ACORN: Input Validation for Secure Aggregation

James Bell[◇], Adrià Gascón[◇], Tancrède Lepoint[†][*], Baiyu Li[◇], Sarah Meiklejohn[◇], Mariana Raykova[◇], Cathie Yun[*]
[◇]Google [†]Amazon

Abstract

Secure aggregation enables a server to learn the sum of client-held vectors in a privacy-preserving way, and has been applied to distributed statistical analysis and machine learning. In this paper, we both introduce a more efficient secure aggregation protocol and extend secure aggregation by enabling *input validation*, in which the server can check that clients' inputs satisfy constraints such as L_0, L_2, and L_∞ bounds. This prevents malicious clients from gaining disproportionate influence on the aggregate statistics or machine learning model.

Our new secure aggregation protocol improves the computational efficiency of the state-of-the-art protocol of Bell et al. (CCS 2020) both asymptotically and concretely: we show via experimental evaluation that it results in 2-8X speedups in client computation in practical scenarios. Likewise, our extended protocol with input validation improves on prior work by more than 30X in terms of client communication (with comparable computation costs). Compared to the base protocols without input validation, the extended protocols incur only 0.1X additional communication, and can process binary indicator vectors of length 1M, or 16-bit dense vectors of length 250K, in under 80s of computation per client.

1 Introduction

Single-server secure aggregation, which enables a server to learn the sum of client-held vectors in a privacy-preserving way, can be used for the secure computation of distributed histograms or for averaging model updates in federated learning systems. As some concrete examples, it supports cryptographic protocols for recommendation systems [37] and time series analysis [46], and is also used in large-scale real-world deployments for predictive typing and selection [41, 28, 26].

In the single-server setting, a powerful server talks to a large number of (resource-constrained) clients with limited connectivity. Along with limited bandwidth, the latter constitutes a central challenge in production systems [28, 7]. The server might be corrupted and even collude with a subset of the clients. This threat model strikes a good balance between trust and efficiency for large-scale distributed computation, and is used by several existing aggregation protocols [5, 6, 44, 46, 43, 37, 48] and more general results [4, 13].

To achieve acceptable levels of accuracy and privacy, the *minimum* number of clients contributing to an aggregation ranges between 100 and 10,000 [28], depending on the application, with larger numbers resulting in better privacy or better trade-offs between privacy and accuracy. On the other hand, input vector sizes correspond to model sizes or histogram sketches, so lengths are easily in the range of hundred of thousands or millions. Therefore, a secure aggregation protocol suitable for practical applications must be *scalable* in terms of being able to tolerate large inputs and a large number of clients, and *dropout-robust* in terms of tolerating a relatively high fraction of clients that abort during the protocol execution. While client computation is a natural concern addressed in several previous works, bandwidth consumption (both in download and upload) is often a determining factor in practice [28]. Achieving both practical computational and communication efficiency is the main focus of this work.

Privacy-preserving input validation. Another aspect crucial for deploying secure aggregation in practice is *correctness in the face of corrupted clients*. This corresponds to enhancing protocols with defenses against *malicious* clients who seek to bias the aggregate data. We discuss such attacks in the context of federated learning in Section 1.2, along with the role that bounding the L_2 or L_∞ norm of client inputs (*norm-bounding*) plays as a defense. In statistics applications such as frequency counting, malicious clients should be prevented from having a disproportionate influence on the output, e.g. contributing a value other than 1 to a histogram bucket or contributing to a large number of buckets. This corresponds to a k-hotness check (i.e. a check that at most k entries are 1 and the rest are 0, which is an L_0 bound) on the input vectors of clients.

To implement these defenses, the server can perform *input validation* on the data sent by clients, relying on zero-

[*]Work done while employed at Google.

knowledge proofs to preserve privacy. Crucially, this must be done without requiring significant client computation. Input validation comes in different forms, ranging from detection of malicious client behavior (but still having it cause the protocol to abort) to both identifying misbehaving clients and removing their contributions from the final aggregate statistics.

1.1 Our Contributions

We introduce and evaluate three protocols in this paper: RLWE-SecAgg, ACORN-detect, and ACORN-robust. Their security and efficiency properties, as well as comparison with existing approaches, are presented in Table 1.

Our first contribution is RLWE-SecAgg, a new secure aggregation protocol based on lattice cryptography that improves the state of the art protocol due to Bell et al. [5] in terms of both concrete and asymptotic efficiency. More precisely, it retains the low communication of this protocol but achieves optimal computational costs.

Our second contribution includes protocol variants ACORN-detect and ACORN-robust with input validation based on zero-knowledge proofs that are practical in terms of both computation and communication. For example, in under 80 seconds, a client running ACORN-detect on a standard laptop can (1) show k-hotness of a binary input of length 1M, or (2) show that a dense vector of length 250K has its L_∞ norm bounded by 2^{16}. In terms of communication, the overhead of ACORN-detect over (non-validated) secure aggregation is roughly 0.1X. This is in contrast with previous works [34, 14] with double-digit factor overheads (see Table 1). To enable this, we provide a new zero-knowledge construction with logarithmic proof size for proving an L_∞ bound on a private vector (committed to in a constant-size commmitment).

In our evaluation of these protocols, we consider two scenarios: *analytics* and *learning*. The former corresponds to the secure computation of a size-ℓ histogram with inputs from $n = 10^4$ devices, where the protocol ensures that each clients contributes no more than once to a bounded number of buckets. The latter corresponds to a federated learning application, where the goal is to average length-ℓ model updates from $n = 500$ devices, while showing a bound on the norm of each client's input. We also provide benchmarks and overheads of the end-to-end performance using four real-world tasks and datasets, showing a bandwidth overhead of at most 1.05X and a manageable computational overhead.

RLWE-SecAgg: Secure aggregation from (R)LWE. As a starting point, we formulate a generalization of the Bell et al. secure aggregation protocol [5], which we refer to as PRG-SecAgg. Our generalized protocol recovers the original PRG-SecAgg construction if we instantiate it using a PRG-based encoding of the input, but we also present a new instantiation—RLWE-SecAgg—that uses a lattice-based encoding. This construction reduces client and server computation costs, both asymptotically and in terms of concrete efficiency. In more detail, for n clients and length-ℓ input vectors, PRG-SecAgg requires clients to do $O(\ell \log n)$ work and the server to do $O(n\ell \log n)$ work. In RLWE-SecAgg these costs improve to $O(\ell + \log n)$ and $O(n(\ell + \log n))$, respectively. This means that for $\log n \leq \ell$, which is the case in all known applications, our new protocol's client computation and communication costs are $O(\ell)$, which is optimal in that it matches the insecure baseline where clients just send their data.

ACORN: Practical private input validation. We propose secure aggregation protocols that support two types of input validation: ACORN-detect supports *detection* of malicious client behavior, while ACORN-robust provides *robustness* to misbehaving clients, as it has the ability to identify them and adaptively exclude their inputs from the final sum.

Our protocols extend our generalized SecAgg protocol and thus can be instantiated with both PRG-SecAgg and RLWE-SecAgg. One complicating factor is that, in SecAgg, clients encode their inputs using pairwise correlated keys. This design decision is justified by its communication efficiency [6], as the correlated randomness can be computed in an input-independent way using constant-sized seeds. Previous works like EIFFeL [14] and RoFL [34] use alternative underlying aggregation schemes (with quadratic and linear dependence in the number of clients, see Table 1) that result in much higher communication than ACORN.

A consequence of using correlated keys is that enforcing correctness becomes complex: clients must *individually* prove a norm bound on the input being encoded but *collectively* prove that the keys used in the encoding step of the protocol are correctly correlated. This latter property would be guaranteed if each client proved individually that it formed its key honestly, but this would be expensive. Instead, in ACORN-detect we use a distributed proof that does not require clients to interact. In ACORN-robust, we instead require neighboring clients to form *identical* commitments to their pairwise shared masks. As long as one of the pair is honest, the server can thus identify a mismatch and exclude the cheating client. This sort of client-aided verification is efficient, but does not work if two malicious clients are neighbors. We thus require clients to commit to shares of their correlated randomness before knowing who their neighbors will be, and also need a logarithm number of rounds to recursively perform this exclusion.

Succinct ZK proofs of bounded norm. Besides using an appropriate underlying SecAgg protocol, an important technique to achieve efficient communication is *ciphertext packing*: encoding several plaintext elements in a single ciphertext. While this keeps ciphertext expansion low even when working in a large group, it complicates the client's proof of correct encoding, as it needs to show an L_∞ bound for correctness of the (linear) packing function. For this we rely on Bulletproofs [9, 11], a discrete log-based zero-knowledge proof system with

			Clear	Bell et al. [5]	RLWE-SecAgg (Sec. 3)	RoFL [34]	EIFFeL [14]	ACORN-detect (Sec. 4)	ACORN-robust (Sec. 4)
Client communication			ℓ	$\ell+\log(n)$	$\ell+\log(n)$	$\ell+n$	ℓn^2	$\ell+\log(n)$	$\ell+\log^2(n)$
n	ℓ	$\gamma=\delta$							
10^2	10^4	0.05	31KB	45KB	45KB	>2MB*†	94MB	47KB	365KB
10^3	2^{18}	0.05	705KB	1030KB	1111KB	>51MB†	240GB*	1032KB	1350KB
10^3	2^{18}	0.33	705KB	1062KB	1144KB	>51MB†	N/A	1064KB	14MB
Client computation			ℓ	$\ell\log(n)$	$\ell+\log(n)$	ℓn	ℓn^2	$\ell\log(n)$	$\ell\log(n)+\log^2(n)$
Secure against									
server & up to γn clients			○	●	●	◐	●	●	◐
up to n clients			✓	✓	✓	✓	×	✓	✓
Input validation			✓	×	×	✓	✓	✓	✓
Robustness									
δn dropouts			✓	✓	✓	✓	✓	✓	✓
δn invalid inputs			✓	×	×	×	✓	×	✓

Table 1: The concrete communication, security properties, and communication and computation asymptotics of various secure aggregation algorithms. The concrete rows are all based on proving an L_∞ bound of $2^{32}/n$, with security against a semi-honest server and malicious clients. The bottom rows show which of the approaches protect client inputs, validate client inputs, function in the presence of (limited) dropouts, and function in the presence of (limited) malicious clients. *Values extrapolated via asymptotics from others in the column. † RoFL doesn't provide details for how to agree on a sharing of zero, so these communication statements don't include that (which is likely only a small fraction of the total cost). By ◐ and ● we denote that the protocol provides semi-honest and malicious security, respectively.

logarithmic proof size.

Using Bulletproofs to prove a bound on each entry of the input vector (as in RoFL [34]) results in linear communication; for a vector of length 2^{20}, for example, the commitments alone would require over 33 MB. The same is true of representing the desired bound as an arithmetic circuit or rank-1 constraint system (R1CS) and then proving it using a state-of-the-art proof system: this compilation process requires one commitment per input (meaning, in our case, one commitment for every entry in the vector), which again results in linear communication. Moreover, proving things entry-wise is not compatible with ciphertext packing.

Instead, we show how the recent techniques by Gentry et al. [20] for approximate proofs of L_∞ bounds via random projections, combined with known tricks for range proofs [23] and other optimizations, allow us to reduce our correct encoding proof to a single linear constraint that can be proved using Bulletproofs. Our approach thus allows us to prove *exact* L_∞ bounds, and furthermore to commit to an arbitrary-length vector using only 256 bits. Moreover, verification of multiple such proofs can be *batched*, which is crucial for our protocol to scale to large cohort sizes.

1.2 Secure Aggregation for FL

Federated learning (FL) is a distributed setting [28] where many clients collaboratively learn a model under the coordination of a central server. In each round of FL, (1) the server broadcasts the current model to all clients involved in that round. Then, in a (2) local training step, clients update the model using their local dataset of examples. Finally, the clients engage in (3) an aggregation step where all locally trained models are pre-processed and aggregated for the server to obtain an updated model. These steps (1-3) are iterated for a number of training rounds.

The role of secure aggregation, and thus of our work, in FL is in step (3) above. A secure summation protocol significantly reduces the leakage to the server with respect to a system where model updates are made available in the clear. Next we discuss poisoning attacks in FL, and how input validation – and in particular norm-bounding – helps to mitigate them.

Poisoning attacks in FL. Attacks can be divided into *untargeted* attacks, in which the goal is to generally degrade the quality of the model, and *targeted* attacks. In targeted attacks (also known as *backdoor* attacks), the goal of the attacker is to induce a given behavior in a particular subtask; e.g., for classification, have examples in a given class (cars) be misclassified (as, for example, birds) while retaining accuracy for the rest of the examples.

Several recent studies have empirically evaluated the effectiveness of imposing bounds on the L_2 norms of clients' model updates as a defense against these types of attacks. Intuitively, the norm bound limits the influence of malicious clients when trying to derail learning. Sun et al. [50] focus on model replacement attacks, where a malicious client provides a scaled malicious model update to effectively replace the current model by a backdoored one. They run a comprehensive study on the EMNIST dataset [12]—a real-life, user-partitioned, and non-IID dataset—and conclude that norm-bounding is an effective defense against backdoor attacks here.

Chowdhury et al. [14] empirically evaluate 7 attacks in the literature (5 untargeted, 2 targeted) on image classification tasks. The results show how norm-bounding helps as a defense, with gaps of more than 20% in accuracy being recovered (for targeted attacks). For the investigated backdoor attacks, the gap in accuracy between the main and backdoor tasks drops from roughly 10% to over 80% when norm-bounding is applied.

Shejwalkar et al. [45] identify parameters where defenses become (in)effective with respect to both existing attacks and new ones they propose. The emphasis of their work is on untargeted attacks operating within real-world deployments. Regarding norm-bounding specifically, they conclude that it "can effectively protect cross-device FL in practice" and more concretely that the "Norm-bounding Aggregation Rule (AGR) is enough to protect production FL against untargeted poisoning, questioning the need for the more sophisticated (and costlier) AGRs".

Finally, Lycklama et al. [34] characterize the classes of targeted attacks that norm-bounding can defend against and provide extensive empirical evaluation and an open-source experimentation framework. The authors conclude that while norm-bounding significantly decreases the available surface for adversarial attacks, it is *not a silver bullet*. In particular, continuous attacks on tail targets [51] remain effective even under norm bounds. Subsequent works have developed defenses against attacks on the tails based on sparsification combined with norm-bounding [52].

Server attacks in FL. The common model for secure aggregation in FL assumes a server that honestly runs step (1) of each FL round, i.e. that broadcasts the model resulting from the previous round to all clients. As shown by Pasquini et al. [39], a malicious server can instead send carefully crafted models to specific clients in order to extract their input. We prove our protocols secure assuming this attack does not take place, but the defenses proposed by Pasquini et al., and in particular model hashing, can be directly applied to our protocols. More concretely, clients can share with each other a hash of the model that they received from the server and verify that it matches their own before proceeding with the protocol.

1.3 Other Related Work

There are several well-known works on verifiable secure aggregation in the two-server or multi-server models [8, 15, 2], but we focus our discussion on the single-server trust model.

Stevens et al. perform differentially private secure aggregation (without input validation) using an LWE-based protocol [49]. This work is similar to our first contribution, RLWE-SecAgg. However, they overlook a subtlety in the security of their scheme, claiming that "[a secure aggregation of keys] reveals nothing about their individual [key] values." This is untrue, because the output itself conveys information even if computed securely. Our security proof addresses this issue.

Lycklama et al. [34] and Chowdhury et al. [14] introduce secure aggregation protocols with input validation called RoFL and EIFFeL, respectively. RoFL requires each client to send commitments to each vector entry to the server. For vectors of length 262,000, they report a 48x increase in required communication (to 51MB) when proving an L_∞ bound, compared to sending the vector in the clear. EIFFeL shares the computation amongst the clients, using them to replace the servers in Prio [15]. This allows them to deal with a constant fraction of malicious clients and dropouts. However, their communication scales quadratically in the number of clients and linearly in the vector length. Thus even for a vector of length 10^4 and 100 clients they report 94MB of communication. This is about three orders of magnitude greater than the cost in the clear. EIFFeL and RoFL suffer from the difficulties of balancing input validation with communication costs, which is a major focus of our work. In Table 1 we offer a detailed comparison in terms of both asymptotic and concrete efficiency.

Ghodsi et. al. [21] propose zPROBE, a secure aggregation protocol that checks that each entry of a client's masked input is constructed honestly from an input of bounded size. They do this by putting the circuit for a pseudorandom generator in a generic proof framework, but as this is very costly they check only a random subset of entries in their experiments. This is enough to prevent submissions where an appreciable fraction of the entries lie outside the desired bound, but this is not sufficient in FL as a model can be corrupted by changing only one or a few entries by a large amount.

Karakoç et. al. [29] also provide secure aggregation with range validation using an oblivious programmable pseudorandom function. They describe this work as a proof of concept and provide experiments only for vectors of length 16 due to the currently prohibitive computational costs.

2 Preliminaries

We denote by $x \leftarrow \chi$ sampling according to a distribution χ. If X is a finite set, we denote by $x \leftarrow X$ uniform sampling from X. By $\approx_{\sigma,\lambda}$ we denote indistinguishability with computational parameter λ and statistical parameter σ; i.e., $D_n \approx_{\sigma,\lambda} F_n$ if there exists another distribution E_n such that D_n is statistically close to E_n ($|\Pr_{x\leftarrow D_n}[A(x)=1]-\Pr_{x\leftarrow E_n}[A(x)=1]|<\sigma(n)$ for an unbounded adversary A) and E_n is computationally indistinguishable from F_n ($|\Pr_{x\leftarrow E_n}[A(x)=1]-\Pr_{x\leftarrow F_n}[A(x)=1]|<\lambda(n)$ for a PPT adversary A). We use the standard simulation-based formalism [22, 33] in our security proofs. We assume key agreement, authenticated encryption, and signature primitives, which we denote as KA, $\mathsf{E}_{\mathsf{auth}}$, and Sig.

2.1 Setting and Threat Model

We consider n clients $1,\dots,n$, each holding a private vector $\mathbf{x_i}\in\mathbb{Z}_t^{\ell}$, and a server with communication channels established with all clients. The goal is for the server to obtain the sum of all client vectors ($\sum_i^n \mathbf{x_i}$), with robustness to a certain fraction of client dropouts. To make this concrete, the functionality is parameterized by a maximum fraction of dropouts $\delta\in[0,1]$, defined as follows:

$$f(\mathbf{x_1},\dots,\mathbf{x_n}) = \begin{cases} \sum_{i\in[n]\setminus D}\mathbf{x_i} & \text{if } |D|\le\delta n \\ \perp & \text{otherwise} \end{cases} \qquad (1)$$

where $D\subseteq[n]$ is the set of clients that dropped out during the protocol execution and the sum happens in $\mathbb{Z}_t^{\ell}$. We aim to withstand an adversary consisting of a coalition of γn clients, for $\gamma\in[0,1]$, and possibly also colluding with the server. As in previous works [6, 5], we assume that corrupt clients are fully malicious. For RLWE-SecAgg and ACORN-detect, we also assume the server is fully malicious,[1] but for ACORN-robust we prove security only in the case of a semi-honest server.

2.2 Lattices and Polynomial Rings

A lattice is a discrete subgroup $\Lambda\subset\mathbb{R}^N$, and it can be represented as the set of all integer combinations of a basis $\mathbf{B}$ such that $\Lambda=\mathbf{B}\mathbb{Z}^N$. We use the cyclotomic ring $R=\mathbb{Z}[X]/(X^N+1)$ for a power-of-two N, and write $R_q=\mathbb{Z}[X]/(q,X^N+1)$ for the residual ring of R modulo q. The *coefficient embedding* of a polynomial $a=\sum_{i=0}^{N-1}a_iX^i\in R$ is the vector $(a_0,a_1,\dots,a_{N-1})$, and we define the L_∞ norm of a as $\|a\|_\infty=\|(a_0,a_1,\dots,a_{N-1})\|_\infty=\max_i|a_i|$. As a convention, we use bold $\mathbf{a}$ to denote the coefficient embedding of a polynomial $a\in R$. We also define the negacyclic matrix representation of $a\in R$ as

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

We can naturally extend the map φ to vectors $\mathbf{a}$ over R such that $\varphi(\mathbf{a})$ is a matrix produced by vertically concatenating $\varphi(a_i)$ for all $a_i\in\mathbf{a}$. Without loss of generality, since the product of two polynomials $a,b\in R$ has the coefficient embedding $\mathbf{A}\cdot\mathbf{b}$ for $\mathbf{A}=\varphi(a)$, we represent $a\cdot b$ as a matrix-vector product $\mathbf{A}\cdot\mathbf{b}$. When $q=1 \pmod{2N}$, this computation can be done more efficiently via Number Theoretic Transformation (NTT) than a naïve matrix-vector multiplication.

2.3 Ring LWE and Encryption

The *ring learning-with-errors* (RLWE) assumption [36], parameterized by a ring R of degree N over $\mathbb{Z}$, an integer modulus $q>0$ defining a quotient ring $R_q=R/qR$, distributions χ_s,χ_e over R, and an integer m, states that for a secret $s\leftarrow\chi_s$, given $m=\mathsf{poly}(N)$ many independent samples from the distribution $A^{\mathsf{RLWE}}_{N,q,\chi_e}(s)=\{(a,as+e)\in R_q^2 \mid a\leftarrow R_q, e\leftarrow\chi_e\}$, it is computationally hard to distinguish them from m uniformly random samples over R_q^2. As we sometimes work with coefficient embedding of polynomials, we can rewrite an RLWE sample as $(\mathbf{A},\mathbf{As}+\mathbf{e})$ for $\mathbf{s}\leftarrow\chi_s\subseteq\mathbb{Z}_q^N$, a matrix $\mathbf{A}=\varphi(a)\in\mathbb{Z}_q^{N\times N}$ for $a\leftarrow R_q$, and an error vector $\mathbf{e}\leftarrow\chi_e\subseteq\mathbb{Z}^N$. In our protocol we treat $\mathbf{A}$ as a public parameter. To encrypt a plaintext message $\mathbf{x}\in\mathbb{Z}_T^N$, we sample $\mathbf{s}\leftarrow\chi_s$ and compute

$$\mathsf{Enc}(\mathbf{s},\mathbf{x})=(\mathbf{A},\mathbf{As}+T\cdot\mathbf{e}+\mathbf{x} \bmod q), \qquad (2)$$

and decrypt using $\mathsf{Dec}(\mathbf{s},(\mathbf{A},\mathbf{y}))=(\mathbf{y}-\mathbf{As}) \bmod T$. For longer messages $\mathbf{x}\in\mathbb{Z}_T^{\ell}$ such that $\ell>N$, we can naturally extend this encryption scheme by using multiple $\mathbf{A}_1,\dots,\mathbf{A}_{\ell/N}$ where $\mathbf{A}_i=\varphi(a_i)$ for $a_i\leftarrow R_q$; equivalently, we sample $\mathbf{s}\leftarrow\chi_s$ and compute an extended ciphertext $\mathsf{Enc}(\mathbf{s},\mathbf{x}\in\mathbb{Z}_T^{\ell})=(\mathbf{A},\mathbf{As}+T\cdot\mathbf{e}+\mathbf{x} \bmod q)$, where $\mathbf{A}=\varphi(\mathbf{a})$ is the vertical concatenation of $\mathbf{A}_1,\dots,\mathbf{A}_{\ell/N}$, and $\mathbf{e}\leftarrow\chi_e^{\ell/N}$. When the plaintext modulus T is coprime to q, the distribution on $(\mathbf{A},\mathbf{As}+T\cdot\mathbf{e})$ with $\mathbf{A}=\varphi(\mathbf{a})$ for $\mathbf{a}\leftarrow R_q^{\ell/N}$ and $\mathbf{e}\leftarrow\chi_e^{\ell/N}$ is indistinguishable from uniform under the RLWE assumption. For simplicity, we omit the public parameter $\mathbf{A}$ from ciphertexts.

