SCIF: Privacy-Preserving Statistics Collection with Input Validation and Full Security

Jianan Su*, Laasya Bangalore†, Harel Berger*, Jason Yi*, Alivia Castor*,
Micah Sherr*, Muthuramakrishnan Venkitasubramaniam*
*Georgetown University
†SandboxAQ

Abstract—**Secure aggregation is the distributed task of securely computing a sum of values (or a vector of values) held by a set of parties, revealing only the output (i.e., the sum) in the computation. Existing protocols, such as Prio (NDSI'17), Prio+ (SCN'22), Elsa (S&P'23), and Whisper (S&P'24), support secure aggregation with input validation to ensure inputs belong to a specified domain. However, when malicious servers are present, these protocols primarily guarantee privacy but not input validity. Also, malicious server(s) can cause the protocol to abort. We introduce SCIF, a novel multi-server secure aggregation protocol with input validation, that remains secure even in the presence of malicious actors, provided fewer than one-third of the servers are malicious. Our protocol overcomes previous limitations by providing two key properties: (1) *guaranteed output delivery*, ensuring malicious parties cannot prevent the protocol from completing, and (2) *guaranteed input inclusion*, ensuring no malicious party can prevent an honest party's input from being included in the computation. Together, these guarantees provide strong resilience against denial-of-service attacks. Moreover, SCIF offers these guarantees without increasing client costs over Prio and keeps server costs moderate. We present a robust end-to-end implementation of SCIF and demonstrate the ease with which it can be instrumented by integrating it in a simulated Tor network for privacy-preserving measurement.**

1. Introduction

Secure multiparty computation (MPC) allows a group of entities to securely compute an arbitrary function operating jointly over their individual inputs with the guarantee that nothing beyond the output of the function is revealed. Secure summation or aggregation is one of the simplest functions to compute via MPC, yet it is already powerful enough to facilitate computation of other aggregate statistics including MEAN, STDDEV, MAX, MIN, (Boolean) AND, OR, HISTOGRAMS, and more. A robust MPC implementation can enable privacy-preserving collection of user travel data for navigation apps [1], vitals in fitness trackers [2], various statistics from web browsers, and, more generally, federated learning [3], [4].

†*Work done while the author was affiliated with Georgetown University.*

In its simplest form, an MPC protocol for aggregation proceeds as follows: Input parties share their input with compute parties (which could be external servers or the input parties themselves) using some linear secret-sharing scheme. The compute parties can compute the sum of secret shares locally and transmit the results to the output party who can reconstruct the result. This simple protocol will guarantee security against semi-honest corruption of all-but-one clients and up to a threshold of servers (implied by the underlying secret-sharing scheme). Standard techniques can be used to boost the security to withstand malicious adversaries. However, there is a simple attack that can disrupt the protocol, namely that the parties can give arbitrary values as inputs. This can completely invalidate the result of the computation (for the underlying target application).

Toward addressing this drawback, Corrigan-Gibbs and Boneh designed Prio, one of the first secure aggregation systems with input validation [5], which has been deployed in various real-world scenarios by organizations such as Apple, Google, Internet Services Research Group (ISRG), and Mozilla [6]. In the Prio model, the task of secure aggregation is delegated to an (external) set of server nodes where correctness and privacy are guaranteed against a semi-honest adversary that corrupts up to all but one of the servers. A crucial ingredient, developed in the Prio work, is a secret-shared (i.e., distributed) non-interactive proof which allows each client to certify its input to the servers that only hold secret-shares of the input while preserving privacy. A non-interactive proof allows the clients to simultaneously share their input and the proof to all servers in a single message.

The main drawback of the Prio system is that their security holds only against semi-honest corruption of the servers, and moreover, if even one of the servers crash, all data is lost, and the computation aborts.

In this work, our goal is to design and implement a concretely-efficient secure aggregation scheme that meets the following criteria:

(a) Security in the presence of malicious adversaries: The security properties and features of the system should hold even in the presence of a malicious (or active) adversary that can arbitrarily deviate from the protocol.

(b) Guaranteed output delivery: An adversary that actively attacks the system should not be able to prevent an honest party from receiving the output. This property is important to prevent denial-of-service attacks. Properties (a) and (b) together are sometimes referred to as *full security*.

(c) Guaranteed input inclusion: The inputs of all honest parties should be included in the computation even if the adversary tries to actively attack the system.[1]

(d) Unreliable network on client side: Input clients are required to participate in at most one round of communication.

(e) Input validation: In a robust secure aggregation system, corrupted clients should be prevented from giving "artificial" inputs. If the underlying domain D can be captured via a predicate $P : \mathbb{F}^m \to \{0, 1\}$ which outputs 1 on all inputs $x \in D$ and 0 otherwise, then a simple form of robustness allows aggregation of inputs if and only if the predicate on its input returns 1 [7], [8], [9].

(f) End-to-end implementation: The full system should be realizable in an end-to-end implementation. The implementation should be deployable on commodity hardware, be sufficiently scalable to support a large number of clients and parallel statistics collection operations, and be sufficiently modular to allow easy integration with existing software.

Secure aggregation has been broadly pursued in two popular settings: (1) in works such as [5], [10], [11] computation is delegated to an external set of servers, and (2) following the line of works starting from Bonawitz et al. [12], [8], [13] where the input parties also play the role of compute parties and are connected in a star network topology with the center of the star designated as the output party. The former works do not guarantee output delivery and the latter do not guarantee input inclusion and/or require client participation in multiple rounds.

The current state of affairs for secure aggregation is that there is no system that sufficiently meets all the desiderata.

1.1. Main Result and Techniques

In this work, we consider a model that is a slight variant of the Prio model and design a secure aggregation system with input validation that meets all our requirements. In slightly more detail, we consider a model where security is maintained even if an attacker simultaneously corrupts all but one of the (input) clients, at most ⅓ of the (compute) servers and the output party. We showcase our system as lightweight via a robust end-to-end implementation.

On a high level, our protocol can be modularly described via the Verifiable Relation Sharing (VRS) functionality as observed in a recent work [14]. Introduced in the work by Applebaum et al. [15], VRS allows a dealer to share a secret with n servers with the guarantee that all (honest) servers either discard the dealer or output valid shares to a secret that satisfies a predefined relation R. Given such a primitive for a linear secret sharing scheme, a robust secure aggregation protocol w.r.t. a predicate P meeting our desiderata can be constructed as follows: (1) Each client acting as a dealer uses a VRS scheme to secret share its input by relying on the predicate to instantiate the relation R. Then, all (honest) servers simply add the secret shares of clients (that were not discarded at the end of the VRS instance) and send the aggregate to the output party that reconstructs the secret. For output to be correctly reconstructed even in the presence of malicious servers, we will need the underlying secret-sharing scheme to have some error reconstruction property.

We design a protocol to realize the VRS functionality by reducing it to the distributed commit and prove functionality (dCP) protocol following the paradigm introduced in recent work [14]. In a dCP protocol, a prover holding a secret witness w wishes to convince n verifiers each with individual inputs $x_1, \ldots, x_n$ that n relations $\mathcal{R}_1, \ldots, \mathcal{R}_n$ hold respectively (i.e., $\mathcal{R}_i(x_i, w)$ holds for each i) w.r.t. to the same witness w. This primitive will be useful for the prover to first give secret shares of its input to n servers and then convince them that the secret encoded in the secret shares satisfies a (certain robustness) predicate P. As an independent technical contribution, we provide a concretely efficient instantiation of the dCP functionality using the Ligero [16] sublinear zero-knowledge argument system, which in turn yields an implementation of a VRS (and a VSS i.e., Verifiable Secret Sharing) protocol for any linear secret-sharing scheme as described above.

Comparison with [14]. As mentioned above, in this work the authors designed a secure aggregation protocol using the VRS and dCP functionalities. Their protocol satisfied (a)-(e) of the desiderata and additionally offered the stronger guarantee of differential privacy. However, the focus was establishing a feasibility result using ad hoc techniques for instantiating the dCP functionality and did not evaluate the concrete efficiency via an implementation.

1.2. Implementation

A core contribution of this paper is the development, evaluation, and concurrent release of privacy-preserving Statistics Collection with Input validation with Full security (SCIF), a ready-to-use open source distributed statistics collection platform. SCIF supports secure and private summation on a large number of devices. To our knowledge, it is the first implemented privacy-preserving statistics collection system that offers protection against malicious participants (both client and server), supports input validation, and guarantees output delivery and input inclusion. We evaluate SCIF using a real-world deployment with geographically distributed clients and demonstrate that SCIF's computational and networking costs are minimal: a measurement involving 500 clients, where each client submits 10^4 inputs, requires

[1] Typically, guaranteed output delivery implies guaranteed input inclusion. However, in scenarios where the input parties can join in a permissionless way and the adversary controls who can join (as is the case in the single server setting described later), guaranteed input inclusion does not hold.

less than 10 seconds of processing time on each client and less than 40 seconds on the server, while consuming slightly more than 2 MB of network bandwidth per client and 200 MB between the servers. Even when up to 40% of the clients behave maliciously, which triggers a share correction operation, the execution time is less than 90 seconds.

