<u>Key Generation Phase</u>

Output: A set of decryptors D, and a public key pk.

1. Every committee member in $j \in [m]$ generates:
   (a) $(\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow \mathsf{AHE.KeyGen}(r_j)$ from randomness r_j;
   (b) secret-shares $\{\mathsf{share}_k^j\}_k \leftarrow \mathsf{Share}(r_j, t)$ within D with threshold t.
   (c) Sends $\big(\mathsf{pk}_j, \pi_j = \mathsf{Prove}_{\mathsf{AHE,KeyGen}}(\mathsf{pk}_j, r_j)\big)$ to Coord.
2. Coord collects messages up to a timeout T. Let $C \subseteq [m]$ be decryptors with correct proofs π_j. Coord aborts if $|C| < t$, and otherwise Coord sets $\mathsf{D} := C$ and $\mathsf{pk} := \sum_{j\in\mathsf{D}} \mathsf{pk}_j$.

<u>Partial Decryption Phase</u>

Output: A partial decryption of ciphertext.

<u>Round 1</u>

3. Coord sends ct^1 to decryptors in D.
4. Every $j \in \mathsf{D}$
   (a) Secret-shares a key $\mathbf{k}_{\mathsf{sym},j}$ for a symmetric encryption scheme Sym within D with threshold t.
   (b) Sends $\bar{\mathsf{pd}}_j \leftarrow \mathsf{Sym.Enc}(m_j, \mathbf{k}_{\mathsf{sym},j})$ to Coord, where $m_j = (\mathsf{pd}_j = \mathsf{AHE.PartialDec}(c^1, \mathsf{sk}_j)$ and $\tau_j = \mathsf{Prove}_{\mathsf{AHE.PartialDec}}(\mathsf{pd}_j, \mathsf{sk}_j))$.
5. Coord collects messages up to a timeout T. Let $P \subseteq \mathsf{D}$ be the subset of decryptors j submitted messages $\bar{\mathsf{pd}}_j$. If $|P| < t$, Coord aborts.

<u>Round 2</u>

6. Coord sends P to all decryptors in P.
7. Every $j \in P$
   (a) Sends shares $\{\mathsf{share}_j^k\}_{k\in\mathsf{D}\setminus P}$ from all decryptors k not in P, i.e. dropouts, to Coord.
   (b) Sends the key shares received in Step 4a from every decryptor in P to Coord.
8. Coord
   (a) Reconstructs $(r_k, \mathsf{pk}_k, \mathsf{sk}_k, \mathsf{pd}_k)$ of all dropout $k \in \mathsf{D} \setminus P$. Aborts if pk_k differs from the one in Step 1c.
   (b) Recovers m_k for all non-dropout $k \in P$ from Step 4a. Aborts if there is an invalid proof of partial decryption.
   (c) Sends $\mathsf{pd} := \sum_{j\in\mathsf{D}} \mathsf{pd}_j$ to S.

Figure 4: Decryptor D by committee. In practice, Coord above is played by the same server playing S.

In the key generation phase the decryptors generate parameters for an AHE scheme with distributed private key among all decryptors. The underlying RWLE encryption constructions enables 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 up all individual public key shares, i.e. $\mathsf{pk} = \sum_j \mathsf{pk}_j$. The AHE scheme should have appropriately chosen parameters which guarantee ciphertext modulus that can accommodate error growth due to the aggregated noise in the public key. During decryption requests from the server, all decryptors provide partial decryptions pd_j of the ciphertext provided by the server.

Additionally during key generation, each decryptor threshold-secret shares its private key with the others to support dropouts among the decryptors. During dropout recovery the decryptors provide to the server shares of the decryption keys of parties who have dropped out and the server has not received their partial decryptions (the semi-honest server correctly reports dropouts).

The only "non-standard" aspect of the decryptor protocols are the required ZK proofs in key generation and decryption. To see why they are needed, let us present a viable attack in their absence. Consider an adversary $\mathcal{A}$ observing the server, i.e., corrupting it passively, and controlling a decryptor d. $\mathcal{A}$ instructs d to delay its message until all other decryptors have sent their message 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 why the proof of knowledge of secret key required by the protocol addresses: even if $\mathcal{A}$ can see public keys of honest decryptors, it cannot cancel the underlying secret key share.

The proof of partial decryption is required to prevent adaptive abort attacks where one of the decryptors colluding with the server gets to see all other AHE partial decryptions, can therefore decrypt the aggregate KAHE ciphertexts, and based on the output value decide to abort.

Our protocol provides security with a semi-honest coordinating server who may be controlling up to $t-1$ decryptors. We present a formal theorem and a simulation-based proof in Appendix B.

6.2 Server and Client roles

We present the protocols for server and clients in Figures 5 and 6, respectively. Each client encrypts its input under a fresh KAHE key $\mathbf{k}_i$. It also encrypts $\mathbf{k}_i$ itself using the public AHE key pk generated by the decryptors, together with a proof of knowledge for the encrypted $\mathbf{k}_i$. The latter is necessary to prevent the server from deriving correlated keys and using those to shift the aggregated KAHE key that will be decrypted by the decryptors (this is very similar to the attack described in the decryptor section).

The server aggregates the AHE ciphertexts encrypting KAHE keys and submits the resulting AHE ciphertext for decryption by the decryptor. It also aggregates the encrypted messages under the KAHE and uses the decrypted key that it obtained from the decryptor, to decrypt the aggregated clients' inputs.

Our results in the semi-honest model are given by the following theorem. An extended version is in Appendix B.

Theorem 1 (Informal)**.** *The protocol formed by Figures 4 (distributed decryptor), 5 (Server), and 6 (client) securely implements the functionality in Figure 1 in the presence of an adversary actively corrupting, simultaneously, at most $t-1$ decryptors and $n-1$ clients.*

In the presence of an adversary that additionally *passively corrupts the Server, the same protocol securely implements the functionality in Figure 11 in the appendix.*

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

We present a formal theorem and the security proofs for a semi-honest server colluding with a number of malicious clients in Appendix B. Note that in case the server is honest, no honest party receives an output. Therefore, the simulator in that case only needs to simulate the real view of malicious clients and decryptors in the protocol. The view of malicious decryptors involves AHE public key shares from honest decryptors as well as an honestly computed AHE ciphertext. The view of malicious clients only involves the aggregated AHE public key and the fact that this is simulatable follows from the security of the decryptor protocol.

```
**Parameters**: Number of clients $n$, input domain $\mathbb{F}^\ell$.
1. Receive key $\mathsf{pk}$ from $\mathsf{D}$.
2. Initialize $nclients, m, \mathsf{ct}^0$, and $\mathsf{ct}^1$ to zero.
3. while $nclients < n$ :
   // Process $i$th client's request
   (a) Send $\mathsf{pk}$ to $\mathsf{C}_i$
   (b) Receive $(m_i, (\mathsf{ct}_i^0, \mathsf{ct}_i^1), p_i)$ from $\mathsf{C}_i$
   (c) If $\mathsf{Verify}_{\mathsf{AHE.Enc}}((\mathsf{ct}_i^0, \mathsf{ct}_i^1), \mathsf{pk}, p_i, \perp)$:
       i. $(\mathsf{ct}^0, \mathsf{ct}^1) \mathrel{+}= (\mathsf{ct}_i^0, \mathsf{ct}_i^1)$
       ii. $m \mathrel{+}= m_i$
       iii. $nclients \mathrel{+}= 1$
   // Decrypt aggregated symmetric key
4. Send $\mathsf{ct}^1$ to $\mathsf{D}$.
5. Receive $\mathsf{pd}$ from $\mathsf{D}$.
6. $\mathbf{k} := \mathsf{AHE.Dec}(\mathsf{ct}^0, \mathsf{pd})$.
7. Output $\mathsf{KAHE.Dec}(m, \mathbf{k})$.
```

Figure 5: Server S. The server processes n asynchronous client contributions. Each clients that contacts the server receives a key pk and submits a message, without coordination with other clients. The server may process client's requests in parallel and in arbitrary order.

```
**Input:** $\mathbf{x_i} \in \mathbb{F}^\ell$.
1. Receive $\mathsf{pk}$ from $\mathsf{S}$
2. Set $\mathbf{k}_i := \mathsf{KAHE.KeyGen}()$
   // $m_i$ is a symmetric key encryption of input $\mathbf{x_i}$
3. Set $m_i := \mathsf{KAHE.Enc}(\mathbf{x_i}, \mathbf{k}_i)$
4. Sample $r$ uniformly at random
   // $(\mathsf{ct}_i^0, \mathsf{ct}_i^1)$ is an encryption of $\mathbf{k}_i$ under $\mathsf{pk}$ with randomness $r$
5. Set $p_i = \mathsf{Prove}_{\mathsf{AHE.Enc}}((\mathsf{ct}_i^0, \mathsf{ct}_i^1), \mathbf{k}_i, \mathsf{pk}, r, \perp)$
6. Send $(m_i, (\mathsf{ct}_i^0, \mathsf{ct}_i^1), p_i)$ to $\mathsf{S}$
```

Figure 6: Client C. Note that all steps, except forming ct_i^0 and sending the result, (but including forming ct_i^1 and p_i) can be done before the pk arrives from S. Thus online time can be very small, and in particular is independent of ℓ.

7 Our Protocol: Active Security

For malicious security we need to remove some of the assumptions we made in the previous protocol. We assume that (i) the Server behaves honestly relaying messages among decryptors and presents to clients the correct public key, (ii) requests decryption of a correctly aggregated sum, and (iii) honestly reports the drop-out clients. In the malicious setting an actively corrupted server could try to decrypt a message sent by a target victim client in Step 5 (Figure 6), or a sum including a given client's input more than once. In the next few subsections we discuss how to drop the above assumptions. We present

- a proof of unique inclusion that the server can generate to convince a verifier that it has used each input from a client at most once in the aggregation. This together with the property that the server cannot generate any new input related to an existing client's contribution, limits the influence of a single client in the final sum, which is an important step in providing differential privacy, as we will discuss later (Section 7.1).

  We note that an actively corrupted server cannot be prevented from ignoring specific client contributions (assuming we require asynchronous single-message clients). Therefore, the ideal functionality achieved by our summation protocol must allow the adversary to exclude honest clients from the final sum, and is presented in Figure 11.

- a distributed verifier construction which can be instantiated by the clients in a way that allows each client to do work sub-linear in the number of contributions, and independent of the vector length (Section 7.2).

In Appendix C we describe additional changes to client, server and decryptors, as well as a full security proof of our malicious protocol.

7.1 Proof of unique inclusion

As shown in the previous section, the proof of encryption in Figure 6 prevents the adversary from "copying" a client contribution, and having a corrupted client deliver the same proof. However, an actively corrupted server can run Steps (i-iii) in Figure 5 many times, to amplify the influence of a client in the result, by adding their contribution more than once. To prevent this we introduce the role of the verifier, whose task in to ensure that the ciphertext ct sent for decryption in Step 4 corresponds to the aggregation of n values from *distinct* clients (Step 5 in Figure 3). The main properties of the verification stage are the following:

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, and client, server, and decryptor costs remain the same as in the previous section.

3. **Distributed verification.** The verifier's work can be distributed among several parties, to amortize costs.

An important component of our approach is the ability to bind a ZK proof to an integer identifier aux. This can be easily done, in general, by having aux be concatenated onto the statement before it is hashed to generate challenges. In practice we will use $\mathsf{aux} = i$ for the contribution of client i. The modification with respect to the semi-honest version for the client is then simply that they submit $\mathsf{Prove}_{\mathsf{AHE.Enc}}(\mathsf{ct}, x, \mathsf{pk}, r, i)$ instead of $\mathsf{Prove}_{\mathsf{AHE.Enc}}(\mathsf{ct}, x, \mathsf{pk}, r, \perp)$, The goal of the Server is then to convince the verifier(s) that the ciphertext sent for decryption corresponds to the aggregation of a set of ciphertexts with unique identifiers. If that is not the case, then the Server should be caught with very high probability. To do this, the server can commit to the set $\{(m_i, (\mathsf{ct}_i^0, \mathsf{ct}_i^1), p_i)\}_i$ with $p_i = \mathsf{Prove}_{\mathsf{AHE.Enc}}(\mathsf{ct}, x_i, \mathsf{pk}, r_i, i)$, i.e. the data received from *all* clients in Step 3b (Figure 5), and prove that the ciphertext sent for decryption indeed equals the sum of the ct_i^1's, *without repetitions.*

We achieve this with very little communication overhead (independent of input length), in a way that can be easily distributed (thus enabling a distributed decryptor implementation). Our data structure, described next, is similar to the one proposed by Honeycrisp [30], but much more succinct.

Aggregation Tree. Let $S = \left\{(\mathsf{ct}^1_i, p_i)\right\}_i$ with $p_i = \mathsf{Prove}_{\mathsf{AHE.Enc}}(\ \mathsf{ct}, x_i, \mathsf{pk}, r_i, i)$ be the encrypted key and proof of knowledge of randomness received by the server (excluding messages whose proof does not verify). An important observation is that p_i can be verified with respect to a commitment of $c_{\mathsf{ct}^1_i}$. This corresponds to significant savings, as the size of ct^1_i is a few KBs in practice, while a Pedersen vector commitment is 16B. Hence, let $\mathtt{Commit}$ be an additive homomophic binding commitment scheme, e.g. Pedersen commitments, and let $c_{\mathsf{ct}^1_i} = \mathtt{Commit}(\mathsf{ct}^1_i)$ $\tilde{S} = \{c_{\mathsf{ct}^1_i}, p_i\}_i$ be the set of commitments and corresponding proofs, indexed by the client identifier i We define $T(\tilde{S})$ to be the binary tree with the pairs in $\tilde{S}$ as leaves, *sorted by identifier* i, and internal nodes containing a single commitment corresponding to the (homomorphic) sum of the commitments in the two children. Therefore, $T(\tilde{S})$ contains n proofs, and $2n$ 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.

The Verifier Role. In order to now prove that all clients are aggregated into a ciphertext ct^1 at most once, it is enough to send $(\mathsf{ct}^1, T(\tilde{S}))$ to a *verifier*, who can then (i) verify all proofs in $\tilde{S}$, (ii) check that all commitments in $\tilde{S}$ add up to a valid commitment to ct^1, and (iii) check that ids i in the proofs of $\tilde{S}$ are all distinct. Altogether, (i-iii) ensure that the sum was correctly computed without "replaying" any client contribution. The important properties for the verifier are (i) that they don't sign a ciphertext that hasn't passed all of the above checks, (2) they never sign more than one ciphertext and (3) that for any honest client they know whether that client was included in the ciphertext they signed. The final condition is a technical restriction that is required for the proof, however all the ideas for the verifiers below incidentally provide it.

7.2 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 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.

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 clients, among which we trust $\gamma_d \geq 1/2$ do not collude with an adversary corrupting the server, are available to serve as a verifier.

Verifier via committee. In this approach the c clients in the pool are grouped in c/k committee of size $k = O(\sigma + \log(c/k))$, 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 $T(\tilde{S})$ are split into c/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, but 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 8.

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 $\gamma_d c$ 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 $s/(\gamma_d c)$ random intervals, the honest majority assumption ensures a cheating server will get caught with probability $\epsilon = 1 - p$.

The following theorem states the security for our malicious protocol. A more detailed version, together with a proof, is presented in Appendix C.

Theorem 2 (Informal)**.** *The protocol formed by Figure 15 (Client), Figure 14 (Server), and Figure 13 (Decryptor) securely implements the aggregation functionality in Figure 1 in the random oracle model against a malicious adversary controlling the server, any number of clients, at most $t - 1$ decryptors, and at most $t - 1$ verifiers per committee.*

The Client role runs in 1 round with cost $O(\ell \log n)$. Decryptors have a 2-round setup, and 3-round decryption, running in $O(d + \log n)$, where d is the number of decryptors. The server runs in $O(n\ell \log n)$. Each verifier committee member runs in $O(nk/c + \log n)$.