Two important properties that we use in our protocol are key homomorphism and message homomorphism, i.e. (informally) $\mathsf{Enc}(\mathbf{s}_1,\mathbf{x}_1)+\mathsf{Enc}(\mathbf{s}_2,\mathbf{x}_2)=\mathsf{Enc}(\mathbf{s}_1+\mathbf{s}_2,\mathbf{x}_1+\mathbf{x}_2)$.

2.4 Commitment and Zero-Knowledge Proofs

Let G be a cyclic group of order q. The vector Pedersen commitment of $\mathbf{v}\in\mathbb{Z}_q^n$ using generators $g_0,g_1,\dots,g_n\leftarrow\mathsf{G}$ and randomness $r\in\mathbb{Z}_q$ is $C=g_0^r\prod_{i=1}^n g_i^{v_i}\in\mathsf{G}$, and we denote the commitment algorithm using the notation $\mathsf{com}=\mathsf{Commit}(\mathbf{v};r)$. It is perfectly hiding and computationally binding under the discrete logarithm assumption. We build our zero-knowledge proofs using Bulletproofs [11], which we describe in more detail in Section 5. Bulletproofs satisfies zero knowledge, meaning a simulator without knowledge of a witness can produce proofs that are indistinguishable from honest ones, and knowledge soundness, meaning it is possible to extract a valid witness from any proof that verifies. We

[1] We technically assume that the server behaves semi-honestly in a key distribution phase but otherwise maliciously, which is implied by assuming a fully malicious server with a PKI.

describe our proof systems as interactive, but make them non-interactive via the Fiat-Shamir heuristic [18], which means we operate in the random oracle model.

3 Generalized Secure Aggregation

In this section we present a generalized version of the secure aggregation protocol of Bell et al. [5] (SecAgg), where we abstract the method used by each party to hide its input as an encoding scheme $(\mathsf{Encode}, \mathsf{Decode})$. This encoding scheme should be additively homomorphic in both keys and values, meaning $\sum_i \mathsf{Encode}(\mathbf{sk}_i, \mathbf{x}_i) = \mathsf{Encode}(\sum_i \mathbf{sk}_i, \sum_i \mathbf{x}_i)$, and thus $\mathsf{Decode}(\sum_i \mathbf{sk}_i, \sum_i \mathsf{Encode}(\mathbf{sk}_i, \mathbf{x}_i)) = \sum_i \mathbf{x}_i$.

A simplified version of SecAgg is in Figure 1, and a full formal specification is in Algorithm 6 (in the appendix). This also contains the additional interactions needed to support input validation, which we ignore for now but describe in the next section. We then provide two examples of how this encoding can be instantiated: the first allows us to recover the original Bell et al. PRG-SecAgg construction, while the second provides a more efficient construction, RLWE-SecAgg (as we confirm experimentally in Section 6).

Commitments, distributed graph generation, and seed sharing. At its heart, SecAgg consists of two interactions between a set of clients and a server. In the first, `ShareSeeds`, each client i takes as input some randomness and learns four pieces of information: (1) a *pairwise seed* $\mathbf{seed}_{i,j}$ that it shares with each neighbor j in some defined communication graph, (2) a *self* seed $\mathbf{seed}_i$, and sets of shares (3) $\mathsf{shares}_{i,\mathcal{D}}$, corresponding to shares of these different seeds that this client should provide to the server for neighbors that *drop out* and (4) $\mathsf{shares}_{i,\mathcal{S}}$ that the client should provide to the server for neighbors that do not. In a slight abuse of notation, we write this as $(\varepsilon, \{\{\mathbf{seed}_{i,j}\}_{j\in \mathsf{N}(i)}, \mathbf{seed}_i, \mathsf{shares}_{i,\mathcal{D}}, \mathsf{shares}_{i,\mathcal{S}}\}_i) \leftarrow \mathtt{ShareSeeds}(\varepsilon, \{\mathsf{rand}_i\}_i)$, where the first input (and output) denote the input (and output) of the server, which in this protocol should learn nothing, and the remaining sets denote the individual inputs (and outputs) of the clients. Some of the main challenges of this first protocol lie in ensuring that honest clients do not have too many malicious neighbors in the communication graph, and that $\mathbf{seed}_{i,j} = \mathbf{seed}_{j,i}$ for all pairs of honest neighbors i and j. This latter property is crucial in ensuring that the derived *masks* cancel when masked inputs are aggregated by the server.

Masking. Using the information learned in this first interactive protocol, client i can then mask its input $\mathbf{x}_i$ using an encoding key computed as

$$\mathbf{sk}_i = \mathbf{s_i} + \sum_{j\in A_i, j<i} \mathbf{s_{ij}} - \sum_{j\in A_i, i<j} \mathbf{s_{ij}}, \tag{3}$$

where $\mathbf{s_{ij}} = \mathsf{F}(\mathbf{seed}_{ij})$, $\mathbf{s_i} = \mathsf{F}(\mathbf{seed}_i)$ for a length-expanding function F, and A_i are the neighbors that i believes to be survivors at this step in the protocol. It then encodes its input as $\mathbf{y}_i = \mathsf{Encode}(\mathbf{sk}_i, \mathbf{G}\mathbf{x}_i)$, where $\mathbf{G}$ is a matrix that allows us to *pack* multiple entries of $\mathbf{x}_i$ into a single plaintext slot; we discuss this in more detail below.

Dropout agreement and unmasking. If all clients are honest and do not drop out, then all their pairwise masks cancel, meaning $\sum_i \mathbf{sk}_i = \sum_i \mathbf{s_i}$. In this case, each client could just provide their individual self mask $\mathbf{s_i}$ to the server at the end of the protocol, who could then take advantage of the dual-homomorphic property of the encoding scheme to compute $\sum_i \mathbf{x}_i = \mathbf{G}^{-1}(\mathsf{Decode}(\sum \mathbf{s_i}, \sum \mathbf{y}_i))$. To account for clients who drop out, however, the server must have a way to recover their pairwise masks in order to cancel them out itself from the keys of *surviving* clients (e.g., if a surviving client i used $\mathbf{s_{ij}}$ for a dropped out client j in forming $\mathbf{sk}_i$, there is no corresponding $\mathbf{sk}_j$ containing $-\mathbf{s_{ij}}$ to cancel out the masks “naturally”).

The second interactive protocol that SecAgg provides is thus $\sum \mathbf{s}_i, \{\mathcal{D}_i, \mathcal{S}_i\} \leftarrow \mathtt{RecoverAggKey}(\varepsilon, \{\mathsf{shares}_{i,\mathcal{D}}, \mathsf{shares}_{i,\mathcal{S}}\})$, which allows the server to recover the aggregate key and thus compute the aggregated input as described above. Intuitively, in this protocol each client i sends a share of the self seed for each surviving neighbor (in $\mathcal{S}_i$) and a share of the pairwise seed for each dropped out neighbor (in $\mathcal{D}_i$), which allows the server to recompute the mask and learn the aggregate encoding key. In more detail, the server computes the aggregate key $\mathbf{sk}$ as

$$\mathbf{sk} = \sum_{i\in\mathcal{S}} (\mathbf{s_i} + \sum_{j\in\mathcal{D}_i, j<i} \mathbf{s_{ij}} - \sum_{j\in\mathcal{D}_i, i<j} \mathbf{s_{ij}}),$$

where $\mathbf{s_i}$ is reconstructed from the shares provided by the neighbors of a surviving client i and $\mathbf{s_{ij}}$ is reconstructed from the shares provided by i for dropped out neighbors j. Crucially, this process requires clients and the server to agree on the set of dropouts and survivors, as otherwise even honest clients could inadvertently reveal information that would allow the server to unmask an individual honest contribution.

3.1 PRG-SecAgg

We can recover the original PRG-SecAgg protocol [5] by instantiating F as a seed-stretching PRG and the encoding scheme as follows, for $\mathbf{sk}_i, \mathbf{x}_i, \mathbf{y} \in \mathbb{Z}_q^\ell$:

$$\begin{aligned} \mathsf{Encode}(\mathbf{sk}_i, \mathbf{x}_i) &:= \mathbf{sk}_i + \mathbf{x}_i \bmod q \\ \mathsf{Decode}(\mathbf{sk}_i, \mathbf{y}) &:= \mathbf{y} - \mathbf{sk}_i \bmod q \end{aligned} \tag{4}$$

3.2 RLWE-SecAgg

Our second SecAgg instantiation, RLWE-SecAgg, leverages an encoding based on RLWE. In this case, the key expansion algorithm samples a key from the appropriate RLWE secret distribution χ_s and then generates the masks as RLWE samples using the expanded key. This combined process of key sampling and mask generation is much more computationally efficient than the key expansion in PRG-SecAgg.

Public parameters: Vector length ℓ, input domain $\mathbb{X}^\ell$, secret distribution χ_s, and seed expansion function $\mathsf{F}\colon \{0,1\}^\lambda \mapsto \mathsf{supp}(\chi_s)^\ell$
Client i's input: $\mathbf{x}_i \in \mathbb{X}^\ell$
Server output: $z \in \mathbb{X}^\ell$

1. Using the server to send messages, clients engage in the `ShareSeeds` protocol, with each surviving client i learning $\{\mathbf{seed}_{i,j}\}_{j\in \mathsf{N}(i)}$, $\mathbf{seed}_i$, $\mathsf{shares}_{i,\mathcal{D}}$, and $\mathsf{shares}_{i,\mathcal{S}}$. The server aborts if there are fewer than $(1-\delta)n$ surviving clients.
2. Each surviving client i performs the following:
   - Computes its *packed encrypted input* $\mathbf{y}_i = \mathsf{Encode}(\mathbf{sk}_i, \mathbf{G}\mathbf{x}_i)$ with key defined as $\mathbf{sk}_i = \mathbf{s}_i + \sum_{j\in A_i, j<i} \mathbf{s}_{ij} - \sum_{j\in A_i, i<j} \mathbf{s}_{ij}$ for $\mathbf{s}_{ij} = \mathsf{F}(\mathbf{seed}_{i,j})$, $\mathbf{s}_i = \mathsf{F}(\mathbf{seed}_i)$ (as in Equation 3).
   - Forms commitments $\mathsf{com}_{\mathbf{sk},i}$ and $\mathsf{com}_{\mathbf{x},i}$ to its key and input respectively.
   - Computes proofs $\pi^{\mathsf{Enc}(\mathbf{sk}_i,\mathbf{x}_i)}$, $\pi^{0\leq \mathbf{x}_i<t}$, and $\pi^{\mathsf{valid}(\mathbf{x}_i)}$ of encoding, smallness, and validity.
   - Sends to the server $\mathbf{y}_i$, $\mathsf{com}_{\mathbf{sk},i}$, $\mathsf{com}_{\mathbf{x},i}$, $\pi^{\mathsf{Enc}(\mathbf{sk}_i,\mathbf{x}_i)}$, $\pi^{0\leq \mathbf{x}_i<t}$, $\pi^{\mathsf{valid}(\mathbf{x}_i)}$.
3. The server aborts if it receives fewer than $(1-\delta)n$ messages or if any of the proofs fail to verify. Otherwise, the server and the clients engage in the `RecoverAggKey` protocol, with the server taking as input the global sets $\mathcal{D}$ and $\mathcal{S}$ of dropouts and survivors and each client i taking as input its sets $\mathsf{shares}_{i,\mathcal{D}}$ and $\mathsf{shares}_{i,\mathcal{S}}$ and providing the appropriate shares to the server according to the status of their neighbors. At the end of the protocol the server learns the aggregate key $\mathbf{sk}$.
4. Each client, acting as a distributed prover, engages with the server (acting as the verifier) in the distributed key correctness protocol. The server aborts if the collective proof fails to verify.
5. The server outputs $\sum_{i\in\mathcal{S}} \mathbf{x}_i$ as $\mathbf{G}^{-1}(\mathsf{Decode}(\mathbf{sk}, \sum_{i\in\mathcal{S}} \mathbf{y}_i))$.

Figure 1: General SecAgg protocol with input verification.

Unlike PRG-SecAgg, this encoding requires a set of public parameters: a polynomial ring $R = \mathbb{Z}[X]/(X^N+1)$ and its residual ring $R_q = \mathbb{Z}[X]/(q, X^N+1)$ for a modulus q, a plaintext modulus T that is coprime to q, a plaintext dimension ℓ, a secret key distribution χ_s and an error distribution χ_e over R, and a matrix $\mathbf{A}$ generated as discussed in Section 2.3. These parameters can be distributed to the clients by the server or through a public channel. They are used in the encoding and decoding algorithms, defined as follows:

$$\begin{aligned} \mathsf{Encode}(\mathbf{sk}_i, \mathbf{x}_i) = \mathbf{y}_i &:= \mathbf{A}\cdot \mathbf{sk}_i + T(\mathbf{e}+\mathbf{f}) + \mathbf{x}_i \bmod q, \\ &\quad \text{where } \mathbf{e}, \mathbf{f} \leftarrow \chi_e^{\ell/N}. \qquad (5) \\ \mathsf{Decode}(\mathbf{sk}, \mathbf{y}) &:= (\mathbf{y} - \mathbf{A}\cdot \mathbf{sk} \bmod q) \bmod T. \end{aligned}$$

We present formal proofs of the correctness and security of this encoding in Appendix A, but provide some intuition for them here.

Correctness. To ensure that the obtained result is the sum of the $\mathbf{x}_i \in \mathbb{Z}_t$ over the integers we need that (i) the sum of errors and messages does not overflow the ciphertext modulus q, and (ii) the sum of the messages does not overflow the plaintext modulus T. These result in the constraints $2nTb_e < q$ and $nt < T$, where b_e is an L_∞ bound on the error $e \leftarrow \chi_e$.

Security. It is tempting to claim that all we need for security is to choose RLWE parameters in a way that ensures the individual encodings $\mathbf{y}_i$ of client contributions are pseudorandom in isolation. However, the server gets more information than just n independent RLWE ciphertexts, as it can also recover $\sum_i \mathbf{e}_i$ from $\tilde{\mathbf{y}} = \sum \mathbf{y}_i$. A common approach to eliminate leakage is to add a large noise to "drown" the error [19, Chapter 21], in a way analogous to how circuit privacy is achieved in some (R)LWE-based homomorphic encryption schemes. The resulting modulus q would be very large, however, which hurts both computation and communication.

Instead, we argue in Appendix A that the encodings of all clients' inputs are indistinguishable from random values that sum up to an encoding of the sum of all inputs. This property can be established from the hardness of an RLWE variant, *Hint-RLWE*, in which samples consist of standard RLWE pairs $(a, as+e) \in R_q^2$ and a "hint" $e+f$, where f is sampled from the same Gaussian distribution as e. The additional noise term f allows us to gradually break the correlation among the shared secrets used in the ciphertexts of neighboring clients, via a carefully constructed hybrid argument (see Lemma 4 for details). Lee et al. [32] showed that the Hint-RLWE problem with error size σ is as hard as the standard RLWE with error size $(1/\sqrt{2})\sigma$. The error terms in our RLWE encodings are thus only slightly larger than standard RLWE encryption, avoiding the need for noise flooding.

Ciphertext expansion. PRG-SecAgg has very limited ciphertext expansion, which is optimal in the sense that the modulus q can be chosen to be exactly tn, to ensure that the result of adding all n values fits in the modulus. This results in only a $1 + \frac{\log_2 n}{\log_2 t}$ factor overhead with respect to an insecure solution where clients just send their values. A naïve encoding in RLWE-SecAgg that puts each entry of $\mathbf{x}_i$ in a polynomial coefficient would result in a $1 + \frac{\log_2 q}{\log_2 t}$ factor overhead. This can be quite wasteful, as q needs to be large ($\geq 2^{51}$) for security. However, we can use a larger plaintext modulus T to *pack* multiple entries of $\mathbf{x}_i$ in a plaintext slot.

In more detail, let $\mathbf{G}$ be the gadget matrix $\mathbf{G} = (1, t, t^2, \ldots, t^{p-1}) \otimes \mathbf{I}_{l/p}$ for $p = \lfloor \log T / \log(nt) \rfloor$. Then by computing $\mu_i = \mathbf{G}\mathbf{x}_i \in [T]^{l/p}$, we effectively pack every p entries of the input $\mathbf{x}_i$ into a single plaintext slot of μ_i while ensuring that the result of the *packed* sum fits in T. To decode from a packed slot, one can apply a digit extraction algorithm for base t, denoted by $\mathbf{G}^{-1}$, which can be naturally extended to a packed vector. Importantly, this packing operations is linear, and thus it can be incorporated into the input validity constraints we consider in the next section.

4 Adding Input Validation

In this section we present ACORN, an extension to the generalized SecAgg protocol that allows for client input validation. Specifically, we provide a way for the server to check that the (hidden) inputs of clients satisfy some pre-defined notion of validity and that their messages in the protocol have been computed according to its specification. We first present ACORN-detect, where the server can detect that misbehavior has occurred but cannot attribute it to an individual client or recover from it, and then present ACORN-robust, in which the server can both identify misbehaving clients and remove their contributions from the final sum.

To achieve this, as described below we require non-interactive zero-knowledge proofs of vector smallness and valid encoding, and an interactive proof for the correctness of an aggregated key. We instantiate these primitives in Section 5 with efficient discrete log-based protocols.

4.1 Detecting Client Misbehavior

We present our summary protocol of ACORN-detect in Figure 1 and our detailed protocol in Figure 6, where the additional steps required for input validation are in red. Across the entire protocol, we require a zero-knowledge proof of the following relation R:

$$\Big\{(x,w) \mid x = ((\mathbf{y}_i, \mathsf{com}_{\mathbf{x},i}, \mathsf{com}_{\mathbf{sk},i})_{i\in\mathcal{S}}, \mathbf{sk}, t, \ell, \mathbf{G}), w = (\mathbf{x}_i, \mathbf{sk}_i, r_i, s_i)_{i\in\mathcal{S}},$$
$$\forall i \in \mathcal{S} : \big(\mathsf{com}_{\mathbf{x},i} = \mathsf{Commit}(\mathbf{x}_i; r_i), \mathsf{com}_{\mathbf{sk},i} = \mathsf{Commit}(\mathbf{sk}_i; s_i),$$
$$\mathbf{y}_i = \mathsf{Encode}(\mathbf{sk}_i, \mathbf{G}\mathbf{x}_i), \mathbf{x}_i \in \mathbb{Z}_t^{\ell}, \mathsf{valid}(\mathbf{x}_i)\big), \sum_{i\in\mathcal{S}} \mathbf{sk}_i = \mathbf{sk}\Big\}$$

We first observe that the witness for this relation is distributed among the clients, with each client i holding $\mathbf{x}_i$ and $\mathbf{sk}_i$ (and the relevant randomness) but being unaware of the other inputs. All the conditions of the relation except the last one, however, are on the individual components and thus each client can prove them independently. This means forming

1. A proof $\pi^{\mathsf{Enc}(\mathbf{sk}_i,\mathbf{x}_i)}$ that $\mathbf{y}_i = \mathsf{Encode}(\mathbf{sk}_i, \mathbf{G}\mathbf{x}_i)$, where $\mathbf{sk}_i$ and $\mathbf{x}_i$ are the values contained in the relevant commitments.

2. A proof $\pi^{\mathsf{valid}(\mathbf{x}_i)}$ that $\mathsf{valid}(\mathbf{x}_i)$ holds.

3. A proof $\pi^{0\leq \mathbf{x}_i < t}$ that $\mathbf{x_i} \in \mathbb{Z}_t^{\ell}$. This condition is needed to prove that no wraparound happens modulo the plaintext space, and thus that the packed sum can be decoded using $\mathbf{G}^{-1}$.

These proofs and the two commitments $\mathsf{com}_{\mathbf{sk},i}$ and $\mathsf{com}_{\mathbf{x},i}$ are sent to the server at the same time as the masked input $\mathbf{y}_i$.

With individual proofs for these individual constraints, the only remaining requirement of R is that $\sum_i \mathbf{sk}_i = \mathbf{sk}$. Clients could prove this individually by proving that they formed $\mathbf{sk}_i$ as specified by the protocol (Equation 3), but as the formation of $\mathbf{sk}_i$ requires key agreements and applications of a length-expanding function F this would be highly inefficient.

Instead, we have the clients prove *collectively* that $\sum_i \mathbf{sk}_i = \mathbf{sk}$, which is the minimal requirement needed for the server to decode and recover the aggregated inputs. This is done by having each client i provide a (partial) proof $\pi_i^{\sum \mathbf{sk}_i}$, which they can do without interacting with other clients. These proofs collectively demonstrate the correctness of the aggregated key $\mathbf{sk}$. Unlike the individual proofs, this proof cannot be made non-interactive, so we instead consider it as an interaction between each client and the server. In our summarized presentation in Figure 1 we present this as a separate step (Step 4), but in our detailed presentation in Figure 6 we show how this protocol can be woven into the broader SecAgg protocol without requiring any additional rounds of interaction.

Security. We formally prove the security of ACORN-detect, following a simulation-based argument [22, 33], in Appendix B. Briefly, security for an honest server follows from the knowledge soundness of the proofs, which gives the simulator the ability (acting as the server) to extract the underlying inputs. Acting as the knowledge extractor means the simulator here needs to rewind, and thus that ACORN-detect is not concurrently secure in the honest server setting. Security in the malicious server setting is largely orthogonal to the question of input validation, and thus our proof follows closely the one of Bell et al. [5], with the simulator additionally relying on zero knowledge to ensure that its interactions with the adversary are indistinguishable from what it expects.

Efficiency. The asymptotic costs for the protocol, using the instantiations in Section 5, are in Table 1. Step 1 requires $\log(n)$ work per client, and is concretely very cheap. Client costs are thus dominated by the following tasks in Step 2: (1) running Encode and expanding seeds to compute $\mathbf{sk_i}$, (2) committing to $\mathbf{x_i}$ and $\mathbf{sk_i}$, and (3) generating proofs. The server's work is dominated by the analogous tasks of (1) verifying proofs in Step 3 and (2) computing $\mathbf{sk}$ in Step 4.

In PRG-SecAgg, the encoding step corresponds to PRG expansions (implemented with AES), and in RLWE-SecAgg it corresponds to the noisy linear transformation in Equation 5. As we show in Section 5, we use a discrete log-based proof system, and thus commitment generation means computing two Pedersen vector commitments (requiring two length-ℓ group multi-exponentiations), and proof generation requires $O(\ell \log(\ell))$ computation to produce a logarithmic size proof.

4.2 Robustness in the Face of Misbehavior

We next present ACORN-robust, which allows the server to not only identify misbehaving clients, but also exclude their input from the result on-the-fly. This property, sometimes refereed to as *guaranteed output delivery* [27], ensures in this context that as long as the number of cheating clients stays below a given threshold α and no more than $(\delta-\alpha)n$ other clients drop out, an honest server is guaranteed a valid output. We present and prove ACORN-robust secure assuming a semi-honest server; an extension to a malicious server seems possible, but we leave this as future work.

In ACORN-detect, clients proved the correctness of the aggregated key by providing the minimal amount of information needed to do so: each client committed to its overall secret key rather than the pairwise masks or seeds used to form it. This allowed the server to be convinced of the correctness of the aggregate key, but not to identify which clients were cheating if the proof failed.

ACORN-robust thus replaces this proof of aggregated key correctness with a more fine-grained approach in which instead of a commitment $\mathsf{com}_{\mathbf{sk},i}$, clients form commitments $\mathsf{com}_{\mathbf{s_{ij}}}$, $\mathsf{com}_{\mathbf{seed}_{i,j}}$, $\mathsf{com}_{\mathbf{s_i}}$, and $\mathsf{com}_{\mathbf{seed}_i}$ to, respectively, their pairwise masks $\mathbf{s_{ij}}$ and seeds $\mathbf{seed}_{i,j}$ and self masks $\mathbf{s_i}$ and seeds $\mathbf{seed}_i$. We see below how this allows honest clients to support the server in verifying each of their neighbors' masks, but first confirm the effect this has on the "individual" proofs used in ACORN-detect.