SCIF is written in Go with scalability and ease-of-use as primary design goals. It is compatible with a large number of platforms (our testing used OSX and Linux), and is easily incorporated into existing systems to enable secure statistics collection. As a proof-of-concept, we integrate SCIF with a private deployment of Tor [17]—a popular anonymity service [18]—and show how SCIF can be used to safely learn information about the behavior of Tor nodes.

SCIF is available for download at https://anonymous.4open.science/r/scif/ (hosted anonymously).

2. Related Work

Single-server setting. The single server setting has been extensively studied in prior works [12], [19]. Initial research developed secure aggregation protocols designed to protect the privacy of client inputs while tolerating potential client dropouts [12], [19], [20], [21], [22]. However, they fell short in ensuring the correctness of the aggregate when faced with malicious clients who can bias the results by sending malformed inputs. To mitigate such attacks in statistical contexts, methods include ensuring well-formed inputs, such as restricting inputs to 0 or 1 in counts, adding only a value of 1 per histogram bucket, and limiting contributions to multiple histogram buckets. In the federated learning context, bounding the norms of inputs (e.g., L_2 and L_{inf} norms) has proven effective against such attacks. Prior works [8], [23], [13], [11] implemented these defenses using *input validation* mechanisms[2] to verify that clients submitted valid inputs, which utilized zero-knowledge proofs. These protocols offer varying levels of security. Some simply detect malicious clients and abort the protocol upon detection (security with abort) [8], [13]. Others go further by identifying and excluding misbehaving clients from the aggregate (full security) [23], [13], [11]. Although these protocols can handle malicious adversaries, they often require multiple rounds of interaction between clients and the server, which is undesirable in settings with unreliable clients. Our focus is on achieving full security while limiting client participation to a single round. This means clients are required to send only a single communication to the server, after which clients' online presence is not necessary.

Many more secure aggregation methods in the single-server setting have been studied and utilized in federated learning. For a comprehensive overview of this literature, refer to the survey by Mansouri et al. [24].

Multi-server setting. In the multi-server setting, considerable research [5], [10], [25] has emerged, employing multi-party computation techniques for computing aggregate statistics. In this setting, clients delegate computation tasks to a small set of servers. Different threat models exist, depending on whether adversaries corrupt parties in a semi-honest or malicious manner, and whether there exists a dishonest or honest majority among the servers.

The seminal work by Corrigan-Gibbs and Boneh [5] introduces an efficient secure aggregation system called Prio in the multi-server model where the adversary can semi-honestly corrupt all but one of the servers and maliciously corrupt the clients. Subsequent research builds upon this foundation. Notably, the works of [5], [10], [25], [26] enhance the original protocol. Prio+, a system introduced by [10], improves upon the client computation costs over Prio by employing boolean secret sharing for input validation, rather than relying solely on zero-knowledge proofs. Yet another system, Elsa, proposed by [25], further improves both Prio and Prio+ in a setting where there are two non-colluding servers and achieves privacy even when one server is maliciously compromised. Developed by [26], the Whisper system aims to scale to millions of clients in a similar two servers setting by improving upon the server-to-server communication and server storage to be sublinear in the number of clients (albeit with a slight increase in client-to-server communication).

On the upside, the aforementioned works in the multi-server setting can tolerate an adversary corrupting any number of servers (i.e., a dishonest majority among the servers). However, they have a limitation: they cannot guarantee output delivery if even a single server is maliciously corrupted. In contrast, our focus is on achieving guaranteed output delivery in a different threat model, where an adversary can only corrupt a minority of the servers maliciously (i.e., an honest majority among the servers).

Additionally, we aim to ensure guaranteed input inclusion, even in the presence of malicious corruption of clients and a minority of the servers. Contrast this with the single-server setting, where the central server can decide which set of clients to include in the final aggregate, (as long as the set meets a minimum size requirement) and could potentially discard many honest clients' inputs. Our work strives to achieve both guaranteed input inclusion and guaranteed output delivery while ensuring that clients' participation remains limited to a single round.

A closely related work is the Flag secure aggregation system by Bangalore et al. [11] who show how to achieve input validation and guaranteed output delivery in the client-server setting where all parties are connected in a star topology with the output party. They demonstrate efficiency by benchmarking components of their system. However, similar to the single-server setting, their work fails to guarantee input inclusion as the output party can censor input clients.

3. Preliminaries

Basic notation. We denote the set of clients by $\mathcal{U} = \{\mathcal{U}_1, \ldots, \mathcal{U}_{\mathsf{n_c}}\}$, servers by $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_{\mathsf{n_s}}\}$ and the output party by $\mathcal{O}$. For simplicity, we assume that each client $\mathcal{U}_i$ is

[2] This is also referred to as input certification, Byzantine resilience, or robustness in prior work.

also identified by a unique integer in $[\mathsf{n_c}]$ and each server $\mathcal{S}_j$ is identified by a unique integer in $[\mathsf{n_s}]$. Although an integer may be associated with a client and a server, we usually specify the type of entity; if not it can be easily inferred from the letters used to specify the identity. $\mathcal{BC}(\cdot)$ denotes the communication over a broadcast channel.

3.1. Our Model

We consider a synchronous network model involving $\mathsf{n_c}$ clients, $\mathsf{n_s}$ servers, and an output party $\mathcal{O}$. The network assumes point-to-point secure and authenticated channels between the following parties: (i) client to server, (ii) server to server, and (iii) server to output party. This can be facilitated with a public-key infrastructure and standard cryptographic primitives such as authenticated encryption. All parties know the identities (i.e., public keys) of the other parties they need to communicate with.

Threat model. Both clients and servers can be malicious at any point in our protocol, meaning they can arbitrarily deviate from the protocol. We assume at most $\mathsf{t_c}$ malicious clients and $\mathsf{t_s}$ malicious servers. More precisely, the adversary can maliciously corrupt all but one of the clients i.e., $\mathsf{t_c} < \mathsf{n_c}$ and maliciously corrupt up to a threshold of $\mathsf{t_s} < \mathsf{n_s}/3$ servers. Additionally, the adversary can maliciously corrupt the output party.

3.2. Secure Aggregation

We consider the problem of adding vectors over integers captured by a large enough field $\mathbb{F}$. In our model, we have $\mathsf{n_c}$ clients, $\mathsf{n_s}$ servers, and an output party. Each client $\mathcal{U}_i \in \mathcal{U}$ has a vector $X_i \in \mathbb{F}^{\mathsf{d}}$ of size d. The servers do not have any inputs, and the output party receives the final aggregate.

Our MPC protocol aims to implement the ideal functionality $\mathcal{F}_{Agg}$ for secure aggregation that is given in Figure 1. This functionality is parameterized by $\mathsf{n_c}$, the (maximum) number of clients, and d, the length of input vector. $\mathcal{F}_{Agg}$ receives the inputs from the clients and stores these values. The functionality will aggregate the inputs of the clients that satisfy the predicate $P(\cdot)$ and sends the result to the output party.

3.3. Replicated Secret Sharing Scheme

We adopt the notation from [27] to describe the Replicated Secret Sharing (RSS) Scheme. RSS, introduced by [28], with threshold t, is defined by the following procedures[3]. We let $\mathbb{R}$ be any finite ring, $\lambda = \binom{n}{t}$ and denote by $T_1, \ldots, T_\lambda \subset [n]$ all subsets of indices of size $n-t$.

- $\mathsf{Enc}(x)$**:** To encode a secret x with threshold t, first generate λ random $x_{T_1}, \ldots, x_{T_\lambda} \in \mathbb{R}$ under the constraint that $x = x_{T_1} + \ldots + x_{T_\lambda}$. The share sh_i is a tuple consisting of all x_{T_j} such that $i \in T_j$. We denote the output of the encoding by $\mathsf{Share} = (\mathsf{sh}_1, \ldots, \mathsf{sh}_n)$. Sometimes, the randomness used for Enc, say r, is explicitly specified as $\mathsf{Enc}(x; r)$.
- $\mathsf{Dec}(\mathsf{Share})$**:** For each subset T holding a value x_T, obtain all the values for x_T repeated across the different share tuple of the encoding Share; if there exists different values for x_T, then set the majority value to be x_T. Finally, P_i sets $x = \sum_{T \subseteq [n]:|T|=n-t} x_T$.

For an encoding of size n and threshold t, there are $\binom{n}{t}$ distributed shares. Each share has $\binom{n-1}{t}$ values. We can check that a encoding is consistent, by ensuring the equality of the joint values in each pair of shares. Note that each pair of shares can check pairwise consistency of an arbitrarily large number of sharings by comparing a hash of the string consisting of all their joint values. Refer to [27] for more details.