8 Experiments

We implement our protocol in C++ and Rust. We use SHELL [32] for RLWE-based KAHE and AHE schemes, and the Bulletproofs [31] implementation by de Valence et al. [29]. We extend the latter to support approximate and exact range proofs over committed vectors. While our protocol can be instantiated with any zero-knowledge proof 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 [33] 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 the standard deviation. Our KAHE scheme achieves the desired leakage-resilient security by Lemma 1. For AHE, we set χ_s to be the uniform ternary and χ_e to be the centered binomial distribution with variance 8, which are standard choices for achieving semantic security in practice [34]. Furthermore, we set our flooding noise parameters to have 40 bits of statistical security according to the analysis in [35, Corollary 2], and we implement the constant-time discrete Gaussian sampler of Micciancio and Walter [36] to sample the flooding noise.

To achieve optimal efficiency in our experiment settings, we estimated the error bound by numerically computing the aggregated errors. In particular, this allows us to tightly bound the errors to be hidden by the flooding noise e_{flood}; in experiments we see that $\|e_{\mathsf{flood}}\|_\infty$ is around 50 bits for up to 10^9 clients with binary input of length up to 10^6. Although this is still large, thanks to using small secret keys in our KAHE scheme, our AHE modulus q_2 is only 58 to 78 bits. So we can set $N_2 = 2^{12}$ and implement our AHE scheme using up to two `uint64_t` RNS moduli. According to our benchmark, KAHE encryption takes at most 15ms, AHE encryption takes at most 10ms, and partial decryption takes up to 258ms mostly due to sampling large Gaussian flooding noises. For communication cost, a single AHE ciphertext is at most 323KB, and similar to KAHE, we can discard unused coefficients in the component ct^0. Figure 10 in the appendix lists the parameters we used in our experiments.

8.1 Microbenchmarks

We present microbenchmarks for our protocol in Figure 7. 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 (1.8s on the client) and verification (407ms per client on the Server and Verifier). We don't 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 even known to the client. Second, proof verification of each client

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		2.7ms	53KB	0.0156ms									
10^5	<1.8s	8.5ms	494KB	0.0435ms	<407ms	<407ms	<765ms	<1.7s	<2.7ms	<120ms	124KB	<407ms	<1.5KB
10^7		260ms	38MB	3.3419ms									

Figure 7: Microbenchmarks of concrete computation and communication costs for several inputs lenghts ℓ. For the client, we measure encryption, proof generation, and communication cost. For the server, we measure aggregation runtime (per client), and runtime of verification of (a) a client's proof, (b) a decryptor's public key proof, and (c) a partial decryption proof. For the decryptor we measure the proof generation costs, at setup and at decryption. Finally, for the verifier we report per-client runtime and communication, assuming a single verifier. See Figure 9 for experiments with a distributed verifier.

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 [29], 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 A.3) only require linear constraints. As observed by Gentry et al. [28], 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 [31, Section 6.3] greatly reduces the server cost.

8.2 Comparison with Prior Work

Before comparing against prior work, we emphasize that our protocol is the first to allow single-server aggregation with asynchronous clients. As we argue in Section 4, this is a major qualitative difference for large-scale deployments, as it allows scaling to millions of clients while tolerating large dropout rates, and as such any quantitative comparison with prior work is inherently inaccurate.

The works closest to ours are LERNA (Li et al. [9]) and Flamingo (Ma et al. [7]). While they do not support fully asynchronous clients, 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 407ms of computation time per client (See Figure 7). While this is over 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 [37], an aggregation of a million clients with vectors of length 10^7 costs less than 50 cents.

In Figure 8, 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 7.2). We consider both a fully malicious server, which requires the verifiers to check all aggregated contributions, and a covert server with that will 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

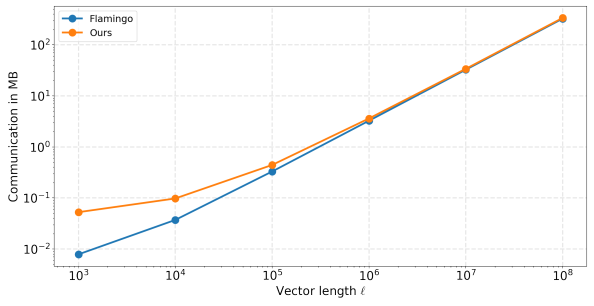

Figure 8: Client communication costs of our protocol and Flamingo for various vector lengths, input range $t = 2^{16}$, statistical security parameter $\sigma = 40$, and 10% dropouts.

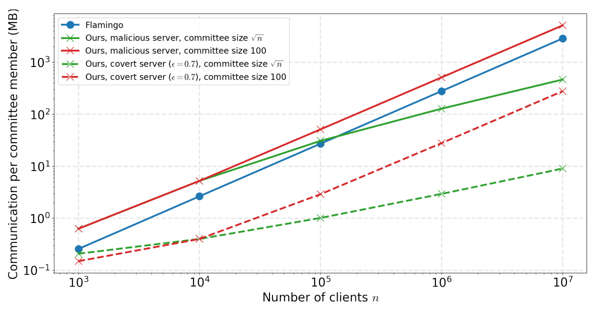

Figure 9: Committee communication cost of different variants of our protocol and Flamingo. This includes both Decryptor and Verifier cost.

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.

9 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] Keith 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] 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.

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

[15] 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.

[16] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications.* Cambridge University Press, 2004.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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`.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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`.

[28] 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.

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

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.

n	c	ℓ	N_1	q_1	**Packing factor**	N_2	q_2	**KAHE comm.**	**AHE comm.**	**log(floodingNoise)**
10^3	10^2	10^3	2048	39	1	4096	60	5.00 KB	46.08 KB	47
10^5	10^2	10^3	2048	49	1	4096	64	6.12 KB	49.15 KB	48
10^7	10^2	10^3	4096	98	2	4096	70	6.12 KB	71.68 KB	51
10^3	10^2	10^5	4096	91	3	4096	60	379.17 KB	61.44 KB	47
10^5	10^2	10^5	8192	212	6	4096	64	441.68 KB	131.07 KB	48
10^7	10^2	10^5	8192	216	5	4096	70	540.00 KB	143.36 KB	51
10^3	10^2	10^7	16384	429	16	4096	60	33.52 MB	245.76 KB	47
10^5	10^2	10^7	16384	408	12	4096	64	42.60 MB	262.14 KB	48
10^7	10^2	10^7	16384	412	10	4096	70	51.50 MB	286.72 KB	51

Figure 10: 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.

[36] 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.

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

[38] 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.

[39] Albert Cheu. Differential privacy in the shuffle model: A survey of separations. *CoRR*, abs/2107.11839, 2021. URL `https://arxiv.org/abs/2107.11839`.

[40] Slawomir Goryczka, Li Xiong, and Vaidy S. Sunderam. Secure multiparty aggregation with differential privacy: a comparative study. In *International Conference on Extending Database Technology*, 2013. URL `https://api.semanticscholar.org/CorpusID:1863871`.

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

A Additional Details on Cryptographic Primitives

We present in this section the additional materials on the cryptographic primitives introduced in Section 5. Figure 10 further presents concrete parameters for our RLWE-based instantiations.

A.1 Key-Additive Homomorphic Encryption (KAHE)

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

- KAHE.Setup() which returns a public parameter. We assume this public parameter is implicit in all the following algorithms.
- 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 1. *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 efficient 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 1.

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 1.*

Our security proof relies on the following Hint-RLWE assumption, which is known to be equivalent to standard RLWE assumption [38].

Definition 2 (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 efficiently 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 [38, 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 efficient 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 a sequence of input $\{x_i\}_{i=1}^n$. Assume $\mathsf{RLWE}_{N_1,q_1,D_\sigma,D_\sigma}$ is hard, and let $\sigma_s = 2\sigma$, $\sigma_e = 2\sqrt{2}\sigma$. We build a simulator $\sim$ as follows.

$$\begin{aligned} \mathsf{Sim}(\mathsf{a}, z = \sum_{i=1}^n x_i) = (\sum_{i=1}^n \mathbf{k}_i, u_1, \ldots, u_{n-1}, u_n')&\text{where} \\ &\forall i \in [n].\mathbf{k}_i \leftarrow D_{\sigma_s}, e_i \leftarrow D_{\sigma_e}, \\ &\forall i \in [n-1].u_i \leftarrow R_{q_1}, \\ &u_n' = -\sum_{i=1}^{n-1} u_i + \mathsf{a} \cdot \sum_{i=1}^n \mathbf{k}_i + \sum_{i=1}^n e_i + t_1 \cdot z \end{aligned}$$

Since $\sigma_e = \sqrt{2}\sigma_s$, sampling from the Gaussian distribution D_{σ_e} is equivalent to the sum of two independent samples from D_{σ_s}. We can rewrite the real distribution as

$$D_0 = \left\{ (\mathsf{a}, \sum_{i=1}^n \mathbf{k}_i, \mathbf{c}_1, \ldots, \mathbf{c}_n) \;\middle|\; \begin{array}{r} \mathsf{a} \leftarrow R_{q_1}, \\ \forall i \in [n].\mathbf{k}_i, e_i, f_i \leftarrow D_{\sigma_s}, \\ \mathbf{c}_i = \mathsf{a} \cdot \mathbf{k}_i + e_i + f_i + t_1 \cdot x_i \end{array} \right\}.$$

We prove that D_0 is indistinguishable from the ideal distribution using the following hybrids. To simplify notations, we assume $e_i, f_i, \mathbf{k}_i$ are all independent samples of D_{σ_s}, and u_i are all independent uniformly random element of R_{q_1}.

- $\mathsf{Hyb}^{(0)} = (\mathsf{a}, z = \sum_{i=1}^n \mathbf{k}_i, y_1 = \mathsf{a} \cdot \mathbf{k}_1 + e_1 + f_1 + t_1 \cdot x_1, \ldots, y_n = \mathsf{a} \cdot z + t_1 \cdot x_n - \sum_{i=1}^{n-1} y_i + \sum_{i=1}^n (e_i + f_i))$. This is exactly D_0.
- $\mathsf{Hyb}^{(j)} = (\mathsf{a}, u_1, \ldots, u_j, y_{j+1} = \mathsf{a} \cdot \mathbf{k}_{j+1} + e_{j+1} + f_{j+1} + t_1 \cdot x_{j+1}, \ldots, y_n = \mathsf{a} \cdot z + t_1 \cdot (x_n + \sum_{i=1}^j x_i) - \sum_{i=1}^j u_i - \sum_{i=j+1}^{n-1} y_i + \sum_{i=1}^n (e_i + f_i))$, for $j = 1, \ldots, n$.

  Assume $\mathcal{A}$ is a distinguisher of $\mathsf{Hyb}^{(j-1)}$ and $\mathsf{Hyb}^{(j)}$. We build a distinguisher $\mathcal{B}$ for the Hint-RLWE problem.

```
$\mathcal{B}(\mathsf{a}, b, \mathbf{k} + r, e + f)$:
  let $z = \sum_{i=1}^{j-1} \mathbf{k}_i + (\mathbf{k} + r) + \sum_{i=j+1}^{n-1} \mathbf{k}_i)$,
    $y_j = b + f_j + t_1 \cdot x_j$,
    $y_{j+1} = \mathsf{a} \cdot \mathbf{k}_{j+1} + e_{j+1} + f_{j+1} + t_1 \cdot x_{j+1}$,
    $\ldots$
     $y_n = \mathsf{a} \cdot z + t_1 \cdot (x_n + \sum_{i=1}^{j-1} x_i) - \sum_{i=1}^{j-1} u_i$
         $- b - \sum_{i=j+1}^{n-1} y_i + \sum_{i=1}^{j-1} (e_i + f_i) + (e + f)$
         $+ \sum_{i=j+1}^{n-1} (e_i + f_i) + e_n$
  run $\mathcal{A}(\mathsf{a}, u_1, \ldots, u_{j-1}, y_j, \ldots, y_n)$
  return the output of $\mathcal{A}$.
```

If $b = \mathsf{a} \cdot \mathbf{k} + e$ as in Hint-RLWE, then $y_j = \mathsf{a} \cdot \mathbf{k} + e + f_j + t_1 \cdot x_j$, and

$$\begin{aligned} y_n &= \mathsf{a} \cdot z + t_1 \cdot (x_n + \sum_{i=1}^{j-1} x_i) - \sum_{i=1}^{j-1} u_i - \sum_{i=j}^{n-1} y_i \\ &\quad + \sum_{i=1}^{j-1}(e_i + f_i) + (e + f) + \sum_{i=j+1}^{n-1} (e_i + f_i) + e_n + f_j. \end{aligned}$$

So $\mathcal{A}$ is given $\mathsf{Hyb}^{(j-1)}$.

If $b = u$ is uniform over R_q, then $y_j = u + f_j + t_1 \cdot x_j = u'$ is uniform over R_q, and

$$\begin{aligned} y_n &= \mathsf{a} \cdot z + t_1 \cdot (x_n + \sum_{i=1}^{j-1} x_i) - \sum_{i=1}^{j-1} u_i - u - \sum_{i=j+1}^{n-1} y_i \\ &\quad + \sum_{i=1}^{j-1}(e_i + f_i) + (e + f) + \sum_{i=j+1}^{n-1} (e_i + f_i) + e_n \\ &= \mathsf{a} \cdot z + t_1 \cdot (x_n + \sum_{i=1}^{j} x_i) - \sum_{i=1}^{j-1} u_i - u' - \sum_{i=j+1}^{n-1} y_i \\ &\quad + \sum_{i=1}^{j-1}(e_i + f_i) + (e + f) + \sum_{i=j+1}^{n-1} (e_i + f_i) + e_n + f_j. \end{aligned}$$

Thus $\mathcal{A}$ is given $\mathsf{Hyb}^{(j)}$. Since $\mathsf{RLWE}_{N_1,q_1,D_\sigma,D_\sigma}$ is hard, we see that $\mathsf{Hyb}^{(j-1)}$ and $\mathsf{Hyb}^{(j)}$ are indistinguishable.

Note that $\mathsf{Hyb}^{(n)}$ is exactly D_1. By the above hybrid argument, we conclude that D_0 and D_1 are indistinguishable. □

A.2 Additive Homomorphic Encryption (AHE) with Distributed Key Generation and Decryption

We use a public key additive homomorphic encryption scheme with additive distributed key generation and decryption procedures. Let $m > 0$ be the number of parties or decryptors. Such cryptosystem is sometimes called a m-out-of-m additive threshold encryption scheme. Formally, an AHE scheme consists of the following algorithms.

- $\mathsf{AHE.Setup}()$ generates a public parameter that is assumed to be implicit in all the following algorithms.
- $\mathsf{AHE.KeyGen}(r)$ takes randomness r and returns a pair $(\mathsf{sk}, \mathsf{pk})$ of secret and public key shares.
- $\mathsf{AHE.KeyAgg}(\{\mathsf{pk}_j\}_j)$ takes in a sequence of public key shares and returns a public key pk.
- $\mathsf{AHE.Enc}(x, \mathsf{pk}, r)$ encrypts a message x using a public key pk and randomness r, returning a ciphertext ct.
- $\mathsf{AHE.PartialDec}(\mathsf{ct}, \mathsf{sk}_j)$ takes a ciphertext ct and a secret key share sk_j and returns a partial decryption pd.
- $\mathsf{AHE.Recover}(\mathsf{ct}, \sum_j \mathsf{pd}_j)$ takes a ciphertext ct and the sum of partial decryptions $\{\mathsf{pd}_j\}_j$, returning the message encrypted in the ciphertext ct.

Note that in our RLWE-based instantiation, a ciphertext consists of two components $\mathsf{ct} = (\mathsf{ct}^0, \mathsf{ct}^1)$, and only ct^1 is needed by $\mathsf{PartialDec}$ whereas only ct^0 is needed by $\mathsf{Recover}$.