1. The proof of valid encoding $\pi^{\mathsf{Enc}(\mathbf{sk}_i,\mathbf{x}_i)}$ is still provided with respect to the commitment $\mathsf{com}_{\mathbf{x},i}$ as before, but commitments $\mathsf{com}_{\mathbf{s_i}}$ and $\{\mathsf{com}_{\mathbf{s_{ij}}}\}_{j\in\mathcal{S}_i}$ replace the single "monolithic" commitment $\mathsf{com}_{\mathbf{sk},i}$. However, the proof does not change, as the server can obtain $\mathsf{com}_{\mathbf{sk},i}$ by combining the commitments to $\mathsf{com}_{\mathbf{s_i}}$ and $\{\mathsf{com}_{\mathbf{s_{ij}}}\}_{j\in\mathcal{S}_i}$ according to the formula for $\mathbf{sk}_i$ (Equation 3).

2. The proof $\pi^{\mathsf{valid}(\mathbf{x}_i)}$ remains the same, with respect to $\mathsf{com}_{\mathbf{x},i}$.

3. The proof $\pi^{0\leq\mathbf{x}_i<t}$ also remains the same, with respect to $\mathsf{com}_{\mathbf{x},i}$.

Whereas ACORN-detect could use the Bell et al. protocols for `ShareSeeds` and `RecoverAggKey` as a black box, ACORN-robust requires changing them. We thus summarize these changes here, focusing on the way they allow the server to robustly reconstruct the aggregated key; the formal protocol descriptions are in Appendix C.

`ShareSeeds` **(Algorithm 2).** Our new protocol variant provides three main guarantees:

Correct seed sharing. All neighbors of a client i receive a *correct sharing* of $\mathbf{seed}_{i,j}$ (resp. $\mathbf{seed}_i$), i.e. a sharing matching $\mathsf{com}_{\mathbf{s_{ij}}}$ (resp. $\mathsf{com}_{\mathbf{seed}_i}$). We achieve this by switching from Shamir secret sharing for pairwise seeds to Feldman's *verifiable secret sharing* (VSS) [17]. If a malicious client does not correctly share a seed they are thus dropped by the `ShareSeeds` protocol.

Seed-mask consistency, for honestly supervised pairwise masks. `ShareSeeds` also ensures that the pairwise mask $\mathbf{s_{ij}}$ used by neighbors i and j was correctly computed as $\mathbf{s_{ij}} = \mathsf{F}(\mathbf{seed}_{i,j})$, as long as either i or j are honest. To do this, client i generates $\mathbf{seed}_{i,j}$ and sends it to j, encrypted under j's public key, along with a deterministic commitment $g^{\mathbf{seed}_{i,j}}$ and a commitment $\mathsf{com}_{\mathbf{s_{ij}}}$ to $\mathbf{s_{ij}}$ that uses randomness $s_{i,j}$ derived from $\mathbf{seed}_{i,j}$. Client j can then decrypt to recover $\mathbf{seed}_{i,j}$, expand it to recover both $\mathbf{s_{ji}}$ and $s_{i,j}$, and form its own commitment $\mathsf{com}_{\mathbf{s_{ji}}}$. It can then check that (1) $\mathbf{seed}_{i,j}$ matches the commitment sent by i and that (2) $\mathsf{com}_{\mathbf{s_{ji}}} = \mathsf{com}_{\mathbf{s_{ij}}}$ (i.e. that the commitments are identical). If either of these checks fails, j can complain to the server by sending it the decryption key; this allows the server to rerun these steps, check the inequalities to confirm that i misbehaved, and drop it.

Independence of supervised and inconsistent masks. The previous guarantee does not ensure consistency of pairwise masks if *both* i and j are malicious clients, as j can simply not supervise i's mask as prescribed. Therefore, it might be the case that $\mathbf{s_{ij}} = \mathsf{F}(\mathbf{seed}_{i,j})$ does not hold after `ShareSeeds`, but in that case both i and j must have misbehaved. The `RecoverAggKey` protocol in ACORN-robust handles that case by checking consistency after reconstruction, as we discuss below. To enable this, client i must commit to $\mathbf{seed}_{i,\cdot}$ *before* it gets assigned to neighbor j, which is done at random. Therefore, i's decision to violate $\mathbf{s_{ij}} = \mathsf{F}(\mathbf{seed}_{i,j})$ is independent of the fact that j is malicious.

`ShareSeeds` only handles seed-mask inconsistency for pairwise masks $\mathbf{s_{ij}}$, but this sort of inconsistency is also a problem for the self-mask $\mathbf{s_i}$. As we see below, however, an inconsistency in a self-mask is analogous to an inconsistency in a pairwise mask, given the guarantees of `ShareSeeds` described above: whenever the server discovers an inconsistent mask in the recovery phase, it also identifies a misbehaving client and proceeds to drop it in an additional round. We describe how this is done next.

`RecoverAggKey` **(Algorithm 4).** When it comes to reconstructing the self masks $\mathbf{s_i}$ for surviving clients i, we consider two cases: one in which the server is unable to reconstruct $\mathbf{seed}_i$ given the shares provided by i's neighbors, and one in which it can reconstruct but $\mathsf{F}(\mathbf{seed}_i) \neq \mathbf{s_i}$. Given commitments and signatures that i provided in `ShareSeeds`, the server can identify i as the only possible misbehaving client in either case and seek to retrospectively drop them from the protocol. This requires one extra round, and is discussed later.

When it comes to reconstructing the pairwise masks $\mathbf{s_{ij}}$, we consider the same two cases: one in which reconstruction fails and the one in which reconstruction succeeds but the reconstructed seed is such that $\mathbf{s_{ij}} \neq \mathsf{F}(\mathbf{seed}_{i,j})$, i.e. inconsistent with the corresponding committed mask. Luckily, we can argue that the former case cannot happen because we assume a sufficient threshold of honest clients (this follows from Bell et al. [5]). We thus focus on the latter case: $\mathbf{s_{ij}} \neq \mathsf{F}(\mathbf{seed}_{i,j})$.

Recall that the server needs to reconstruct $\mathbf{seed}_{i,j}$ to recover $\mathbf{s_{ij}}$ because i dropped out but j did not. If $\mathbf{s_{ij}} \neq \mathsf{F}(\mathbf{seed}_{i,j})$, the server can conclude that j is also dishonest thanks to the guarantees of `ShareSeeds`, and proceed to drop it.

Retrospectively dropping clients. In three cases discussed above we need to drop a client i *after* it submits its masked input. This involves reaching out to all neighbors j of i to obtain shares to recover $\mathbf{seed}_{i,j}$. In doing so, however, the server might discover that a neighbor j is dishonest and also needs to be dropped, which requires another round. In this process, the server uncovers all remaining inconsistent seed-mask pairs, i.e. pairs such that $\mathbf{s_{ij}} \neq \mathsf{F}(\mathbf{seed}_{i,j})$, which necessarily correspond to pairs of dishonest client i and j. Since the total number of dishonest clients is $\gamma n = O(n)$, naïvely the adversary could delay output delivery for $O(n)$ rounds. This is exactly why we require clients to commit to their seeds before being assigned their neighbors, and in particular is the benefit of our guaranteed independence of supervised and inconsistent masks. By doing this a corrupted client has at most constant probability of having another corrupted client as its neighbor, and thus we can show via a random graphs argument that with high probability the number of rounds required to exclude αn clients is $O(\log_{\alpha^{-1}}(n))$. We discuss this in more detail below.

Security. We formally prove the security of ACORN-robust in Appendix C.3. The proof for the honest server case is very similar to the proof for ACORN-detect, with the main difference being that the server aborts only if there have been too many dropouts (i.e., it does not abort just because proofs fail to verify). The proof for the semi-honest server closely follows the proof by Bell et al. As expected, the main difference with the detection-only case is that the ideal functionality for ACORN-robust is slightly modified to account for the fact that invalid inputs are dropped from the final sum.

Efficiency. We now discuss the overhead of ACORN-robust on top of ACORN-detect. Asymptotic costs are in Table 1.

Additional commitments. As opposed to just committing to $\mathbf{sk}$, clients need to commit to self and pairwise masks, resulting in $k/2$ additional length-ℓ vector commitments in the first step of the protocol, where $k = O(\log n)$ is the number of neighbors. For reference, $k \leq 70$ and $k \leq 150$ are large enough to provide *statistical* security $\sigma = 40$ up to $n = 10^5$, for a semi-honest and malicious server, respectively.

Checking commitments. Recall that clients check commitments for seed-mask consistency, and the server verifies inconsistencies. This is another k length-ℓ multi-exponentiations for each client and $O(kn)$ for the server. These computations can be batched, however, and can of course be avoided if no inconsistent masks are found. It is cheaper than the first step as long as the fraction of misbehaving clients is below a third. It is possible to optimize the task of efficiently finding inconsistent masks (i.e. identifying which m commitments amongst n are bad), but as a rough bound it can be done with $O(m + \log(n))$ multi-exponentiations. Each client must also expand $k+1$ seeds.

Additional secret-sharing. There are $k = O(\log n)$ times as many secrets to share as in the ACORN-detect case (i.e. $O(k^2)$ shares rather than $O(k)$), but this is still independent of ℓ and share generation is very efficient. The total communication is thus still dominated by the masked input for even moderately large values of ℓ.

Round complexity. ACORN-detect takes five rounds, while ACORN-robust takes six rounds if no inconsistent masks are found. If they are, however, the total number of rounds increases according to the number of rounds needed to retrospectively drop clients. We can bound this number of rounds using the following theorem, which we prove in Appendix C.

Theorem 4.1. *Consider an execution of ACORN-robust where at most $\alpha n < n/3$ clients misbehave, where α plus the fraction of dropouts is less than δ. Then the probability that ACORN-robust requires at least $6 + r$ rounds to finish is bounded by $\mathrm{negl}(\sigma) + (\alpha n/2 + n/k)(\sqrt{8}\alpha)^{r-1}$, where σ is a statistical security parameter.*

5 Zero-Knowledge Constructions

In this section we provide constructions of the zero-knowledge proofs required in ACORN. Specifically, we instantiate proofs of *aggregated keys*, *correct encoding*, *input smallness*, and *input validity*. For ACORN-detect, we present the distributed proof of aggregated key correctness below, which is a variant of a Schnorr proof with an additional step to ensure that the interactive protocol achieves (full) zero knowledge. For ACORN-robust, this proof is integrated into the protocol itself. For the latter three predicates, we leverage the Bulletproofs system, which we present before presenting our proofs for these individual predicates.

5.1 Distributed Proof of Aggregated Key Correctness

In ACORN-detect, each client i commits to its encoding key sk_i, which incorporates its self mask and pairwise masks, using some randomness r_i; in other words, it creates an Pedersen commitment $C_i = \mathbf{g}^{\mathsf{sk}_i} h^{r_i}$. At the end of the protocol, the server learns the sum of self masks $\mathbf{s}$ in the clear. Our distributed key correctness protocol, presented in Figure 2, thus aims to convince the server that $\prod_i C_i$ is a commitment to $\mathbf{s}$, where the witness for this proof (consisting of the opening of $\prod_i C_i$) is additively shared among all the clients.

Our protocol uses the Schnorr proof of knowledge, with each client proving knowledge of the randomness r_i used to form its commitment. The server, acting as the verifier, can combine these proofs using the fact that the challenge

Public parameters: group $\mathbb{G}, \mathbb{H}$ with generators $\mathbf{g} \in \mathbb{G}^{\ell}$ and $h \in \mathbb{H}$

Client i's input: $\mathbf{sk}_i$, r_i, $C_i = \mathsf{Commit}(\mathbf{sk}_i; r_i)$

Server's input: $\mathbf{s} \in |\mathbb{G}|^{\ell}$, $\{C_i\}_i$

1. Client i samples α_i randomly and sends $h_i = h^{\alpha_i}$ to the server.
2. The server samples a random challenge e and forms a commitment $\mathsf{com}_{i,\mathsf{chl}} = g^e h_i^{s_i}$ for each client i, using some randomness s_i. The server sends $\mathsf{com}_{i,\mathsf{chl}}$ to client i.
3. Each client i samples a random value $k_i \in |\mathbb{H}|$ and sends $K_i = h^{k_i}$ to the server.
4. The server opens the committed challenge by sending e and s_i to client i.
5. Client i checks that $\mathsf{com}_{i,\mathsf{chl}} = g^e h_i^{s_i}$, where $\mathsf{com}_{i,\mathsf{chl}}$ is the value it received in Step 2. If not, it aborts. Otherwise, it computes $t_i = r_i \cdot e + k_i$ and sends it and α_i to the server.
6. The server checks that $h_i = h^{\alpha_i}$. If so, it computes $C = \prod_{i \in [n]} C_i / \mathbf{g}^{\mathbf{s}}$ and $t = \sum_{i \in [n]} t_i$. It checks that $h^t = C^e \prod_{i \in [n]} K_i$ and outputs 1 if this check passes and 0 otherwise.

Figure 2: Distributed key correctness protocol for proving that $\sum_i \mathbf{sk}_i = \mathbf{s}$.

response in Schnorr is a linear function of the committed value.

To tolerate a malicious server, we need this proof to be (fully) zero knowledge, which the sigma protocol can be when instantiated as a non-interactive proof using the Fiat-Shamir transformation. However, this does not work in our setting since clients act as distributed provers and do not have the same view that could then be hashed to form the challenge. Trying to send the required information to all clients to obtain such a common view is not viable since it incurs a prohibitive communication overhead.

Instead, we modify the execution so that the server commits to its challenge ahead of time using parameters provided by each client. To retain the property that this is a proof of knowledge, we require that this commitment is *equivocable*, as the knowledge extractor needs to be able to send two different challenges consistent with the same commitment. For example, Pedersen commitments provide equivocation when the discrete logarithm between the generators g and h is known.

Correctness. We can verify that

$$h^t = \prod_{i \in [n]} h^{t_i} = (\prod_{i \in [n]} h^{r_i})^e (\prod_{i \in [n]} h^{k_i}) = C^e \prod_{i \in [n]} K_i.$$

Knowledge soundness. The soundness of the protocol follows similarly to the soundness of the single prover Schnorr protocol. The extractor can rewind the execution of steps 3 and 4 with the i-th client and provide two different openings e_1 and e_2 for the committed challenge using the equivocability of the commitment scheme, and obtain two different values $t_{i,1}$ and $t_{i,2}$. If the proof verifies in both cases, then $h^{t_{i,1}} = h^{t_{i,2}}$, and the extractor can compute $r_i = (t_1 - t_2)(e_1 - e_2)^{-1}$ since $e_1 - e_2 \neq 0$.

Zero knowledge. The simulator for client i rewinds step 2 and 3 after it has obtained the opening of the commitment for challenge e. It generates t_i at random and sets $K_i = h^{t_i}(h^{r_i})^{-e}$.

5.2 Inner Product Proofs

We build our zero-knowledge proofs using Bulletproofs [11]. In the context of this work, similarly to Gentry et al. [20], we regard Bulletproofs as a zero-knowledge proof of knowledge of vectors $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_q^n$ that satisfy an inner product constraint $\langle \mathbf{x}, \mathbf{y} \rangle = a$ in an order-q group G, where a is a public scalar.

At a high level, to prove $\langle \mathbf{x}, \mathbf{y} \rangle = a$ the prover recursively computes a new equation $\langle \mathbf{x}', \mathbf{y}' \rangle = a'$ for vectors of half the length, and computes commitments given a challenge sent by the verifier. This requires both the prover and verifier to compute new generators at each recursive step, with the prover also computing new commitments to $\mathbf{x}'$ and $\mathbf{y}'$. In the non-interactive variant using the Fiat-Shamir heuristic the generator computation can be unfolded, resulting in a single multi-exponentiation of length $2n$. We state the concrete costs for Bulletproofs in terms of multi-exponentiation operations, for which efficient sublinear algorithms are known [40]. The full protocol details are given by Gentry et al. [20, Section E.2].

Lemma 1 ([11, 20])**.** *Let $C \in \mathsf{G}$ be a group element, and let $\mathbf{h} \in \mathsf{G}^2, \mathbf{g} \in \mathsf{G}^{2n}$ be sets of generators in G known to both the prover and verifier. Bulletproofs allows the prover to prove knowledge of vectors $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_q^n$ and randomness $r \in \mathbb{Z}_q$ such that $C = h_0^r h_1^{\langle \mathbf{x}, \mathbf{y} \rangle} \prod_{i=1}^n \mathbf{g}_i^{\mathbf{x}_i} \mathbf{g}_{i+n}^{\mathbf{y}_i}$. It satisfies perfect completeness, statistical zero knowledge, and computational knowledge soundness under the discrete logarithm assumption.*

When compiled into a NIZK proof, the prover performs $6n + 8(\log_2(n) + 1)$ group exponentiations (computed as several multi-exponentiations), and the verifier performs a single multi-exponentiation of length $2(n + \log_2(n) + 3)$. Moreover, batched verification of m proofs requires a single multi-exponentiation of length $2n + 2 + m\log_2(2n + 4)$. The proof size is $2\log n + 4$ group elements.

5.3 Proofs of Smallness

We consider two variants of the problem of proving in zero knowledge that $\mathbf{x}_i \in [0, t-1]$ for all i: the first is efficient for $t = 2$, which is useful for applications that rely on binary k-hot encodings, while the second works for arbitrary values of t, which is useful in the learning setting and in our proof of correct encoding.

5.3.1 Proof for $t = 2$

We describe the protocol for $t = 2$ for simplicity, which closely follows the approach to range proofs in Bulletproofs [11]. Let $C \in \mathsf{G}$ be a group element, and let $h \in \mathsf{G}$ and $\mathbf{g}, \mathbf{h} \in \mathsf{G}^\ell$ be generators in G (public parameters).

1. The prover finds $\mathbf{y} \in \mathsf{G}^\ell$ satisfying (i) $\mathbf{x} \circ \mathbf{y} = 0$ and (ii) $\mathbf{x} = \mathbf{1}^\ell + \mathbf{y}$. These properties hold if and only if $\mathbf{x}$ is binary. The prover commits to $\mathbf{x}|\mathbf{y}$ as $C = h^r \prod_{i=1}^{\ell} \mathbf{g}_i^{\mathbf{x}_i} \mathbf{h}_i^{\mathbf{y}_i}$ and sends this to the verifier.

2. The verifier sends random challenge scalars $\tau, \rho \in \mathbb{Z}_q$ to the prover. Define $\mathbf{r} = (\tau^{i-1})_{i \in [\ell]}$.

3. By Schwartz-Zippel, $\langle \mathbf{x}, \mathbf{y} \circ \mathbf{r} \rangle + \rho(\langle \mathbf{x}, \mathbf{r} \rangle + \langle -\mathbf{1}^\ell, \mathbf{y} \circ \mathbf{r} \rangle) = \langle \mathbf{1}^\ell, \mathbf{r} \rangle$ holds if and only if (i) and (ii) hold, except with probability $(\ell + 1)/q$. This can be rewritten as a single constraint $\langle \mathbf{x}', \mathbf{y}' \rangle = (1 - \rho^2)\langle \mathbf{1}^\ell, \mathbf{r} \rangle$, for $\mathbf{x}' := \mathbf{x} - \mathbf{1}^\ell \rho$ and $\mathbf{y}' := \mathbf{y} \circ \mathbf{r} + \rho \mathbf{r}$. The prover and verifier can obtain a commitment C' to $\mathbf{x}'|\mathbf{y}'$ by computing $C' = C \cdot \prod_{i=1}^{\ell} \mathbf{g}_i^{-\rho} \mathbf{h}_i^{\rho}$. This is a commitment to $\mathbf{x}'$ and $\mathbf{y}'$ using generators $(h, \mathbf{g}, \mathbf{h}')$ where $\mathbf{h}'_i = \mathbf{h}_i^{\tau^{1-i}}$. The prover then uses Bulletproofs as described in Lemma 1 to prove that $\langle \mathbf{x}, \mathbf{y}' \rangle = 0$ with respect to C'.

This proof is thus a reduction to Bulletproofs (Lemma 1), requiring three additional length-ℓ multi-exponentiations in Step 3 by both the prover and verifier. However, this overhead can be reduced to only two length-ℓ multi-exponentiations for the verifier, as both the generator switch and commitment update can be combined with the analogous operations in the outer loop of Bulletproofs. Moreover, the proof can be made non-interactive via Fiat-Shamir, by deriving τ, ρ from the protocol transcript and proof statement.

5.3.2 Proof for arbitrary t

To show that $\mathbf{x}_i \in [a, b]$ it suffices to show that $(\mathbf{x}_i - a)(b - \mathbf{x}_i)$ is non-negative. We thus show that $c_i := \mathbf{x}_i(t - 1 - \mathbf{x}_i) \geq 0$. A common way to do this is to exhibit a decomposition of c_i into four squares. However, a useful optimization consists of showing that $c'_i := 4c_i + 1 \geq 0$ [23]. These two conditions are equivalent over the integers, but because $c'_i \equiv 1 \bmod 4$ it can be written as a sum of three squares, where the three squares can be efficiently determined [42].

For convenience, we write c'_i as $1 + (t-1)^2 - (2\mathbf{x}_i - t + 1)^2$. The protocol thus proceeds by having the client prove that $c'_i \geq 0$ for all i, by showing that it knows $\mathbf{u}, \mathbf{v}, \mathbf{w}$ such that

$$\mathbf{x}' \circ \mathbf{x}' + \mathbf{u} \circ \mathbf{u} + \mathbf{v} \circ \mathbf{v} + \mathbf{w} \circ \mathbf{w} = \mathbf{a} \tag{6}$$

holds over the integers, where $\mathbf{x}' := 2 \cdot \mathbf{x} - (t-1) \cdot \mathbf{1}$ and $\mathbf{a} := (-(1 + (t-1)^2)) \cdot \mathbf{1}$ is public. The prover must also show that these computations do not wrap around the modulus q, which means showing that

$$\|\mathbf{x}'|\mathbf{u}|\mathbf{v}|\mathbf{w}\|_\infty < \sqrt{q}/4 \tag{7}$$

At this point it might seem that we're going in circles, as we reduced a range proof to two constraints, one of which is itself a range proof. However, this second bound is very loose, because $\sqrt{q}/4 \gg t$. In our implementation of Bulletproofs we have $q > 2^{250}$, while we are interested in values of t for natural datatypes, e.g. $t = 2^{16}$. Therefore we can take advantage of approximate range proofs introduced by Gentry et al. [20] and also used in [35], whose properties (assuming Fiat-Shamir) are summarized in the following lemma.

Lemma 2 ([20, Lemma 3.5]). *Fix a security parameter λ. Let $\mathbf{z} \in \mathbb{Z}^\ell$ be a vector such that $\|\mathbf{z}\|_\infty \leq t$, and let $\gamma > 1$ be such that $\gamma > 2500\sqrt{\ell}$. There is a ZK proof system to show $\|\mathbf{z}\|_\infty \leq \gamma \cdot t$ by proving a single constraint $\langle \mathbf{z}|\mathbf{y}, \mathbf{b} \rangle = s$ given vector commitments to $\mathbf{z}$ and $\mathbf{y}$, where $\mathbf{b} \in \mathbb{Z}_q^{\ell+\lambda}$ is a public vector, $\mathbf{y} \in [\pm\gamma t/2(1 + 1/\lambda)]^\lambda$, and $s \in \mathbb{Z}_q$.*

The requirement in Lemma 2 that $\sqrt{q}/(4t) > 2500\sqrt{4\ell}$ holds for Equation 7 as long as $\log_2(q) > 2\log_2(2500) + 2\log_2(t) + \log_2(\ell) + 6$, which combined with the fact that $q > 2^{250}$ means that even for $t = 2^{64}$ this approach can handle vectors of length $\ell > 2^{90}$ (well beyond realistic input sizes).

We can now describe our range proof protocol for large t. Let $h \in \mathsf{G}$ and $\mathbf{g} \in \mathsf{G}^{8\ell+\lambda}$ be public generators in G. We denote by $\mathsf{Commit}(h, \mathbf{g}[i : i + \ell - 1]; \mathbf{v})$ the vector Pedersen commitment to $\mathbf{v}$ of length ℓ, using generators h and $\mathbf{g}_i, \ldots, \mathbf{g}_{i+\ell-1}$.