Non-interactive Generation of Random RSS Sharings. Let $\mathcal{F} = F_k \mid k \in 0, 1^\kappa, F_k : 0, 1^\kappa \to \mathbb{F}$ be a family of pseudo-random functions. Let the random values associated with the RSS encoding of k are $k_{T_1}, \ldots, k_{T_\lambda}$. To generate the ℓ-th random encoding, compute the value r_T^ℓ as $r_T^\ell = F_{k_T}(\ell)$. The newly generated random sharing, denoted by $(\mathsf{rsh}_1^\ell, \ldots, \mathsf{rsh}_n^\ell)$, is such that rsh_i^ℓ consists of all $r_{T_j}^\ell$ where $i \in T_j$, as defined in $\mathsf{Enc}(\cdot)$.

4. Secure Aggregation with Input Validation

In this section, we introduce our secure aggregation protocol, which incorporates input validation and full security.

We start with a high-level overview of the protocol. Similar to standard MPC techniques, the basic structure consists of two main phases: input sharing and output reconstruction. In the input sharing phase, each client secret-shares its input among the servers using a linear secret sharing scheme (which we instantiate with a replicated secret sharing scheme). During the output reconstruction phase, the servers aggregate the shares received from all clients and send the aggregated shares to the output party, who then reconstructs the final aggregate. By setting the threshold of the secret-sharing scheme to $\mathsf{t_s} < \mathsf{n_s}/3$, our protocol can tolerate the malicious corruption of up to $\mathsf{t_s}$ servers, ensuring guaranteed output delivery.

Input validation via zero-knowledge proofs. To ensure that clients submit valid inputs, our protocol employs zero-knowledge proofs. Clients must demonstrate to each server that their input is well-formed and that the input shares distributed among the servers are consistent with this input. This is achieved through a distributed Commit and Prove (dCP) functionality, detailed in Section 4.1. The dCP functionality involves two phases. During the Commit phase, the client commits to its input. Subsequently, during the Prove phase, the client proves to each server that the committed input is well-formed with respect to some predicate $P(\cdot)$ and consistent with the input shares distributed among the servers. Each server then accepts or rejects the proof and outputs the shares if the proof is

[3] Rather than using the standard share and reconstruct terminology, we define RSS using slightly different terminology: encode $\mathsf{Enc}(\cdot)$ and decode $\mathsf{Dec}(\cdot)$. This ensures consistency with the coding scheme notation used in our dCP.

Functionality $\mathcal{F}_{\mathsf{Agg}}$

The functionality $\mathcal{F}_{\mathsf{Agg}}$ communicates with the set of clients $\mathcal{U} = \{\mathcal{U}_1, \ldots, \mathcal{U}_{\mathsf{n_c}}\}$, an output party $\mathcal{O}$ and an adversary $\mathcal{A}$. It is parameterized by $P : \mathbb{F}^d \to \{0, 1\}$, $\mathsf{n_c}$ and d, where P is the predicate used for certify the inputs, $\mathsf{n_c}$ is the number of clients and d denotes the size of client's input vector.

1) Upon receiving input $(\text{"Input"}, \mathsf{sid}, \mathcal{U}_i, X_i)$ from some new client $\mathcal{U}_i \in \mathcal{U}$ where $X_i \in \mathbb{F}^{\mathsf{d}}$, store the client's input X_i.
2) Upon receiving $(\text{"Output"}, \mathcal{O})$ from the output party $\mathcal{O}$, proceed as follows:
   - Compute the aggregate $\mathsf{Y}_{agg} = \sum_{\mathcal{U}_i \in \mathcal{U}} P(X_i) \cdot X_i$ (note that this is equivalent to aggregating only inputs of clients that satisfy the predicate P).
   - Send $(\text{"Output"}, \mathsf{sid}, \mathsf{Y}_{agg})$ to the output party $\mathcal{O}$ and halt.

Figure 1. Ideal Functionality for Secure Aggregation with Input Certification

accepted. After the dCP, servers need to agree on a set of valid clients whose inputs will be included in the final aggregate. Servers broadcast complaints against clients for whom proof verification failed. Based on the number of complaints, clients are either discarded if the complaints are too high or included in the valid set otherwise. We will rely on a zk-SNARK so that only a single message is required to generate a proof. Our dCP protocol is inspired by the work of Zhang et al. [29] who rely a similar primitive towards designing a VSS protocol.

Ensuring all honest servers possess valid shares. For the final aggregate to be reconstructible, all honest servers must have valid input shares from clients in the valid set. If any honest server lacks valid input shares, it cannot compute its aggregate share. This could prevent the output party from reconstructing the aggregate if the number of missing aggregate shares exceeds the reconstruction threshold. To address this, we employ a verifiable relation sharing (VRS) scheme that enhances the dCP to ensure that all honest servers receive valid shares, guaranteeing output delivery.

We begin by discussing our dCP and VRS constructions in Sections 4.1 and 4.2 respectively and then show how to integrate them into our secure aggregation protocol with input validation.

4.1. Distributed Commit-and-Prove (dCP)

We construct a dCP protocol involving a prover and n verifiers. This protocol serves as a foundational component in validating the inputs with respect to a predicate $P(\cdot)$. In our scenario the prover secret-shares its input and demonstrates that both the input and its shares, distributed among the verifiers, are "well-formed". To elaborate, the prover secret-shares its input x and sends the share sh_j to verifier $\mathcal{V}_j$, where $(\mathsf{sh}_1, \ldots, \mathsf{sh}_n) \leftarrow \mathsf{Enc}(x; r_{\mathsf{dcp}})$ and r_{dcp} is the randomness used by Enc. Subsequently, the prover needs to prove the following two properties to each verifiesr $\mathcal{V}_j \in \mathcal{V}$.

1) $P(x) = 1$
2) $\mathsf{sh}_j = [\mathsf{Enc}(x; r_{\mathsf{dcp}})]_j$, indicating that sh_j is a valid share with respect to input x and randomness r_{dcp}.

We now construct the dCP protocol based on the Ligero zk-SNARK proof system. First, we first establish terminology. the circuit and the associated distributed relations which will be used for our dCP construction. We repeat this verbatim from [14].

Notation 4.1 (Circuit Description given a predicate P). We consider a circuit C that takes $\mathsf{w} = (x, r_{\mathsf{dcp}})$ as input and yields output $(\mathbf{out}_1, \ldots, \mathbf{out}_n)$ such that if $P(x) = 1$, then $\mathbf{out}_j = \mathsf{sh}_j$, otherwise $\mathbf{out}_j = \perp$ for all $j \in [n]$ where $(\mathsf{sh}_1, \ldots, \mathsf{sh}_n) \leftarrow \mathsf{Enc}(\mathsf{w})$.

We employ the following notation to specify a set of relations $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ via a circuit C.

Notation 4.2 (Determination of Distributed Relations from Circuits). Consider a circuit $\mathsf{C} : \mathbb{F}^{\mathsf{in}} \to \mathbb{F}^{\mathsf{out}}$ that takes an input w and produces $(\mathbf{out}_1, \ldots, \mathbf{out}_n)$, where $\mathbf{out}_j = [C(\mathsf{w})]_j$ represents the jth output of the circuit for input w. A pair $(\mathbf{out}_j, \mathsf{w})$ *belongs to* $\mathcal{R}_j$ if the evaluation by $\mathsf{C}(\mathsf{w})$ does not result in $(\perp, \ldots, \perp)$. Additionally, we assert that the distributed relation $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ *is determined by* a circuit C if $(\mathbf{out}_j, \mathsf{w})$ *belongs to* $\mathcal{R}_j$ for each $j \in [n]$.

With the terminology established, our focus now is on constructing a dCP protocol for a set of relations determined by a circuit C as described previously. This circuit C takes input $\mathsf{w} = (x, r_{\mathsf{dcp}})$ and verifies if w satisfies the predicate $P(\cdot)$. If $P(x) = 1$, the circuit internally computes and outputs the shares $(\mathbf{out}_1, \ldots, \mathbf{out}_n)$, otherwise, it outputs $(\perp, \ldots, \perp)$.

Initially, let us consider how a prover with input w can prove to a single verifier, denoted as $\mathcal{V}_j$, that the two aforementioned properties hold for w. The prover can execute the Ligero proof generation outlined in Appendix B with respect to circuit C and input w. All constraints imposed by the circuit are enforced via code, linear, and quadratic tests, akin to the standard Ligero proof system. Additionally, the column consistency check aligns with Ligero's protocol. Note that randomness for tests and opening columns is generated via Fiat-Shamir transformation.