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

Simulation Security. As a scheme with distributed decryption, we require partial decryptions to preserve individual decryptors' secret, while allowing a separate party (the server in our case) to recover the plaintext message from partial decryptions. In the following security definition, we consider a semi-honest adversary corrupting a subset of decryptors $\mathbb{M} \subset [m]$, where the adversary can ask honest decryptors to partially decrypt a ciphertext encrypting a message chosen by the adversary. Note that our security formulation is based on partial decryptions of a single ciphertext, which is sufficient for our use case.

Definition 3. *For any $m, n > 0$, let $\mathcal{E}$ be a public key AHE scheme with additive distributed key generation and decryption with m decryptors and with at most n homomorphic additions. Let κ and λ be computational and statistical security parameters, respectively. The scheme $\mathcal{E}$ is* secure *if*

- *it is semantic secure when considering just* KeyGen*,* KeyAgg*, and* Enc *algorithms, and*
- *for any subset $\mathcal{H} \subseteq [m]$, there exists an efficient simulator* Sim *such that, for any $0 < k \leq n$ and for all plaintext messages $x_1, \ldots, x_k$, the following distribution*

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

is computationally indistinguishable from

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

In the above definition, we consider an AHE scheme with a priori bounded number of homomorphic addition operations. This is mainly for simplicity such that we can use fixed parameters to implement $\mathsf{PartialDec}$. We can loss such restriction by parameterize $\mathsf{PartialDec}$ with the number of homomorphic additions used to generate ct.

RLWE-based instantiation. We use the following RLWE-based AHE scheme.

Definition 4. *Let N_2 be a power of two, and let $R_{q_2} = \mathbb{Z}[X]/(q_2, X^{N_2} + 1)$ be a quotient ring 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}. For any distribution χ, we denote using $s \leftarrow \chi(r)$ the process of sampling from χ using randomness r. We use a uniformly random* u *as the public parameter that is implicit in all algorithms. Our AHE consists of the following algorithms:*

- $\mathsf{AHE.KeyGen}(r) = (\mathsf{sk}, \mathsf{pk})$*: Parses $(r_1, r_2) = r$, samples $\mathsf{sk} \leftarrow \chi_s(r_1)$ and $e \leftarrow \chi_e(r_2)$, and sets* $\mathsf{pk} = -\mathsf{u} \cdot \mathsf{sk} + e$.
- $\mathsf{AHE.KeyAgg}(\{\mathsf{pk}_j\}_{j=1}^m) = \mathsf{pk}$*: The aggregated public key is* $\mathsf{pk} = \sum_{j=1}^m \mathsf{pk}_j \in R_{q_2}$.
- $\mathsf{AHE.Enc}(x, \mathsf{pk}, r) = (\mathsf{ct}^0, \mathsf{ct}^1)$*: Parses $(r_1, r_2) = r$, samples $v \leftarrow \chi_s(r_1)$ and $e^0, e^1 \leftarrow \chi_e(r_2)$, and computes* $\mathsf{ct}^0 = \mathsf{pk} \cdot v + e^0 + \Delta \cdot x \in R_{q_2}^2$ *and* $\mathsf{ct}^1 = \mathsf{u} \cdot v + e^1 \in R_{q_2}^2$.
- $\mathsf{AHE.PartialDec}(\mathsf{ct}^1, \mathsf{sk}_j) = \mathsf{ct}^1 \cdot \mathsf{sk}_j + e_{\mathsf{flood}}$*: To mask a ciphertext component* ct^1 *using a secret key share* sk_j*, this algorithm samples $e_{\mathsf{flood}} \leftarrow \chi_{\mathsf{flood}}$, and returns* $\mathsf{ct}^1 \cdot \mathsf{sk}_j + e_{\mathsf{flood}}$.

- $\mathsf{AHE.Recover}(c, \{\mathsf{pd}_j\}_{j=1}^m) = x$*: Parses* $(\mathsf{ct}^0, \mathsf{ct}^1) = \mathsf{ct}$*, and returns* $\left\lfloor (\mathsf{ct}^0 + \sum_{j=1}^m \mathsf{pd}_j)/\Delta \right\rceil$.

The above scheme is message-additive homomorphic: for any messages x and x', if ct and ct' are ciphertexts encrypting x and x' respectively, then $\mathsf{ct} + \mathsf{ct}'$ encrypts $x + x'$. It is also easy to see that the public key of this AHE scheme is additively homomorphic: for any two pairs of key shares $(\mathsf{sk}_1, \mathsf{pk}_1)$ and $(\mathsf{sk}_2, \mathsf{pk}_2)$, the sum $\mathsf{pk} = \mathsf{pk}_1 + \mathsf{pk}_2$ is a public key that corresponds to the sum of secret key shares $\mathsf{sk}_1 + \mathsf{sk}_2$. Furthermore, if $(\mathsf{sk}_3, \mathsf{pk}_3)$ is another pair of key shares, then $\mathsf{pk} + \mathsf{pk}_3$ is a public key corresponding to the secret key $\mathsf{sk}_1 + \mathsf{sk}_2 + \mathsf{sk}_3$. To decrypt a ciphertext ct encrypted under a public key pk, the holder of each secret key share sk_j executes $\mathsf{AHE.PartialDec}(\mathsf{ct}^1, \mathsf{sk}_j)$ and obtains pd_j, and then one can recover (without knowledge of any secret key share) the underlying plaintext by invoking $\mathsf{AHE.Recover}(\mathsf{ct}^0, \{\mathsf{pd}_j\}_j)$. To ensure correctness and leakage-resilient partial decryptions, we will need to do error analysis and set parameters with respect to the error term in an aggregated public key.

The following lemma shows that our AHE scheme is secure with respect to Definition 3.

Lemma 3. *Assume* $\mathsf{RLWE}_{N_2, q_2, \chi_s, \chi_e}$ *is* κ*-bit hard. Let* β *be an upper bound on* $\|e \cdot s + f \cdot v\|_\infty$ *for* $e_1, \ldots, e_n, f \leftarrow \chi_e$*,* $e = \sum_{i=1}^n e_i$*,* $s, v_1, \ldots, v_n \leftarrow \chi_s$*, and* $v = \sum_{i=1}^n v_i$*. Let* $s_{\mathsf{flood}} = \sqrt{24 N_2} 2^{\lambda/2} \cdot \beta$*, and let* χ_{flood} *be the discrete Gaussian distribution with parameter* s_{flood}*. Then the AHE scheme is* (κ, λ)*-bit secure with respect to Definition 3.*

Proof. Assume the RLWE problem with the given secret and error distributions is hard. Then it is known that the AHE scheme without $\mathsf{PartialDec}$ and $\mathsf{Recover}$ is semantic secure. So it remains to show that the joint distribution with partial decryptions can be simulated.

Without loss of generality, assume $\mathcal{H} = \{1, \ldots, h\}$. We build a simulator Sim as follows.

$$\begin{aligned}
\mathsf{Sim}(\mathsf{pk}, \{\mathsf{sk}_j\}_{j \notin \mathcal{H}}, z) = (\{\tilde{\mathsf{pk}}_j, \tilde{z}_j + e_j'''\}_{j \in \mathcal{H}}) &\text{ where} \\
&\forall h < j < m. \tilde{\mathsf{pk}}_j, \tilde{z}_j \leftarrow R_{q_2}, \\
&\tilde{\mathsf{pk}}_m = \mathsf{pk} - \sum_{j \in \mathcal{H}} \mathsf{pk}_j - \sum_{h<j<m} \tilde{\mathsf{pk}}_j, \\
&\tilde{z}_j = z - \mathsf{ct}_0 - \mathsf{u} \cdot \sum_{j \in \mathcal{H}} \mathsf{sk}_j - \sum_{h<j<m} \tilde{z}_j
\end{aligned}$$

Fix $0 < k \le n$ and a sequence of inputs $\{x_i\}_{i=1}^k$. Let $z = \sum_{i=1}^k x_i$. The real distribution is

$$D_0 = \left\{ \begin{array}{r} (\{\mathsf{sk}_j\}_{j=1}^m, \mathsf{pk}, \mathsf{ct}, \{\mathsf{pk}_j, \mathsf{ct}^1 \cdot \mathsf{sk}_j + e_j'''\}_{j \in \mathcal{H}}) : \\ \forall j \in [m]. r_j \leftarrow \{0,1\}^\kappa, (\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow \mathsf{KeyGen}(r_j), \\ \mathsf{pk} = \sum_{j=1}^m \mathsf{pk}_j, \\ \forall i \in [k]. \mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}, x_i), \mathsf{ct} = \sum_{i=1}^k \mathsf{ct}_i, \\ \forall j \in \mathcal{H}. e_j''' \leftarrow \chi_{\mathsf{flood}} \end{array} \right\}.$$

For simplicity, let us fix $\mathsf{sk}_j \leftarrow \chi_s$ for all $j \in [m]$. The real distribution conditioned on these fixed secret key shares is

$$\mathsf{Hyb}^{(0)} = \left\{ \begin{array}{c} (\mathsf{ct}^0, \mathsf{ct}^1, \{\mathsf{pk}_j, \mathsf{pd}_j = \mathsf{ct}^1 \cdot \mathsf{sk}_j + e_j'''\}_{j \in \mathcal{H}}) : \\ \forall i \in [k]. \mathsf{ct}_i \leftarrow \mathsf{Enc}(\mathsf{pk}, x_i), \mathsf{ct} = \sum_{i=1}^k \mathsf{ct}_i \end{array} \right\}.$$

We can write $\mathsf{ct} = (\mathsf{ct}^0 = v \cdot \mathsf{pk} + e' + \Delta z, \mathsf{ct}^1 = v \cdot \mathsf{u} + e'')$, where $v = \sum_{i=1}^k v_i$ for $v_i \sim \chi_s$, $e' = \sum_{i=1}^k e_i'$ and $e'' = \sum_{i=1}^k e_i''$ for $e_i', e_i'' \sim \chi_e$. Furthermore, for all $j \in [m]$, we can write $\mathsf{pk}_j = -\mathsf{u} \cdot \mathsf{sk}_j + e_j$ for $e_j \sim \chi_e$. Then, for all $j \in \mathcal{H}$, the partial decryption pd_j in $\mathsf{Hyb}^{(0)}$ can be expressed as

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

By our assumption that $\beta \geq \|v \cdot e_j + e'' \cdot \mathsf{sk}_j\|_\infty$ and χ_{flood} is discrete Gaussian with parameter $s_{\mathsf{flood}} = \sqrt{24N_2}2^{\lambda/2} \cdot \beta$, it can be shown that (following an analysis as in [35, Corollary 2]) $\mathsf{Hyb}^{(0)}$ is λ-bit statistically indistinguishable from the following hybrid:

$$\mathsf{Hyb}^{(1)} = \left\{ \begin{array}{r} (\mathsf{ct}^0, \mathsf{ct}^1, \{\mathsf{pk}_j, d_j = -v \cdot \mathsf{pk}_j + e'''_j\}_{j\in\mathcal{H}}) : \\ \forall i \in [k].\mathsf{ct}_i = (v_i \cdot \mathsf{pk} + e'_i + \Delta x_i, v_i \cdot \mathsf{u} + e''_i), \\ \mathsf{ct} = \sum_{i=1}^k \mathsf{ct}_i, v = \sum_{i=1}^k v_i \end{array} \right\}.$$

Next, notice that for $j \in \mathcal{H}$, pk_j in $\mathsf{Hyb}^{(1)}$ is pseudorandom conditioned on $\sum_{j=1}^m \mathsf{pk}_j = \mathsf{pk}$. So $\mathsf{Hyb}^{(1)}$ is indistinguishable from

$$\mathsf{Hyb}^{(2)} = \left\{ \begin{array}{r} (\mathsf{ct}^0, \mathsf{ct}^1, \{\tilde{\mathsf{pk}}_j, d_j = -v \cdot \tilde{\mathsf{pk}}_j + e'''_j\}_{j\in\mathcal{H}}) : \\ \forall j \in [m].e_j \leftarrow \chi_e, \\ \mathsf{pk} = \mathsf{u} \cdot \mathsf{sk} + \sum_{j=1}^m e_j, \\ \forall j \in [m-1].\tilde{\mathsf{pk}}_j \leftarrow R_{q_2}, \\ \tilde{\mathsf{pk}}_m = \mathsf{pk} - \sum_{j<m} \tilde{\mathsf{pk}}_j, \\ \forall i \in [k].\mathsf{ct}_i = (v_i \cdot \mathsf{pk} + e'_i + \Delta x_i, v_i \cdot \mathsf{u} + e''_i), \\ \mathsf{ct} = \sum_{i=1}^k \mathsf{ct}_i, v = \sum_{i=1}^k v_i \end{array} \right\}.$$

Since all $\tilde{\mathsf{pk}}_j$ for $j < m$ are uniformly random, we see that $\mathsf{Hyb}^{(2)}$ is equivalent to the simulator output. Our proof is now complete. □

A.3 Zero-Knowledge Proof of Knowledge (ZKPoK)

Next we discuss how we instantiate the three ZKPoK constructions that we defined in Section 5.3.

Succinct Proof of RLWE-based AHE via linear constraints. In the context of our RLWE-based instantiation of AHE (Section 5.2), the three required ZKPoK boil down to proving inner product constraints on vectors corresponding to polynomial coefficients. This approach has been used in previous works [8, 28] 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 $a \in R$ with coefficient vector $\mathbf{a} = (a_0, \ldots, a_{N-1})$, the *negacyclic matrix* representation of a is

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

For any $a, b \in R$ with coefficient vectors $\mathbf{a}$ and $\mathbf{b}$, $\varphi(a) \cdot \mathbf{b}$ 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 $\mathbf{A} = \varphi(a) \in \mathbb{Z}_q^{N\times N}$ 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 arithmetic relations in $\mathbb{Z}_q^N$.

In our RLWE-based AHE scheme, both Proof of plaintext knowledge and Proof of key generation boil down to the following. Let $(a, ar + e) \in R_q^2$ be an RLWE sample. For public polynomials a, c, the prover shows knowledge of a polynomial r and a *small* polynomial e such that the relation $c = ar + e$ holds in R_q. By *small* here we mean that $\|\mathbf{e}\|_\infty$ is bounded by a public parameter t. The following definition captures the relation of interest formally, which we denote as $R_{N,q,t,\mathbf{c},\mathbf{A}}$, as it is parameterized by the public values $N, q, t, \mathbf{c}$ and $\mathbf{A}$. Let us remark that the statement in the relation is over the naturals, and that the $\mathbf{w}q$ term allows to simulate reduction modulo q.

Definition 5 (Knowledge of secret)**.** *Let $N, q, t > 0$ be integers, and let $\mathbf{A} := \varphi(a) \in \mathbb{Z}^{N \times N}$ and $\mathbf{c} \in \mathbb{Z}^N$ be the randomness and ciphertext corresponding to an RLWE sample $c := (a, ar + e) \in R_q^2$. We define the* knowledge of secret *relation as $R_{N,q,t,\mathbf{c},\mathbf{A}} := \big\{(\mathbf{r}, \mathbf{e}, \mathbf{w}) \in \mathbb{Z}^{3N} \mid \mathbf{c} = \mathbf{A}\mathbf{r} + \mathbf{e} - \mathbf{w}q \wedge ||\mathbf{e}||_\infty < t\big\}$.*

In the following, we will define proofs based on the Bulletproofs framework [31] for the above relation that have size sub-linear in N. We stress, however, that there are many ways to prove the above relation in zero-knowledge, and we leave the application-specific optimization of this step to future work.

As described in Section 5.3, we need three kinds of zero-knowledge proofs for AHE operations. Our first observation is that all three share the same structure given in Definition 5. That is, we want to prove knowledge of a secret $\mathbf{r}$ subject to the linear relation $\mathbf{A}\mathbf{r} + \mathbf{e} - \mathbf{w}q$ and the inequality $||\mathbf{e}||_\infty < t$. However, while Definition 5 is stated over the naturals, the proofs we use work over elliptic curve groups of a finite order P. Instead of proving $R_{N,q,t,\mathbf{c},\mathbf{A}}$ directly, we prove the following three constraints hold:

1. $\mathbf{c} = \mathbf{A}\mathbf{r} + \mathbf{e} - \mathbf{w}q \pmod P$
2. $||\mathbf{e}||_\infty < t \pmod P$
3. $||(\mathbf{r}|\mathbf{w})||_\infty < P/6Nq \pmod P$

Note that the third constraint ensures that the first and second one hold over the naturals even if we prove them mod P, so long as $t \ll P/3$. The reason is that (i) entries in $\mathbf{A}\mathbf{r}$ are bounded by the sum of N independent and uniformly random elements of $\mathbb{Z}_q$, and therefore upper bounded by $2Nq$, and (ii) entries in $\mathbf{w}$ are bounded by that same value.