1. The prover finds auxiliary vectors $\mathbf{u}, \mathbf{v}, \mathbf{w}$ such that Equation 6 holds and "double-commits" to $\mathbf{z} := \mathbf{x}|\mathbf{u}|\mathbf{v}|\mathbf{w}$ as $C_1 := \mathsf{Commit}(h, \mathbf{g}[\ell : 4\ell]; \mathbf{z})$ and $C_2 := \mathsf{Commit}(h, \mathbf{g}[4\ell + 1 : 8\ell]; \mathbf{z})$. Let $\langle \mathbf{z}|\mathbf{y}, \mathbf{b} \rangle = s$ be the constraint that proves Equation 7 as in Lemma 2. Then the prover computes $C_y = \mathsf{Commit}(h, \mathbf{g}[8\ell + 1 : 8\ell + \lambda]; \mathbf{y})$ and sends C_1, C_2, and C_y to the verifier.

2. The verifier sends random challenge scalars $\sigma, \tau, \rho \in \mathbb{Z}_q$ to the prover. Let $\mathbf{r}_\mathbf{x}|\mathbf{r}_\mathbf{u}|\mathbf{r}_\mathbf{v}|\mathbf{r}_\mathbf{w} = (\sigma^{i-1})_{i \in [4\ell]}$ and $\mathbf{r} = (\tau^{i-1})_{i \in [\ell]}$ be the corresponding challenge vectors.

3. The following constraint is equivalent to Equation 6, except with probability bounded by $(\ell + 4)/q$:

   $$\langle \mathbf{x}', \mathbf{x}' \circ \mathbf{r} \rangle + \langle \mathbf{u}, \mathbf{u} \circ \mathbf{r} \rangle + \langle \mathbf{v}, \mathbf{v} \circ \mathbf{r} \rangle + \langle \mathbf{w}, \mathbf{w} \circ \mathbf{r} \rangle = \langle \mathbf{a}, \mathbf{r} \rangle.$$

   Instead of proving this directly, we use a trick from Gentry et al. [20] to express it as

   $$\langle \mathbf{x}' + \mathbf{r}_\mathbf{x}, (\mathbf{x}' - \mathbf{r}_\mathbf{x}) \circ \mathbf{r} \rangle + \langle \mathbf{u} + \mathbf{r}_\mathbf{u}, (\mathbf{u} - \mathbf{r}_\mathbf{u}) \circ \mathbf{r} \rangle + \\ \langle \mathbf{v} + \mathbf{r}_\mathbf{v}, (\mathbf{v} - \mathbf{r}_\mathbf{v}) \circ \mathbf{r} \rangle + \langle \mathbf{w} + \mathbf{r}_\mathbf{w}, (\mathbf{w} - \mathbf{r}_\mathbf{w}) \circ \mathbf{r} \rangle = s,$$

   where $s = \langle \mathbf{a}, \mathbf{r} \rangle - (\|\mathbf{r}_\mathbf{x}\|^2 + \|\mathbf{r}_\mathbf{u}\|^2 + \|\mathbf{r}_\mathbf{v}\|^2 + \|\mathbf{r}_\mathbf{w}\|^2)$ is a public value. This constraint can be rewritten as a single inner product $\langle \bar{\mathbf{x}}|\bar{\mathbf{u}}|\bar{\mathbf{v}}|\bar{\mathbf{w}}, \bar{\mathbf{x}}'|\bar{\mathbf{u}}'|\bar{\mathbf{v}}'|\bar{\mathbf{w}}' \rangle = a$ for some $a \in \mathbb{Z}_q$, where $\bar{\mathbf{x}} := \mathbf{x} + \mathbf{r_x}$ and $\bar{\mathbf{x}}' := (\mathbf{x} - \mathbf{r_x}) \circ \mathbf{r}$, and analogously for the rest. By proving these constraints against C_1 and C_2, the prover is effectively showing that C_1 and C_2 are commitments to the same vector $\mathbf{z}$. Next, to incorporate the constraint $\langle \mathbf{z}|\mathbf{y}, \mathbf{b} \rangle = s$ that proves Equation 7, the prover can replace $\mathbf{z}$ with $\bar{\mathbf{z}} := \bar{\mathbf{x}}|\bar{\mathbf{u}}|\bar{\mathbf{v}}|\bar{\mathbf{w}}$ and prove an equivalent constraint $\langle \bar{\mathbf{z}}|\mathbf{y}, \mathbf{b} \rangle = a'$, where $a' := s - \langle \mathbf{b}, \mathbf{r_x}|\mathbf{r_u}|\mathbf{r_v}|\mathbf{r_w}|\mathbf{0}^{|\mathbf{y}|} \rangle$.

At this point, the prover has the constraints $\langle \bar{\mathbf{z}}, \bar{\mathbf{x}}'|\bar{\mathbf{u}}'|\bar{\mathbf{v}}'|\bar{\mathbf{w}}' \rangle = a$ and $\langle \bar{\mathbf{z}}|\mathbf{y}, \mathbf{b} \rangle = a'$, which can be merged into a single constraint using ρ, resulting in a single inner product:

$$\langle \bar{\mathbf{z}}|\mathbf{y}, \bar{\mathbf{x}}'|\bar{\mathbf{u}}'|\bar{\mathbf{v}}'|\bar{\mathbf{w}}'|\rho\mathbf{b} \rangle = a + \rho a'. \tag{8}$$

The prover and the verifier both obtain a commitment C' to $\bar{\mathbf{z}}|\mathbf{y}|\bar{\mathbf{x}}'|\bar{\mathbf{u}}'|\bar{\mathbf{v}}'|\bar{\mathbf{w}}'|\rho\mathbf{b}$ from C_1, C_2 and C_y, which can be done using only linear operations. The prover can then use Bulletproofs to prove the constraint in Equation 8.

In addition to using Bulletproofs, the above protocol requires three length-8ℓ multi-exponentiations by the prover and verifier to compute C' in step 3. As in the protocol for $t = 2$, the prover and verifier also need to switch the generators to match the commitment C'. The prover can again combine the process of both switching generators and updating the commitment with the analogous operations in the outer loop of Bulletproofs, thus computing them almost for free. The only remaining overhead is two multi-exponentiations of length 8ℓ for the verifier, but this overhead can be batched as it depends on public (not proof-specific) generators.

For $\mathbf{x} \in \mathbb{Z}_t^\ell$, the overall proof system $\pi^{0 \le \mathbf{x} < t}$ thus has the costs stated in Lemma 1, using $n = 8\ell + \lambda$, with two additional multi-exponentiations of length 8ℓ for the verifier and commitment cost (to vectors of total length $8\ell + \lambda$ with entries in $[t]$) for the prover.

5.4 Proofs of Validity of Encoding

In all variants of the protocol, $\pi^{\mathsf{Enc}(\mathbf{sk},\mathbf{x})}$ reduces to proving an inner product constraint involving *public* packing matrix $\mathbf{G} \in \mathbb{Z}_q^{\bar{\ell} \times \ell}$, masked input $\mathbf{y_i} \in \mathbb{Z}_q^{\bar{\ell}}$, and a *private* committed input vector $\mathbf{x_i}$, for each client i. In the PRG-based ACORN-detect, for example, we rely on the constraint $\mathbf{y_i} = \mathbf{sk}_i + \mathbf{G}\mathbf{x_i}$ for a committed key $\mathbf{sk}_i \in \mathbb{Z}_q^{\bar{\ell}}$. For the RLWE-based variant, we rely on the constraint $\mathbf{y_i} = \mathbf{A} \cdot \mathbf{sk}_i + T\mathbf{e_i} + \mathbf{G}_k\mathbf{x_i} + Q\mathbf{d_i} \wedge \|\mathbf{e_i}\|_\infty < \beta_e$ for committed key $\mathbf{sk}_i$ and error term $\mathbf{e_i}$, and for some private $\mathbf{d_i} \in \mathbb{Z}_q^{\bar{\ell}}$, where Q is the RLWE ciphertext modulus and β_e is a high-probability bound on $\mathbf{e_i} \in \mathbb{Z}_q^{\bar{\ell}}$. Note that, unlike in [20], we do not require the RLWE modulus Q to match the group order q underlying the Bulletproofs, which allows us to choose NTT-friendly Q for achieving optimal encoding efficiency.

In general, these required constraints can be written as a single constraint $\langle \mathbf{x}|\mathbf{v_1}|\cdots|\mathbf{v_k}|, \mathbf{b} \rangle = a$ using Schwartz-Zippel as in the smallness proofs in Section 5.3, and then using the smallness proofs directly to prove that $\|\mathbf{e}_i\|_\infty < b_e$. For PRG-SecAgg, these reductions to a single constraint do not have any overhead. For RLWE-SecAgg, the combined proof requires additional multi-exponentiations due to the secret multiplier $\mathbf{d_i}$ and the smallness proof of $\mathbf{e_i}$.

5.5 Other Validity Predicates

We have already presented proofs of one useful validity predicate: $\mathsf{valid}(\mathbf{x}) := \mathbf{x} \in [0,t)^\ell$, which extends to $\mathsf{valid}(\mathbf{x}) := \|\mathbf{x}\|_\infty = t$. We now discuss useful variants related to bounding L_0 and L_2 norms. We first observe that proving k-hotness, i.e. $\mathsf{valid}(\mathbf{x}) := \mathbf{x} \in \{0,1\}^\ell \wedge \|\mathbf{x}\|_0 = k$, can be achieved by just merging the constraint $\langle \mathbf{x}, \mathbf{1}^\ell \rangle = k$ with the proof of Section 5.3.1, which does not add any overhead. Let us also consider how to prove $\mathsf{valid}(\mathbf{x}) := \mathbf{x} \in [0,t) \wedge \|\mathbf{x}\|_2 \le b$ for some public bound b, where t could be replaced by some natural bit-width like 2^{16} or 2^{32}. This can be done in two steps: first we establish that $\|\mathbf{x}\|_2 \le \eta b$ using an approximate L_2 proof such that $\eta b < q/2$ for some gap parameter $\eta > 1$, and then we apply Lagrange's four-square theorem and prove that

$$\langle \mathbf{x}|\upsilon_0|\upsilon_1|\upsilon_2|\upsilon_3, \mathbf{x}|\upsilon_0|\upsilon_1|\upsilon_2|\upsilon_3 \rangle = b \tag{9}$$

where $\upsilon_0, \ldots, \upsilon_3$ are integers guaranteed to exist if $\|x\|_2 \le b$. Recently, Lyubashevsky et al. [35, Lemma 2.9] showed that the approximate L_2 bound proof can be adopted from the approximate L_∞ bound proof of Lemma 2. These two proofs can be combined with the proofs of Section 5.3 to show that Equation 9 holds over the integers, and the overhead is the additional commitments to $\upsilon_0, \ldots, \upsilon_3$ (twice using different sets of generators) and the increased inner product constraint length (by 4). The details of this extension were given by Gentry et al. [20, Section 3.5].

6 Implementation and Evaluation

In this section we present experimental results for our new protocols, focusing on RLWE-SecAgg (in Section 6.1) and ACORN-detect (in Section 6.2). We do not benchmark ACORN-robust, but as described in Section 4.2 these costs can be derived from those of ACORN-detect (taking into account the additional vector commitments clients must form).

We focus on two scenarios when setting experiment parameters: (1) federated learning (FL) applications with $n = 500$ clients and input vectors $\mathbf{x}$ containing 16-bit integers (i.e. $t = 2^{16}$); and 2) federated aggregation (FA) applications with $n = 10000$ clients and input vectors $\mathbf{x}$ containing binary values (i.e. $t = 2$). In both settings, we consider input vectors of length ℓ ranging from 2^{10} to 2^{20}, which covers a wide range of real-world scenarios. Our experiments were performed on a laptop with an Intel i7-1185G7 CPU running at 3GHz and with 16GB memory, in single thread mode, and we take advantage of SIMD instructions such as AVX512. For the input validation steps we also performed experiments on a Pixel 6 Pro smartphone.

6.1 RLWE-SecAgg

RLWE parameters. We first set the error distribution χ_e for sampling $\mathbf{e}$ and $\mathbf{f}$ in our encryption scheme to be a discrete Gaussian D_{σ_1} with standard deviation $\sigma_1 = 4.5$. As shown in Appendix A, the security level of our RLWE encryption with this error distribution can be derived from the hardness of solving RLWE with a discrete Gaussian error distribution of standard deviation $\sigma = 3.2$. In the implementation we use a tail-cut discrete Gaussian with support $[-60, 60]$ to sample $\mathbf{e}+\mathbf{f}$, which is statistically close to $D_{2\sigma_1}$ with distance at most 2^{-30}. When input validation is not required, we choose a power-of-2 ring degree $N \in \{2^{11}, 2^{12}, 2^{13}\}$, and we pick a prime modulus $q = 1 \pmod{2N}$ with 155 bits of security according to the lattice estimator [3]; such q takes advantage of Number Theoretic Transform (NTT) for fast polynomial multiplication. With these parameters our RLWE encoding scheme achieves at least 128 bits of security with $n \leq 10^4$ clients(Appendix A). Furthermore we pick ring parameters such that it can optimally accommodate input messages via packing. With input validation, we choose $N = 2^{12}$ and a prime q of at most 96 bits and achieve the same level of security as above. These parameters allow us to prove valid RLWE encodings using Bulletproofs based on curve25519.

Ciphertext expansion. Since the RLWE modulus q is usually much larger than the input bound t, we pack multiple input entries into a single plaintext slot. In addition, when the packed input vector has length $\bar{\ell} < N$, each client i sends just the first $\bar{\ell}$ coefficients $\mathbf{y}'_i = \mathbf{y}_i[1 \ldots \bar{\ell}]$ instead of the full $\mathbf{y}_i$. The server can still recover the aggregated input from $\sum_i \mathbf{y}'_i$ and the first ℓ rows of the public randomness $\mathbf{A}$. In contrast, in PRG-SecAgg we have more choices for the modulus q. For PRG-SecAgg without input validation, we can set the modulus $q = nt$ to achieve the optimal ciphertext expansion ratio, which is the total ciphertext bit-size over the input bit-size; when input validation is required, we set q to the group size of curve25519.

Experimental results. We benchmarked the RLWE encoding step for individual clients, which involves expanding seeds to secret keys and encoding the packed input with the properly aggregated secret keys. For comparison, we also benchmarked the PRG encoding step, where seeds are expanded by repeatedly calling AES to the desired length, and masking is done via modular addition. The results are in Figure 3.

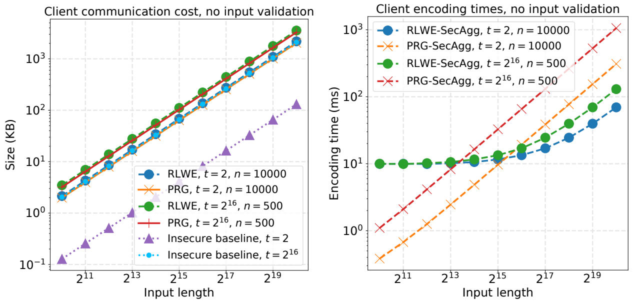

Figure 3: Averaged over 100 runs and plotted on a logarithmic scale, the message size (on the left) and encoding times (on the right) for our two scenarios for RLWE-SecAgg, PRG-SecAgg, and the insecure baseline where clients send the (uncompressed) input vector in the clear.

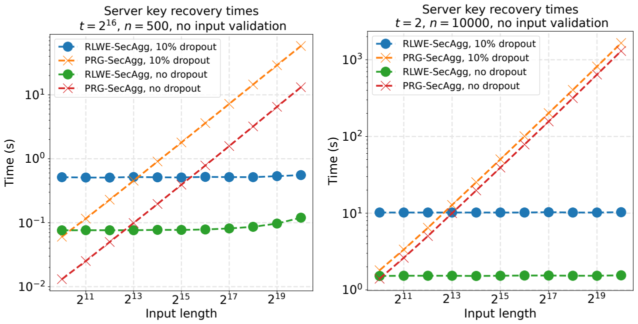

Figure 4: Server key recovery experiment results for RLWE-SecAgg and PRG-SecAgg. The runtimes are averaged over 100 runs and plotted on a logarithmic scale. The diagram on the left shows runtimes for the FL scenario ($t = 2^{16}$ and $n = 500$), and the diagram on the right for the FA scenario ($t = 2$ and $n = 10^4$). We consider cases with no dropouts and with a $\delta = 1/10$ fraction of dropouts.

For example, in the FL use case where $t = 2^{16}$ and $n = 500$, when the input has length $\ell = 2^{16}$, RLWE encoding takes only 17ms while PRG encoding takes 65ms; for an input of length $\ell = 2^{20}$, RLWE encoding takes 130ms while PRG encoding takes 1.06s. Figure 3 shows our encoding benchmark results of both RLWE-SecAgg and PRG-SecAgg. Overall, RLWE encoding achieves roughly up to 5x speedup in the FA setting for $\ell \geq 2^{15}$, and up to more than 8x speedup in the FL setting for $\ell \geq 2^{13}$; for shorter input $\mathbf{x}$ the time spent on RLWE secret sampling is more significant than the PRG mask expansion.

We also benchmarked the server key recovery step for RLWE-SecAgg and PRG-SecAgg. The results are shown in Figure 4. For RLWE-SecAgg, the key recovery step includes expanding seeds to RLWE secrets of length N and decoding the RLWE masked sum, which involves an NTT operation per RLWE ciphertext. We see from the results that the RLWE key recovery times are dominated by seed expansion, which is independent of the input length ℓ, and the time spent on decoding the masked sum was not significant except when n is small and ℓ is very large. Compared to PRG-SecAgg, the RLWE key recovery step is much more efficient for long inputs: for

	Client computation time			Client bandwidth	
Type	Round (s)	Total (m)	Factor	Total (MB)	Factor
MNIST (19k params, 160 rounds, training time per round 1.17s)					
SA	1.19	3.22	1x	5.41	1x
L_2	6.65	17.79	5.52x	5.68	1.05x
L_∞	6.59	17.63	5.47x	5.67	1.05x
CIFAR-10 S (62k params, 100 rounds, training time per round 0.83s)					
SA	0.86	1.48	1x	11.02	1x
L_2	18.81	31.39	21.28x	11.20	1.02x
L_∞	18.28	30.50	20.68x	11.19	1.02x
CIFAR-10 L (273k params, 160 rounds, training time per round 20.75s)					
SA	20.91	55.82	1x	77.66	1x
L_2	100.33	267.60	4.79x	77.97	1.01x
L_∞	96.55	257.53	4.61x	77.96	1.01x
Shakespeare (818k params, 20 rounds, training time per round 284.16s)					
SA	284.64	94.89	1x	29.08	1x
L_2	519.92	173.31	1.83x	29.13	1.01x
L_∞	510.42	170.15	1.79x	29.12	1.01x

Table 2: End-to-end model training with ACORN-detect. "SA" denotes experiments using PRG-SecAgg, which includes local training and secure aggregation. L_2 and L_∞ denote experiments using ACORN-detect with L_2 and L_∞ validity proofs. Computation time and bandwidth are reported per client, and overhead factors are with respect to secure aggregation (SA).

example, with $n = 10000$ it takes only 10.3s to recover all secrets and decode the masked sum in RLWE-SecAgg when $\ell = 2^{20}$ for all $n = 10000$ clients with 10% dropout rate, while in PRG-SecAgg the same step requires 1650s.

6.2 ACORN-detect

While RLWE-SecAgg is more efficient without input validation, it becomes less efficient on the client when input validation is required due to additional proofs of smallness of the error terms. For example, when input $\mathbf{x}$ is a binary vector of length 2^{16}, generating the input validation proof for RLWE-SecAgg takes 8.4s whereas it takes 4.6s for PRG-SecAgg. For input $\mathbf{x}$ of length 2^{16} and l_∞ norm $t = 2^{16}$, it takes 26.3s to generate RLWE-SecAgg input validation proofs and 18.7s for PRG-SecAgg. We thus focus our results only on the PRG variant of ACORN-detect. Experimental results for the RLWE variant can be found at Appendix F.

We benchmarked the main components of ACORN-detect with graph parameters $\gamma = \delta = 1/10$. For the client, these consist of the encoding step and generating the necessary commitments to **sk** and **x** and proofs $\pi^{0 \leq \mathbf{x} < t}$ and $\pi^{\mathsf{Enc}(\mathbf{sk},\mathbf{x})}$. As discussed in Section 5.5, these costs also cover $\pi^{\mathsf{valid}(\mathbf{x})}$ for various validity predicates (e.g., one-hotness and both L_∞ and L_1 bounds). For the server, this consists of proof verification and key recovery steps. Figure 5 shows the client and server runtimes as well as the client communication costs for both settings we consider, where all benchmarks were run on a laptop. When running the client computations on the Pixel 6 Pro smartphone, we observed an average slowdown of 3X.

Encoding. We set the mask modulus q to the group size of curve25519 to match our Bulletproofs implementation. Comparing to PRG-SecAgg without input validation, this modulus q is less optimal in terms of packing capacity, and as a result, encoding times are increased by 40% to 70%. Regardless, encoding still takes less than one second for all but one input lengths (the exception being vectors of length 2^{20} in the federated learning use case).

Commitment generation. When $t = 2$, commitment generation is fast and grows slowly even for long inputs: for inputs of length $\ell = 2^{20}$ the commitments can be generated in 404ms. When input entries are large ($t = 2^{16}$), commitment generation is slower, but can still finish in 1.13s for $\ell = 2^{17}$. On the Pixel 6 Pro, commitments can be generated in 734ms for $t = 2$ and $\ell = 2^{20}$, and in 2.1s for $t = 2^{16}$ and $\ell = 2^{17}$.

Proof generation. We implemented the more efficient Bulletproofs variant due to Gentry et al. [20, Section E.2]. Our implementation [16] further optimizes proof generation by not requiring the client to pad the inner product constraints to a power-of-2 length, which saves almost half of the proof generation time when the input is exactly or slightly longer than a power of 2. When $t = 2$, all proofs can be generated in 572ms for inputs of length $\ell = 2^{13}$ and in 70s for length $\ell = 2^{20}$. When $t = 2^{16}$, the combined linear constraint is roughly four times longer than in the $t = 2$ case, so proof generation is slower: it runs in 2.27s for inputs of length 2^{13} and in 285s for length 2^{20}. For comparison, proof generation on the Pixel 6 Pro for $\ell = 2^{13}$ takes 2.1s for $t = 2$ and 8.2s for $t = 2^{16}$.

Proof verification. The verification step also takes advantage of the lightweight linear proof optimization, and we benchmarked the batched verification of proofs from all n clients using the techniques mentioned in Section 5.2. As we can see, batched proof verification in the binary case is very efficient due to the smaller size of proofs and the SIMD acceleration: verifying all proofs from 10,000 clients takes 1.7s for inputs of length $\ell = 2^{13}$ and 133.2s for $\ell = 2^{20}$. When $t = 2^{16}$, the proof is longer and hence verification requires more time: it takes 9.1s for $\ell = 2^{13}$ and 131.1s for $\ell = 2^{18}$. Note that the server can divide client proofs in many small batches and fully parallelize the proof verification process.