However, this process is not sufficient, as we still need to ensure consistency between:

- the share sh_j received by verifier $\mathcal{V}_j$
- the output $\mathbf{out}_j = [C(\mathsf{w})]_j$ which is included in the extended witness

Formally, we need to ensure that $\mathsf{sh}_j = [C(\mathsf{w})]_j$. This equality between the share and portion of the extended witness can be checked via an additional linear test to confirm

Protocol Π_{dCP}

This protocol involves a prover $\mathcal{P}$ and n verifiers $\mathcal{V} = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$. It is parameterized by n relations $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ which are determined by a circuit $\mathsf{C} : \mathbb{F}^{\mathsf{n}_1} \to \mathbb{F}^{\mathsf{n}_2}$. C takes as input w and outputs $(\mathbf{out}_1, \ldots, \mathbf{out}_n)$, where $\mathbf{out}_j = [C(\mathsf{w})]_j$ is the jth output of the circuit given input w. Recalling Definition 4.2, a pair $(\mathbf{out}_j, \mathsf{w})$ *belongs to* $\mathcal{R}_j$ if the evaluation by $\mathsf{C}(\mathsf{w})$ does not output $(\perp, \ldots, \perp)$. Refer to Appendix B for a self-contained description of Ligero proof system.

Input & Output The prover has input $w \in \mathbb{F}^{\mathsf{n}_1}$ and the verifiers have no inputs. Each verifier $\mathcal{V}_j \in \mathcal{V}$ outputs $(\mathbf{out}_j, \mathsf{accept})$ if $(\mathbf{out}_j, \mathsf{w}) \in \mathcal{R}_j$; otherwise outputs reject.

Commit Phase: The prover $\mathcal{P}$ does the following.

1) Compute the commitment to the proof oracle following the procedure outlined in Appendix B. Specifically, for a given circuit C with input w, $\mathcal{P}$ performs the following steps:
   - Compute the extended witness $\mathsf{w}_{\mathsf{ext}}$.
   - Encode the extended witness row by row using the Reed-Solomon scheme.
   - Commit to the encoded extended witness column-wise using a Merkle-tree-based hash.
   - The resulting Merkle root, denoted by com, serves as the commitment to the proof oracle.
2) Broadcast $(\mathsf{Commit}, \mathsf{sid}, \mathcal{P}, \mathsf{com})$ to all the verifiers $\mathcal{V}$.

Prove Phase: $\mathcal{P}$ proves to $\mathcal{V}_j$ that $(\mathsf{sh}_j, \mathsf{w}) \in \mathcal{R}_j$ for each of the verifiers $\mathcal{V}_j \in \mathcal{V}$ where $(\mathsf{sh}_1, \ldots, \mathsf{sh}_n) \leftarrow \mathsf{Enc}(\mathsf{w})$ This phase proceeds as follows.

1) The proof is generated according to the Ligero proof system specifications and includes the following components for each verifier $\mathcal{V}_j$:
   - Outputs of the standard Ligero tests associated with circuit C: code, linear, quadratic, and linear-share tests, which are common to all verifiers.
   - Output of the column consistency check, as specified in Ligero, including the authentication paths for opening the columns of the extended witness (which was committed to using the Merkle root com).
   - Output of an additional linear test, verifying that $\mathsf{sh}_j = \mathbf{out}_j$, where sh_j is the received share and $\mathbf{out}_j$ is the jth output of the circuit C (recall that $\mathbf{out}_j$ is included in the extended witness).
   - The Merkle root com' and authentication paths associated with opening the output of the additional linear test for $\mathcal{V}_j$. Here, com' is the root of the Merkle tree, with each of the n outputs of the additional linear test as the leaves.

   Note that the first two components of the proof are common across all verifiers, while the third component differs for each verifier.
2) Finally, $\mathcal{P}$ sends the proof and the share sh_j to $\mathcal{V}_j$ for all $j \in [n]$.
3) Each verifier $\mathcal{V}_j \in \mathcal{V}$ verifies the proof received in the previous step and outputs the $(\mathsf{accept}, \mathbf{out}_j)$ if the proof verification passes, otherwise outputs reject.

Figure 2. A Distributed Commit-and-Prove Protocol

that the share sh_j received by $\mathcal{V}_j$ matches the jth output of $C(\mathsf{w})]_j$ i.e., $\mathbf{out}_j$, which is included in the extended witness.

By introducing this additional linear test to ensure share consistency, we ensure the prover can convince a single verifier $\mathcal{V}_j$ of the aforementioned properties. Extending this proof among n verifiers is our next step. The straightforward approach would be to repeat the aforementioned proof generation process for each verifier. However, this method fails because of the following reason. Note that using the Fiat-Shamir transform to obtain the non-interactive variant of the Ligero Proof system needs some more work to extend to multiple verifiers. This is because the additional components of the linear test are different for different verifiers, which results in different columns being opened for different verifiers. This can violate the zero-knowledge property. To avoid this, we need to ensure that all the verifiers open the same columns. We do so by building a Merkle tree with the output of the linear test (for share consistency) as the leaves and use the Merkle root in the Fiat-Shamir transform. When the prover sends the proof to each verifier, it also presents the authentication path corresponding to the output of the linear test for share-consistency to each verifier. This addresses the issue of differing randomness across verifiers.

In summary, the additional linear test for share-consistency and the modification of the Fiat-Shamir transform allow us to extend the Ligero proof system to accommodate multiple verifiers. Leveraging this, we can construct a dCP protocol as follows. During the Commit phase, the prover commits to the proof oracle corresponding to circuit C and input w using the Merkle tree-based hash. The resulting Merkle root is then broadcasted to all the verifiers. Subsequently, during the Prove phase, the prover generates the proof following the extension of Ligero to multiple verifiers and forwards the proof, along with the respective shares, to each of the verifiers.

The ideal functionality for Distributed Commit-and-Prove (dCP), represented by $\mathcal{F}_{\mathsf{dCP}}$, is provided in Figure 10 of Appendix A. The corresponding protocol Π_{dCP}, which securely implements $\mathcal{F}_{\mathsf{dCP}}$, is detailed in Figure 2. The formal theorem and the proof sketch are presented below.

Theorem 4.1. Given a predicate $P : \mathbb{F}^{\mathsf{d}} \to \{0, 1\}$, we first instantiate the circuit C and distributed relation

$(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ as per definitions 4.1 and 4.2.
Then, the Π_{dCP} protocol, given in Figure 2, involving a prover $\mathcal{P}$ and n verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$, securely realize the $\mathcal{F}_{\mathsf{dCP}}$ functionality (given in Figure 10) against a static malicious adversary $\mathcal{A}$ who controls the prover and at most t verifiers.
The communication between the prover and verifier $\mathcal{V}_i$ is $O(|\mathbf{out}_i| + \sqrt{|\mathsf{C}| \cdot \kappa} + \mathcal{BC}(h))$ field elements where κ is the security parameter, $|\mathsf{C}|$ is the number of multiplication gates in circuit C and h is the output length of the hash function.

Communication efficiency. The prover initially broadcasts the commitments com, the root of a Merkle tree with a size of h, incurring a cost of $\mathcal{BC}(h)$. During the proof phase, the prover sends the input share and the proof to each verifier. The input share sent to each verifier $\mathcal{V}_i$ is of size $O(|\mathbf{out}_i|)$. The proof size is $O(\sqrt{|\mathsf{C}| \cdot \kappa})$ (follows from the Ligero proof system). The additional linear test and authentication paths in our variant of Ligero do not alter the costs asymptotically. Thus, the total costs align with those specified in Theorem 4.1.

Proof Sketch. We briefly discuss the key ideas of our proof. Let $\mathcal{A}$ be the static adversary who maliciously corrupts parties in protocol Π_{dCP}. Let $\mathcal{C}$ be the subset of verifiers corrupted by $\mathcal{A}$ where $|\mathcal{C}| \leq t$.

We now describe the simulator Sim. Sim internally invokes the adversary $\mathcal{A}$ with some auxilliary input z. We consider two cases depending on whether $\mathcal{A}$ corrupts the prover or not.

Simulating the case where the Prover is honest. Since the dCP is based on the Ligero proof system, we first describe how to adapt the simulator for the zero-knowledge property in the Ligero proof system to our setting. In our setting, the simulator needs to simulate the proofs for each corrupt verifiers. Recall that we deviate from the standard Ligero proof system in two ways. First, the prover sends a separate proof to each of the verifiers, differing only in the output of an additional linear test that we introduce. The common part of the proof across all verifiers is simulated as in Ligero and remains the same for all verifiers. The additional linear test, specific to each verifier, is simulated similarly to Ligero's linear test; this part differs across verifiers. Second, the outputs of the additional linear tests are committed via a merkle tree and later revealed towards their respective verifiers. This additional merkle root, computed using a collision resistant hash function (modeled as a random oracle), does not reveal any additional information.