In our security proof we require t to be equal to p/n, where p is a high-probability bound on the flooding noise in threshold decryption (with a value of about 2^{50} in our concrete parameters, see Section 8), and n is the number of clients.

When $\mathbf{e}$ is large, we can, in the same way as Bell et al. [8], combine inner product Bulletproofs [31] and the optimized range proofs of Gentry et al. [28]. This is what we use for $\mathsf{Prove}_{\mathsf{AHE.PartialDec}}$, and we refer the reader to [8, Section 5.3.2] for the details.

An improved construction for small error vectors. When $\mathbf{e}$ is small, we can do better than Bell et al. [8] by using approximate range proofs.

Lemma 4 (Approximate proof of smallness ([28], 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 $||\mathbf{z}||_\infty \leq b/2$.*

Since $\mathbf{e}$ in $\mathsf{AHE.KeyGen}$ and $\mathsf{AHE.Enc}$ is sampled from a centered binomial distribution with variance 8, it has support in $[-16, 16]$. 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 10). Finally, recall that $\mathbf{r}$ is a ternary vector.

We now observe that the bounds (2) and (3) that we need to prove are loose, in the sense that honest clients will hold much smaller $\mathbf{r}, \mathbf{e}, \mathbf{w}$ and therefore we can employ efficient approximate proofs. Recall that we need the multiplicative gap between these two quantities to be at least $2500\sqrt{N}$, as per Lemma 4. This is easy to verify for (3), given the concrete values of N, q and P discussed above. For (2), recall that $t = p/n$ in our application, where p is a high-probability bound on the flooding noise in threshold decryption and n is the number of clients. An important observation is that p in fact grows with n since the flooding noise in partial decryptions should be large enough to mask a term $\mathsf{X}s$, where X is a sum of n binomials and s is a share of the decryption key(see Lemma 3) . Therefore, $t > 2^{20} \gg ||\mathbf{e}||_\infty \leq 16$, which implies the required gap. The following lemma states this reduction to inner product constraints, which then can be offloaded to Bulletproofs.

Relaxed Aggregation Functionality $\mathcal{F}_{\mathsf{agg}}$

Setup: n clients identified by indices $1, 2, \ldots, n$ holding private inputs $\mathbf{x_1}, \ldots, \mathbf{x_n}$. Adversary $\mathcal{A}$, controlling a subset $\mathcal{C} \subseteq [n]$ of the clients, and possibly corrupting the server S.

1. Functionality $\mathcal{F}_{\mathsf{agg}}$ receives all honest clients' inputs.
2. If S is *actively* corrupted, $\mathcal{A}$ chooses a subset $S \subseteq [n] \setminus \mathcal{C}$ to drop. Otherwise $S = [n] \setminus \mathcal{C}$.
3. $\mathcal{F}_{\mathsf{agg}}$ generates $\mathbf{k} \leftarrow \mathsf{KAHE.KeyGen}()$ and sets $\tilde{m} := \mathsf{KAHE.Enc}(\sum_{i \in S} \mathbf{x_i}, \mathbf{k})$.
4. if S is corrupted, $\mathcal{F}_{\mathsf{agg}}$ sends $\tilde{m}$ to $\mathcal{A}$.
5. $\mathcal{A}$ chooses inputs of corrupted clients $I_{\mathcal{A}} := (\mathbf{y_i})_{i \in \mathcal{C}}$.
6. $\mathcal{A}$ sends $(I_{\mathcal{A}}, d)$ to $\mathcal{F}_{\mathsf{agg}}$, with $d \in \{\texttt{continue, abort}\}$.
7. If $d = \texttt{continue}$ then $\mathcal{F}_{\mathsf{agg}}$ sends $\sum_{i \in \mathcal{C}} \mathbf{y_i} + \sum_{i \in S} \mathbf{x_i}$ to S, otherwise $\mathcal{F}_{\mathsf{agg}}$ sends $\perp$ to S.
8. If S is corrupted and $d = \texttt{continue}$, $\mathcal{F}_{\mathsf{agg}}$ sends $\mathbf{k}$ to $\mathcal{A}$.

Figure 11: The relaxed summation functionality with a possible malicious server.The adversary (i) observes an encryption of the honest client's input sum before sending its inputs, (ii) gets to abort the protocol after observing the result, and (iii) gets to choose which honest clients to drop.

Lemma 5 (Proof of knowledge of secret)**.** *Let* $\mathbf{A} := \varphi(a) \in \mathbb{Z}_q^{N \times N}$ *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* $N, q, t \in \mathbb{N}$ *be such that* $P > 3 \cdot 10^3 q N^{5/2}$ *and* $t > 4 \cdot 10^3 \sqrt{N}$*. Then, there is a proof system for* $R_{N,q,t,\mathbf{c},\mathbf{A}}$ *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. Note that (1) can be written as a conjunction of constraints $\bigwedge_{i \in [N]} \langle \mathbf{r}, \mathbf{M}_i \rangle + \langle \mathbf{e}, 0^{i-1} 1 0^{N-i} \rangle + \langle \mathbf{w}, 0^{i-1}(-q)0^{N-i} \rangle = \langle \mathbf{c}, 0^{i-1} 1 0^{N-i} \rangle$ which by taking random linear combinations as in Gentry et al. [28] 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 7.1). Next, we show that constraints (2,3) can be reduced to an inner product constraint. First not that (2) satisfies the constraints of Lemma 4, as $40000\sqrt{N} = 2500 \cdot 16\sqrt{N} < t$, and therefore it is equivalent to a constraint $\langle \mathbf{e}|\mathbf{y}|\mathbf{c}, \mathbf{b}' \rangle = c'$, with public $\mathbf{b}', c'$, along with a vector $\mathbf{z}$ with norm bounded by $t/2$. Finally, note that again we can apply Lemma 4 to handle (3), given that $24000qN^3 < P$. The reason, as mentioned above, is that $||\mathbf{r}|\mathbf{w}||$ is bounded by $N + 16/q$. Let $\langle \mathbf{r}|\mathbf{w}, \mathbf{b}'' \rangle = c''$, with public $\mathbf{b}'', c''$. be the resulting constraint. We can then apply another linear combination with random challenges to merge all three constraints into a final inner product constraint of the form $\langle (\mathbf{r}|\mathbf{e}|\mathbf{w}|\mathbf{c}), \hat{\mathbf{b}} \rangle = \hat{c}$, with public $\hat{\mathbf{b}}, \hat{c}$. Then the proof consists on proving the validity of this one constraint of length $4N$, along with the fact that $||\mathbf{z}|| < t/2$ $||\mathbf{z}'|| < P/12Nq$, for vectors $\mathbf{z}, \mathbf{z}'$ in $[t]^{128}$ and $[P/6Nq]^{128}$, respectively. By offloading this proof to Bulletproofs we get the costs in the statement of the Lemma. $\square$

B Details and Proofs of Security for Semi-Honest Protocol

In this section we provide the security theorems and proofs for the constructions from Section 6.

B.1 Decryptor

Recall that in this section we assume an adversary with access to the protocol transcript of a semi-honest server and fully controlling no more than $t-1$ decryptors. This in particular means that malicious decryptors might correlate their behaviour with the transcript of the server. This situation introduces some subtleties

Distributed Decryptor Functionality $\mathcal{F}_{\mathsf{D}}$

Key Generation

1. $\mathcal{F}_{\mathsf{D}}$ samples key pairs $(\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow \mathsf{AHE.KeyGen}$ for all $j \in \mathcal{H}$, and sends $\mathsf{pk}_{\mathcal{H}} = \sum_{j\in\mathcal{H}} \mathsf{pk}_j$ to $\mathcal{A}$.
2. $\mathcal{F}_{\mathsf{D}}$ receives from $\mathcal{A}$ key pairs $\{(\mathsf{sk}_k, \mathsf{pk}_k)\}_{k\in\mathcal{C}}$.
3. $\mathcal{F}_{\mathsf{D}}$ outputs $\mathsf{pk} = \mathsf{pk}_{\mathcal{H}} + \sum_{k\in\mathcal{C}} \mathsf{pk}_k$ to S.

Decryption

4. $\mathcal{F}_{\mathsf{D}}$ receives ct^1 from S.
5. If $\mathcal{F}_{\mathsf{D}}$ receives `abort` from $\mathcal{A}$, then aborts without output anything.
6. Otherwise, $\mathcal{F}_{\mathsf{D}}$ receives $e_{\mathcal{A}}$ from $\mathcal{A}$. If $\|e_{\mathcal{A}}\|_\infty > \Delta(m-h)/m$ for $h = |\mathcal{H}|$, then $\mathcal{F}_{\mathsf{D}}$ aborts without output anything.
7. $\mathcal{F}_{\mathsf{D}}$ outputs $\mathsf{pd} = \sum_{j\in\mathcal{H}} \mathsf{AHE.PartialDec}(\mathsf{ct}^1, \mathsf{sk}_j) + \sum_{k\in\mathcal{C}} \mathsf{AHE.PartialDec}(\mathsf{ct}^1, \mathsf{sk}_k) + e_{\mathcal{A}}$ to Coord.

Figure 12: The Decryptor ideal functionality, with $\mathcal{H}$ the set of honest decryptors and $\mathcal{C}$ the set of corrupted decryptors.In the case where the adversary $\mathcal{A}$ does not corrupt any decryptors then the messages from $\mathcal{A}$ are empty and therefore it does not have any influence.

that need to be reflected in the ideal functionality. Concretely, the ideal functionality allows an adversary that controls at least one decryptor to modify an honestly sampled public key. This is captured in steps 3-4 of the ideal functionality. While this is a technical detail in the decryptor ideal functionality, this sort of situation will lead to security considerations when discussing the server. Other than that, the ideal functionality does what one would expect: generate public key and decrypt with the corresponding secret key. The decryption is done with noise corresponding to a committee of m decryptors.

The ideal functionality implemented by the decryptor role is presented in Figure 12. This definition captures the ideal-world notion of privacy with respect to which we prove security of our protocols, i.e., we prove that our protocol leaks to the adversary no more that the ideal functionality.

Theorem 3. *Assume* Coord *is semi-honest and no more than* $t-1$ *decryptors are actively corrupted. Furthermore, assume* $\mathsf{RLWE}_{N_2,q_2,\chi_s,\chi_e}$ *is* κ*-bit hard, and let* AHE *be as in Section 5.2 instantiated over ring* R_{q_2} *with secret distribution* χ_s*, error distribution* χ_e*, and a flooding noise distribution* χ_{flood} *defined as*

- *Let* β *be an upper bound on* $\|s \cdot \sum_{i=1}^n e_i + f \cdot \sum_{i=1}^n v_i\|_\infty$ *for* $e_1, \ldots, e_n, f \leftarrow \chi_e$ *and* $s, v_1, \ldots, v_n \leftarrow \chi_s$*.*
- *Let* $s_{\mathsf{flood}} = \sqrt{24N_2}2^{\lambda/2} \cdot \beta$*, and let* χ_{flood} *be the discrete Gaussian distribution with parameter* s_{flood}*.*

Then, the Decryptor functionality (Figure 12) is implemented securely with abort by the protocol in Figure 4.

For efficiency, the Decryptor protocol have a 1-round setup, and 2-round decryption, running in $O(m + \log n)$*, where* m *is the number of decryptors.*

Proof. Assume that the honest decryptors are the set $\mathcal{H} \subseteq [m]$, and the malicious decryptors are $\mathcal{C} \subset [m]$. Let $\mathcal{A}$ be any adversary controlling the malicious decryptors and the server (semi-honestly).

For any real world adversary $\mathcal{A}$, we build the following simulator Sim that plays the role of an adversary with the ideal functionality (Figure 12). For this purpose, Sim has oracle access to $\mathcal{A}$, and it simulates honest decryptors and a semi-honest Coord in interaction with malicious decryptors controlled by $\mathcal{A}$.

1. On input $\mathsf{pk}_{\mathcal{H}}$ received from $\mathcal{F}_{\mathsf{D}}$, Sim samples random but correlated $\{\tilde{\mathsf{pk}}_j\}_{j\in\mathcal{H}}$ such that $\sum_{i\in\mathcal{H}} \tilde{\mathsf{pk}}_j = \mathsf{pk}_{\mathcal{H}}$, and for each $j \in \mathcal{H}$, Sim invokes the ZK simulator to generate a proof $\tilde{\pi}_j$. Then Sim sends $\{(\tilde{\mathsf{pk}}_j, \tilde{\pi}_j)\}_{j\in\mathcal{H}}$ to the server.

2. Sim runs $\mathcal{A}$ and receives the malicious decryptors' messages $\{(\mathsf{pk}_k, \pi_k)\}_{k\in\mathcal{C}}$. Let $\hat{\mathcal{C}}$ be the set of malicious decryptors whose proof π_k verifies. For all $k \in \hat{\mathcal{C}}$, Sim invokes the ZK extractor on π_k to extract sk_k. Then Sim sends $\{(\mathsf{sk}_k, \mathsf{pk}_k)\}_{k\in\mathcal{C}}$ to $\mathcal{F}_{\mathsf{D}}$.

3. Sim receives pk from $\mathcal{F}_{\mathsf{D}}$. This finishes the key generation phase.

4. At the beginning of the decryption phase, Sim samples symmetric encryption keys $\mathbf{k}_j$ for all $j \in \mathcal{H}$, and for all $j \in \mathcal{H}$ and for all $k \in \mathcal{C}$, it samples uniformly random share_k^j and sends them to $\mathcal{A}$. Sim then receives the secret shares $\{\mathsf{share}_j^k\}_{k\in\mathcal{C},j\in\mathcal{H}}$ of malicious decryptors' symmetric encryption keys from $\mathcal{A}$. Since $|\mathcal{H}| \geq t$, Sim recovers $\mathbf{k}_k$ from $\{\mathsf{share}_j^k\}_{j\in\mathcal{H}}$ for all $k \in \mathcal{C}$.

5. Sim receives ct^1 from the server. For all $j \in \mathcal{H}$, Sim simulates proofs of partial decryptions τ_j on the fake partial decryption value 0, and sets $\bar{\mathsf{pd}}_j \leftarrow \mathsf{Enc}(\mathbf{k}_j, (0, \tau_j))$. Sim then sends $\{\bar{\mathsf{pd}}_j\}_{j\in\mathcal{H}}$ to the server.

6. Sim receives malicious decryptors' messages $\{\bar{\mathsf{pd}}_k\}_{k\in\mathcal{C}}$, and for all $k \in \mathcal{C}$, Sim decrypts $\bar{\mathsf{pd}}_k$ using $\mathbf{k}_k$ reconstructed in Step 4 and obtains (pd_k, τ_k). If there exists $k \in \mathcal{C}$ such that τ_k does not verify, then Sim sends `abort` to $\mathcal{F}_{\mathsf{D}}$.

We now show that the real world is indistinguishable from the ideal world, using the following hybrids.