End-to-end performance in FL. Following Lycklama et al. [34], we demonstrate the effectiveness of ACORN-detect for four practical use cases in federated learning. Concretely, we consider the following three tasks on image-based datasets: training (1) a convolutional neural network (CNN) on the Federated-MNIST dataset [12], (2) the LeNet-5 [31] and (3) the ResNet-20 CNN [24] on the CIFAR-10 dataset [30]. We also consider (4) the task of training an LSTM [25] model on the text-based Shakespeare dataset [12]. We use Tensor-

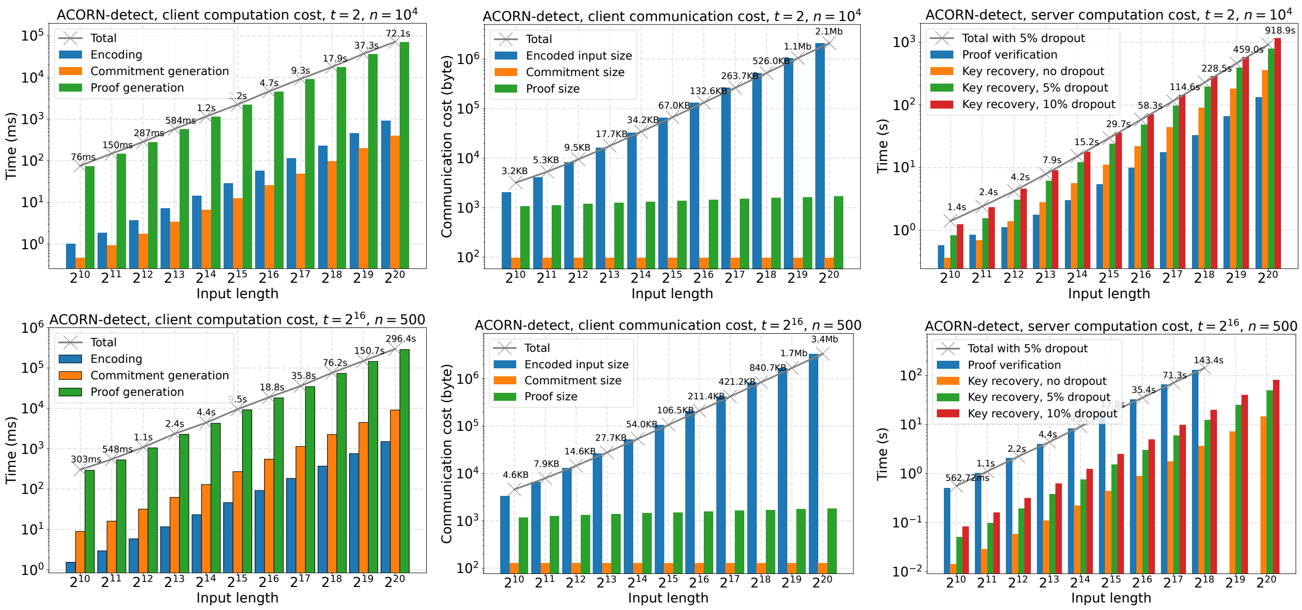

Figure 5: On a logarithmic scale, benchmarks for ACORN-detect for the FA use case where $t = 2$ and $n = 10000$ (on the top row) and the FL use case where $t = 2^{16}$ and $n = 500$ (on the bottom). In both cases we measured (1) the client runtime (on the left); (2) the client communication cost (in the middle); and (3) the server runtime (on the right), considering proof verification and key reconstruction for three different levels of dropouts. Our proof verification experiments for $t = 2^{16}$ and input length $\ell \geq 2^{19}$ ran out of memory, which is why these bars are missing in the bottom right diagram. The encoding experiments were repeated 100 times for each parameter set, and the other experiments were repeated for at least 5 seconds or 10 iterations; the average running times are estimated using the bootstrap resampling method with 95% confidence level.

Flow [1] to train neural networks, where the hyperparameters are set as in [34]. To account for network latency we introduce delays of 0.5ms as in [34]. We report the per-client running time that includes the local training and aggregation steps (as described in Section 1.2). The results are in Table 2, and a more detailed experimental setup can be found in Appendix E.

For each use case, we compare ACORN-detect with both L_∞- and L_2-norm based validity proofs against PRG-SecAgg without input validation. A single round on each client consists of receiving the updated model, performing local training using its samples, and participating in the secure aggregation protocol using its model updates as input. As we can see, the local training time dominates the running time of our SA baseline. Furthermore, in all cases the bandwidth overhead of ACORN-detect is very modest (at most 1.05x), and in most cases the computational overhead is also fairly low (at most 5.52x). The exception is for CIFAR-10 S, where the higher overhead (21.28x) is due to a relatively higher learning rate ($\eta = 0.01$) and the small sample count (1024) and epochs per round (2) required to reach the desired accuracy. This makes the local training highly efficient, and thus generating validity proofs becomes a more dominant cost.

7 Conclusion and Open Problems

We presented a new secure aggregation protocol, RLWE-SecAgg, along with extensions, ACORN, that allow the server to perform validity checks on the inputs provided by clients. Our benchmarks demonstrate that the overheads of these checks are practical. Other zero-knowledge protocols offer lower prover runtimes, however, and may do so without making other costs impractical for our setting. For example, lattice-based proofs may offer a better balance between computational and communication overheads, and would also offer the advantage when combined with RLWE-SecAgg of providing plausible post-quantum security.

Acknowledgements

We would like to thank Michael Specter for benchmarking our code on a Pixel device, and our anonymous reviewers and shepherd for their helpful comments and suggestions.

References

[1] M. Abadi et al. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”. In:

CoRR abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467.

[2] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou. “Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares”. In: *SCN*. Vol. 13409. Lecture Notes in Computer Science. Springer, 2022, pp. 516–539.

[3] M. R. Albrecht, R. Player, and S. Scott. “On the concrete hardness of Learning with Errors”. In: *J. Math. Cryptol.* 9.3 (2015), pp. 169–203.

[4] B. Alon, M. Naor, E. Omri, and U. Stemmer. “MPC for Tech Giants (GMPC): Enabling Gulliver and the Lilliputians to Cooperate Amicably”. In: *IACR Cryptol. ePrint Arch.* (2022), p. 902.

[5] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova. “Secure Single-Server Aggregation with (Poly)Logarithmic Overhead”. In: *CCS*. ACM, 2020, pp. 1253–1269.

[6] K. A. Bonawitz et al. “Practical Secure Aggregation for Privacy-Preserving Machine Learning”. In: *CCS*. ACM, 2017, pp. 1175–1191.

[7] K. A. Bonawitz et al. “Towards Federated Learning at Scale: System Design”. In: *MLSys*. mlsys.org, 2019.

[8] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. “Lightweight Techniques for Private Heavy Hitters”. In: *IEEE Symposium on Security and Privacy*. IEEE, 2021, pp. 762–776.

[9] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. “Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting”. In: *Proceedings of Eurocrypt 2016*. 2016.

[10] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. “(Leveled) Fully Homomorphic Encryption without Bootstrapping”. In: *ACM Trans. Comput. Theory* 6.3 (2014), 13:1–13:36.

[11] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. “Bulletproofs: Short Proofs for Confidential Transactions and More”. In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 315–334.

[12] S. Caldas et al. *Leaf: A benchmark for federated settings*. arXiv preprint arXiv:1812.01097. 2018.

[13] A. R. Choudhuri, A. Goel, M. Green, A. Jain, and G. Kaptchuk. “Fluid MPC: Secure Multiparty Computation with Dynamic Participants”. In: *CRYPTO (2)*. Vol. 12826. Lecture Notes in Computer Science. Springer, 2021, pp. 94–123.

[14] A. R. Chowdhury, C. Guo, S. Jha, and L. van der Maaten. “EIFFeL: Ensuring Integrity for Federated Learning”. In: *CCS*. ACM, 2022.

[15] H. Corrigan-Gibbs and D. Boneh. “Prio: Private, Robust, and Scalable Computation of Aggregate Statistics”. In: *NSDI*. USENIX Association, 2017, pp. 259–282.

[16] *Efficient Bulletproofs in Rust*. Anonymized; available upon acceptance for double blinded review. 2022.

[17] P. Feldman. “A Practical Scheme for Non-interactive Verifiable Secret Sharing”. In: *Proceedings of the 28th Annual Symposium on Foundations of Computer Science, SFCS ’87*. 1987, 427–438.

[18] A. Fiat and A. Shamir. “How to Prove Yourself: Practical Solutions to Identification and Signature Problems”. In: *CRYPTO*. Vol. 263. Lecture Notes in Computer Science. Springer, 1986, pp. 186–194.

[19] C. Gentry. “A fully homomorphic encryption scheme”. PhD thesis. Stanford University, USA, 2009. URL: https://searchworks.stanford.edu/view/8493082.

[20] C. Gentry, S. Halevi, and V. Lyubashevsky. “Practical Non-interactive Publicly Verifiable Secret Sharing with Thousands of Parties”. In: *EUROCRYPT (1)*. Vol. 13275. Lecture Notes in Computer Science. Springer, 2022, pp. 458–487.

[21] Z. Ghodsi, M. Javaheripi, N. Sheybani, X. Zhang, K. Huang, and F. Koushanfar. *zPROBE: Zero Peek Robustness Checks for Federated Learning*. 2022. DOI: 10.48550/ARXIV.2206.12100. URL: https://arxiv.org/abs/2206.12100.

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

[23] J. Groth. “Non-interactive Zero-Knowledge Arguments for Voting”. In: *International Conference on Applied Cryptography and Network Security (ACNS)*. Vol. 3531. Lecture Notes in Computer Science. 2005, pp. 467–482.

[24] K. He, X. Zhang, S. Ren, and J. Sun. *Deep Residual Learning for Image Recognition*. arXiv preprint arXiv:1512.03385. 2015.

[25] S. Hochreiter and J. Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[26] *How Messages improves suggestions with federated technology*. https://support.google.com/messages/answer/9327902. Accessed: 2022-10-06.

[27] Y. Ishai, E. Kushilevitz, Y. Lindell, and E. Petrank. “On Combining Privacy with Guaranteed Output Delivery in Secure Multiparty Computation”. In: *CRYPTO*. Vol. 4117. Lecture Notes in Computer Science. Springer, 2006, pp. 483–500.

[28] P. Kairouz et al. “Advances and Open Problems in Federated Learning”. In: *Found. Trends Mach. Learn.* 14.1-2 (2021), pp. 1–210.

[29] F. Karakoç, M. Önen, and Z. Bilgin. “Secure Aggregation Against Malicious Users”. In: *SACMAT ’21: The 26th ACM Symposium on Access Control Models and Technologies, Virtual Event, Spain, June 16-18, 2021*. Ed. by J. Lobo, R. D. Pietro, O. Chowdhury, and H. Hu. ACM, 2021, pp. 115–124. DOI: 10.1145/3450569.3463572. URL: https://doi.org/10.1145/3450569.3463572.

[30] A. Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. 2009.

[31] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE*. 1998.

[32] J. Lee, D. Kim, D. Kim, Y. Song, J. Shin, and J. H. Cheon. “Instant Privacy-Preserving Biometric Authentication for Hamming Distance”. In: *IACR Cryptol. ePrint Arch.* (2018), p. 1214.

[33] Y. Lindell. “How to Simulate It - A Tutorial on the Simulation Proof Technique”. In: *Tutorials on the Foundations of Cryptography*. Springer International Publishing, 2017, pp. 277–346.

[34] H. Lycklama, L. Burkhalter, A. Viand, N. Küchler, and A. Hithnawi. “RoFL: Attestable Robustness for Secure Federated Learning”. In: *CoRR* abs/2107.03311 (2021).

[35] V. Lyubashevsky, N. K. Nguyen, and M. Plançon. “Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General”. In: Lecture Notes in Computer Science (2022).

[36] V. Lyubashevsky, C. Peikert, and O. Regev. “On Ideal Lattices and Learning with Errors over Rings”. In: *J. ACM* 60.6 (2013), 43:1–43:35.

[37] L. Melis, G. Danezis, and E. D. Cristofaro. “Efficient Private Statistics with Succinct Sketches”. In: *NDSS*. The Internet Society, 2016.

[38] D. Micciancio and M. Walter. “On the Bit Security of Cryptographic Primitives”. In: *EUROCRYPT (1)*. Vol. 10820. Lecture Notes in Computer Science. Springer, 2018, pp. 3–28.

[39] D. Pasquini, D. Francati, and G. Ateniese. “Eluding Secure Aggregation in Federated Learning via Model Inconsistency”. In: *CCS*. ACM, 2022, pp. 2429–2443.

[40] N. Pippenger. “On the Evaluation of Powers and Monomials”. In: *SIAM J. Comput.* 9.2 (1980), pp. 230–250.

[41] *Predicting Text Selections with Federated Learning*. https://ai.googleblog.com/2021/11/predicting-text-selections-with.html. Accessed: 2022-10-06.

[42] M. O. Rabin and J. O. Shallit. “Randomized algorithms in number theory”. In: *Communications on Pure and Applied Mathematics* 39.S1 (1986), S239–S256. DOI: https://doi.org/10.1002/cpa.3160390713. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpa.3160390713. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpa.3160390713.

[43] L. Reyzin, A. D. Smith, and S. Yakoubov. “Turning HATE Into LOVE: Homomorphic Ad Hoc Threshold Encryption for Scalable MPC”. In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 997.

[44] E. Roth, H. Zhang, A. Haeberlen, and B. C. Pierce. “Orchard: Differentially Private Analytics at Scale”. In: *OSDI*. USENIX Association, 2020, pp. 1065–1081.

[45] V. Shejwalkar, A. Houmansadr, P. Kairouz, and D. Ramage. “Back to the Drawing Board: A Critical Evaluation of Poisoning Attacks on Federated Learning”. In: (2022). To appear.

[46] E. Shi, T. H. Chan, E. G. Rieffel, R. Chow, and D. Song. “Privacy-Preserving Aggregation of Time-Series Data”. In: *NDSS*. The Internet Society, 2011.

[47] M. Skala. *Hypergeometric tail inequalities: ending the insanity*. 2013. DOI: 10.48550/ARXIV.1311.5939. URL: https://arxiv.org/abs/1311.5939.

[48] J. So, B. Güler, and A. S. Avestimehr. “Turbo-Aggregate: Breaking the Quadratic Aggregation Barrier in Secure Federated Learning”. In: *IEEE J. Sel. Areas Inf. Theory* 2.1 (2021), pp. 479–489.

[49] T. Stevens, C. Skalka, C. Vincent, J. Ring, S. Clark, and J. Near. “Efficient Differentially Private Secure Aggregation for Federated Learning via Hardness of Learning with Errors”. In: *CoRR* abs/2112.06872 (2021). arXiv: 2112.06872. URL: https://arxiv.org/abs/2112.06872.

[50] Z. Sun, P. Kairouz, A. T. Suresh, and H. B. McMahan. “Can You Really Backdoor Federated Learning?” In: *CoRR* abs/1911.07963 (2019).

[51] H. Wang et al. “Attack of the Tails: Yes, You Really Can Backdoor Federated Learning”. In: *NeurIPS*. 2020.

[52] C. Xie, M. Chen, P.-Y. Chen, and B. Li. “CRFL: Certifiably Robust Federated Learning against Backdoor Attacks”. In: *Proceedings of the Thirty-eighth International Conference on Machine Learning (ICML)*. 2021.

A Security Proof of RLWE-SecAgg

In this section we provide the proofs of correctness and security of the RLWE-SecAgg construction in Section 3. For our proofs we assume the "good" graph properties defined by Bell et al. [5] for the output of GENERATEGRAPH suffice for the correctness and security of the RLWE-SecAgg construction as well.

Theorem A.1 (Correctness). *Assume Algorithm 6 is instantiated with a good graph generation algorithm* GENERATEGRAPH. *If less than a fraction δ of the clients dropout, i.e., $|A_3'| \geq (1-\delta)n$, then the server does not abort and obtains $\mathbf{z} = \sum_{i \in A_2'} \mathbf{x}_i$ with overwhelming probability.*

Correctness follows from the key and input homomorphic properties of the RLWE encodings which are analogous to one-time pad, which enables the server to obtain the appropriate key for decoding of the aggregated value.

Next we prove that security would rely on the fact that the encodings under keys that cannot be canceled hide the encoded messages.

RLWE Encoding Properties. Below, we first establish the security properties of our RLWE encoding, which will then be used to prove the semi-honest security of the RLWE-SecAgg protocol.

Definition 1 (HintRLWE). *For any $N, q \geq 1$, any $\sigma_1, \sigma_2 > 0$, and a distribution χ_s over R, the $\mathsf{HintRLWE}_{N,q,\sigma_1,\sigma_2}$ problem is to distinguish, given arbitrary number of samples, the following two distributions for $\mathbf{s} \leftarrow \chi_s$:*

$$A^{\mathsf{HintRLWE}}_{N,q,\sigma_1,\sigma_2}(\mathbf{s}) = \left\{ \begin{array}{l} (\mathbf{A}, \mathbf{A}\mathbf{s} + \mathbf{e}, \mathbf{e} + \mathbf{f}) : \\ \mathbf{A} \leftarrow R_q, \mathbf{e} \leftarrow D_{\sigma_1}, \mathbf{f} \leftarrow D_{\sigma_2} \end{array} \right\},$$

and

$$A^{\mathsf{random}}_{N,q,\sigma_1,\sigma_2} = \left\{ \begin{array}{l} (\mathbf{A}, \mathbf{u}, \mathbf{e} + \mathbf{f}) : \\ \mathbf{A} \leftarrow R_q, \mathbf{u} \leftarrow R_q, \mathbf{e} \leftarrow D_{\sigma_1}, \mathbf{f} \leftarrow D_{\sigma_2} \end{array} \right\}.$$

Lee at al. [32] showed that the integer lattice version of HintRLWE problem reduced from the standard LWE problem, preserving the sample complexity and the adversary's distinguishing advantage. Such reduction can be naturally adapted to the power-of-two cyclotomic ring setting. Furthermore, our RLWE encryption algorithm encodes the plaintext in the lower order bits of the ciphertext, and thus it relies on pseudorandomness of tuples $(\mathbf{A}, \mathbf{A}\mathbf{s} + T\mathbf{e})$ as in the BGV homomorphic encryption scheme [10]. When the plaintext modulus T is coprime to the ciphertext modulus q, $T^{-1} \pmod{q}$ always exists, and we can further extend the above reduction to the following form.

Lemma 3. *For any T coprime to q, any $\sigma_1, \sigma_2 > 0$, let $\sigma = \sigma_1\sigma_2/\sqrt{\sigma_1^2 + \sigma_2^2}$, and let χ_s be any distribution over R. There exists an efficient reduction from $\mathsf{RLWE}_{N,q,\sigma}$ to the problem of distinguishing $A^{\mathsf{random}}_{N,q,\sigma_1,\sigma_2}$ and the following for $\mathbf{s} \leftarrow \chi_s$:*

$$B^{\mathsf{HintRLWE}}_{N,q,\sigma_1,\sigma_2}(\mathbf{s}) = \left\{ \begin{array}{l} (\mathbf{A}, \mathbf{A}\mathbf{s} + T \cdot \mathbf{e}, \mathbf{e} + \mathbf{f}) : \\ \mathbf{A} \leftarrow R_q, \mathbf{e} \leftarrow D_{\sigma_1}, \mathbf{f} \leftarrow D_{\sigma_2} \end{array} \right\}.$$

Furthermore, the reduction preserves the distinguishing advantage.

As in [32], we can set $\sigma_1 = \sigma_2$, and thus $\sigma = 1/\sqrt{2}\sigma_1$. We obtain the following technical lemma that we will use later in the proof of Theorem A.2. Intuitively the lemma states that the joint distribution of encrypted inputs from honest clients in A_2' is indistinguishable from random conditioned on the sum of the random inputs is the same as the sum of the honest inputs in A_2'.

Lemma 4. *For any $\sigma_1 > 0$, for any $m, N, q, T, \ell \geq 1$ such that T is coprime to q, let $k = \ell/N$ and let $\mathbf{x}_i, \ldots, \mathbf{x}_m \in \mathbb{Z}_T^\ell \equiv R_T^k$. Assume $\mathsf{RLWE}_{N,q,\sigma}$ is hard for $\sigma = 1/\sqrt{2}\sigma_1$. Then, the following two distributions are indistinguishable*

$$\left\{ \begin{array}{l} (\mathbf{A}, \mathbf{A}\mathbf{s}_1 + T(\mathbf{e}_1 + \mathbf{f}_1) + \mathbf{x}_1, \ldots, \\ \quad \mathbf{A}\mathbf{s}_{m-1} + T(\mathbf{e}_{m-1} + \mathbf{f}_{m-1}) + \mathbf{x}_{m-1}, \\ \quad -\mathbf{A}\sum_{i=1}^{m-1} \mathbf{s}_i + T(\mathbf{e}_m + \mathbf{f}_m) + \mathbf{x}_m) \bmod q : \\ \mathbf{A} \leftarrow R_q^k, \mathbf{s}_1, \ldots, \mathbf{s}_{m-1} \leftarrow \chi_s \ \wedge\ \forall i.\ \mathbf{e}_i, \mathbf{f}_i \leftarrow D_{\sigma_1}^k \end{array} \right\}$$

$$\approx \left\{ \begin{array}{l} (\mathbf{A}, \mathbf{u}_1, \ldots, \mathbf{u}_{m-1}, \\ -\sum_{i=1}^{m-1} \mathbf{u}_i + T\sum_{i=1}^{m}(\mathbf{e}_i + \mathbf{f}_i) + \sum_{i=1}^{m} \mathbf{x}_i) \bmod q : \\ \mathbf{A} \leftarrow R_q^k, \mathbf{u}_1, \ldots, \mathbf{u}_{m-1} \leftarrow R_q^k \ \wedge\ \forall i.\ \mathbf{e}_i, \mathbf{f}_i \leftarrow D_{\sigma_1}^k \end{array} \right\}.$$

Furthermore, if $\mathsf{RLWE}_{N,q,\sigma}$ for χ_s is κ-bit hard, where $\sigma = \sigma_1^2/\sqrt{2\sigma_1^2}$, then the bit security of distinguishing these two distributions is $\kappa - 2(\log m + 1)$.

Proof. Denote the two distributions as $\mathcal{D}_0$ and $\mathcal{D}_1$, and we use the following hybrids to show that $\mathcal{D}_0 \approx \mathcal{D}_1$. Let $\mathcal{H}_0 = \mathcal{D}_0$, and for $1 \leq i \leq m-1$, let $\mathcal{H}_i$ be

$$\left\{ \begin{array}{c} (\mathbf{A}, \mathbf{u}_1, \ldots, \mathbf{u}_i, \mathbf{y}_{i+1}, \ldots, \mathbf{y}_{m-1}, \\ -\mathbf{w}_i - \mathbf{A}\mathbf{r} + T(\sum_{j=1}^{i}(\mathbf{e}_j + \mathbf{f}_j) + \mathbf{e}_m + \mathbf{f}_m) + \mathbf{z}) : \\ \mathbf{A} \leftarrow R_q^k, \mathbf{w}_i = \sum_{j=1}^{i} \mathbf{u}_j, \mathbf{r} = \sum_{j=i+1}^{m-1} \mathbf{s}_j, \\ \mathbf{e}_m, \mathbf{f}_m \leftarrow D_{\sigma_1}^k, \mathbf{z} = \sum_{j=1}^{i} \mathbf{x}_j \\ \wedge \forall i < j < m.\ \mathbf{s}_j \leftarrow \chi_s, \mathbf{u}_j \leftarrow R_q^k, \mathbf{e}_j, \mathbf{f}_j \leftarrow D_{\sigma_1}^k, \\ \mathbf{y}_j = \mathbf{A}\mathbf{s}_j + T(\mathbf{e}_j + \mathbf{f}_j) + \mathbf{x}_j \bmod q \end{array} \right\}.$$

Note that the final hybrid $\mathcal{H}_{m-1}$ is exactly $\mathcal{D}_1$.