Now, we describe the simulation of the Commit and Prove phases in the dCP protocol: Whenever an honest prover $\mathcal{P}$ commits to a value w, Sim receives a message $(\text{receipt}, \mathit{sid}, \mathcal{P})$. Upon receiving the receipt message, Sim simulates the proof according to Ligero, which includes a common portion for all verifiers and the additional linear test outputs that differ across verifiers. Let $\widetilde{\mathsf{com}}$ be the Merkle commitment to the extended witness, generated as part of the common portion of the simulated proof. Sim simulates a broadcast of $\widetilde{\mathsf{com}}$ among the verifiers on behalf of the honest prover $\mathcal{P}$.

Next, whenever an honest $\mathcal{P}$ sends the $(\text{Prove}, \mathit{sid}, \mathcal{P}, \mathcal{V}_j, \mathsf{sh}_j)$ message to $\mathcal{F}_{\mathsf{dCP}}$, Sim receives the message $(\text{Proof}, \mathit{sid}, \mathcal{P}, \mathcal{V}_j, \mathsf{sh}_j, \mathsf{accept})$ from $\mathcal{F}_{\mathsf{dCP}}$ on behalf of $\mathcal{V}_j \in \mathcal{C}$. For each $\mathcal{V}_j \in \mathcal{C}$, then Sim internally sends the message $(\mathsf{sh}_j, \widetilde{\pi}_j)$ to $\mathcal{A}$ on behalf of $\mathcal{P}$, where $\widetilde{\pi}_j$ is the simulated proof associated with verifier $\mathcal{V}_j \in \mathcal{C}$.

Simulating the case where the Prover is corrupted. The simulation of Commit Phase is as follows. Whenever $\mathcal{A}$ (controlling $\mathcal{P}$) wants to commit to a value, Sim obtains the commitment $\mathsf{com}_{\mathbb{V}_d}$ that $\mathcal{A}$ broadcasts to all verifiers. Since the proof system is non-interactive and operates in the random oracle model, then Sim can observe all of the queries that P makes to the random oracle while computing the proof, and use it to obtain the witness w. Then, Sim externally sends the message $(\mathsf{commit}, \mathsf{sid}, \mathcal{P}, \mathsf{w})$ to $\mathcal{F}_{\mathsf{dCP}}$ and keeps track of the value w.

Next, the Prove Phase is simulated as follows. Whenever $\mathcal{A}$ wants to prove a statement to a verifier $\mathcal{V}_j$, Sim receives from $\mathcal{A}$ the share sh_j and the proof π_j on behalf of the honest verifier $\mathcal{V}_j$. Sim first verifies the proof π_j as per the verification steps described in Figure 2. If the proof verification passes but $\mathcal{R}_j(\mathsf{sh}_j, \mathsf{w}) \neq 1$ for any of the honest verifiers $\mathcal{V}_j$, then Sim aborts. If Sim does not abort, it externally sends the message $(\text{Prove}, \mathit{sid}, \mathcal{P}, \mathcal{V}_j, \mathsf{sh}_j)$ to $\mathcal{F}_{\mathsf{dCP}}$ on behalf of $\mathcal{A}$ for each honest verifier $\mathcal{V}_j$. Finally, Sim outputs whatever $\mathcal{A}$ outputs and halts.

We now prove that the real world view is computationally indistinguishable from the ideal world view. When the prover is uncorrupted, the key difference between the ideal and real executions is that the commitment $\widetilde{\mathsf{com}}$ and proof $\{\widetilde{\pi}_j\}_{\mathcal{V}_j \in \mathcal{C}}$ are both simulated in the former and generated as per the protocol in the latter. It follows from the zero-knowledge property of the Ligero proof system that the **REAL** and **IDEAL** distributions are indistinguishable (when the prover is honest).

When the prover is corrupted, we first claim that Sim aborts with negligible probability: if $\mathcal{R}_j(\mathsf{sh}_j, \mathsf{w}) \neq 1$, then proof verification on input (sh_j, π_j) corresponding to an honest verifier $\mathcal{V}_j$ fails, except with negligible probability. This follows from the soundness of Ligero.

Assuming that Sim does not abort, we need to show that the outputs of the honest verifiers are the same in both the real execution with $\mathcal{A}$ and the ideal execution with Sim. In the ideal execution, upon receiving (sh_j, π_j) internally from $\mathcal{A}$, Sim will send $(\text{dCP-Prove}, \mathsf{sid}, \mathcal{P}, \mathcal{V}_j, \mathsf{sh}_j)$ to $\mathcal{F}_{\mathsf{dCP}}$ for each honest verifier $\mathcal{V}_j$. Recall that during the simulation, if the proof verification passes, then $\mathcal{R}_j(\mathsf{sh}_j, \mathsf{w}) = 1$; similarly, if the proof verification fails, $\mathcal{R}_j(\mathsf{sh}_j, \mathsf{w}) \neq 1$. This holds because we assume Sim does not abort. As per the simulation, $\mathcal{F}_{\mathsf{dCP}}$ functionality sends $(\text{Proof}, \mathsf{sid}, \mathcal{P}, \mathcal{V}_j, \mathsf{sh}_j, \mathsf{accept})$ whenever $\mathcal{R}_j(\mathsf{sh}_j, \mathsf{w}) = 1$ (equivalently, when proof verification passes); otherwise sends $(\text{Proof}, \mathsf{sid}, \mathcal{P}, \mathcal{V}_j, \bot, \mathsf{reject})$ to each honest verifier $\mathcal{V}_j$. Therefore, the accept/reject outcome depends on whether the proof verification passes or fails in both the real and ideal

world executions. As a result, the honest parties have the same output in both worlds, assuming Sim does not abort.

4.2. Verifiable Relation Sharing (VRS) from dCP

The problem of Verifiable Relation Sharing (VRS), introduced by [15], allows a client (prover) to share a vector of secret data items among multiple servers (the verifiers) while proving in zero knowledge that the shared data adheres to certain properties. This combined task of sharing and proving generalizes notions like verifiable secret sharing and zero-knowledge proofs over secret-shared data.

We use the framework established by [14], which demonstrates the construction of a Verifiable Random Secret (VRS) from a dCP. At a high level, the VRS construction involves the prover invoking the dCP ideal functionality with its input. Note that the dCP is weaker than a VRS in that some honest verifiers might output "accept" while others output “reject.” In contrast, a VRS requires unanimous agreement among honest verifiers: either all accept and hold a valid share, or all reject.

To ensure this unanimous agreement in a VRS, we implement a share recovery mechanism whenever a verifier gets “reject” from the dCP functionality invoked by the prover. This mechanism ensures that verifiers are able to recover their shares or collectively discard the prover and output “reject.” At a high level, this recovery process involves all verifiers masking their input shares with a random share pre-computed during the offline phase using a Verifiable Secret Sharing (VSS) scheme. These masked shares are broadcasted and subsequently used by the verifiers to either unanimously reject (if the masked shares are malformed) or to recover their respective shares.

Our construction is similar to that in [14], but it uses replicated secret sharing instead of Shamir’s secret sharing scheme. We chose replicated secret sharing for its simplicity and because it allows for an offline phase that can be reused across executions, with a cost that is independent of input size, unlike the approach in [14]. The ideal VRS functionality is provided in Figure 11 of Appendix A and our VRS construction is given in Figure 3.

Optimization. The protocol described in Figure 3 requires the prover to broadcast the commitment which will require the prover to participate in more than one round. Since we need the prover to be involved only in a single round, we modify the protocol slightly to leverage the broadcast channel that exists between the verifiers. In slightly more detail, the prover sends the commitment to each of the verifiers over a point-to-point channel (instead of a broadcast). The verifiers then execute step (1) of the sharing phase of protocol Π_{VRS} in Figure 3. Next, each verifier $\mathcal{V}_j$ broadcasts to all verifiers: If the verifier received $\mathsf{happy}_j = \mathsf{accept}$, then broadcast the commitment.Otherwise, broadcast a complaint message.

If there are more than $\mathsf{t_s}$ complaints, then verifiers agree to discard the prover and output $(\mathsf{reject}, \perp)$. Otherwise, the verifiers proceed to executing steps (2) to (4) of the sharing phase of Π_{VRS} and output accept along with input share obtained at the end of step (4) (i.e., share recovery phase) of protocol Π_{VRS}. We now state the formal theorem with this optimization incorporated in Appendix C and discuss the complexity.

4.3. Secure Aggregation from VRS

In this section, we describe our secure aggregation protocol, leveraging the VRS functionality discussed earlier. Recall that we outlined the basic structure of our protocol in the overview of Section 4. At its core, our protocol unfolds in two primary phases:

- Input Sharing: Clients share their inputs using the VRS functionality $\mathcal{F}_{\mathsf{VRS}}$. Here, each client acts as a prover and servers act as verifiers during the invocation of the $\mathcal{F}_{\mathsf{VRS}}$. At the end of each invocation of $\mathcal{F}_{\mathsf{dCP}}$ by a client, the servers receive $\mathsf{accept}/\mathsf{reject}$ along with the input share associated with this client.
- Output Reconstruction: Servers determine a set of valid clients if they received the output accept from $\mathcal{F}_{\mathsf{dCP}}$. Then, they sum the shares received from invocation of $\mathcal{F}_{\mathsf{dCP}}$ corresponding to all the valid clients and send their aggregate shares to the output party. The output party collects and error-corrects the received shares to reconstruct the aggregate output.