- $\mathsf{Hyb}^{(0)}$: This is the real world execution, and the view consists of the AHE public parameter u, honest decryptors' public key shares $\{\mathsf{pk}_j\}_{j\in\mathcal{H}}$ and proofs of correct public key shares $\{\pi_j\}_{j\in\mathcal{H}}$, the public key $\mathsf{pk} = \sum_{j\in\mathcal{H}} \mathsf{pk}_j + \sum_{k\in\mathcal{C}} \mathsf{pk}_k$, the ciphertext ct, the symmetric encryptions $\{\bar{\mathsf{pd}}_j\}_{j\in\mathcal{H}}$ where $\bar{\mathsf{pd}}_j = \mathsf{Enc}(\mathbf{k}_j, (\mathsf{pd}_j, \tau_j))$, secret shares of the honest symmetric keys $\{\mathsf{share}_k^j\}_{j\in\mathcal{H},k\in\mathcal{C}}$, and the partial decryption output $\mathsf{pd} = \sum_{j\in\mathcal{H}} \mathsf{pd}_j + \sum_{k\in\mathcal{C}} \mathsf{pd}_k$. In particular, they satisfy
  - For all $j \in \mathcal{H}$, $\mathsf{pk}_j = -\mathsf{u}\cdot\mathsf{sk}_j + e_j$ for some $e_j \sim \chi_e$, where u is the public random polynomial of AHE;
  - $\mathsf{ct} = (\mathsf{ct}^0, \mathsf{ct}^1)$ where $\mathsf{ct}^0 = \mathsf{pk}\cdot v + e' + \Delta\cdot\mathbf{x}$ and $\mathsf{ct}^1 = \mathsf{u}\cdot v + e''$, for plaintext $\mathbf{x}$, encryption randomness v, and error terms e', e'';
  - For all $j \in \mathcal{H}$, $\mathsf{pd}_j = \mathsf{sk}_j \cdot \mathsf{ct}^1 + e_j'''$ for $e_j''' \sim \chi_{\mathsf{flood}}$.

- $\mathsf{Hyb}^{(1)}$: In this hybrid we set $\mathsf{pd}_j = -v\cdot\mathsf{pk}_j + e_j'''$ for all $j \in \mathcal{H}$. 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 assumptions on the l_∞ norm of $v\cdot e_j + \mathsf{sk}_j\cdot e''$ and the flooding noise distribution, pd_j in the above expression is statistically close to $\mathsf{pd}_j = -v\cdot\mathsf{pk}_j + e_j'''$ as in $\mathsf{Hyb}^{(1)}$.

- $\mathsf{Hyb}^{(2)}$: In this hybrid we first compute $\mathsf{sk}_{\mathcal{H}} = \sum_{j\in\mathcal{H}} \mathsf{sk}_j$ and $\mathsf{pk}_{\mathcal{H}} = -\mathsf{u}\cdot\mathsf{sk}_{\mathcal{H}} + \sum_{j\in\mathcal{H}} e_j$ for all $e_j \leftarrow \chi_e$; then we sample random pk_j for all $j \in \mathcal{H}$ conditioned on $\sum_{j\in\mathcal{H}} \mathsf{pk}_j = \mathsf{pk}_{\mathcal{H}}$. Furthermore, in this hybrid we simulate proofs of correct public key shares π_j. This hybrid is indistinguishable from $\mathsf{Hyb}^{(1)}$ because pk_j in $\mathsf{Hyb}^{(1)}$ are pseudorandom conditioned on the sum of pk_j's included in pk, and because of the fact that $\mathsf{Hyb}^{(1)}$ does not depend on sk_j for $j \in \mathcal{H}$, and simulated proofs are indistinguishable from real proofs in $\mathsf{Hyb}^{(1)}$.

- $\mathsf{Hyb}^{(3)}$: In this hybrid we replace $\{\mathsf{share}_k^j\}_{j\in\mathcal{H},k\in\mathcal{C}}$ with random and independent elements, which are the secret shares of $\mathbf{k}_j$ sent to the malicious decryptors. By assumption that $|\mathcal{C}| < t$, the corresponding secret shares in $\mathsf{Hyb}^{(2)}$ are indistinguishable from random and independent elements. So this hybrid is indistinguishable from $\mathsf{Hyb}^{(2)}$.

- $\mathsf{Hyb}^{(4)}$: In this hybrid we set for all $j \in \mathcal{H}$ that $\bar{\mathsf{pd}}_j \leftarrow \mathsf{Enc}(\mathbf{k}_j, (0, \tau_j))$ where τ_j is the simulated proof of partial decryption, and we set $\mathsf{pd} = \sum_{k \in \mathcal{C}} \mathsf{pd}_k + \mathsf{ct}^1 \cdot \mathsf{sk}_{\mathcal{H}} + \sum_{j \in \mathcal{H}} e_j'''$. Note that, except for $\{\bar{\mathsf{pd}}_j\}_{j \in \mathcal{H}}$, other components in $\mathsf{Hyb}^{(3)}$ do not depend on the symmetric encryption keys $\mathbf{k}_j$. So, by the semantic security of Enc, the encryptions $\bar{\mathsf{pd}}_j$ in this hybrid are indistinguishable from those in $\mathsf{Hyb}^{(3)}$. Note that this is exactly the ideal world execution, where pk_H is generated as in $\mathcal{F}_{\mathsf{D}}$, $\bar{\mathsf{pd}}_j$ for all $j \in \mathcal{H}$ are generated as in Sim, and pd is computed in the same way as in $\mathcal{F}_{\mathsf{D}}$.

For efficiency, since we aggregate n KAHE secret keys under AHE, the ciphertext modulus q_2 must be $O(\log n)$, and thus each decryptor takes $O(\log n)$ time for their AHE operations. Each decryptor in addition secret shares their randomness for generating AHE secret keys with all m decryptors; hence the extra $O(m)$ running time. $\square$

B.2 Server and Clients

As in the previous section, we start by describing our ideal functionality. Intuitively, Functionality $\mathcal{F}_{\mathsf{agg}}$ from Figure 1 is what we would like to be able to implement. However, the following theorem shows that this is impossible within the other constraints we are aiming for, namely cost for D that is sublinear in input length ℓ, and therefore cost for server after receiving inputs sublinear in ℓ.

Theorem 4. *There does not exist a protocol for vector aggregation (c.f. Functionality 1) in the standard model with asynchronous client contributions and total communication after the client contributions sublinear in vector length, that is secure against a semi-honest server colluding with one malicious client.*

Proof. To formally prove this claim, we show that, for any protocol π, there exists a real-world adversary $\mathcal{A}_R$ that can't be successfully simulated in the ideal world, i.e. for which a distinguisher D with non-negligible advantage in distinguishing $\mathrm{IDEAL}_{\mathcal{F}^{\mathsf{Sim}}}\big((\mathbf{x_i})_i, \lambda\big)$ and $\mathrm{REAL}_{\pi^{\mathcal{A}_R}}\big((\mathbf{x_i})_i, \lambda\big)$ for every simulator Sim, where $\mathcal{F}^{\mathsf{Sim}}$ denotes the functionality of Figure 1 with Sim playing the role of the adversary $\mathcal{A}$. Recall (Definitions 7 and 8 in Section D) that for a semi-honest server $\mathrm{IDEAL}_{\mathcal{F}^{\mathsf{Sim}}}\big((\mathbf{x_i})_i, \lambda\big) = (\mathtt{output}_{\mathsf{S}_I}, \mathtt{output}_{\mathsf{Sim}})$. i.e., the *joint* distribution of server and adversary outputs when interacting with the ideal functionality $\mathcal{F}^{\mathsf{Sim}}$, and $\mathrm{REAL}_{\pi^{\mathcal{A}_R}}\big((\mathbf{x_i})_i, \lambda\big) = (\mathtt{output}_{\mathsf{S}}, \mathtt{output}_{\mathcal{A}_R})$, i.e. the *joint* distribution of server and adversary outputs when running protocol π.

Next, we define $\mathcal{A}_R$. This attacker only needs to control one malicious client c, and semi-honestly/passively corrupt the server. $\mathcal{A}_R$ proceeds by observing the transcript/view $\mathcal{V}_{\mathsf{S}}$ of the server. Let us write $\mathcal{V}_{\mathsf{S}}$ as $\mathcal{V}_{\mathsf{S},1} || \mathcal{V}_{\mathsf{S},2} || \mathtt{output}_{\mathsf{S}}$, where $\mathcal{V}_{\mathsf{S},1}$ denotes the view of the server until the last honest client submits their last input message. Note that this is well defined for any asynchronous protocol π. Also, recall that $\mathcal{V}_{\mathsf{S},2}$ is sublinear in ℓ. Concretely, $\mathcal{A}_R$ instructs c to wait until all honest clients have reported, and runs the client code of π with input $\mathbf{x_c} := H(\mathcal{V}_{\mathsf{S},1})$, where H is a collision-resistant hash function with output space matching the space of inputs $\mathbb{F}^\ell$. $\mathcal{A}_R$ outputs the server's view, i.e., $\mathtt{output}_{\mathcal{A}_R} = \mathcal{V}_{\mathsf{S}}$. Note that the distinguisher D can recover $\mathbf{x_c}$ from any view $\mathcal{V}_{\mathsf{S}}$ it is given.

For the sake of reaching a contradiction, let Sim be a successful simulator, i.e. one such that $\mathrm{IDEAL}_{\mathcal{F}^{\mathsf{Sim}}}\big((\mathbf{x_i})_i, \lambda\big) = (\mathtt{output}_{\mathsf{S}_I}, \mathtt{output}_{\mathsf{Sim}})$ is indistinguishable $\mathrm{REAL}_{\pi^{\mathcal{A}_R}}\big((\mathbf{x_i})_i, \lambda\big)$. Note that since $\mathtt{output}_{\mathsf{S}}$ includes $\mathbf{x_c}$ then Sim must have provided $\mathbf{x_c}$ to the functionality in step 6, otherwise $\mathtt{output}_{\mathsf{S}}$ and $\mathtt{output}_{\mathsf{S}_I}$ would differ. By the collision-resistance property of H, $\mathbf{x}$ acts as a commitment to $\mathcal{V}_{\mathsf{S},1}$. Only after making that "commitment" the functionality enables Sim to recover the output that the server receives to Sim, by revealing $\mathbf{k}$ in the last step. Let $\mathtt{output}^{\mathsf{Sim}}$ denote that output, i.e. $\mathsf{KAHE.Dec}(\tilde{s}, \mathbf{k}) + \mathbf{x_c}$. Note that since $\mathcal{A}_R$ outputs the view of the server, Sim should do that too, otherwise $\mathtt{output}_{\mathsf{Sim}}$ and $\mathtt{output}_{\mathcal{A}_R}$) would differ. Let $\mathcal{V}_{\mathsf{S}}^{\mathsf{Sim}} = \mathcal{V}_{\mathsf{S},1}^{\mathsf{Sim}} || \mathcal{V}_{\mathsf{S},2}^{\mathsf{Sim}} || \mathtt{output}_{\mathsf{S}}^{\mathsf{Sim}}$ be the view for the server that Sim hands to the distinguisher.

As mentioned above, $\mathtt{output}^{\mathsf{Sim}}$ must include $\mathbf{x_c} = H(\mathcal{V}_{\mathsf{S},1})$, but the set of possible values of $\mathtt{output}^{\mathsf{Sim}}$ that include $\mathbf{x_c}$ is bounded by $2^{|\mathcal{V}_{\mathsf{S},2}^{\mathsf{Sim}}|}$. This is because $\mathbf{x_c}$ is determined by the first portion of the view, i.e. $\mathcal{V}_{\mathsf{S},1}^{\mathsf{Sim}} = \mathcal{V}_{\mathsf{S},1}$. On the other hand, the set of possible values of $\mathbf{x_c}$ is $\mathbb{F}^\ell$ (as D could have chosen any element of $\mathbb{F}^\ell$ for the sum of the honest clients inputs). Since $|\mathcal{V}_{\mathsf{S},2}^{\mathsf{Sim}}|$ is sublinear in ℓ, a distinguisher D that simply checks whether $\mathbf{x_c}$ is in the output must have non-negligible success probability. Therefore, Sim can't be successful: a contradition. $\square$

We will therefore instead consider the relaxation of the aggregation functionality shown in Figure 11. We show security with respect to the relaxed aggregation functionality next. The proof is in the "decryptor"-hybrid model, as it assumes a secure implementation of the decryptor.

Theorem 5. *Given an honest* D *implementing Functionality 12 and an at least semi-honest server, the protocol formed by figures 5 and 6 securely implements the relaxed aggregation functionality (Functionality 11).*

Furthermore, the Client role runs in 1 *round with cost* $O(\ell \log n)$*, and the server runs in* $O(n\ell \log n)$*.*

Proof of Theorem 5. Assume the decryptor protocol in Figure 4 securely implements the decryptor functionality $\mathcal{F}_{\mathsf{D}}$ of Figure 12. Then, for any decryptor adversary there exists a decryptor simulator. In particular, consider a dummy adversary $\mathcal{A}_{\mathsf{D}}$ that simply passes through the messages from the distinguisher and decryptors / coordinator; then there exists $\mathsf{Sim}_{\mathsf{D}}$ that can simulate this dummy adversary.

Consider the protocol in Figures 5 and 6 that uses the decryptor protocol as a secure implementation of D. Let $\mathcal{A}$ be an adversary to this protocol. Let $\mathcal{C}$ and $\mathcal{H}$ be the sets of corrupted and honest clients, and assume w.l.o.g. $1 \in \mathcal{H}$. We build the following simulator Sim in the $\mathcal{F}_{\mathsf{D}}$-hybrid model, which interacts with $\mathcal{F}_{agg}$ and has oracle access to $\mathcal{A}$ and $\mathsf{Sim}_{\mathsf{D}}$.

1. Sim simulates server following the server protocol, and runs $\mathcal{A}$ such that any interactions with decryptors and Coord are passed through $\mathsf{Sim}_{\mathsf{D}}$. At the end of the key generation phase, $\mathcal{F}_{\mathsf{D}}$ outputs pk to the simulated server.

2. Sim receives $\tilde{m}$ from $\mathcal{F}_{agg}$, which is an KAHE encryption of $\sum_{i\in\mathcal{H}} \mathbf{x_i}$ under key $\mathbf{k}$. Note that $\mathbf{k}$ is private to $\mathcal{F}_{agg}$ at this point.

3. Sim simulates all honest clients: Let Ctxt be the KAHE ciphertext space.

   - for all $i \neq 1$, $m_i \leftarrow \mathsf{Ctxt}$, and $m_1 = \tilde{m} - \sum_{i\neq 1} m_i$;
   - for all $i \in \mathcal{H}$, $r_i \leftarrow \{0,1\}^\lambda, \mathsf{ct}_i \leftarrow \mathsf{AHE.Enc}(0, \mathsf{pk}, r_i)$;
   - for all $i \in \mathcal{H}$, $p_i \leftarrow \mathsf{Prove}_{\mathsf{AHE.Enc}}(\mathsf{ct}_i, 0, \mathsf{pk}, r_i, \perp)$.

   In particular, the proofs p_i are generated honestly.

4. Sim sends $\{(m_i, \mathsf{ct}_i, p_i)\}_{i\in\mathcal{H}}$ to $\mathcal{A}$ as honest clients messages to the server.

5. Sim then runs $\mathcal{A}$ to let it generate $\{(m_i, \mathsf{ct}_i, p_i)\}_{i\in\mathcal{C}}$ which are messages sent from the corrupted clients to the server. Let $\hat{C}$ be the subset of $\mathcal{C}$ containing corrupted clients i whose messages are valid and p_i verifies.

6. Sim simulates the server and computes $m = \sum_{i\in\hat{C}} m_i + \tilde{m}$.

7. Sim simulates the server and computes $\mathsf{ct}^1 = \sum_{i\in\hat{C}} \mathsf{ct}_i^1 + \sum_{i\in\mathcal{H}} \mathsf{ct}_i^1$, and sends ct^1 to $\mathcal{F}_{\mathsf{D}}$. Note that $\mathsf{ct} = (\mathsf{ct}^0, \mathsf{ct}^1)$ should decrypt to the same value as $\sum_{i\in\hat{C}} \mathsf{ct}_i$.

8. Sim then runs $\mathcal{A}$ to interact with the decryptor protocol, where the messages between $\mathcal{A}$ and the malicious decryptors are passed through $\mathsf{Sim}_{\mathsf{D}}$ who interacts with $\mathcal{F}_{\mathsf{D}}$. If $\mathsf{Sim}_{\mathsf{D}}$ aborts, Sim sends $\mathtt{abort}$ to $\mathcal{F}_{agg}$.