For all $1 \leq i \leq m-1$, we build a reduction $\mathcal{B}_i$ takes as input a tuple $(\mathbf{A}, \mathbf{b}, \mathbf{v})$ that is either from $B^{\mathsf{HintRLWE}}_{N,q,\sigma_1,\sigma_1}(\mathbf{s})$ for some $\mathbf{s} \leftarrow \chi_s$ as in Lemma 3 or from $A^{\mathsf{random}}_{N,q,\sigma_1,\sigma_1}$:

```
$\mathcal{B}_i(\mathbf{A}, \mathbf{b}, \mathbf{v}) = (\mathbf{A}, \mathbf{u}_1, \ldots, \mathbf{u}_{i-1}, \mathbf{y}_i, \ldots, \mathbf{y}_m)$ where
    For $j = 1 \ldots i-1$: $\mathbf{u}_j \leftarrow R_q$;
    For $j = i+1 \ldots m-1$: $\mathbf{s}_j \leftarrow \chi_s$ , $\mathbf{e}_j, \mathbf{f}_j \leftarrow D_{\sigma_1}^k$, and
        $\mathbf{y}_j = \mathbf{A}\mathbf{s}_j + T(\mathbf{e}_j + \mathbf{f}_j) + \mathbf{x}_j$;
    $\mathbf{y}_i = \mathbf{b} + T \cdot \mathbf{f}_i + \mathbf{x}_i$ for $\mathbf{f}_i \leftarrow D_{\sigma_1}^k$;
    $\mathbf{y}_m = -\mathbf{w} - \mathbf{A}\mathbf{r} - \mathbf{b} + \mathbf{v} + T(\sum_{j=1}^{i-1}(\mathbf{e}_j + \mathbf{f}_j) + \mathbf{e}_m) + \mathbf{z}$
        for $\mathbf{w} = \sum_{j=1}^{i-1} \mathbf{u}_j$, $\mathbf{r} = \sum_{j=i+1}^{m-1} \mathbf{s}_j$, $\mathbf{e}_m \leftarrow D_{\sigma_1}^k$,
        and $\mathbf{z} = \sum_{j=1}^{i-1} \mathbf{x}_j$.
```

First, consider $(\mathbf{A}, \mathbf{b}, \mathbf{v}) \leftarrow B^{\mathsf{HintRLWE}}_{N,q,\sigma_1,\sigma_1}(\mathbf{s})$, i.e., $\mathbf{b} = \mathbf{A}\mathbf{s} + T \cdot \mathbf{e}$ and $\mathbf{v} = \mathbf{e} + \mathbf{f}$ for $\mathbf{e}, \mathbf{f} \leftarrow D_{\sigma_1}^k$. We rewrite using fresh variable names $\mathbf{s}_i = \mathbf{s}$, $\mathbf{e}_i = \mathbf{e}$, and $\mathbf{f}_m = \mathbf{f}$. As a result we have $\mathbf{y}_i =$

$\mathbf{b} + T \cdot \mathbf{f}_i + \mathbf{x}_i = \mathbf{A}\mathbf{s}_i + T(\mathbf{e}_i + \mathbf{f}_i) + \mathbf{x}_i$ and

$$\mathbf{y}_m = -\sum_{j=1}^{i-1} \mathbf{u}_j - \mathbf{A} \sum_{j=i}^{m-1} \mathbf{s}_j + T(\sum_{j=1}^{i-1} (\mathbf{e}_i + \mathbf{f}_i) + \mathbf{e}_m + \mathbf{f}_m) + \sum_{j=1}^{i-1} \mathbf{x}_j;$$

so the output of the reduction is the same as $\mathcal{H}_{i-1}$.

On the other hand, consider $(\mathbf{A}, \mathbf{b}, \mathbf{v}) \leftarrow A^{\mathsf{random}}_{N,q,\sigma_1,\sigma_1}$, i.e., $\mathbf{b} \leftarrow R_q$ and $\mathbf{v} = \mathbf{e} + \mathbf{f}$ for $\mathbf{e}, \mathbf{f} \leftarrow D^k_{\sigma_1}$. Denote $\mathbf{u}_i = \mathbf{b}$, $\mathbf{e}_i = \mathbf{e}$, and $\mathbf{f}_m = \mathbf{f}$. Then the output of the reduction is

$$w = (\mathbf{A}, \mathbf{u}_1, \ldots, \mathbf{u}_{i-1}, \mathbf{u}_i + T\mathbf{f}_i + \mathbf{x}_i, \mathbf{y}_{j+1}, \ldots, \mathbf{y}_{m-1}, \mathbf{y}_m),$$

where

$$\mathbf{y}_m = -\sum_{j=1}^{i} \mathbf{u}_j - \mathbf{A} \sum_{j=i+1}^{m-1} \mathbf{s}_j + T(\sum_{j=1}^{i} \mathbf{e}_j + \sum_{j=1}^{i-1} \mathbf{f}_j + \mathbf{e}_m + \mathbf{f}_m) + \sum_{j=1}^{i-1} \mathbf{x}_j.$$

Since $\mathbf{u}_i = \mathbf{b}$ is uniformly random, w follows exactly the distribution $\mathcal{H}_i$. By the hardness assumption of $\mathsf{RLWE}_{N,q,\sigma}$ and by Lemma 3, $\mathcal{H}_{i-1}$ and $\mathcal{H}_i$ are indistinguishable, and hence $\mathcal{D}_0$ and $\mathcal{D}_1$ are indistinguishable.

To estimate the bit security loss, notice that the above proof involves $m-1$ hybrids. By assumption, the decisional $\mathsf{HintRLWE}$ problem is κ-bit secure, so the advantage of any time t adversary distinguishing $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$ is at most $\varepsilon = t/2^\kappa$. By [38], any time T adversary distinguishing $\mathcal{H}_0 = \mathcal{D}_0$ and $\mathcal{H}_{m-1} = \mathcal{D}_1$ has advantage at most $3m^2t/2^\kappa$; so our encryption scheme can achieve $\kappa - 2(\log m - 1)$ bit security. □

Now we state and prove the security of RLWE-SecAgg.

Theorem A.2 (Semi-Honest Security). *Assume Algorithm 6 is instantiated with a good graph generation algorithm* GENERATEGRAPH*, a semantic secure authenticated encryption scheme $\mathsf{E}_{\mathsf{auth}}$, a secure key agreement protocol $\mathcal{KA}$, and the RLWE encryption instantiated with parameters $N, q, \sigma > 0$ and noise distribution $\chi_e = D_{\sqrt{2}\sigma}$ such that $\mathsf{RLWE}_{N,q,\sigma}$ is hard. There exists a PPT simulator Sim such that for all sets of surviving clients A_1, A_2, A_2', A_3 as defined in Algorithm 6, all inputs $\mathcal{X} = (\mathbf{x}_i)_{i\in[n]}$, and all sets of corrupted clients $\mathcal{C}$ with $|\mathcal{C}| \leq \gamma n$, the output of $\mathsf{Sim}(\mathbf{z}, \mathcal{C}, A_1, A_2, A_2', A_3)$ is computationally indistinguishable from the joint view of the server and the corrupted clients, where $\mathbf{z} = \sum_{i\in A_2'\setminus\mathcal{C}} \mathbf{x}_i$ is the sum of inputs of surviving honest clients.*

Proof. We follow the blueprint of the hybrid argument in [5, Theorem 3.6] with the following modification.

- Hyb_1 to Hyb_7: We use the same hybrids as in [5] except that $\mathbf{s}_u$ in Hyb_4 for $u \in A_2 \setminus A_2'$ are sampled from χ_s, and that $\mathbf{s}_{ij}$ for all honest $i < j \in A_2'$ are sampled from χ_s and $\mathbf{s}_{ji} = -\mathbf{s}_{ij}$ for all $j < i$. Indistinguishability still follows from the fact that F is a secure PRG.
- Hyb_8[2]: In this hybrid we set $\mathbf{y}_i$ for all honest parties $i \in A_2'$ to be $\mathbf{y}_i = \mathbf{u}_i$, where $\mathbf{u}_i$ are chosen at random subject to

$$\sum_{i\in A_2'\setminus\mathcal{C}} \mathbf{u}_i = T \cdot \sum_{i\in A_2'\setminus\mathcal{C}} (\mathbf{e}_i + \mathbf{f}_i) + \mathbf{G}\mathbf{z} \pmod{q}.$$

  By the property of GENERATEGRAPH, with overwhelming probability the graph on $A_2' \setminus \mathcal{C}$ is connected, and thus $\sum_{i\in A_2'\setminus\mathcal{C}}(\sum_{\substack{j\in A_2'\setminus\mathcal{C}\\ j<i}} \mathbf{s}_{ij} - \sum_{\substack{j\in A_2'\setminus\mathcal{C}\\ i<j}} \mathbf{s}_{ij}) = 0$. So, by Lemma 4 this hybrid is indistinguishable from Hyb_7.

- Hyb_9: In this hybrid we set $\mathbf{y}_i$ for all honest parties $i \in A_2 \setminus A_2'$ to be uniformly random. Since no honest party sends shares of $\mathbf{seed}_i$, and due to the property of GENERATEGRAPH, with overwhelming probability the server does not receive a sufficient number of shares of $\mathbf{seed}_i$. Thus this hybrid is indistinguishable from Hyb_8 by the pseudorandomness of $\mathsf{RLWE}_{N,q,\sigma}$ samples.

The last hybrid Hyb_9 can be computed from the simulator's input. Therefore the security claim follows. □

B Proofs of Security for ACORN-detect

In our security proofs, we assume that the set of honest clients dropouts is public information, and thus we don't prevent the adversary from using that information. However, whether malicious clients drop out is decided by the adversary, in a possibly input-dependent way. We also assume that honest clients always provide valid inputs, as expected.

To prove security, we follow the standard simulation-based argument [22, 33] and show that every attacker against our protocol can be simulated by an attacker in an ideal world where a trusted party $\mathcal{T}$ computes a function F (vector summation in our case) on the clients' inputs $\mathcal{X}$. Recall that we consider an attacker $\mathcal{A}$ controlling at most γn clients and possibly also the server. The ideal world consists of the following steps (see Lindell [33] for details), which are adapted from the general case to our simpler setting where only one party (the server) has an output: (a) the honest clients send their inputs to $\mathcal{T}$, (b) $\mathcal{A}$ chooses which corrupted clients send their input to $\mathcal{T}$ and which ones abort, (c) if the server is corrupted $\mathcal{A}$ gets to choose whether to abort the protocol or continue, and (d) if the protocol is not aborted, $\mathcal{T}$ gives the server its prescribed output $F(\mathcal{X})$. Finally, (e) if the server is not corrupted then it outputs what it received from $\mathcal{T}$.

In our case, F is parameterized by the list of inputs $\mathcal{X} = (\mathbf{x_i})_{i\in[n]}$, the set of dropouts $\mathcal{D} \subseteq [n]$, and the maximum frac-

[2] There appears to be a typo in the proof of [5, Theorem 3.6] that Hyb_9 therein is a duplicate of Hyb_8, so we skip it and refer to the following hybrid as Hyb_8 in our proof.

tion of dropouts $\delta \in [0,1]$, and is defined as follows:

$$F_{D,\delta}(\mathcal{X}) = \begin{cases} \sum_{i\in\overline{\mathcal{D}}} \mathbf{x_i} & \text{if } |\mathcal{D}| \leq \delta n \wedge \mathsf{valid}(\mathbf{x_i}) = 1 \ \forall i \in \overline{\mathcal{D}}, \\ \bot & \text{otherwise.} \end{cases} \tag{10}$$

We first consider the honest server case, which is the most relevant one for input validation. We denote by $\mathsf{Real}_{\mathcal{A}(\mathbf{z}),C}(\mathcal{X})$ (resp. $\mathsf{Ideal}_{\mathcal{A}(\mathbf{z}),C}(\mathcal{X})$) the real (respectively ideal) executions of our protocol where the adversary $\mathcal{A}$ has full control over clients in C but not the server.

Theorem B.1 (Honest server)**.** *For any C such that $|C| \leq \gamma n$, dropouts $\mathcal{D} \subseteq [n]$, and input $\mathcal{X}$, there exists a PPT simulator* Sim *such that* $\mathsf{Ideal}_{\mathsf{Sim}(\mathbf{z}),C}(\mathcal{X}) \approx \mathsf{Real}_{\mathcal{A}(\mathbf{z}),C}(\mathcal{X})$.

Proof. Sim starts by internally invoking $\mathcal{A}$, setting the inputs of honest clients to be some fixed valid vector $\mathbf{v}$ and following the protocol honestly. As Sim honestly follows the protocol and messages between clients are independent of their inputs, this interaction is identical to what $\mathcal{A}$ expects.

Since Sim controls the server, it learns $\mathsf{com}_{\mathbf{sk},i}$, $\mathsf{com}_{\mathbf{s},i}$, $\pi^{\mathsf{Enc}(\mathbf{sk}_i,\mathbf{x}_i)}$, $\pi^{\mathsf{valid}(\mathbf{x}_i)}$, $\pi^{0\leq\mathbf{x}_i<t}$, and $\pi_i^{\sum\mathbf{sk}_i}$ from all corrupted clients i that do not drop out. From any valid $\pi^{\mathsf{Enc}(\mathbf{sk}_i,\mathbf{x}_i)}$ it can use the extractor guaranteed by knowledge soundness to learn the witness for i, and in particular the input $\mathbf{x}_i$. Sim thus learns for each corrupted client i whether it should drop out, provide an invalid input (in the case that its proofs fail to verify), or provide a valid input (and what that input should be).

Using this information, Sim has the appropriate clients abort in step (b). For surviving clients whose proofs failed to verify, Sim picks arbitrary invalid inputs and sends them to $\mathcal{T}$. For each remaining client i, Sim sends the extracted input $\mathbf{x}_i$. In step (d), the ideal-world server gets $F(\mathcal{X})$ from the trusted party $\mathcal{T}$. The simulator's choice of inputs match the real-world inputs, and thus the real-world and ideal-world server have the same output.

In the ideal world, the server outputs $F(\mathcal{X})$, and the Sim outputs whatever $\mathcal{A}$ outputted in the internal simulation. Again, this is identical to its output in the real world as Sim followed the protocol honestly. □

We now discuss the case of a malicious server. We denote by $\mathsf{Ideal}_{\mathcal{A}(\mathbf{z}),C\cup\{S\}}(\mathcal{X})$ the output received by an adversary $\mathcal{A}$ with auxiliary input $\mathbf{z}$ controlling *both* a set of $C \subseteq [n]$ corrupted clients and the server in an ideal world execution. Analogously, the view of such an adversary in the real world is denoted by $\mathsf{Real}_{\mathcal{A}(\mathbf{z}),C\cup\{S\}}(\mathcal{X})$.

The next theorem states that our protocol achieves malicious security with the exact same assumption as Bell et al., i.e. semi-honest server behaviour in the key distribution phase, which is implied by a Public Key Infrastructure (PKI). In the case of semi-honest server and fully malicious clients the theorem follows without that assumption. Such an attacker can fully control clients and additionally has read access to the protocol execution of the server. For the malicious server, the ideal functionality computed by $\mathcal{T}$ is a generalization the one in Equation 10 and is described in Bell et al. [5] [Definition 4.1]. In that functionality the trusted party outputs large partial sums instead of a single sum, but the argument is analogous, so we stick to the simpler notion from Equation 10 for simplicity.

The proof requires from the simulator the ability to, given a length ℓ sum vector $\mathbf{s_H}$ corresponding to the sum of inputs of honest clients, come up with a set of plausible inputs for the honest clients, i.e. produce $(1-\gamma)n$ *valid* inputs that add up to $\mathbf{s_H}$, i.e. In our proof we assume for simplicity that this task can be done in polynomial time by the simulator. It is easy to see that this is indeed the case for a validity predicate valid that corresponds to an L_0, L_∞, or L_1 bound, and not it is not a loss of generality for the purpose of proving security, as we can equip the ideal functionality, and thus the trusted party $\mathcal{T}$ in the ideal world, with the ability to output this set of plausible inputs to the simulator.

Theorem B.2 (Malicious server, with PKI)**.** *For any C such that $|C| \leq \gamma n$, dropouts $\mathcal{D} \subseteq [n]$, and input $\mathcal{X}$, there exists a PPT simulator* Sim *such that* $\mathsf{Ideal}_{\mathsf{Sim}(\mathbf{z}),C\cup\{S\}}(\mathcal{X}) \approx \mathsf{Real}_{\mathcal{A}(\mathbf{z}),C\cup\{S\}}(\mathcal{X})$.

Proof. (Sketch.) The proof is analogous to the one by Bell et al., shown in [5] as a series of hybrids. The key idea is that Sim can *extract* the sum $\mathbf{s_H}$ of inputs of honest surviving clients by querying the trusted party with $\mathbf{0}$ as input for all corrupted clients. More precisely, Sim proceeds as follows:

1. Sim does not instruct any corrupted client to abort in step (b), and sets their inputs to be all $\mathbf{0}$ (we assume w.l.o.g. that $\mathsf{valid}(\mathbf{0}) = 1$). In step c), Sim does not abort the server.

2. In step d) Sim learns $\mathbf{s_H} := F(\mathcal{X})$. Note that all honest clients provide valid inputs, and thus $\mathbf{s_H}$ is the sum of inputs of honest surviving clients.

3. Sim internally invokes $\mathcal{A}$ controlling all honest clients, setting their inputs arbitrarily, as long as they're all valid and add up to $\mathbf{s_H}$ (we assume finding such fake inputs can be done efficiently as discussed above). Sim then outputs whatever $\mathcal{A}$ in its internal simulation outputs.

Note that the internal execution by Sim in step 3. accounts for the fact that the output of $\mathcal{A}$ in the real world might depend on the view (including output) of the (corrupted) real-world server. This view in turn depends on honest clients' inputs. Since Sim set honest inputs in the internal execution to match the sum of inputs in the real world, then by the hybrid argument in [5] the output of $\mathcal{A}$ in Sim's internal execution is indistinguishable of that of $\mathcal{A}$ in the real world, which concludes the proof. That this hybrid argument still holds in our case follows trivially from the fact that the proofs that we added for validity checking are all zero-knowledge. □

C ACORN-robust: Details and Proofs

C.1 The ACORN-robust Construction

In this section we give a detailed description of ACORN-robust, along with security and robustness proofs. The main protocol is described in Algorithm 1, and it is almost identical to ACORN-detect, with the exception that the `ShareSeeds` and `RecoverAggKey` correspond to the modified version described in Section 4, as opposed to the ones from Bell et al. [5].

Algorithm 2 describes the `ShareSeeds` subprotocol. Note that, as discussed in Section 4, in Step 1 clients commit to seeds before they know which neighbors will be assigned to those seeds, which happens in step 2. Moreover, a key pair is generated for each neighbor, which is used to communicate with that neighbor via the server securely, using authenticated encryption. As we will see later, clients might reveal a secret key to the server, as a way to open the communication, and expose their neighbor as a cheater, e.g., a client having produced an inconsistent seed-mask pair. The choice of neighboring random graph is the same of [5]. More precisely, the communication graph is a k-regular Harary graph with n nodes. Assuming even k for simplicity, this graph results from arranging n nodes in a circle and having the neighbors of each nodes be (a) the subsequent $k/2$ nodes clockwise, and (b) the $k/2$ preceding nodes counter-clockwise. So far this is a deterministic structure, the randomness comes into the labeling of the nodes, i.e. the placement of clients $1,\ldots,n$ in the nodes of the graph. Our proof of Theorem 4.1 (which we give in the next subsection) is tailored to this random graph construction.

The seed exchange and seed sharing stages are also analogous to [5], the only differences are that (i) the sharing of seeds is done via Feldman's verifiable secret sharing scheme, described in Algorithm 3, and (ii) clients check/supervise that the received masks match the commitment they expect. In these steps, clients secret-share their random seeds across their neighbors, using the server as a relay (communication channel), and relying on public keys sent in the first step. Thanks to Feldman's VSS, shared seeds are consistent with their commitments after this point, as otherwise the server drops those clients (see Algorithm 3), and any pairs seed-mask for all pairwise masks $\mathbf{s_{i,j}}$ involving an honest client are consistent, as otherwise the server drops the dishonest client in the second step of the seed sharing stage.

Step 2. of Algorithm 1 is analogous to ACORN-detect: $\mathbf{y_i}$ is formed in the exact same way and the same validity proofs are provided (excluding the distributed Schnorr proof).

Finally, the `RecoverAggKey` subprotocol is described in Algorithm 4. The server first recovers self masks $\mathbf{s_i}$ from the corresponding seeds and drops clients for which the recovered masks are not consistent with the commitments that where submitted (and consequently with respect to which client proofs were made in Step 2 of Algorithm 1). What remains is recovering pairwise masks from dropped out clients in Steps 3 and 4 in Algorithm 4, possibly dropping misbehaving clients along the way, and thus requiring additional rounds of interaction with their neighbors to recover their pairwise masks. In the next section we show the bound on the number of additional rounds (Theorem 4.1).

C.2 Proof of Theorem 4.1

Let us restate the theorem in a slightly more explicit way for convenience:

Theorem C.1. *Suppose at most $\alpha < 1/3$ fraction of clients are malicious and that α plus the fraction of dropouts is less than δ. Then the probability that ACORN-robust requires at least $6+r$ rounds to finish is bounded by a term that is negligible in k and thus the security parameter plus $(\alpha n/2 + n/k)(\sqrt{8}\alpha)^{r-1}$.*

Consider the following graph model with parameters n,k,m and $(a_1,\ldots,a_m)$. Place n vertices in a circle, label the vertices with labels 1 through n uniformly at random. For each vertex $i \in [m]$, let S_i be the set of the $k/2$ vertices clockwise from i and i itself. Then for each $i \in [m]$, choose a_i vertices from S_i and add an edge between each of them an i. Finally remove from the graph the vertices $[n]-[m]$ along with any vertices neighboring them. Call the distribution of the resulting graph $G(n,k,m,(a_i)_{i\in[m]})$.

Let P_l be the path of $l+1$ vertices and L_l be given by P_{l-1} with a self edge added to exactly one of its end points.

Theorem C.2. *Consider Algorithm 1 run with n clients, each with k neighbors. Suppose at most m clients are malicious and m plus the number of honest dropouts is less than δ. Let a_i be the number of bad mask commitments the ith malicious client sends in the first step of the protocol.*

The probability that step 3 will be executed $\geq l$ times (i.e. the number of rounds of the whole protocol is $\geq 6+l$) is less than by the probability that $G(n,k,m,(a_i)_{i\in[m]})$ contains either P_l or L_l as an induced subgraph.

Proof. Note that the graph model can be given by running the protocol with the given parameters: having the clients correspond to nodes, malicious clients to the nodes labelled by $[m]$, the as to the number of mismatching commitments each client sends in the first round and the edges to the pairs these are assigned to. The malicious clients who survive for masked input encoding then induce the appropriate random graph.