For completeness, we provide a detailed protocol description using VRS in Figure 4, as taken verbatim from [14]. The formal theorem statement below.

Theorem 4.2. Let $\mathsf{n_s}, \mathsf{t_s}, \mathsf{d} \in \mathbb{N}$ such that $\mathsf{t_s} < \mathsf{n_s}/3$ and $P : \mathbb{F}^{\mathsf{d}} \to \{0, 1\}$ be an arbitrary predicate. Let $\mathcal{F}_{\mathsf{Agg}}$ be the ideal functionality given in Figure 1. The protocol Π_{Agg}, as outlined in Figure 4, securely realizes $\mathcal{F}_{\mathsf{Agg}}$ in the $\mathcal{F}_{\mathsf{VRS}}$-hybrid model among $\mathsf{n_c}$ clients each holding input vectors of length d with elements in some finite field $\mathbb{F}$, $\mathsf{n_s}$ servers, and an output party $\mathcal{O}$, which is secure against a static, malicious rushing adversary that can maliciously corrupt an arbitrary number of clients, up to $\mathsf{t_s}$ servers and the output party and ensures guaranteed output delivery where κ is the security parameter. Additionally, a client is required to engage in only a single round of communication.
The communication between the prover and each of the verifiers is $O(\mathsf{d} \cdot \binom{\mathsf{n_s}-1}{\mathsf{t_s}} + \rho)$ field elements where κ and $O(1)$ field elements is the security parameter and $|P|$ is the number of gates in the circuit associated with predicate P. The total communication among the servers is as follows (in terms of field elements):

– Offline phase: $O(\mathsf{n_s^3} + \mathsf{n_s} \cdot \mathcal{BC}(\mathsf{n_s^2}))$
– Online phase in the worst case: $O(\mathsf{n_c} \cdot \mathsf{n_s} \cdot \rho + \mathsf{n_c} \cdot \mathsf{n_s} \cdot \mathcal{BC}(h) + \cdot \mathsf{n_c} \cdot \mathsf{n_s} \cdot \mathsf{d} \cdot \mathcal{BC}(\binom{\mathsf{n_s}-1}{\mathsf{t_s}}))$
– Online phase in the optimistic setting (when the all the servers are honest) and γ fraction of the clients are malicious: $O(\mathsf{n_c} \cdot \mathsf{n_s} \cdot (\mathsf{d} \cdot \binom{\mathsf{n_s}-1}{\mathsf{t_s}} + \rho) + \mathsf{n_c} \cdot \mathsf{n_s} \cdot \mathcal{BC}(h) + \gamma \cdot \mathsf{n_c} \cdot \mathsf{n_s} \cdot \mathsf{d} \cdot \mathcal{BC}(\binom{\mathsf{n_s}-1}{\mathsf{t_s}})))$. (the main difference is the additional γ factor in the second term).

Protocol Π_{VRS}

This protocol allows a prover $\mathcal{P}$ with input $x \in \mathbb{F}$ to verifiably secret share $x \in \mathbb{F}$ among n verifiers $\mathcal{V} = \{\mathcal{V}_1, \dots, \mathcal{V}_n\}$ and prove that $P(x) = 1$ to all the verifiers.

Public Parameters. It is parameterized by a bound $n \geq 3t+1$ where n is the number of verifiers, t is the number of corrupt verifiers and a predicate $P : \mathbb{F} \to \{0,1\}$. For the predicate P, we can obtain a circuit $\mathsf{C} : \mathbb{F}^{t+1} \to \mathbb{F}^{\mathsf{n}_2}$ and n relations $(\mathcal{R}_1, \dots, \mathcal{R}_n)$ as per Definition 4.2. All the parties have access to an distributed commit-and-prove ideal functionality $\mathcal{F}_{\mathsf{dCP}}$, which is parameterized by $(\mathcal{R}_1, \dots, \mathcal{R}_n)$.

Input & Output. $\mathcal{P}$ has an input $x \in \mathbb{F}$ and the verifiers $\mathcal{V}$ have no inputs. If $P(x) = 1$ holds, then each verifier $\mathcal{V}_j \in \mathcal{V}$ outputs $(\mathsf{Share}_j,\ \mathsf{accept})$ where $(\mathsf{Share}_1, \dots, \mathsf{Share}_n) \leftarrow \mathsf{Enc}(x)$. Otherwise, all verifiers output reject.

Offline Phase. The parties interactively generate a valid sharing of a random (unknown) value $k \in \mathbb{F}$ where $(\mathsf{ksh}_1, \dots, \mathsf{ksh}_{n_s}) \leftarrow \mathsf{Enc}(k)$ and each verifier $\mathcal{V}_j$ receives shares rsh_i for all $i \in [n]$. Each verifier $\mathcal{V}_i \in \mathcal{V}$ proceeds as follows:

1) Sample a random value $k_i \in \mathbb{F}$ and secret-share using RSS it among the other verifiers using a VSS scheme such that $\mathcal{V}_j$. Note that $\mathcal{V}_i$ acts as a dealer here.
2) The verifiers run a pair-wise consistency check where the parties exchange the common values between their shares. If there are any inconsistencies, broadcast a complaint with identities of the pair of parties. Then, the dealer broadcasts all the shares held by the pair of parties againts whom a complaint was raised.
3) Upon the completion of all n instances of secret sharing and pair-wise consistency checks, $\mathcal{V}_i$ computes and outputs a sharing of $k = \sum_{i \in [n]} k_i$ from the RSS shares of $\{k_i\}_{i \in [n]}$ because of the linearity property of RSS.

Later, during the sharing phase, the verifiers can use the replicated secret sharing of k to generate an RSS sharing $(\mathsf{rsh}_1, \dots, \mathsf{rsh}_n)$ non-interactively (as described in Section 3.3).

Sharing Phase.

1) **[Input Sharing]** Prover $\mathcal{P}$ with a secret x proceeds as follows.
   - Sample randomness r_{vrs} and encode the input x as follows: $(\mathsf{sh}_1, \dots, \mathsf{sh}_n) \leftarrow \mathsf{Enc}(x; r_{\mathsf{vrs}})$. Let $(\mathsf{sh}_1, \dots, \mathsf{sh}_n)$ be denoted by Shares.
   - Invoke the Commit Phase of $\mathcal{F}_{\mathsf{dCP}}$ as the prover with input $(\mathsf{Commit}, \mathsf{sid}, \mathcal{P}, x, r_{\mathsf{vrs}})$.
   - Invoke the Prove phase of $\mathcal{F}_{\mathsf{dCP}}$ as a prover with input $(\mathsf{Prove}, \mathsf{sid}, \mathcal{P}, \mathcal{V}_j, \mathsf{sh}_j)$.
2) Upon receiving the message $(\mathsf{Proof}, \mathsf{sid}, \mathsf{sh}_j, \mathsf{happy}_j)$ from $\mathcal{F}_{\mathsf{dCP}}$, each verifier $\mathcal{V}_j$ proceed as follows.
   - If $\mathsf{happy}_j = \mathsf{accept}$, then broadcast $(\mathsf{Masked\text{-}Share}, \mathsf{sid}, \mathcal{V}_j, \mathsf{msh}_j)$ where the masked share $\mathsf{msh}_j := \mathsf{sh}_j + \mathsf{rsh}_j$
   - Otherwise, set $\mathsf{msh}_j := \perp$ and broadcast nothing.
3) **[Consistency Check]** Let the broadcasted message from each verifier $\mathcal{V}_j \in \mathcal{V}$ be denoted by $(\mathsf{Masked\text{-}Share}, \mathsf{sid}, \mathcal{V}_j, \mathsf{msh}'_j)$. If the masked shares obtained from verifiers' broadcasts, denoted by $(\mathsf{msh}'_1, \dots, \mathsf{msh}'_n)$, form a valid encoding i.e., decoding succeeds on shares $(\mathsf{msh}'_1, \dots, \mathsf{msh}'_n)$.
4) **[Share Recovery]** Each verifier $\mathcal{V}_j$ *locally* computes its output as follows:
   a) If the consistency check fails, then set $\mathsf{Share}_j := \perp$ and output $(\mathsf{reject}, \perp)$.
   b) If the consistency check passes, then each verifier $\mathcal{V}_j \in \mathcal{V}$ outputs $(\mathsf{accept}, \mathsf{Share}_j)$ where Share_j is computed as follows:
      i) **Keep Existing Share:** If $\mathsf{happy}_i = \mathsf{accept}$, then set $\mathsf{Share}_j := \mathsf{sh}_j$, or
      ii) **Recover Share:** If $\mathsf{happy}_i = \mathsf{reject}$, then $\mathcal{V}_j$ needs to recover its share by computing $\mathsf{Share}_j := \mathsf{msh}''_j - \mathsf{rsh}_j$ where $(\mathsf{msh}''_1, \dots, \mathsf{msh}''_n)$ is obtained by error-correcting $(\mathsf{msh}'_1, \dots, \mathsf{msh}'_n)$.