9. Sim receives pd as the message from $\mathcal{F}_{\mathsf{D}}$ to the server, and computes $\mathbf{k}' = \mathsf{AHE.Recover}(\mathsf{pd})$. Note that this message pd received by Sim is not sent to the simulated server.

10. Sim decrypts $\sum_{i\in\hat{C}} m_i$ using $\mathbf{k}_{\mathcal{A}}$ and gets $\mathbf{x}_{\hat{C}}$, which is the sum of inputs of corrupted clients in $\hat{C}$. Sim then sends $(\mathbf{x}_{\hat{C}}, 0, \ldots, 0)$ to $\mathcal{F}_{agg}$, and receives $\mathbf{k}$ and $s = \sum_{i\in\mathcal{H}} \mathbf{x_i} + \mathbf{x}_{\hat{C}}$.

11. Sim then sets $\mathsf{pd}' = \mathsf{pd} - \sum_{i\in\mathcal{H}} \mathbf{k}_i + \mathbf{k}$, and sends pd' to the simulated server as the partial decryption output from $\mathcal{F}_{\mathsf{D}}$.

12. Sim then runs $\mathcal{A}$ and Sim_D to the end, and outputs the output of $\mathcal{A}$.

We now prove that the real world (consisting of clients, server, decryptors, and $\mathcal{A}$) is indistinguishable from the ideal world (consisting of Sim, $\mathcal{F}_{agg}$, $\mathcal{F}_\mathsf{D}$, and Sim_D).

- $\mathsf{Hyb}^{(0)}$: This is the real world execution consisting of the clients, server, decryptors, and $\mathcal{A}$. A transcript contains the KAHE public parameter a, AHE public key pk, honest clients' encoded input $\{m_i = \mathsf{KAHE.Enc}(\mathbf{x_i}, \mathbf{k}_i)\}_{i\in\mathcal{H}}$ and AHE ciphertexts $\{\mathsf{ct}_i = \mathsf{AHE.Enc}(\mathbf{k}_i, \mathsf{pk})$, and partial decryptions from honest decryptors $\{\mathsf{pd}_j\}_{j\le h}$.

- $\mathsf{Hyb}^{(1)}$: In this hybrid we add a dummy adversary $\mathcal{A}_\mathsf{D}$ who simply passes through messages from and to $\mathcal{A}$ intending to the decryptors. This is exactly the same as $\mathsf{Hyb}^{(0)}$.

- $\mathsf{Hyb}^{(2)}$: In this hybrid we replace the decryptors with $\mathcal{F}_\mathsf{D}$, and $\mathcal{A}_D$ with Sim_D. By the assumption that the decryptors securely implement $\mathcal{F}_\mathsf{D}$, this hybrid is indistinguishable from $\mathsf{Hyb}^{(0)}$. Note that the transcript of this hybrid is the same as in $\mathsf{Hyb}^{(0)}$ and $\mathsf{Hyb}^{(1)}$ except that partial decryptions $\{\mathsf{pd}_j\}_{j\le h}$ are generated from Sim_D on input pd, which is given to Sim_D by $\mathcal{F}_\mathsf{D}$; that is, $\{\mathsf{pd}_j\}_{j\le h}$ in $\mathsf{Hyb}^{(2)}$ are independent of honest decryptors' AHE secret keys.

- $\mathsf{Hyb}^{(3)}$: In this hybrid we replace honest clients' KAHE ciphertexts using random but correlated values. Specifically, we set $m_i = u_i$ for all $i \in \mathcal{H} \setminus \{1\}$ and $m_1 = -\sum_{i\in\mathcal{H}\setminus\{1\}} u_i + \sum_{i\in\mathcal{H}} \mathbf{x_i}$, where u_i are independently and uniformly sampled elements from the KAHE's ciphertext space. Note that $\mathcal{A}$ learns the sum of honest clients' KAHE secret keys $\mathbf{k} = \sum_{i\in\mathcal{H}} \mathbf{k}_i$. By the special leakage property of KAHE as described in Lemma 1, we have that the joint distribution of a, $\{m_i\}_{i\in\mathcal{H}}$, and $\mathbf{k} = \sum_{i\in\mathcal{H}} \mathbf{k}_i$ in $\mathsf{Hyb}^{(2)}$ (which are KAHE ciphertexts) is indistinguishable from that in $\mathsf{Hyb}^{(3)}$ (which are random but correlated elements) due to the HintRLWE assumption.

- $\mathsf{Hyb}^{(4)}$: In this hybrid we replace honest clients' AHE ciphertexts using independent AHE ciphertexts encrypting 0. Specifically, we $\mathsf{ct}_i \leftarrow \mathsf{AHE.Enc}(0, \mathsf{pk})$ for all $i \in \mathcal{H}$, and we generate the proofs p_i accordingly. This hybrid is indistinguishable with $\mathsf{Hyb}^{(3)}$ due to the semantic security of AHE: the AHE secret key $\mathsf{sk} = \mathsf{sk}_\mathsf{H} + \mathsf{sk}_\mathsf{C}$ is hidden from the adversary and thus the distinguisher, where H is the set of honest decryptors. Note that the partial decryption pd is the same as in $\mathsf{Hyb}^{(3)}$, which reconstructs to $\mathbf{k}_\mathcal{A} + \sum_{i\in\mathcal{H}} \mathbf{k}_i$. Note that $\mathsf{Hyb}^{(4)}$ is exactly the transcript of the ideal world.

For efficiency, the KAHE ciphertext modulus must be at least $O(\log n)$ large to support sum of n input values, and since each input $\mathbf{x_i} \in \mathbb{F}^\ell$, it takes $O(\ell \log n)$ time for each client to encrypt its input. The server aggregates KAHE ciphertexts from all n clients, so its running time is $O(n\ell \log n)$. $\square$

C Details and Proofs of Security for Malicious Protocol

In this section we include details of the variant of our protocol that is secure against an actively corrupted server.

Figure 13 presents the decryptor protocol, and we present the formal definitions of the Server and the Client roles in Figure 14 and Figure 15.

C.1 Sybil attacks and Differential Privacy

To prevent revealing the sum of just one client's data the verifier could check that many clients are included in the sum. However, a malicious server could easily fake many malicious clients in a Sybil attack. If we have no means of verifying the identity of clients this attack can't be avoided. We could, however, change the functionality to include noise for Differential Privacy (DP). It is easy to extend our protocol to add that in. Using infinite divisibility of common distributions used for DP [39, 40] any group of entities we trust most of to be honest can provide noise contributions (just like the client provide input) and the verifier can check they are included. In particular the decryptors or verifiers themselves could provide it (though it would require order ℓ communication from them).

C.2 Proof of Malicious Security

Recall that the properties we require of the verifier are:

1. They refuse to sign any ciphertext that isn't a sum of honest contributions at most once and some ciphertext of which the server knows the plaintext.
2. They refuse to sign more than one ciphertext.
3. Upon signing a ciphertext they can report which zero-knowledge proofs (from the clients) were used in it.

Theorem 6. *Assume* KAHE *is an KAHE scheme satisfying leakage-resilient security of Definition 1, and assume* AHE *is an AHE scheme as in Section 5.2 instantiated over ring* R_{q_2} *with secret distribution* χ_s*, error distribution* χ_e*, and a flooding noise distribution* χ_{flood} *defined as*

- *Let* β *be an upper bound on* $\|s \cdot \sum_{i=1}^{n} e_i + f \cdot \sum_{i=1}^{n} v_i\|_\infty$ *for* $e_1, \ldots, e_n, f \leftarrow \chi_e$ *and* $s, v_1, \ldots, v_n \leftarrow \chi_s$*.*
- *Let* $s_{\mathsf{flood}} = \sqrt{24N_2}2^{\lambda/2} \cdot \beta$*, and let* χ_{flood} *be the discrete Gaussian distribution with parameter* s_{flood}*.*

Furthermore, assume there exist an symmetric encryption scheme Sym *and a zero-knowledge proof of knowledge scheme, and assume there exist* c *clients grouped in* c/k *committees of size* $k = O(\sigma + \log(c/k))$ *each, where* σ *is the statistical security parameter and each group is randomly selected to implement the Verifier for the proof of unique inclusion of AHE ciphertexts as in Section 7.1.*

Then, the protocol formed by Figure 15 (Client), Figure 14 (Server), and Figure 13 (Decryptor) securely implements the aggregation functionality in Figure 1 in the random oracle model against a malicious adversary controlling the server, any number of asynchronous clients, at most $\min(m-1, 2t-m-1)$ *decryptors, and at most* $t-1$ *verifiers per committee.*

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* $O(m + \log n)$*, where* m *is the number of decryptors. The server runs in* $O(n\ell \log n)$*. Each verifier committee member runs in* $O(nk/c + \log n)$*.*

We now prove Theorem 6 in the (Verifier, RO)-hybrid model.

Proof of Theorem 6. 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. By assumption we have $|\mathcal{H}|, |\mathcal{H}_D| \geq t$. Without loss of generality, we assume that $1 \in \mathcal{H}$. Assume $\mathcal{A}$ is any real world adversary to the protocol. We build the following simulator 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.

1. For all $j \in \mathcal{H}_D$, Sim simulates honest decryptor j:
   - $(\mathsf{sk}_j, \mathsf{pk}_j) \leftarrow \mathsf{AHE.KeyGen}(r_j)$ for random r_j;
   - $\mathsf{share}^j \leftarrow \mathsf{Share}(r_j)$;
   - $(\mathsf{sk}_{\mathtt{mask},j}, \mathsf{pk}_{\mathtt{mask},j}) \leftarrow \mathsf{AHE.KeyGen}(r_{\mathtt{mask},j})$ for random $r_{\mathtt{mask},j}$;
   - $\mathsf{share}^{\mathtt{mask},j} \leftarrow \mathsf{Share}(r_{\mathtt{mask},j})$;
   - sends $h_j = (\mathsf{pk}_j, \{\mathsf{share}^j_k\}_{k \in \mathcal{C}_D}, \mathsf{pk}_{\mathtt{mask},j}, \{\mathsf{share}^{\mathtt{mask},j}_k\}_{k \in \mathcal{C}_D})$ and a signature $\mathrm{sig}(h_j)$ on h_j to $\mathcal{A}$
2. For all $k \in \mathcal{C}_D$, Sim receives $\{\mathsf{share}^k_j\}_{j \in \mathcal{H}_D}$ and $\{\mathsf{share}^{\mathtt{mask},k}_j\}_{j \in \mathcal{H}_D}$, as malicious decryptors' shares on their AHE key generation randomness, from $\mathcal{A}$.

3. Sim receives $\{S^j\}_{j \in \mathcal{H}_D}$ from $\mathcal{A}$, as the broadcast messages to all $j \in \mathcal{H}_D$, where S^j should contain decryptors' messages h_ℓ for all $\ell \in \mathsf{D}$.
   - For all $j \in \mathcal{H}_D$, if $|S^j| < t$ or there exists a proof in S^j that does not verify, Sim removes j from $\mathcal{H}_D$ (i.e. j aborts).
4. For all $j \in \mathcal{H}_D$, Sim sends $s_j = \text{sig}(\sum_{\ell \in S^j} \mathsf{pk}_\ell)$ and $s_{\mathsf{mask},j} = \text{sig}(\sum_{\ell \in S^j} \mathsf{pk}_{\mathtt{mask},\ell})$ to $\mathcal{A}$ as honest decryptor j's message. This finishes the key generation phase of the decryptor protocol.
5. For all $i \in \mathcal{H}$, Sim receives $\mathtt{signedpk}_i$ from $\mathcal{A}$ as the AHE public key message to honest client i. Note that all $\mathtt{signedpk}_i$ contains a set S_D of signatures signatures s_j and $s_{\mathsf{mask},j}$ from decryptors j on $\sum_{\ell \in S} \mathsf{pk}_\ell$ and and $\sum_{\ell \in S} \mathsf{pk}_{\mathtt{mask},\ell}$, respectively.
   - For each $i \in \mathcal{H}$, parse $(\mathsf{D}, S_\mathsf{D}, \mathsf{pk}, \mathsf{pk}_{\mathtt{mask}}) = \mathtt{signedpk}_i$. If there are at least t many signatures $s_j \in S_\mathsf{D}$ such that s_j does not verify on pk, or if there are at least t invalid signatures $s_{\mathsf{mask},j} \in S_\mathsf{D}$, then remove i from $\mathcal{H}$ (i.e. client i aborts).
6. For all $i \in \mathcal{H}$, Sim simulates the honest client i:
   - Samples a random seed_i from RO's domain;
   - $\tilde{x}_i \leftarrow \mathsf{RO}(\mathsf{seed}_i)$, which effectively sets the input of simulated client i to 0;
   - $\mathbf{k}_i \leftarrow \mathsf{KAHE.KeyGen}$;
   - $m_i \leftarrow \mathsf{KAHE.Enc}(\tilde{x}_i, \mathbf{k}_i)$;
   - $\mathsf{ct}_i \leftarrow \mathsf{AHE.Enc}(\mathbf{k}_i, \mathsf{pk}; \rho)$ for some randomness ρ;
   - $p_i = \mathsf{Prove}_{\mathsf{AHE.Enc}}(\mathsf{ct}_i, \mathbf{k}_i, \mathsf{pk}, \rho, i)$
   - $\widetilde{\mathsf{seed}}_i \leftarrow \mathsf{AHE.Enc}(\mathsf{seed}_i, \mathsf{pk}_{\mathtt{mask}})$;
   - sends $(m_i, \mathsf{ct}_i, p_i, \widetilde{\mathsf{seed}}_i)$ to $\mathcal{A}$.
7. Let $\mathcal{A}$ query the ideal verifier oracle, receive ct^1 and S from $\mathcal{A}$, and sends $s_{\mathsf{ct}^1} \leftarrow \text{sig}(\mathsf{ct}^1)$ to $\mathcal{A}$.
8. Sim sends S to $\mathcal{F}_{agg}$ to drop the clients in S.
9. Sim then sends $(0, \ldots, 0)$ as malicious clients' inputs to $\mathcal{F}_{agg}$.
10. Sim receives the sum $s_\mathcal{H}$ of inputs of all honest surviving clients and malicious clients from $\mathcal{F}_{agg}$.
11. If $S \neq \emptyset$, let $i^* \in \mathcal{H} \setminus S$ be some honest surviving client, and Sim programs the random oracle such that $\mathsf{RO}(\mathsf{seed}_{i^*}) = s_\mathcal{H} - \tilde{x}_{i^*}$. This finishes the client phase.
12. Sim receives $\mathtt{signedct}_j = (\mathsf{ct}^1, s_{\mathsf{ct}^1})$ from $\mathcal{A}$ as broadcast messages to $j \in \mathcal{H}_D$.
    - For all $j \in \mathcal{H}_D$, if s_{ct^1} in $\mathtt{signedct}_j$ contains any signature that does not pass verification, Sim removes j from $\mathcal{H}_D$ (i.e. j aborts).
13. For all $j \in \mathcal{H}_D$, Sim then simulates honest decryptor j:
    - samples a symmetric encryption key $\mathbf{k}_{\mathsf{sym},j}$;
    - generates $\mathsf{share}^{\mathsf{sym},j} \leftarrow \mathsf{Share}(\mathbf{k}_{\mathsf{sym},j})$;
    - $\mathsf{pd} \leftarrow \mathsf{AHE.PartialDec}(\mathsf{sk}_j, \mathsf{ct}^1)$;
    - $\bar{\mathsf{pd}}_j \leftarrow \mathsf{Sym.Enc}(\mathsf{pd}_j, \mathbf{k}_{\mathsf{sym},j})$;
    - $\pi_j \leftarrow \mathsf{Prove}_{\mathsf{AHE.PartialDec}}(\mathsf{pd}_j, \mathsf{sk}_j)$;
    - sends $(\{\mathsf{share}_k^{\mathsf{sym},j}\}_{k \in \mathcal{C}_D}, \{\mathsf{share}_j^{\mathtt{mask},h}\}_{h \in \mathsf{D}}, \bar{\mathsf{pd}}_j, \pi_j)$ to $\mathcal{A}$.