Let i_l be (one of) the client(s) whose edge/pairwise mask shares are requested in the lth iteration of Step 3. For $k>1$ if client i_k had their shares requested in the kth iteration then there must be some client with whom they share an edge (with mismatched commitments) whose shares were requested in the $k-1$th iteration, call this client i_{k-1}. Clients $i_1,\ldots,i_l$ form a subgraph isomorphic to P_{l-1}. In order for i_1 to have their

Algorithm 1: ACORN-robust, Secure Aggregation with Input Verification Robust to Malicious Clients.

```
Parties: Clients $1,\dots,n$, and Server.
Public Parameters: Vector length $\ell$, input domain $\mathbb{X}^\ell$, and PRG $\mathsf{F}\colon \{0,1\}^\lambda \mapsto \mathsf{supp}(\chi_s)^\ell$
Input: $\mathbf{x}_i \in \mathbb{X}^\ell$ (by each client $i$).
Output: $z \in \mathbb{X}^\ell$ (for the server).
  1. Using the server to send messages, clients engage in a ShareSeeds protocol, with each surviving client $i$ learning $\{\mathbf{seed}_{i,j}\}_{j\in\mathsf{N}(i)}$, $\mathbf{seed}_i$, $\mathsf{shares}_{i,\mathcal{D}}$, and $\mathsf{shares}_{i,\mathcal{S}}$. The server receives commitments $\mathsf{com}_{\mathbf{seed}_{i,j}}$ and $\mathsf{com}_{\mathbf{seed}_i}$ to $\mathbf{seed}_{i,j}$ and $\mathbf{seed}_i$ and also $\mathsf{com}_{\mathbf{s}_{i,j}}$ and $\mathsf{com}_{\mathbf{s}_i}$ which if $i,j$ are honest are commitments to $\mathbf{s}_{i,j} = \mathsf{F}(\mathbf{seed}_{i,j})$ and $\mathbf{s}_i = \mathsf{F}(\mathbf{seed}_i)$. The server aborts if there are fewer than $(1-\delta)n$ surviving clients.
  2. Each surviving client $i$ performs the following:
       • Computes its packed encrypted input $\mathbf{y}_i = \mathsf{Encode}(\mathbf{sk}_i, \mathbf{G}\mathbf{x}_i)$ with key defined as $\mathbf{sk}_i = \mathbf{s}_i + \sum_{j\in A_i, j<i} \mathbf{s}_{ij} - \sum_{j\in A_i, i<j} \mathbf{s}_{ij}$.
       • Forms a commitment $\mathsf{com}_{\mathbf{x}_i}$ to its input.
       • Computes proofs $\pi^{\mathsf{Enc}(\mathbf{sk}_i,\mathbf{x}_i)}$, $\pi^{0\le\mathbf{x}_i<t}$, and $\pi^{\mathsf{valid}(\mathbf{x}_i)}$ of encoding, smallness, and validity.
       • Sends to the server $\mathbf{y}_i$, $\mathsf{com}_{\mathbf{x}_i}$, $\pi^{\mathsf{Enc}(\mathbf{sk}_i,\mathbf{x}_i)}$, $\pi^{0\le\mathbf{x}_i<t}$, $\pi^{\mathsf{valid}(\mathbf{x}_i)}$.
  3. The server aborts if it receives fewer than $(1-\delta)n$ messages. The server counts any client whose proofs fail as a dropout. Otherwise, the server and the clients engage in the corresponding RecoverAggKey protocol. The server takes as input the global sets $\mathcal{D}$ and $\mathcal{S}$ of dropouts and survivors and its commitments from the first round. Each client $i$ takes as input its sets $\mathsf{shares}_{i,\mathcal{D}}$ and $\mathsf{shares}_{i,\mathcal{S}}$. At the end of the protocol the server learns a set $R$ of remaining clients and the aggregate key $\mathbf{sk} = \sum_{i\in R} \mathbf{sk}_i$.
  4. The server outputs $\sum_{i\in R} \mathbf{x}_i$ as $\mathbf{G}^{-1}(\mathsf{Decode}(\mathbf{sk}, \sum_{i\in R} \mathbf{y}_i))$.
```

edge masks requested for the first iteration they must either have a bad self mask (and thus a self edge) or a bad edge mask with someone, call them i_0, who failed to provide a masked input. This i_0 must also be malicious to have not reported the mismatch earlier. Hence there must be either a a copy of P_l or L_l.

Finally, this subgraph must be an induced subgraph because if there is an edge between i_k and i_{k+c} then i_{k+c} will be requested in the $k+1$th iteration, therefore $c = 1$. □

Lemma 5. *Assume $m < n/3$ then with probability at least $1 - m\exp\left(-2(1/3 - m/n)^2 k\right)$ every vertex in $[m]$ has $\le k/3$ neighbors in $[m]$.*

Proof. For each $i \in [m]$ the number of neighbors of i in $[m]$ is given by a Hypergeometric distribution with parameters $n-1$,$m-1$ and k. The probability that that random variable is $\ge k$ is bounded by $\exp\left(-2(1/3 - m/n)^2 k\right)$ according to the tail bound in [47]. The result follows by a union bound. □

Theorem C.3. *If $m \le n/3$ then the probability that $G(n,k,m,(a_i)_{i\in[m]})$ contains either P_l or L_l as an induced subgraph is at most*

$$\left(\frac{m}{2} + \frac{n}{k}\right)\left(\frac{\sqrt{8}m}{n}\right)^{l-1} + m\exp\left(-2\left(\frac{1}{3} - \frac{m}{n}\right)^2 k\right).$$

Proof. We will bound the probability of seeing a copy of P_l by taking a union bound over ordered sequences of vertices in $[m]$. There are $m!/(m-l-1)!$ sequences of $l+1$ such vertices.

For a specific instance of a sequence, consider assigning each of the vertices to a place around the circle one by one. There are 2^l ways to choose whether each vertex in the sequence is placed clockwise or anticlockwise from the previous one. Therefore there are $2^l m!/(m-l-1)!$ possible choices of sequences together with whether they are clockwise or anticlockwise.

For all but the first vertex the probability that they are placed as a neighbor in the Harary graph to the previous vertex, and in the correct direction, is at most $k/2$ divided by the number of remaining vertices. Thus they form a path in the Harary graph with probability bounded by $(k/2)^l (n-l-1)!/(n-1)!$.

Fix a sequence and a choice of whether each is clockwise or anticlockwise from the previous. Assume wlog that the vertices in the sequence are $1,\dots,l+1$, in that order. For each $i \in [l+1]$ vertex i is required to be the source of r_i edges in the path for r_i either 0, 1 or 2.

Vertex i sends out $a_i - r_i$ other edges all of which must be to vertices in $[m] \setminus [l+1]$. Let α_i be fraction of is neighbors in the Harary graph (other than $i+1$ and $i-1$) that are in $[m] \setminus [l+1]$. The probability that vertex i is only connects to such vertices is bounded by $\alpha_i^{a_i - r_i}$.

The probability that vertex i sends edges to the clockwise neighbors in the path is

$$\frac{\binom{k/2-r_i}{a_i-r_i}}{\binom{k/2}{a_i}} = \frac{(k/2-r_i)!a_i!}{(k/2)!(a_i-r_i)!} \le \frac{2^{r_i} a_i!}{k^{r_i}(a_i-r_i)!} \tag{11}$$

Assume that $\alpha_i \le 1/3$ for all $i \in [l+1]$ by Lemma 5 this will hold with all but a small probability. Then a bound on the probability that the edges from i are as required is given by

$$\frac{2^{r_i} a_i!}{k^{r_i}(a_i-r_i)!} 3^{r_i - a_i}$$

which is maximised when $a_i = r_i$.

We assume that the a_i happen to be in this worst case arrangement[3]. As the r_i sum to l the product of these probabilities is at most $(\sqrt{8}/k)^l$.

We now take a union bound over all length $l+1$ sequences in $[m]$ and choices for clockwise or anticlockwise at each step. Giving the probability of having a copy of P_l as bounded by

$$\frac{2^l m!}{(m-l-1)!}\frac{(k/2)^l(n-l-1)!}{(n-1)!}\left(\frac{\sqrt{8}}{k}\right)^l \leq \frac{\sqrt{8}^l m^{l+1}}{n^l}. \quad (12)$$

However we have counted every sequence twice (once forward and once in reverse) so we can halve this to get a bound of $\frac{m}{2}\left(\frac{\sqrt{8}m}{n}\right)^l$.

The analysis for the probability of L_l is very similar and gives a bound of $\frac{n}{k}\left(\frac{\sqrt{8}m}{n}\right)^l$.

Adding these two bounds and the bound from Lemma 5 on the probability of any $\alpha_i > 1/3$ gives the result. □

Theorem 4.1 follows immediately from combining these two results.

Finally, we proof security of ACORN-detect in the next section.

C.3 Proofs of Security for ACORN-robust

In proving the security of ACORN-robust, we follow the same approach as for ACORN-detect. We define the a slightly different functionality. Given a set of inputs $\mathcal{X}$ and dropouts $\mathcal{D}$, let $\mathcal{V}(\mathcal{X},\mathcal{D}) = \{i \in \overline{\mathcal{D}} | \mathsf{valid}(\mathbf{x_i}) = 1\}$, i.e. the set of valid inputs from non-dropouts.

$$F_{D,\delta}(\mathcal{X}) = \begin{cases} \sum_{i\in\mathcal{V}(\mathcal{X},\mathcal{D})} \mathbf{x_i} & \text{if } |\mathcal{V}(\mathcal{X},\mathcal{D})| \geq (1-\delta)n, \\ \perp & \text{otherwise.} \end{cases} \quad (13)$$

We define $\mathsf{Ideal}_{\mathcal{A}(\mathbf{z}),C}(\mathcal{X})$, $\mathsf{Real}_{\mathcal{A}(\mathbf{z}),C}(\mathcal{X})$,$\mathsf{Ideal}_{\mathcal{A}(\mathbf{z}),C\cup\{S\}}(\mathcal{X})$ and $\mathsf{Real}_{\mathcal{A}(\mathbf{z}),C\cup\{S\}}(\mathcal{X})$ as in Section 4.1 except with this new functionality F in place of the old one.

Theorem C.4 (Honest server)**.** *For any C such that $|C| \leq \gamma n$, dropouts $\mathcal{D} \subseteq [n]$, and input $\mathcal{X}$, there exists a PPT simulator* Sim *such that* $\mathsf{Ideal}_{\mathsf{Sim}(\mathbf{z}),C}(\mathcal{X}) \approx \mathsf{Real}_{\mathcal{A}(\mathbf{z}),C}(\mathcal{X})$.

Proof. Sim starts by internally invoking $\mathcal{A}$, setting the inputs of honest clients to be some fixed valid vector $\mathbf{v}$ and following the protocol honestly. As Sim honestly follows the protocol (with honest dropouts specified by $\mathcal{D}$) and messages between clients are independent of their inputs, this interaction is identical to what $\mathcal{A}$ expects.

Since Sim controls the server, it learns $\pi^{\mathsf{Enc}(\mathbf{sk}_i,\mathbf{x}_i)}$, $\pi^{\mathsf{valid}(\mathbf{x}_i)}$ and $\pi^{0\leq\mathbf{x}_i<t}$, from which it can extract any valid inputs from adversarial clients by knowledge soundness. Sim also keeps track of which of the clients are dropped in its internal protocol run.

Assume the protocol finishes with the server learning a sum of all honest (simulated with zero input) non-dropout and some malicious clients. Sim has the dropped clients abort in step (b). For each remaining client i, Sim sends the extracted input $\mathbf{x}_i$. In step (d), the ideal-world server gets $F(\mathcal{X})$ from the trusted party $\mathcal{T}$. The simulator's choice of inputs match the real-world inputs, and thus the real-world and ideal-world server have the same output, because the effect of non-zero honest non-dropout inputs is to be added to the output of the protocol.

In the ideal world, the server outputs $F(\mathcal{X})$, and the Sim outputs whatever $\mathcal{A}$ outputted in the internal simulation. Again, this is identical to its output in the real world as Sim followed the protocol honestly.

It remains to see that the internal and thus real protocol ends with the server learning a sum including all honest non-dropout clients. To see this note that at each point where the server drops a client it is only after seeing some proof of malicious behaviour on that clients part. □

Theorem C.5 (Semi-honest server)**.** *For any C such that $|C| \leq \gamma n$, dropouts $\mathcal{D} \subseteq [n]$, and input $\mathcal{X}$, there exists a PPT simulator* Sim *such that* $\mathsf{Ideal}_{\mathsf{Sim}(\mathbf{z}),C\cup\{S\}}(\mathcal{X}) \approx \mathsf{Real}_{\mathcal{A}(\mathbf{z}),C\cup\{S\}}(\mathcal{X})$.

Proof. (Sketch.) This proof goes exactly like the proof of Theorem B.2, except we have to worry that the extra mask revelations at the end of the protocol might leak more information about honest clients. In fact, this can't happen because the server is semi-honest and will only request both sets of mask for a client if it has been given some kind of proof that that client is malicious. Specifically, it will only request pairwise mask shares for a client who it has requested the self-mask for if one of their masks doesn't expand from the appropriate shared seed. That can only happen if the client confirmed that the seed with a given commitment expanded to a vector with a given commitment when it didn't. That is malicious behaviour so no honest client can be a victim of this. □

D Additional Zero-Knowledge Proofs

D.1 Proof of Valid RLWE Encoding

We now describe in details the inner product constraints needed for proving valid RLWE encodings $\pi^{\mathsf{Enc}(\mathbf{sk},\mathbf{x})}$. Recall that the RLWE encoding of input $\mathbf{x} \in [t]^\ell$ is

$$\mathsf{Encode}(s,\mathbf{x}) := \mathbf{a}\cdot s + T(\mathbf{e}_0+\mathbf{e}_1) + \mathbf{m} \bmod q,$$

where $s \in R_q = \mathbb{Z}[X]/(q, X^N+1)$ is the client's RLWE secret, $\mathbf{a} \in R_q^{\ell/N}$ is a public vector of random polynomials,

[3] Arranging this would require choosing the a_i after seeing the random placement around the circle, therefore we believe this step is probably quite lossy.

$\mathbf{e}_0, \mathbf{e}_1 \leftarrow \chi_\sigma^{\ell/N}$ are vectors of discrete Gaussian polynomials in R, and $\mathbf{m} \in R_q^{\ell/N}$ are polynomials whose coefficients are packed input $\mathbf{Gx} \in [T]^\ell$. Here we use the gadget vector $\mathbf{g} = [1, B, B^2, \ldots, B^{\log_B q}]$ and gadget matrix $\mathbf{G} = \mathbf{I} \otimes \mathbf{g}^t$. Since the coefficients of noises $\mathbf{e}_0$ and $\mathbf{e}_1$ are independently sampled from the discrete Gaussian of standard deviation σ, we can replace $\mathbf{e}_0 + \mathbf{e}_1$ with $\mathbf{f} \leftarrow \chi_{2\sigma}^{\ell/N}$. Furthermore, since the above RLWE encoding is the concatenation of equations over R_q, for the remaining section we simply consider

$$y(X) = a(X) \cdot s(X) + T \cdot f(X) + m(X) \bmod q \tag{14}$$

as the RLWE encoding equation for a block $\tilde{\mathbf{x}}$ of input such that $\mathbf{G}\tilde{\mathbf{x}} \in [T]^N$.

To build the proof $\pi^{\mathsf{Enc}(\mathbf{s},\mathbf{x})}$, the client commits to its private values $\mathbf{s}, \mathbf{f}$ and $\mathbf{x}$, and then it generates inner product constraints in the Bulletproofs system. Let P be the group size underlying the Bulletproofs system. In practice we have $q \neq P$ for efficient RLWE encoding, and so we need to express the above equation as constraints modulo P.

First, notice that Equation 14 is equivalent to

$$y(X) = a(X) \cdot s(X) + T \cdot f(X) + m(X) + q \cdot b(X), \tag{15}$$

for some private polynomial $b(X) \in R$, over the integers. The components in this equation satisfy the following bounds:

$$\|y\|_\infty \leq q, \|a\|_\infty \leq q, \|s\|_\infty \leq q, \|m\|_\infty \leq T.$$

In addition, the error polynomial f satisfies $\|f\|_\infty \leq \beta_e$, where $\beta_e \leq 12\sigma$ is a high probability bound for discrete Gaussian coefficients with standard deviation 2σ. By expanding Equation 15, we get

$$y(X) = \sum_i (\sum_j s_j a_{i-j} \cdot (-1)^{i<j} + T \cdot f_i + m_i + q \cdot b_i) \cdot X^i.$$

If $2(Nq^2 + T\beta_e + T) < P$, then the coefficients of $y(X) - q \cdot b(X)$ do not wrap around $\bmod P$, and hence we can use inner product constraints modulo P to prove the relation in Equation 14.

Proof of smallness of f. The error bound $\|f\|_\infty \leq \beta_e$ is equivalent to a range constraint $f_i \in [-\beta_e, \beta_e]$ for all coefficients f_i of $f(X)$. We use a similar strategy as the proof of smallness of $\mathbf{x}$, by rewriting the range constraint as $c_i := (\beta_e - f_i)(\beta_e + f_i) \geq 0$ and considering $c_i' = 4c_i + 1 \geq 0$ for all i. Since $c_i' \equiv 1 \bmod 4$, it can be written as the sum of three squares. Let $\mathbf{f}' = 2\mathbf{f}$. The client can show that it knows secret vectors $\mathbf{u}, \mathbf{v}, \mathbf{w}$ such that

$$\mathbf{f}' \circ \mathbf{f}' + \mathbf{u} \circ \mathbf{u} + \mathbf{v} \circ \mathbf{v} + \mathbf{w} \circ \mathbf{w} = \alpha,$$

where $\alpha = (4\beta_e^2 + 1) \cdot \mathbf{1}$.

Following the blueprint of Section 5.3.2, the client first generates an approximate range proof showing

$$\|\mathbf{f}'||\mathbf{u}||\mathbf{v}||\mathbf{w}\|_\infty < \sqrt{P}/4.$$

Then, the client can generate proofs for the sum of three squares. By using a challenge vector $\mathbf{r} = (\tau^{i-1})_i$, we can replace the earlier constraint as

$$\langle \mathbf{f}', \mathbf{f}' \circ \mathbf{r} \rangle + \langle \mathbf{u}, \mathbf{u} \circ \mathbf{r} \rangle + \langle \mathbf{v}, \mathbf{v} \circ \mathbf{r} \rangle + \langle \mathbf{w}, \mathbf{w} \circ \mathbf{r} \rangle = \langle \alpha, \alpha \circ \mathbf{r} \rangle.$$

By using another challenge vector $\mathbf{r}_\mathbf{x} \| \mathbf{r}_\mathbf{u} \| \mathbf{r}_\mathbf{v} \| \mathbf{r}_\mathbf{w} = (\sigma^{i-1})_i$, we can rewrite the above as

$$\begin{aligned} &\langle \mathbf{f}' + \mathbf{r}_\mathbf{x}, (\mathbf{f}' - \mathbf{r}_\mathbf{x}) \circ \mathbf{r} \rangle + \langle \mathbf{u} + \mathbf{r}_\mathbf{x}, (\mathbf{u} - \mathbf{r}_\mathbf{u}) \circ \mathbf{r} \rangle \\ &+ \langle \mathbf{v} + \mathbf{r}_\mathbf{v}, (\mathbf{v} - \mathbf{r}_\mathbf{v}) \circ \mathbf{r} \rangle + \langle \mathbf{w} + \mathbf{r}_\mathbf{w}, (\mathbf{w} - \mathbf{r}_\mathbf{w}) \circ \mathbf{r} \rangle = s, \end{aligned}$$

where $s = \langle \alpha, \mathbf{r} \rangle + (\|\mathbf{r}_\mathbf{x}\|^2 + \|\mathbf{r}_\mathbf{u}\|^2 + \|\mathbf{r}_\mathbf{v}\|^2 + \|\mathbf{r}_\mathbf{w}\|^2)$.

The client needs to commit twice to **f**, **u**, **v**, and **w**. Then, given the challenge values τ and σ, it can generate a proof of the constraint

$$\langle \bar{\mathbf{f}} \| \bar{\mathbf{u}} \| \bar{\mathbf{v}} \| \bar{\mathbf{w}}, \bar{\mathbf{f}}' \| \bar{\mathbf{u}}' \| \bar{\mathbf{v}}' \| \bar{\mathbf{w}} \rangle = \gamma,$$

for some γ.

This constraint can be merged together with the approximate range proof:

$$\langle \bar{\mathbf{z}} \| \mathbf{y}, \bar{\mathbf{f}}' \| \bar{\mathbf{u}}' \| \bar{\mathbf{v}}' \| \bar{\mathbf{w}} \| \rho \mathbf{b} \rangle = \gamma + \rho \gamma'.$$

Combined Proof. To complete the proof, the client then needs to merge the above constraints with the those enforcing Equation 15. Let $\mathbf{a}, \mathbf{s}, \mathbf{f}, \mathbf{b} \in \mathbb{Z}_q^N$ be the coefficient embeddings of polynomials a, s, f, b, and let $\mathbf{a}^{(i)}$ be the ith negacyclic shift of $\mathbf{a}$. Then, Equation 15 is equivalent to the following set of inner product constraints:

$$\begin{aligned} \langle \mathbf{s} \| \mathbf{f} \| \mathbf{x} \| \mathbf{b}, \mathbf{a}^{(0)} \| T0 \ldots 0 \| \mathbf{g}\, \mathbf{0} \ldots \mathbf{0} \| q0 \ldots 0 \rangle = y_0 \quad (\bmod P), \\ \langle \mathbf{s} \| \mathbf{f} \| \mathbf{x} \| \mathbf{b}, \mathbf{a}^{(1)} \| 0T \ldots 0 \| \mathbf{0}\, \mathbf{g} \ldots \mathbf{0} \| 0q \ldots 0 \rangle = y_1 \quad (\bmod P), \\ \ldots \end{aligned}$$

Note that all these constraints can be merged into one inner product by using a fresh random challenge.

To count the size of the resulting inner product proof for the entire input **x**, let L be the length of **x** and assume **Gx** has length ℓ. Furthermore, let $k = \log_B q$ be the dimension of the gadget vector **g**. Each of the above inner product is on a vector of dimension $3N + kN$. Since the secret **s** is the same in all these inner products, the merged constraint has just one copy of **s**. Thus, the length of the combined constraint is an inner product of length $N + 2\ell + L$.

For the proof of smallness of **f**, we can prove the concatenated error vector is bounded. Such a constraint can be written as an inner product of length $8\ell + \lambda$.

Finally, we can merge these proofs with the proof of smallness of input **x**:

- Combined with the case where **x** is binary: The proof $\pi^{0 \leq \mathbf{x} < t}$ has input length $2L = 2ckN$, where **x** is part of the constraint. The proof of small **f** has **f** as part of the constraint. So the combined proof can reuse the **x** and **f** parts and has input length $2L + N + 9\ell + \lambda$.

- Combined with the case where $\mathbf{x}$ is in a large domain, i.e. t is larger: The proof $\pi^{0 \le \mathbf{x} < t}$ has input length $8L + \lambda$, where $\mathbf{x}$ is part of the constraint. Similar to the above, the proof of small $\mathbf{f}$ has $\mathbf{f}$ as part of the constraint. So the combined proof has input length $8L + N + 9\ell + 2\lambda$.

E End-to-End Experiment Setup

Here we give more details about our end-to-end experiments on the four federated learning tasks in Section 6.2. Our experiments ran on a laptop with a quad-core Intel i7-1185G7 CPU and 16GB memory. Neural network training was performed using Python in the TensorFlow [1] framework without GPU acceleration. We used the same training parameters as Lycklama et al. [34] and simulated the same network latency (0.5ms). In particular, we used SDG as the training algorithm and used the same local learning rates η: $\eta = 0.05$ for MNIST, $\eta = 0.01$ for CIFAR-10 S, $\eta = 0.05$ for CIFAR-10 L, and $\eta = 0.3$ for Shakespeare. We set the number of epochs per round for MNIST to 5, for CIFAR-10 S and CIFAR-10 L to 2, and for Shakespeare to 1. During each epoch, a client runs the training algorithm on 1248 samples for MNIST, 1024 samples for CIFAR-10 S and L, and 69440 samples for Shakespeare; such training tasks are done in batches of size 32 for MNIST and 64 for others. Model parameter updates are converted to 8-bit fixed point values by applying 8-bit probabilistic quantization with 7 fractional bits. In each of our experiments, we ran the federated learning task with a full dataset using a corresponding secure aggregation protocol, and the per-client performance was measured using the average of five end-to-end runs.

F Benchmarks of RLWE Variant of ACORN-detect

We present our benchmarks of the main components of the RLWE variant of ACORN-detect. As in Section 6.2, we set the graph parameters $\gamma = \delta = 1/10$. The client computation includes RLWE encoding, generating commitments to sk, $\mathbf{x}$, to the RLWE error vectors $\mathbf{e}$, to the private multiplier $\mathbf{b}$ in Equation 15, and to other private values needed by proofs as mentioned in Appendix D.1, as well as generating proofs $\pi^{0 \le \mathbf{x} < t}$, $\pi^{\mathsf{Enc}(\mathbf{sk},\mathbf{x})}$, and $\pi^{\mathsf{valid}(\mathbf{x})}$. The server computation includes batched proof verification and key recovery. Figure 7 shows the client and server running times as well as the client communication costs for both the FA setting ($t = 2, n = 10000$) and the FL setting ($t = 2^{16}, n = 500$).