Figure 3. A VRS Protocol for predicate P

where h is the output length of the hash function and $\rho = \sqrt{(\mathsf{d} \cdot \binom{\mathsf{n_s}-1}{\mathsf{t_s}}) + |P|) \cdot \kappa}$

The costs in the above theorem are obtained by multiplying the costs in Theorem C.1 by a factor of $\mathsf{n_c}$.

5. Implementation and Evaluation

We demonstrate the efficacy, performance, and practicality of our protocol through an implementation that we call SCIF. The core aim of our implementation is to enable analysts to easily and securely compute aggregate statistics over data that is distributed potentially across a large number of parties. SCIF is ready-to-deploy software released under a permissive open-source license, and is available for use now at https://anonymous.4open.science/r/scif/.

We begin by describing SCIF's architecture and operation (Section 5.1). We then explore the operational costs of using SCIF via a real-world deployment consisting of hundreds of distributed clients (Section 5.2). Finally, we describe a case-study in which we use SCIF to securely and privately compute statistics about the performance of a private Tor network (Section 5.3).

5.1. Architecture and Operation

SCIF is constructed with security, scalability, and ease-of-use as principle design goals. We build SCIF in 7.7k

Input-Certified Secure Aggregation Protocol Π_{Agg}

Public parameters. Security parameter λ, field $\mathbb{F}$, input vector length d, server corruption threshold $\mathsf{t_s}$, predicate $P : \mathbb{F} \to \{0, 1\}$.
Parties. Clients $\mathcal{U} = \{\mathcal{U}_1, \ldots, \mathcal{U}_{\mathsf{n_c}}\}$, servers $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_{\mathsf{n_s}}\}$ and an output party $\mathcal{O}$.
Input & Output. $\{x_1, \ldots, x_{\mathsf{n_c}}\}$ where $x_i \in \mathbb{F}$ is client $\mathcal{U}_i$'s input. The output party $\mathcal{O}$ receives $\sum_{\mathcal{U}_i \in \mathcal{U}} P(x_i) \cdot x_i$.
Setup. Each client has a point-to-point private, authenticated channel with every server. Also, the servers have point-to-point, private authenticated channels with all other servers and a broadcast channel.

Input Sharing. $[\mathcal{U} \to \mathcal{S}]$ The input sharing proceeds as follows.

1) Each client $\mathcal{U}_i$ acts as a dealer and invokes an instance of the VRS functionality $\mathcal{F}_{\mathsf{VRS}}$ with input $x_i \in \mathbb{F}$. This calls for the client $\mathcal{U}_i$ to send the tuple $(\mathsf{Input}, \mathsf{sid}, \mathcal{U}_i, x_i, r_{\mathsf{vrs},i})$ to $\mathcal{F}_{\mathsf{VRS}}$ where $r_{\mathsf{vrs},i}$ is the randomness sampled by the $\mathcal{U}_i$.
2) Each server $\mathcal{S}_j$ participates in $\mathcal{F}_{\mathsf{VRS}}$ as a verifier and receives the tuple $\left(\mathsf{Output}, \mathsf{sid}, \mathcal{S}_j, \mathsf{Share}_j^{(i)}, \mathsf{happy}_j^{(i)}\right)$ for all $j \in [\mathsf{n_s}]$. As per the properties of the $\mathcal{F}_{\mathsf{VRS}}$, all servers output the same happy bit i.e., $\mathsf{happy}_j^{(i)} = \mathsf{happy}_{j'}^{(i)}$ for $j, j' \in [\mathsf{n_s}]$. So, we drop the subscript j while referring to the happy bit.

Output Reconstruction. $[\mathcal{S} \to \mathcal{O}]$

1) At the end of all the invocations to $\mathcal{F}_{\mathsf{VRS}}$, the servers define a set Valid to comprise of all clients $\mathcal{U}_i$ such that $\mathsf{happy}^{(i)} = \mathsf{accept}$.
2) Each server $\mathcal{S}_j$ sums the shares it received from all clients in the set Valid i.e., $\mathsf{osh}_j = \sum_{\mathcal{U}_i \in \mathsf{Valid}} \mathsf{Share}_j^{(i)}$ and sends its output share osh_j to the output party.
3) The output party collects shares of the output osh_j from each server $\mathcal{S}_j \in \mathcal{S}$. If no share is received from the server $\mathcal{S}_j$, then the output party sets $\mathsf{osh}_j := \bot$. Finally, the output party error-corrects the vector $(\mathsf{osh}_1, \ldots, \mathsf{osh}_{n_s})$ to reconstruct Y_{agg} and sets Y_{agg} as the output.

Figure 4. An Input-Certified Secure Aggregation Protocol from VRS

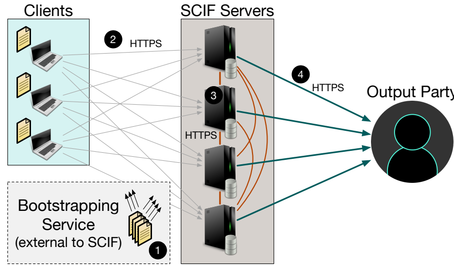

Figure 5. High-level architecture of SCIF. After bootstrapping (❶), clients send secret shares and proofs to each server (❷). Servers then construct their outputs (❸) and communicate their outputs to the output party (❹).

lines (as measured by scc) of Go and make extensive use of the language's memory safety and parallelism features (i.e., goroutines) to optimize CPU resources and support large numbers of clients. Cryptographic functions used Go's crypto package, including crypto/tls, crypto/sha256, and crypto/rand, and PRFs were constructed using ChaCha20.

For our implementation, we prioritized simplicity, focusing on essential features and deployment. We built our protocol in a modular fashion, with separate packages for packed secret sharing, replicated secret sharing (RSS), and the Ligero proof system. Some choices we made, while simplifying the process, might not be ideal: (1) We used a the näive polynomial interpolation algorithm that runs in quadratic time for packed secret sharing instead of a more efficient Fast Fourier Transform (FFT)-based approach that runs in quasilinear time. As we built our code in a modular fashion, we can easily replace the packed secret sharing protocol with the FFT-based approach, enhancing performance without major code overhauls. (2) We opted for RSS due to its simplicity and efficiency, especially suitable for a small number of servers and field sizes. RSS facilitates an offline phase independent of input size and enables non-interactive random sharing generation for share recovery. If alternative schemes become more suitable, RSS can be easily substituted, given our modular implementation.

The high-level architecture of SCIF is shown in Figure 5. The three principle components—clients, servers, and the output party—are all implemented as web services, and use an object-relational mapping (ORM) model to persist program state. SCIF is compatible with any backend supported by GORM [30]; we use MySQL 8.0.36 for servers and the output party. Messages exchanged between parties are transmitted via HTTPS (i.e., TLSv1.3). SCIF supports using its own PKI, but for simplicity, we use certificates from LetsEncrypt in our test deployment.

SCIF assumes a *bootstrapping phase* in which participants (clients, servers, and the output party) receive the parameters and credentials that are necessary to participate in an experiment (Figure 5, ❶). The use of credentials is explained below. Parameters include a unique experiment ID (exp); the time by which clients must submit their inputs' shares; the times by which servers must submit their complaints, the masked shares of clients that correspond to received complaints, and aggregates of clients' shares; the public parameters for the Ligero proof system; and the network identifiers (e.g., hostnames) and public keys of the SCIF servers.

To restrict which clients may participate in a particular experiment, SCIF supports optional client authentication via a modular authentication API. The API consists of a single function, $\top/\bot \leftarrow \mathrm{auth}(\mathrm{exp}, \mathrm{cred})$, where auth returns true ($\top$) iff the credential cred is valid for an experiment exp.

We have implemented a simple token-based authentication scheme. Adding support for additional authentication mechanisms (e.g., a university login) simply requires overloading the auth function.

Importantly, distributing the parameters and credentials is handled externally to SCIF. This provides maximum flexibility as it allows distribution to be carried out using mechanisms that best match the particular deployment (e.g., over-the-air updates for an experiment involving smartphone users vs. a browser extension that contains experiment configurations for web users).