14. Sim receives $\{\mathsf{share}_j^{\mathsf{sym},k}\}_{j\in\mathcal{H}_D}$ from $\mathcal{A}$ as malicious decryptor k's broadcast messages.
15. Each simulated honest decryptor j who receives a P_j from $\mathcal{A}$ will then:
    - check $|P_j| \geq t$ aborting if this doesn't hold;
    - send $\{\mathsf{share}_j^{\mathtt{dropout}}\}$ for all $\mathtt{dropout} \in \mathcal{H}_D \setminus P_j$ to $\mathcal{A}$;
    - send $\{\mathsf{share}_j^{\mathsf{sym},h}\}$ for all $h \in P_j$ to $\mathcal{A}$.
16. Sim then answers random oracle queries from $\mathcal{A}$.
17. Sim runs $\mathcal{A}$ to the end, and outputs whatever $\mathcal{A}$ outputs.

We now show that the real world execution is indistinguishable from the ideal world.

- $\mathsf{Hyb}^{(0)}$: This is the real world execution. In particular, the view contains the adversary's output, the public AHE random parameter u, and
  - for all $j \in \mathcal{H}_D$, $\mathsf{pk}_j = -\mathsf{u}\cdot\mathsf{sk}_j + e_j$ for some $e_j \sim \chi_e$;
  - $\mathsf{ct} = (\mathsf{ct}^0, \mathsf{ct}^1)$ where $\mathsf{ct}^0 = \mathsf{pk}\cdot v + e' + \Delta\cdot\mathbf{x}$ and $\mathsf{ct}^1 = \mathsf{u}\cdot v + e''$, for plaintext $\mathbf{x}$, encryption randomness v, and error terms e', e'';
  - for all $j \in \mathcal{H}_D$, honest decryptor j computes its partial decryption as $\mathsf{pd}_j = \mathsf{sk}_j \cdot \mathsf{ct}^1 + e_j'''$ for $e_j''' \sim \chi_{\mathsf{flood}}$.
- $\mathsf{Hyb}^{(1)}$: In this hybrid we compute honest decryptor j's partial decryption pd_j using pk_j and the term v used in ct^1, and then we simulate proofs of partial decryptions. Specifically, for all $j \in \mathcal{H}$ we set $\mathsf{pd}_j = -v\cdot\mathsf{pk}_j + e_j'''$ and simulate a proof of partial decryption $\tilde{\pi}_j$ on the value pd_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 assumptions on the l_∞ norm of $v\cdot e_j + \mathsf{sk}_j\cdot e''$ and the flooding noise distribution, pd_j in the above expression is statistically close to $\mathsf{pd}_j = -v\cdot\mathsf{pk}_j + e_j'''$ as in $\mathsf{Hyb}^{(1)}$.
- $\mathsf{Hyb}^{(2)}$: Note that in $\mathsf{Hyb}^{(1)}$, for all honest decryptors $j \in \mathcal{H}_D$, only pk_j depends on sk_j. So in $\mathsf{Hyb}^{(2)}$, 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 of jth public key share on pk_j. That is, in this hybrid we compute $\mathsf{pk}_j = -\mathsf{u}\cdot\mathsf{sk}_j + e_j$ as in $\mathsf{Hyb}^{(1)}$, but we do not include pk_j in the view of the distinguisher. Instead we then compute $\{\tilde{\mathsf{pk}}_j\}_{j\in\mathcal{H}_D} \leftarrow \mathsf{Share}(\sum_{j\in\mathcal{H}_D}\mathsf{pk}_j)$, and include $\tilde{\mathsf{pk}}_j$ in the view. By the pseudorandomness of the public key shares pk_j, this hybrid is indistinguishable from $\mathsf{Hyb}^{(1)}$.
- $\mathsf{Hyb}^{(3)}$: In this hybrid we replace honest partial decryptions pd_j's with random values whose sum is $\sum_{j\in\mathcal{H}_D}\mathsf{pd}_j$. Specifically, we first compute $\mathsf{pd}_{\mathcal{H}_D} = \sum_{j\in\mathcal{H}_D}\mathsf{pd}_j$, and then
  - sample random $\widehat{\mathsf{pd}_j}$ for all $j \in \mathcal{H}_D$ conditioned on $\sum_{j\in\mathcal{H}_D}\widehat{\mathsf{pd}_j} = \mathsf{pd}_{\mathcal{H}_D}$; and
  - for all $j \in \mathcal{H}_D$, set $\bar{\mathsf{pd}}_j \leftarrow \mathsf{Sym.Enc}(\widehat{\mathsf{pd}_j}, \mathbf{k}_{\mathsf{sym},j})$, and we simulate proofs $\tilde{\pi}_j$ on the fake partial decryption value $\widehat{\mathsf{pd}_j}$.

Note that, for all honest $j \in \mathcal{H}_D$, in $\mathsf{Hyb}^{(2)}$ their partial decryptions pd_j are random conditioned on $\sum_{j\in\mathcal{H}_D} \mathsf{pd}_j = -v \cdot \sum_{j\in\mathcal{H}_D} \mathsf{pk}_j + \sum_{j\in\mathcal{H}_D} e_j'''$, where all pk_j are random. So $\{\widehat{\mathsf{pd}_j}\}_{j\in\mathcal{H}_D}$ has the same distribution as $\{\mathsf{pd}_j\}_{j\in\mathcal{H}_D}$ in $\mathsf{Hyb}^{(2)}$, and hence the symmetric encryptions $\bar{\mathsf{pd}}_j$ are identically distributed in $\mathsf{Hyb}^{(2)}$ and $\mathsf{Hyb}^{(3)}$. So, this hybrid is identical to $\mathsf{Hyb}^{(2)}$.

- $\mathsf{Hyb}^{(4)}$: In this hybrid we sample a fresh AHE key pair $(\mathsf{sk}_{\mathcal{H}_D}, \mathsf{pk}_{\mathcal{H}_D})$, and we compute $\widehat{\mathsf{pd}_j}$ for honest j using $\mathsf{pk}_{\mathcal{H}_D}$ and v. Specifically,
  - we sample $(\mathsf{sk}_{\mathcal{H}_D}, \mathsf{pk}_{\mathcal{H}_D}) \leftarrow \mathsf{AHE.KeyGen}$;
  - compute $\mathsf{pd}_{\mathcal{H}_D} = -v \cdot \mathsf{pk}_{\mathcal{H}_D} + \sum_{j\in\mathcal{H}_D} e_j'''$ for $e_j''' \leftarrow \chi_{\mathsf{flood}}$;
  - for all $j \in \mathcal{H}_D$, sample random $\widehat{\mathsf{pd}_j}$ conditioned on $\sum_{j\in\mathcal{H}_D} \widehat{\mathsf{pd}_j} = \mathsf{pd}_{\mathcal{H}_D}$; and
  - for all $j \in \mathcal{H}_D$, set $\bar{\mathsf{pd}}_j \leftarrow \mathsf{Sym.Enc}(\widehat{\mathsf{pd}_j}, \mathbf{k}_{\mathsf{sym},j})$, and we simulate proofs $\tilde{\pi}_j$ on the fake partial decryption value $\widehat{\mathsf{pd}_j}$.

  The difference of this hybrid with $\mathsf{Hyb}^{(3)}$ is that $\mathsf{pk}_{\mathcal{H}_D}$ is now pseudorandom. Note that everything else are computed exactly the same as in $\mathsf{Hyb}^{(3)}$, and the only term depends on $\mathsf{sk}_{\mathcal{H}_D}$ in $\mathsf{Hyb}^{(4)}$ is $\mathsf{pk}_{\mathcal{H}_D}$. In addition, in this hybrid we no longer have sk_j for honest decryptors j. This hybrid is indistinguishable from $\mathsf{Hyb}^{(3)}$ under RLWE assumption that $\mathsf{pk}_{\mathcal{H}_D}$ is pseudorandom.

- $\mathsf{Hyb}^{(5)}$: In this hybrid we no longer run honest decryptors as oracles, and instead we generate shares and handle decryptor aborts directly in the hybrid. In addition, we also handle client aborts in this hybrid. Specifically, for all $j \in \mathcal{H}_D$:
  - let $\{\mathsf{share}_k^j\}_{k\in\mathcal{C}_D}$ be a set of random values;
  - generate $(\mathsf{sk}_{\mathsf{mask},j}, \mathsf{pk}_{\mathsf{mask},j}) \leftarrow \mathsf{AHE.KeyGen}(r_{\mathsf{mask},j})$ for random $r_{\mathsf{mask},j}$;
  - let $\mathsf{share}^{\mathsf{mask},j} \leftarrow \mathsf{Share}(r_{\mathsf{mask},j})$;
  - let $\mathsf{share}^{\mathsf{sym},j} \leftarrow \mathsf{Share}(\mathbf{k}_{\mathsf{sym},j})$.

  When $\mathcal{A}$ sends $\texttt{signedpk}_i = (\mathsf{D}, S_{\mathsf{D}}, \mathsf{pk}, \mathsf{pk}_{\mathsf{mask}})$ to each honest client $i \in \mathcal{H}$, we let client i abort if there are at least t signatures s_j in S_{D} that are invalid.

  Then, during the decryption phase, for all honest $j \in \mathcal{H}_D$:
  - when $\mathcal{A}$ broadcasts S^j to j, we check in this hybrid that $|S^j| \geq t$ and the proofs in S^j verify. If the verification does not pass, we remove j from $\mathcal{H}_D$, i.e. let j abort;
  - when $\mathcal{A}$ broadcasts $\texttt{signedct}_j = (\mathsf{ct}^1, s_{\mathsf{ct}^1})$ to j, if s_{ct^1} contains an invalid signature, we remove j from $\mathcal{H}_D$, i.e. let j abort;
  - if we do not abort in the previous step, then send $\{\mathsf{share}_h^{\mathsf{mask},j}\}_{h\in\mathcal{H}_D}$ to $\mathcal{A}$ along with encryptions of honest partial decryptions $\{\bar{\mathsf{pd}}_j\}_{j\in\mathcal{H}_D}$.

  For each $\ell, j \in \mathcal{H}_D$, $\mathcal{A}$ is given either $\mathsf{share}_j^{\mathsf{sym},\ell}$ (if $\ell \in P_j$) or share_j^{ℓ} (if $\ell \notin P_j$). However, in order to avoid aborts, for each j it is only able to request share_j^{ℓ} for $m - t$ different ℓ.

  Comparing with $\mathsf{Hyb}^{(4)}$, we see that this hybrid handles aborts in the same way as in $\mathsf{Hyb}^{(4)}$:
  - for all $j \in \mathcal{H}_D$, $\mathsf{share}^{\mathsf{mask},j}$ and $\mathsf{share}^{\mathsf{sym},j}$ are generated in the same way as in $\mathsf{Hyb}^{(4)}$;
  - honest decryptors abort under the same condition about proofs on pk_j and $\mathsf{pk}_{\mathsf{mask},j}$;
  - honest clients abort under the same condition about signatures on pk;

  - $\mathcal{A}$ receives at most $|\mathcal{C}_D| * (m - |\mathcal{C}_D|)$ values share_k^j for $k \in \mathcal{C}_D$ and at most $(m - |\mathcal{C}_D|) * (m - t)$ such values from honest clients. That is at most $m - t + |\mathcal{C}_D|$ shares per honest client, by the bound on $|\mathcal{C}_D|$ that is at most $t - 1$ shares per honest client. Therefore, for each $j \in \mathcal{H}_D$, $\{\mathsf{share}_j^j\}_{k \in \mathcal{C}_D}$ is uniformly random in both $\mathsf{Hyb}^{(4)}$ and $\mathsf{Hyb}^{(5)}$; and thus $\mathcal{A}$ cannot decrypt all of the additive shares of the master secret key.

  So $\mathsf{Hyb}^{(5)}$ is identical to $\mathsf{Hyb}^{(4)}$.

- $\mathsf{Hyb}^{(6)}$: In this hybrid we compute $\mathsf{pd}_{\mathcal{H}_D}$ using $\mathsf{sk}_{\mathcal{H}_D}$:

  - $\mathsf{pd}_{\mathcal{H}_D} = \mathsf{AHE.PartialDec}(\mathsf{ct}^1, \mathsf{sk}_{\mathcal{H}_D}) + \sum_{j=1}^{|\mathcal{H}_D|-1} e_j'''$ for $e_j''' \leftarrow \chi_{\mathsf{flood}}$.

  Note that in $\mathsf{AHE.PartialDec}$ we already have one instance of flooding noise sampled from χ_{flood}; so we add $|\mathcal{H}_D| - 1$ many more to match $\mathsf{Hyb}^{(5)}$. We can express $\mathsf{pd}_{\mathcal{H}_D}$ in this hybrid as

$$\mathsf{pd}_{\mathcal{H}_D} = -v \cdot \mathsf{pk}_{\mathcal{H}_D} + v \cdot e + \mathsf{sk}_{\mathcal{H}_D} \cdot e'' + \sum_{j \in \mathcal{H}_D} e_j'''.$$

  where e is the error term in $\mathsf{pk}_{\mathcal{H}_D} = -\mathsf{u} \cdot \mathsf{sk}_{\mathcal{H}_D} + e$, and e'' is the error term in ct^1. Note that in $\mathsf{Hyb}^{(5)}$ we have

$$\mathsf{pd}_{\mathcal{H}_D} = -v \cdot \mathsf{pk}_{\mathcal{H}_D} + \sum_{j \in \mathcal{H}_D} e_j'''.$$

  Using the argument similar to $\mathsf{Hyb}^{(1)}$, we see that $\mathsf{pd}_{\mathcal{H}_D}$ in $\mathsf{Hyb}^{(5)}$ and $\mathsf{Hyb}^{(6)}$ are statistically close. So $\mathsf{Hyb}^{(6)}$ is indistinguishable from $\mathsf{Hyb}^{(5)}$.

- $\mathsf{Hyb}^{(7)}$: In this hybrid, we set all dropout clients' input to 0, and we program the RO accordingly. For all $i \in S$ we set $\mathsf{RO}(\mathsf{seed}_i) = x_i$ and $\tilde{x}_i = \mathsf{PRG}.\mathtt{Expand}(\mathsf{seed}_i)$. This effectively replaces input x_i with 0 for these dropout clients i. Since all seed_i are sampled at uniformly random from an exponentially large domain, and since PRG is modeled as a random oracle such that its output is truely random, the probability of collisions is negligible, and $\mathsf{Hyb}^{(1)}$ is indistinguishable from $\mathsf{Hyb}^{(0)}$.

- $\mathsf{Hyb}^{(8)}$: In this hybrid, we pick any surviving honest client $i^* \in \mathcal{H} \setminus S$ and replace its input with $s = \sum_{i \in \mathcal{H} \setminus S} x_i$, and we replace the input of other honest surviving clients $i \neq i^*$ with 0. Since all $\tilde{x}_i$ remain pseudorandom and correlated in the same way, this hybrid is indistinguishable from $\mathsf{Hyb}^{(7)}$.

- $\mathsf{Hyb}^{(9)}$: In this hybrid we program the RO such that $\mathsf{RO}(\mathsf{seed}_{i^*}) = s = \sum_{i \in \mathcal{H} \setminus S} x_i$, and $\tilde{x}_{i^*} = \mathsf{PRG}.\mathtt{Expand}(\mathsf{seed}_{i^*})$. This effectively replaces the input of i^* with 0. Since seed_{i^*} is uniformly random from an exponentially large domain, the collision probability with RO queries from $\mathcal{A}$ is negligible, and thus $\mathsf{Hyb}^{(9)}$ is indistinguishable from $\mathsf{Hyb}^{(8)}$. Note that, this hybrid does not depend on the real honest clients' private information anymore.

- $\mathsf{Hyb}^{(10)}$: In this hybrid we replace honest surviving clients' KAHE ciphertexts m_i to (i.e. $i \in \mathcal{H} \setminus S$)

  - for all $i \neq i^*$, let m_i be a random element in KAHE's ciphertext space;
  - let $m_{i^*} = -\sum_{i \in S} m_i$.