Comparing with the PRG variant of ACORN-detect, whose benchmarks are shown on Figure 5, we see that the client encoding time and the server key reconstruction time are both significantly reduced, especially as the input $\mathbf{x}$ becomes longer. The client total running time is increased by 0.5X to 1X, due to more complicated constraints for the proof of valid encoding $\pi^{\mathsf{Enc}(\mathbf{sk},\mathbf{x})}$. For example, for input length 2^{16}, the total client time is 8.4s for the FA use case (compared with 4.7s for the PRG variant), and it is 26.4s for the FL use case (compared with 18.8s for the PRG variant). Client communication costs also increase accordingly, but with a smaller increment. On the server, despite the fact that ZK proof verification now runs longer, when the number of clients is large, the total server running time is now lower than the PRG variant for medium to long inputs. This is due to the much more efficient RLWE encoding. For the FL use case with $n = 500$ clients, the saving by RLWE encoding is less apparent, and the total server time is larger than the PRG variant.

Public parameters: Vector length ℓ, input domain $\mathbb{X}^\ell$, secret distribution χ_s, and seed expansion function $\mathsf{F}\colon \{0,1\}^\lambda \mapsto \mathsf{supp}(\chi_s)^\ell$
Client i's input: $\mathbf{x}_i \in \mathbb{X}^\ell$
Server output: $z \in \mathbb{X}$

Commitments

1. Client i generates keypairs $(\mathsf{sk}_{i,1}, \mathsf{pk}_{i,1}), (\mathsf{sk}_{i,2}, \mathsf{pk}_{i,2}) \leftarrow \mathsf{Sig.KeyGen}(1^\lambda)$ and sends $(\mathsf{pk}_{i,1}, \mathsf{pk}_{i,2})$ to the server. It performs the first step in the distributed key correctness protocol, which results in it sending a message h_i to the server.
2. The server commits to the public key vectors $\mathsf{pk}_1 = (\mathsf{pk}_{i,1})_i$ and $\mathsf{pk}_2 = (\mathsf{pk}_{i,2})_i$ using a Merkle tree. It sends the root hashes $h_{\mathsf{root},1}$ and $h_{\mathsf{root},2}$ to each client. It also performs the second step of the distributed key correctness protocol, which means sampling its random challenge e and forming and sending $\mathsf{com}_{i,\mathsf{chl}}$ to client i.

Distributed graph generation

3. Client i selects k neighbors by sampling randomly and without replacement k times from the set of n clients, and sends the resulting set $\mathsf{N}_\rightarrow(i)$ of outgoing neighbors to the server. Denote by $\mathsf{N}(i)$ all neighbors of client i (consisting of their outgoing edges and implicitly defined incoming edges).
4. The server sends $\mathsf{N}_\leftarrow(i), (j, \mathsf{pk}_{j,1}, \pi_{j,1}, \mathsf{pk}_{j,2}, \pi_{j,2})_{j \in \mathsf{N}(i)}$ to client i, where $\pi_{j,1}$ and $\pi_{j,2}$ are Merkle inclusion proofs with respect to roots $h_{\mathsf{root},1}$ and $h_{\mathsf{root},2}$.
5. Client i aborts if the server has sent more than $3k+k$ keys, if there is an index $j \in \mathsf{N}_\rightarrow(i)$ that is not reflected in the keys sent by the server, or if the Merkle inclusion proofs fail to verify.

Seed sharing

6. Each client i that has not dropped out performs the following:
   - Generates a random seed $\mathbf{seed}_i$.
   - Computes two sets of shares $H_i^{\mathbf{seed}} = \{h_{i,1}^{\mathbf{seed}}, \ldots, h_{i,k}^{\mathbf{seed}}\} = \mathtt{ShamirSS}(t, k, \mathbf{seed}_i)$ and $H_i^s = \{h_{i,1}^s, \ldots, h_{i,k}^s\} = \mathtt{ShamirSS}(t, k, sk_{i,1})$.
   - Sends to the server messages $m_j = (j, c_{i,j})$ for each $j \in \mathsf{N}_\rightarrow(i)$, where $c_{i,j} \leftarrow \mathsf{E}_{\mathsf{auth}}.\mathsf{Enc}(k_{i,j}, i \| j \| h_{i,j}^{\mathbf{seed}} \| h_{i,j}^s)$ for $k_{i,j} = \mathsf{KA.Agree}(\mathsf{sk}_{i,2}, \mathsf{pk}_{j,2})$.
7. If the server receives messages from fewer than $(1-\delta)n$ clients, it aborts. Otherwise, it sends all messages $(j, c_{i,j})$ to client j. Denote by $A_j \subseteq \mathsf{N}(j)$ the set of neighbors for whom client j received such a message.

Masking

8. Each client i that has not dropped out performs the following:
   - Computes a shared random seed $\mathbf{seed}_{i,j}$ as $\mathbf{seed}_{i,j} = \mathsf{KA.Agree}(sk_{i,1}, pk_{j,1})$.
   - Computes its *packed encrypted input* $\mathbf{y}_i = \mathsf{Encode}(\mathbf{sk}_i, \mathbf{Gx}_i)$ with key defined as $\mathbf{sk}_i = \mathbf{s}_i + \sum_{j \in A_i, j<i} \mathbf{s}_{ij} - \sum_{j \in A_1, i<j} \mathbf{s}_{ij}$ for $\mathbf{s}_{ij} = \mathsf{F}(\mathbf{seed}_{i,j})$, $\mathbf{s}_i = \mathsf{F}(\mathbf{seed}_i)$ (as in Equation 3).
   - Forms $\sigma_{i,j}^{\mathsf{incl}} \leftarrow \mathsf{Sig.Sign}(\mathsf{sk}_{i,2}, m_{i,j} = \text{“included”} \| i \| j)$ for all $j \in A_i$.
   - Forms commitments $\mathsf{com}_{\mathbf{sk},i} \leftarrow \mathsf{Commit}(\mathbf{sk}_i; r_i)$ and $\mathsf{com}_{\mathbf{x},i} \leftarrow \mathsf{Commit}(\mathbf{x}_i)$ to its key and input respectively.
   - Computes proofs $\pi^{\mathsf{Enc}(\mathbf{sk}_i, \mathbf{x}_i)}$, $\pi^{0 \le \mathbf{x}_i < t}$, and $\pi^{\mathsf{valid}(\mathbf{x}_i)}$ of encoding, smallness, and validity.
   - Performs the third step of the distributed key correctness protocol to form K_i.
   - Sends to the server $\mathbf{y}_i$, $(m_{i,j}, \sigma_{i,j}^{\mathsf{incl}})_j$, $\mathsf{com}_{\mathbf{sk},i}, \mathsf{com}_{\mathbf{x},i}, K_i, \pi^{\mathsf{Enc}(\mathbf{sk}_i, \mathbf{x}_i)}, \pi^{0 \le \mathbf{x}_i < t}, \pi^{\mathsf{valid}(\mathbf{x}_i)}$.

Dropout agreement and unmasking

9. The server collects packed encoded inputs for a determined time period. If it receives fewer than $(1-\delta)n$, it aborts. Otherwise, it defines a global set of dropouts $\mathcal{D}$ and a set of survivors $\mathcal{S}$. It then sends the messages and signatures $(m_{j,i}, \sigma_{j,i}^{\mathsf{incl}})$ to every client $i \in \mathcal{S}$, along with the sets $\mathcal{D}_i = \mathsf{N}(i) \cap \mathcal{D}$ (its incoming neighbors that are dropouts) and $\mathcal{S}_i = \mathsf{N}(i) \cap \mathcal{S}$ (its incoming neighbors that are not). It also sends the opening of the commitment for the distributed key correctness protocol (following the fourth step), containing its challenge e.
10. Each client i that has not dropped out performs the following:
    - Checks that $\mathcal{D}_i \cap \mathcal{S}_i = \emptyset$, that $\mathcal{S}_i, \mathcal{D}_i \subseteq \mathsf{N}(i) \cap A_i$, and that all signatures $\sigma_{j,i}^{\mathsf{incl}}$ are valid on message $m_{j,i}$ for all $j \in \mathcal{S}_i$, aborting if any of these checks fail.
    - Computes $\sigma_{i,j}^{\mathsf{ack}} \leftarrow \mathsf{Sign}(\mathsf{sk}_{i,2}, \text{“ack”} \| i \| j)$ for all $j \in \mathcal{S}_i$.
    - Performs the fifth step of the distributed key correctness protocol, which means forming values t_i and α_i.
    - Sends $(m_{i,j}, \sigma_{i,j}^{\mathsf{ack}})_j$ and t_i, α_i to the server.
11. The server aborts if it receives fewer than $(1-\delta)n$ responses. It verifies all received proofs $\pi^{\mathsf{Enc}(\mathbf{sk}_i, \mathbf{x}_i)}$, $\pi^{0 \le \mathbf{x}_i < t}$, $\pi^{\mathsf{valid}(\mathbf{x}_i)}$ and aborts if any of them fails. Otherwise it forwards all messages $(j, m_{i,j}, \sigma_{i,j}^{\mathsf{ack}})$ to client j.
12. Each remaining client verifies its received signatures using $\mathsf{pk}_{j,2}$, aborting if they fail to verify. Once a client receives p valid signatures from its neighbors, it sends $\{i, h_{i,j}^{\mathbf{seed}}\}_{j \in \mathcal{D}_i}$ and $\{i, h_{i,j}^s\}_{j \in \mathcal{S}_i}$ to the server, which it has obtained by decrypting the ciphertexts $c_{i,j}$ received in Step 6.
13. The server aborts if it receives fewer than $(1-\delta)n$ responses, and otherwise:
    - Collects, for each client $i \in \mathcal{D}$, the set of all received shares in $H_i^{\mathbf{seed}}$, and aborts if there are fewer than t. If not it recovers $\mathbf{seed}_i$ and $\mathbf{s}_i$ using the t shares received from the lowest client IDs.
    - Collects, for each client $i \in \mathcal{S}$, the set of all shares in H_i^s, and aborts if there are fewer than t. If not it recovers $sk_{i,1}$ and $\mathbf{s}_{ij}$ for all $j \in \mathsf{N}(i)$.
    - Computes a decryption key $\mathbf{sk} = \sum_{i \in \mathcal{S}} (\mathbf{s}_i + \sum_{j \in \mathcal{D}_i, j<i} \mathbf{s}_{ij} - \sum_{j \in \mathcal{D}_i, i<j} \mathbf{s}_{ij})$.
    - Using $\mathbf{sk}$, performs the final step of the distributed key correctness protocol and aborts if verification fails.
    - Outputs $\sum_{i \in \mathcal{S}} \mathbf{x}_i$ as $\mathbf{G}^{-1}(\mathsf{Decode}(\mathbf{sk}, \sum_{i \in A_2'} \mathbf{y}_i))$.

Figure 6: Maliciously secure SecAgg from homomorphic encodings with input verification.

Algorithm 2: `ShareSeeds` Robust to Malicious Clients.

Parties: Clients $1,\ldots,n$, and Server.
Public Parameters: Vector length ℓ, input domain $\mathbb{X}^\ell$, secret distribution χ_s, and PRG $\mathsf{F}\colon \{0,1\}^\lambda \mapsto \mathsf{supp}(\chi_s)^\ell$
Input: N/A
Output: $\mathsf{N}(i)$, $\{\mathbf{seed}_{i,j}\}_{j\in\mathsf{N}(i)}$, $\mathbf{seed}_i$, $\mathsf{shares}_{i,\mathcal{D}}$, and $\mathsf{shares}_{i,\mathcal{S}}$ (to Client i). $\mathsf{com}_{\mathbf{seed}_{i,j}}$, $\mathsf{com}_{\mathbf{seed}_i}$, $\mathsf{com}_{\mathbf{s}_{i,j}}$ and $\mathsf{com}_{\mathbf{s}_i}$ (to the Server). Note that if i and j are honest $\mathbf{s}_{i,j} = \mathsf{F}(\mathbf{seed}_{i,j})$ and $\mathbf{s}_i = \mathsf{F}(\mathbf{seed}_i)$.

Communication Graph Generation and Seed and Public Key Distribution

1. Client $i \in [n]$ generates $k/2+1$ seeds $\mathbf{seed}_{i,\cdot}$ and computes $g^{\mathbf{seed}_{i,\cdot}}$ and a commitment to $\mathsf{F}(\mathbf{seed}_{i,\cdot})$ with blinding also derived from $\mathbf{seed}_{i,\cdot}$. It then generates k key pairs $(\mathsf{sk}_{i,\cdot}, \mathsf{pk}_{i,\cdot})$. It then sends the public keys and all commitments to the server.
2. The server:
   - Generates a k-regular Harary graph G as in [5]. Let $\mathsf{N}^+(i)$ be the neighbors of i clockwise from i and $\mathsf{N}^-(i)$ be the other neighbors and $\mathsf{N}(i)$ the union of these.
   - Passes to each client $j \in \mathsf{N}^+(i)$ the commitment pair for one of the $\mathbf{seed}_{i,\cdot}$ and one of the public keys $\mathsf{pk}_{i,\cdot}$, which are henceforth denoted $\mathbf{seed}_{i,j}$ and $\mathsf{pk}_{i,j}$. The final seed is denoted $\mathbf{seed}_i$.
   - Passes to each client $j \in \mathsf{N}^-(i)$ a public key from i henceforth denoted $\mathsf{pk}_{i,j}$.
   - Informs client i which seed commitment was sent to whom.

Seed Exchange

3. Each client i computes, for each $j \in \mathsf{N}^+(i)$, $\mathsf{Enc}(k_{i,j}, \mathbf{seed}_{i,j})$ where $k_{i,j} = \mathsf{KA.Agree}(\mathsf{sk}_{i,j}, \mathsf{pk}_{j,i})$.
4. The server forwards all these messages to the corresponding clients j.

Seed Sharing

5. Each client i:
   - Decrypts the value of each $\mathbf{seed}_{j,i}$. Checks that this gives the correct $g^{\mathbf{seed}_{j,i}}$, if not it sends $\mathsf{sk}_{i,j}$ to the server.
   - Computes each $\mathsf{F}(\mathbf{seed}_{j,i})$ and checks they give the right commitments, if not it sends $\mathsf{sk}_{i,j}$ to the server.
   - Computes shares: $\{h_{i,1},\ldots,h_{i,k}\} = \mathtt{ShamirSS}(t,k,\mathbf{seed}_i)$,
   - Computes shares $(h_{i,j,m})_{m\in\mathsf{N}(i)}$ of each $\mathbf{seed}_{i,j}$ along with deterministic coefficient commitments as per Algorithm 3.
   - Computes shares $(\tilde{h}_{i,j,m})_{m\in\mathsf{N}(i)}$ of $\mathbf{seed}_{j,i}$ with deterministic commitments, for each j that sent the correct $\mathbf{seed}_{j,i}$.
   - For each $m \in \mathsf{N}(i)$ sends $c_{i,m} = \mathsf{Enc}(k_{i,m}, (\mathbf{seed}_{i,m}, (h_{i,j,m})_{j\in\mathsf{N}^+(i)}, (\tilde{h}_{i,j,m})_{j\in\mathsf{N}^-(i)}, (h_{i,m})_{\mathsf{sgn}(\mathsf{sk}_i)}))$ to the server.
   - Sends the server the deterministic coefficient commitments.
6. The server aborts if fewer than $(1-\delta)n$ clients remain otherwise it:
   - Forwards $(c_{i,m})_{i\in\mathsf{N}(m)}$ to client m.
   - Checks that the coefficient commitments match the original seed commitments and computes $g^{h_{i,j,m}}$ and $g^{\tilde{h}_{i,j,m}}$ and sends it to client m

Bad Message Resolution

7. Client m opens each $c_{i,m}$ and checks that the $h_{i,j,m}$ gives the correct $g^{h_{i,j,m}}$ and that the signature on $h_{i,m}$ is valid. If either check fails it sends $\mathsf{sk}_{m,i}$ to the server.
8. The server uses each $\mathsf{sk}_{m,i}$ it has received (which it checks match the $\mathsf{pk}_{m,i}$ from earlier) to check whether i sent a bad $\mathbf{seed}_{i,m}$ or $c_{i,m}$ and if so it drops i, otherwise it drops m. It informs each client i of the set N_i of their remaining neighbors.

Algorithm 3: Sharing of seeds with verification (Feldman's scheme).

Parties: Client i, their neighbours $m \in \mathsf{N}(i)$ (including j) and Server.
Public Parameters: t
Input: $\mathbf{seed}_{i,j}$ from client i.
Output: A sharing of $\mathbf{seed}_{i,j}$ amongst the clients in $\mathsf{N}(i)$.

1. Client i, chooses a random degree $t-1$ polynomial P subject to $P(0) = a_0 = \mathbf{seed}_{i,j}$ given by $P(x) = a_0 + a_1 x + ... + a_{t-1} x^{t-1}$.
2. Client i then evaluates $P(m)$ for every $m \in \mathsf{N}(i)$, encrypts it using a shared key with party m and sends it to the server.
3. Client i also computes g^{a_l} for each $l \in [0, ..., t-1]$ and sends these to the server.
4. For each $m \in \mathsf{N}(i)$ the server computes $g^{P(m)}$ from the g^{a_l} and sends it along with the encryption of $P(m)$ to client m.
5. Each client m decrypts the value of $P(m)$ and from it computes $g^{P(m)}$ checking that it matches the $g^{P(m)}$ provided by the server. If it doesn't match, client m sends $\mathsf{sk}^1_{m,i}$ to the server so they can confirm this, otherwise they report that it matched.
6. If any client m reports that there was a mismatch the server uses $\mathsf{sk}^1_{m,i}$ to check this. If there is indeed a mismatch the server labels client i dishonest. In Algorithm 1 client i is dropped at this point.
7. The server informs all clients $m \in \mathsf{N}(i)$ whether i has been labelled dishonest or not.
8. If i is dishonest the other clients abort else they output their share $P(m)$.

Algorithm 4: `RecoverAggKey` Robust to Malicious Clients.

Parties: Clients $1, \ldots, n$, and Server.
Public Parameters: Vector length ℓ, input domain $\mathbb{X}^\ell$, secret distribution χ_s, and PRG $\mathsf{F}\colon \{0,1\}^\lambda \mapsto \mathsf{supp}(\chi_s)^\ell$
Input: A set S of remaining clients, $\mathsf{N}(i)$, $\{\mathbf{seed}_{i,j}\}_{j \in \mathsf{N}(i)}$, $\mathbf{seed}_i$, $\mathsf{shares}_{i,\mathcal{D}}$, and $\mathsf{shares}_{i,\mathcal{S}}$ (to Client i). $\mathsf{com}_{\mathbf{seed}_{i,j}}$, $\mathsf{com}_{\mathbf{seed}_i}$, $\mathsf{com}_{\mathbf{s}_{i,j}}$ and $\mathsf{com}_{\mathbf{s}_i}$ (to the Server). Note that if i and j are honest $\mathbf{s}_{i,j} = \mathsf{F}(\mathbf{seed}_{i,j})$ and $\mathbf{s}_i = \mathsf{F}(\mathbf{seed}_i)$.
Output: A set R of clients (including all honest clients that didn't dropout) and the aggregate key $\mathbf{sk} = \sum_{i \in R} \mathbf{sk}_i$.

Removing Some Masks from Clients

1. Each client m sends $\{(h_{i,j,m}, \tilde{h}_{i,j,m})\}_{i \in N_m \setminus F_m, j \in \mathsf{N}(i)} \cup \{(i, (h_{i,m})_{\mathsf{sgn}(\mathsf{sk}^2_i)})\}_{i \in F_m}$, obtained by decrypting $c_{i,j}$ from Step 5.
2. The server
   - For each client $i \in S$, if possible using the shares provided with good signatures, recovers $\mathbf{seed}_i$ and $\mathbf{s}_i$, and checks them against their commitments from earlier.
   - Drop all caught clients, remove them from S and request their edge mask sharing.
   - If no client cheated the server tells the clients they are done and skips to Step 4.

Loop to Remove Edge Masks

3. Client m provides any requested $\{(h_{i,j,m}, \tilde{h}_{i,j,m})\}_{j \in \mathsf{N}(i)}$
4. The server then:
   - If there are sufficient shares, recovers $\mathbf{seed}_{i,j}$, $\mathbf{seed}_{j,i}$ and $\mathbf{s}_{ij}$ for all $j \in \mathsf{N}(i)$.
   - Computes $\mathbf{sk}_i$ as client i did in Step 2 of Algorithm 1.
   - For all $\mathbf{sk}_i$ that don't match their commitments: drop client i, remove i from F and request their edge mask secrets.
   - If any new requests were sent return to Step 3.
   - Outputs $(S, \sum_{i \in S} \mathsf{sk}_i)$.

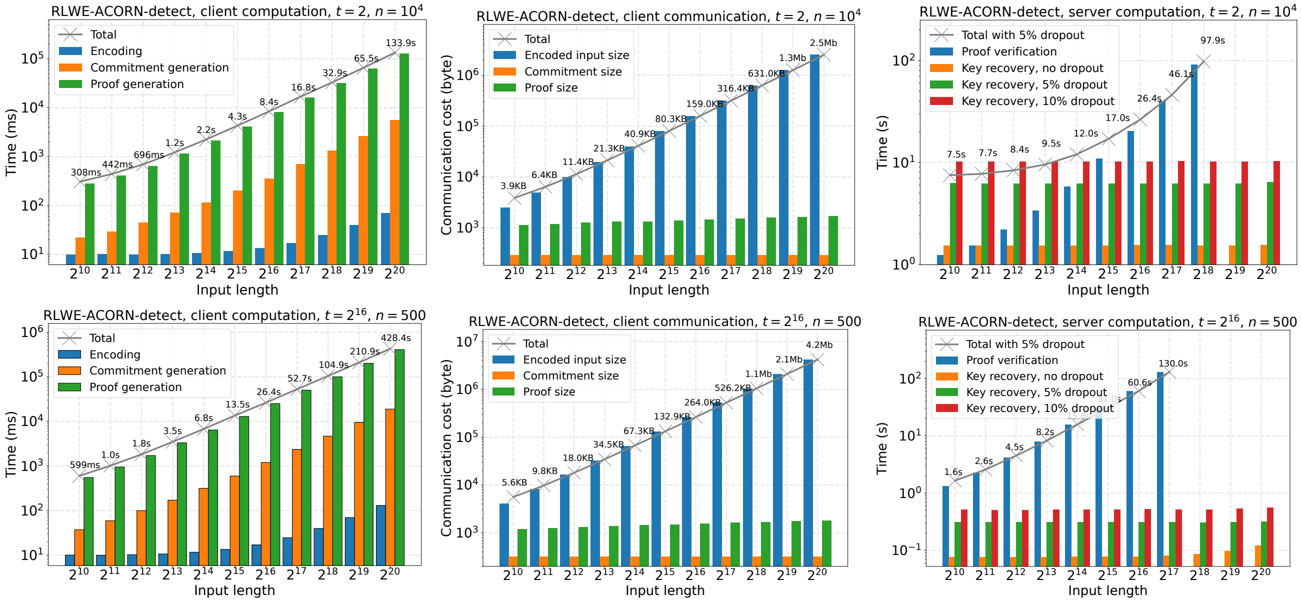

Figure 7: Benchmarks for RLWE variant of ACORN-detect, on a logarithmic scale, for the FA use case where $t = 2$ and $n = 10000$ (on the top row) and the FL use case where $t = 2^{16}$ and $n = 500$ (on the bottom). In both cases we measured (1) the client runtime (on the left); (2) the client communication cost (in the middle); and (3) the server runtime (on the right), considering amortized batch proof verification and key reconstruction for three different levels of dropouts. Our proof verification experiments for input length $\ell \geq 2^{19}$ for FA and $\ell \geq 2^{18}$ for FL ran out of memory, and hence these bars are missing in corresponding diagrams. The encoding experiments were repeated 100 times for each parameter set, and the other experiments were repeated for at least 5 seconds or 10 iterations; the average running times are estimated using the bootstrap resampling method with 95% confidence level.