  Note that the view to the distinguisher contains all m_i and the sum of KAHE secret keys $\mathbf{k}_i$ for all honest surviving clients $i \in \mathcal{H} \setminus S$; but nothing else depends on the KAHE secret keys of these clients i. In $\mathsf{Hyb}^{(9)}$, the view contains the sum of $\mathbf{k}_i$, and for all $i \in \mathcal{H} \setminus S$, m_i is a KAHE ciphertext under $\mathbf{k}_i$. By the HintRLWE assumption and special leakage property of KAHE in Lemma 1, the real KAHE ciphertexts m_i in $\mathsf{Hyb}^{(9)}$ are indistinguishable from the random but correlated values m_i in $\mathsf{Hyb}^{(10)}$. So, this hybrid is indistinguishable from $\mathsf{Hyb}^{(9)}$.

- $\mathsf{Hyb}^{(11)}$: In this hybrid we replace $\tilde{\mathsf{pk}}_j$ from random shares of $\mathsf{pk}_{\mathcal{H}_D}$ to be honestly sampled from $\mathsf{AHE.KeyGen}$, we replace pd_j be honestly computed j'th partial decryption, and we replace the proof of partial decryption π_j using a honest generated proof. Using arguments similar to $\mathsf{Hyb}^{(0)}$ to $\mathsf{Hyb}^{(4)}$ (in reverse order), we see that this hybrid is indistinguishable from $\mathsf{Hyb}^{(10)}$. Note that this hybrid depends on only simulated honest decryptors, and it depends on honest clients as in the Sim.

For efficiency, note that the client's running time is $O(\ell \log n)$, the same as in the semi-honest server case. The server can compute the aggregation tree in time $O(n\ell \log n)$, and thus the asymptotic running time remains the same as in the semi-honest server case. For the decryptors, we now require an extra round in both the key generation and the decryption phases, but since we require the same AHE parameters as in the semi-honest server case, the asymptotic running time remains the same at $O(m + \log n)$. $\square$

C.3 Distributed KeyGen with Untrusted Proxy

Analogously to the work of Flamingo [7], we present a protocol for key generation that can support an actively corrupted coordinator/proxy. Note that actively malicious behavior by decryptors (along with collusion with a semi-honest server) is already supported by the protocol of the previous section. Here we extend the protocol to handle the situation where a malicious server doesn't honestly relay messages between decryptors possibly inducing drop-outs, i.e., the role of the coordinator Coord in Figure 4. Let us remark that instances of our protocol with a single decryptor (2-server model), or where decryptors have means to communicate among themselves independently of the Server would have a significantly simpler protocol than the one presented next.

For space reasons the protocol in full detail is presented in Appendix C but the main differences from Figure 4 are as follows. Firstly, the decryptors must sign their key contribution and then check that the overall key is constructed by contributions from enough of them, before signing the resulting key so that clients know it is valid. At the end they must check that they have (mostly) been given the same view of which of them dropped out without providing a partial decryption. This prevents the server from recovering more shares than it should. Finally, the decryptors must generate two keys rather than one, the second is used by the clients as explained in the next section.

C.4 Server and Client

Another challenge in the malicious server setting that we need to tackle, is coming from the fact that the server can maliciously drop clients including after seeing their (encrypted) contributions. This creates difficulties for the simulation proof, which needs to simulate the honest parties without knowing which of them will be dropped by the adversary and which ciphertexts will be included in the final sum to be decrypted. While in the semi-honest setting the set of drop-outs could be treated as an input for the ideal functionality and thus the simulator could invoke the ideal functionality and obtain the sum of the inputs of the honest clients that will be included in the sum, this is no longer the case in the malicious setting

To handle this we introduce a different type of encoding for the clients' inputs: Each client additively secret shares its input into a share generated from a PRG invocation, and the difference (line 3 in Figure 15). All clients send their seeds used for the PRG-generated share individually encrypted with the new shared key (line 8 in Figure 15). The remaining shares of the inputs are provided to the server via the aggregation protocol (lines 4-7 in Figure 15).

The server recovers its final output by obtaining shares of the decryption key for the PRG shares from the decryptors, decrypting the seeds and evaluating the PRG. The results of these are aggregated together with the output of the aggregation protocol ran on the rest of the input shares (line 8-10 in Figure 14).

Looking ahead what the above constructions enable us to do in the proof is to instantiate the PRG as a random oracle that the simulator can program to embed any value that it wants in a simulated input for an honest client (see details in the proof in Appendix C.2).

D Security definitions

Our security proofs are in the ideal vs. real paradigm. We follow closely the definitions in Lindell's tutorial [15], 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 $\mathrm{IDEAL}_{\mathcal{F}_I^{\mathcal{A}}}\big((\mathbf{x_i})_i, \lambda\big)$ 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 $\mathrm{REAL}_{\pi_R^{\mathcal{A}}}\big((\mathbf{x_i})_i, \lambda\big)$ 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 6. *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 $(\mathbf{x_i})_i$,*

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

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

Deterministic vs. probabilistic adversaries. In accordance to the previous definition, to prove that a protocol π is secure, one exhibits a simulator Sim. 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 [15]). 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 (ii) termination possibly producing an output. An important observation is that for passively corrupted parties the `next-action` function is known to the simulator, and its outcome can be predicted, as the corrupted party must follow the prescribed protocol. This is particularly important in our protocol in the case of a semi-honest server, as it takes the role of the verifier in ZK proofs. This means that in that corruption model we can rely on Honest-Verifier Zero knowledge. Finally, we should emphasize that the randomness tape of a passively corrupted party is not observable to the adversary, and in particular, an adversary passively corrupting the server can't predict challenges that the Server will generate, which prevents actively corrupted clients from forging proofs.

D.1 Definitions in the Single-Server setting

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

D.1.1 (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 7 ((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 $\boldsymbol{output}_{\mathsf{S}_I}$. Let $\boldsymbol{output}_{\mathcal{A}_I}$ 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*

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

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 8 ((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 $\boldsymbol{output}_{\mathsf{S}_R}$. Let $\boldsymbol{output}_{\mathcal{A}_R}$ 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*

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

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

D.1.2 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

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

D.2 Random Oracle Model

Like previous works [1, 3], our maliciously secure protocol is proven secure in the Random Oracle Model (RO) [41].

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

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

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

Setup: A committee $\mathcal{C}$ with members $1\ldots,m$, with secure authenticated channels among themselves, and a coordinator Coord forwarding messages. A PKI holding public signing keys for committee members and verifier V.

Parameters: Public AHE parameters, timeout T, and threshold t.

Key Generation Phase

Output: A set of decryptors $\mathsf{D} \subseteq \mathcal{C}$, public keys $\mathsf{pk}, \mathsf{pk}_{\mathsf{mask}}$, and $\geq t$ signatures on $\mathsf{pk}, \mathsf{pk}_{\mathsf{mask}}$.

Round 1: Share partial keys

1. Every committee member $j \in [m]$:
   (a) Computes $(\mathsf{sk}_j, \mathsf{pk}_j) := \mathsf{AHE.KeyGen}(r_j)$ from randomness r_j,
   (b) Secret-shares r_j within $\mathcal{C}$ with threshold t.
   (c) Computes $(\mathsf{sk}_{\mathsf{mask},j}, \mathsf{pk}_{\mathsf{mask},j}) := \mathsf{AHE.KeyGen}(r_{\mathsf{mask},j})$ from randomness $r_{\mathsf{mask},j}$,
   (d) Secret-shares $r_{\mathsf{mask},j}$ within $\mathcal{C}$ with threshold t.
   (e) Sends $h_j := \big(\mathsf{pk}_j, \mathsf{Prove}_{\mathsf{AHE.KeyGen}}(r_j), \mathsf{pk}_{\mathsf{mask},j}, \mathsf{Prove}_{\mathsf{AHE.KeyGen}}(r_{\mathsf{mask},j})\big)$ and $\mathrm{sig}(h_j)$ 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 $\mathsf{pk}_{\mathsf{mask}} := \sum_{j\in\mathcal{C}_s} \mathsf{pk}_{\mathsf{mask},j}$.

Round 2: Verify global keys $\mathsf{pk}, \mathsf{pk}_{\mathsf{mask}}$

3. Coord broadcasts $S = \{h_j | j \in \mathsf{D}\}$ within D.
4. Every decryptor $j \in \mathsf{D}$: // Decryptors independenly check the server's work
   (a) Checks $|S| \geq t$ and that it has received valid proofs from each member, aborting if any check fails.
   (b) Sends $s_j := \mathrm{sig}(\sum_{i\in S} \mathsf{pk}_j)$ and $s_{\mathsf{mask},j} := \mathrm{sig}(\sum_{i\in S} \mathsf{pk}_{\mathsf{mask},j})$ 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 the signed keys are not all equal or if $|S_{\mathsf{D}}| < t$ Coord aborts, otherwise it outputs $(\mathsf{D}, S_{\mathsf{D}}, \mathsf{pk}, \mathsf{pk}_{\mathsf{mask}})$.

Decryption Phase

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

Output: Aggregated partial decryption of ciphertext $(\mathsf{ct}^0, \mathsf{ct}^1)$.

Round 1: Collect encrypted partial decryptions

6. Coord receives $(\mathsf{ct}^1, s_{\mathsf{ct}^1})$ and broadcasts it within D.
7. Every $j \in \mathsf{D}$:
   // Decryptors provide an *encrypted* partial decryption, only if the ciphertext has been verified by V, and secret share the corresponding key.
   (a) Checks that s_{ct^1} contains appropriate signature(s), otherwise aborts.
   (b) Secret-shares a key $\mathbf{k}_{\mathsf{sym},j}$ for a symmetric encryption scheme Sym within D with threshold t.
   (c) Sends $\big(\bar{\mathsf{pd}}_j := \mathsf{Sym.Enc}(\mathsf{pd}_j, \mathbf{k}_{\mathsf{sym},j}), p_j := \mathsf{Prove}_{\mathsf{AHE.PartialDec}}(\mathsf{pd}_j, \mathsf{sk}_j)\big)$, where $\mathsf{pd}_j = \mathsf{AHE.PartialDec}(c^1, \mathsf{sk}_j)$ to Coord.
   (d) Send shares of mask keys received in step 1d to Coord to enable recovery of $\mathsf{sk}_{\mathsf{mask}}$.
8. Coord collects messages up to a timeout T. Let P be the set of decryptors that reply. If $|P| < t$, Coord aborts.
9. Coord reconstructs $\mathsf{sk}_{\mathsf{mask}}$.

Round 2: Recovery of partial decryptions

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 not in P, i.e. dropouts, to Coord.
    (c) Sends key shares received in step 7b from every decryptor in P 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. Coord recovers $\mathbf{k}_{\mathsf{sym},j}, \mathsf{pd}_j$ for every non-dropout $j \in P$ (from step 7c). If some proof p_j does not verify Coord aborts.
13. Coord sends $\mathsf{pd} := \sum_{j\in\mathsf{D}} \mathsf{pd}_j$ and $\mathsf{sk}_{\mathsf{mask}}$ to S.

Figure 13: Decryptor D by committee, with unreliable proxy Coord.

```
Parameters: Number of clients $n$, input domain $\mathbb{F}^\ell$.
1. Receive key $\texttt{signedpk} = (\mathsf{D}, S_\mathsf{D}, \mathsf{pk}, \mathsf{pk}_\texttt{mask})$ from $\mathsf{D}$.
2. Initialize $nclients, m, \mathsf{ct}^0$, and $\mathsf{ct}^1$ to zero.
3. while $nclients < n$ :
   // Process $i$th client's request
     (a) Send $\texttt{signedpk}$ to $\mathsf{C}_i$
         // $\widetilde{\mathsf{seed}}_i$ is the encryption of a PRG seed $\mathsf{seed}_i$ under $\mathsf{sk}_\texttt{mask}$
     (b) Receive $(m_i, (\mathsf{ct}_i^0, \mathsf{ct}_i^1), p_i, \widetilde{\mathsf{seed}}_i)$ from $\mathsf{C}_i$
     (c) If $\mathsf{Verify}_{\mathsf{AHE.Enc}}((\mathsf{ct}_i^0, \mathsf{ct}_i^1), \mathsf{pk}, p_i, i)$:
           i. $(\mathsf{ct}^0, \mathsf{ct}^1) \mathrel{+}= (\mathsf{ct}_i^0, \mathsf{ct}_i^1)$
          ii. $m \mathrel{+}= m_i$
         iii. $nclients \mathrel{+}= 1$
   // Decrypt aggregated symmetric key
4. Compute Aggregation tree $T$.
5. Send $\mathsf{ct}^1, T$ to $\mathsf{V}$.
6. Receive $s_{\mathsf{ct}^1} = \mathrm{sig}(\mathsf{ct}^1)$ from $\mathsf{V}$.
7. Send $\mathsf{ct}^1, s_{\mathsf{ct}^1}$ to $\mathsf{D}$.
8. Receive $\mathbf{k}$ and $\mathsf{AHE}.sk_\texttt{mask}$ from $\mathsf{D}$.
9. Recover $\texttt{mask} = \sum_i \mathsf{PRG}.\texttt{Expand}(\mathsf{seed}_i, \ell)$.
10. Output $\mathsf{KAHE.Dec}(m, \mathbf{k}) - \texttt{mask}$.
```

Figure 14: Server S. The server processes n asynchronous client contributions.

```
Input: $\mathbf{x_i} \in \mathbb{F}^\ell$.
1. Receive $\texttt{signedpk} = (\mathsf{D}, S_\mathsf{D}, \mathsf{pk}, \mathsf{pk}_\texttt{mask})$ from $\mathsf{S}$. If there are at least $t$ signature $s_j \in S_\mathsf{D}$ or at least $t$ signatures $s_{\texttt{mask},j}$ that fail to verify on $\mathsf{pk}$, client $\mathsf{C}$ aborts.
2. Set $\mathbf{k}_i := \mathsf{KAHE.KeyGen}()$ and $\mathsf{seed}_i := \mathsf{PRG.KeyGen}()$.
   // Compute masked input $\mathbf{x_i}$, mask with $\mathsf{seed}_i$.
3. $\tilde{\mathbf{x}}_\mathbf{i} := \mathsf{PRG}.\mathsf{Expand}(\mathsf{seed}_i, \ell) + \mathbf{x_i}$.
   // $m_i$ is an symmetric key encryption of the masked input.
4. Set $m_i := \mathsf{KAHE.Enc}(\tilde{\mathbf{x}}_\mathbf{i}, \mathbf{k}_i)$.
5. Sample $r \leftarrow \{0,1\}^\lambda$ uniformly at random
   // $(\mathsf{ct}_i^0, \mathsf{ct}_i^1)$ is an encryption of $\mathbf{k}_i$ under $\mathsf{pk}$ with randomness $r$
6. Set $(\mathsf{ct}_i^0, \mathsf{ct}_i^1) := \mathsf{AHE.Enc}(\mathbf{k}_i, \mathsf{pk}, r)$
7. Set $p_i = \mathsf{Prove}_{\mathsf{AHE.Enc}}((\mathsf{ct}_i^0, \mathsf{ct}_i^1), \mathbf{k}_i, \mathsf{pk}, r, i)$
8. Set $\widetilde{\mathsf{seed}}_i = \mathsf{AHE.Enc}(\mathsf{seed}_i, \mathsf{pk}_\texttt{mask})$
9. Send $(m_i, (\mathsf{ct}_i^0, \mathsf{ct}_i^1), p_i, \widetilde{\mathsf{seed}}_i)$ to $\mathsf{S}$
```

Figure 15: Client C. Note that all steps, except forming ct_i^0 and $\widetilde{\mathsf{seed}}_i$ and sending the result, (but including forming ct_i^1 and p_i) can be done before the $\mathsf{pk}, \mathsf{pk}_\texttt{mask}$ arrive from S. Thus online time can be very small, and in particular is independent of ℓ.