After the bootstrapping phase, SCIF performs the operations described in Section 4. Clients submit their shares and proofs to each SCIF server (❷) and optionally include an authentication credential with their submission. Servers verify the credential (if applicable), and perform proof verification and complaint generation (if necessary) when shares are received. After the shares are due, the servers initiate a *server processing phase* (❸) in which they first broadcast their complaints and then perform masked share generation and broadcasting, followed by share correction[4], and aggregation. In the current implementation, we did not implement a protocol to realize broadcast between the servers but rather accomplished broadcasting by sending point-to-point messages. After the server processing phase, the servers send their output (aggregate shares) to the output party (❹) which then computes the aggregate result.

5.2. Distributed Cloud Deployment

To evaluate its performance under real-world conditions, we deployed SCIF across multiple data centers and geographic locations using Google Cloud.

Setup. SCIF servers and the output party used fixed instances in the us-east1 (South Carolina) and us-west1 (Oregon) regions, while clients were located on regions selected randomly from those available in Google Cloud. SCIF uses TLS for secure communication; we registered domain names for the servers and the output party and configured our SCIF instances with certificates issued from LetsEncrypt. Servers and the output party were c2d-standard-32 instances with 32 VCPUs (16 cores) and 128 GB of RAM. Unless otherwise specified, each client was a e2-standard-8 instance with 8 VCPUs (4 cores) and 32 GB of RAM. All machines ran Ubuntu 22.04 with kernel 6.5.0-1020-gcp.

The values of clients' inputs were generated uniformly at random. SCIF's performance and operation do not depend on the particular inputs chosen by clients. (We do explore how the *size* of clients' inputs affects SCIF's performance.)

SCIF utilizes deadlines (i.e., for clients' input shares submissions, servers' complaints, and masked shares and aggregate shares submissions) to achieve synchronization among all parties. Through extensive experiments, we found optimal deadlines that minimize the waiting time at each phase. When measuring the execution time of the system, we subtract the waiting time from our results. Our results thus are informative of how quickly experiments could be carried out (under our experimental setup). We emphasize that SCIF supports parallel experiments, and thus the cost of idling could be amortized away if several measurements are conducted in parallel.

The current system did not implement offline phase so we assume the pairwise PRF keys were already available for the system to use. This is a one-time setup cost that can be used across multiple experiments and therefore will not affect our benchmarks.

The performance of SCIF depends on the client's input length, number of servers, and public parameters for the Ligero proof system. The default values of these parameters, as used in our experiments, are listed in Appendix D.

Proof generation and verification. We first consider the cost of generating and verifying proofs—operations that occur respectively at the client and the servers. Figure 6 shows the median time required to generate *(top)* and verify *(bottom)* a proof for varying client input sizes. Error bars represent the range of values across five executions. (Many error bars are not visible due to their low magnitude.)

We find that the generation and verification times are modest and our measurements confirm that they grow sublinearly with respect to input length (note that the x-axis in Figure 6 is in log-scale). Generating and verifying a client's input of 10,000 values requires less than two seconds in total. Even with very large inputs consisting of 10^6 values, the median time for the client to generate the proof is 65 seconds. Verification time, at the server, for the proof over 10^6 values is only 13 seconds. In Appendix E, we show similar results using 10 clients and 7 servers.

Performance in the presence of malicious behavior. We test our system's performance in the presence of a malicious adversary as follows. To simulate a malicious adversary, we can either have a malicious client send a malformed message (such as an input share or proof) to the server, or introduce a malicious server that deviates from the protocol (e.g. by becoming unreachable or sending unnecessary complaints). Our system's performance depends on the number of the clients for which the share recovery mechanism is triggered, regardless of whether it was triggered by a malicious client or server. Thus, we study the performance of our system by varying the percentage of clients for which share recovery is triggered and we trigger the share recovery by having malicious clients send malformed proofs to a server.

First, we evaluate SCIF's performance when a fraction of the clients behave maliciously and transmit malformed proofs to one of the servers. Our experiments use four servers and 500 clients, the latter of which are distributed across six machine. Each of the client machines was provisioned with 16 cores and 128 GB of RAM.

SCIF's performance when 50 (10%) of the clients are malicious is presented in Figure 7 *(top)*. The Figure shows the constituent costs of SCIF's operations; the overall cost

[4] Servers issue complaints against a client if the proof associated with the client fails to verify (see Section 4). The share recovery phase, which includes mask generation and correction, is executed by the servers for clients who have had complaints raised against them.

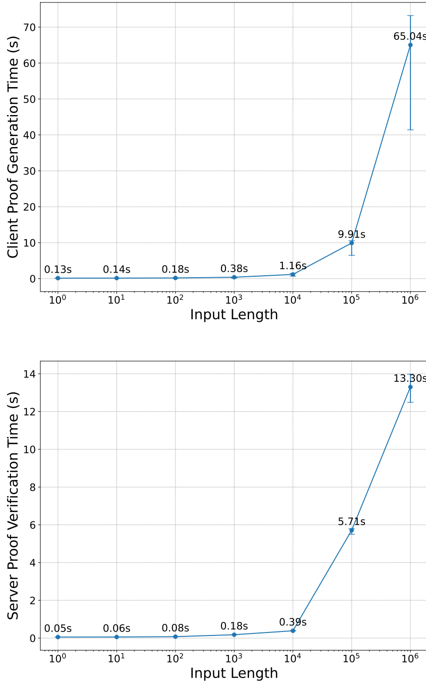

Figure 6. The time required for a client to generate a proof *(top)* and for a server to verify a client's proof *(bottom)* for various sized client inputs (in log-scale).

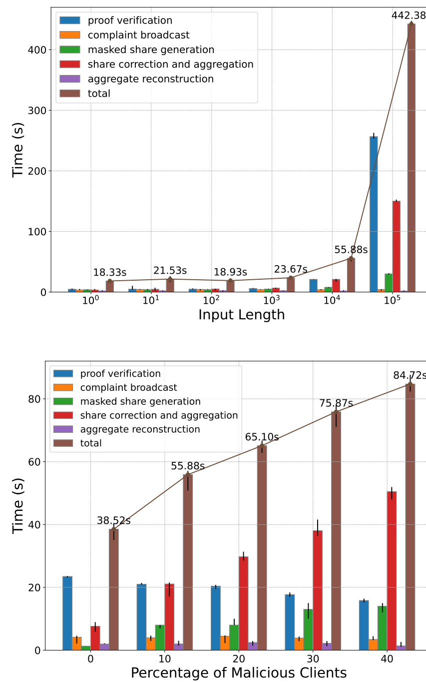

Figure 7. *Top:* Execution time (log-scale) with 500 clients and varying client input lengths, when 50 clients (10%) submit invalid proofs. *Bottom:* Execution time for various percentages of malicious clients, and a client input length of 10^4.

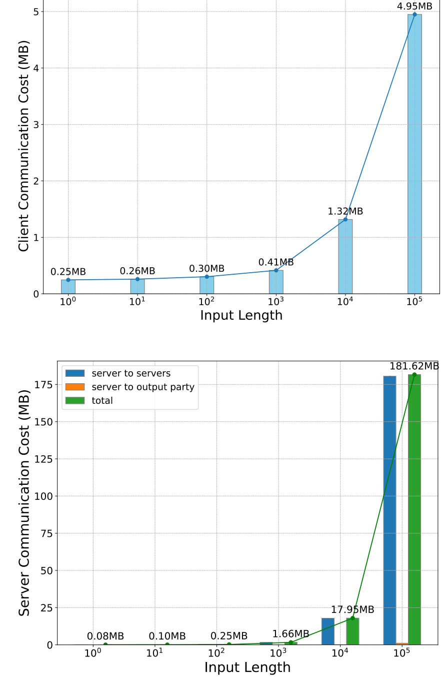

Figure 8. *Top:* Client communication cost as the input length varies (log-scale). *Bottom:* Server communication cost for 500 clients with varied input lengths (log-scale). Fifty (10%) of the clients are malicious.

is shown as "total". When each client's input vector has 10^5 values, the total execution time for a server is 442.38 seconds, which involves verifying 500 proofs in parallel, generating 500 complaints, executing share recovery for 50 malicious clients' shares, aggregating all valid shares, and then sending the aggregated shares to the output party, as well as any time due to network communication. The offline phase[5] is not factored into the server costs as this is a one-time expense and can be reused across multiple experiments.

We also explored how the system scales with various percentages of malicious clients. As the percentage of malicious clients increases from 10 to 40%, the total execution time grows linearly, as is shown in Figure 7 *(bottom)*. Even when 40% of the clients provide malicious inputs, the total execution time is less than 1.5 minutes.

Communication cost. A practical secure aggregation system should not impose excessively high communication costs. To understand SCIF's communication overhead, we examine the total communication cost produced by our system for each party, as measured by (pcap) packet traces we record on each node. Our experimental setup consisted of four servers, one output party, and 500 clients; 50 (10%) of the clients were configured to be malicious. The results are shown in Figure 8. For client input vectors of 10^5 values, the total communication cost for a client is 4.95 MB (Figure 8, *top*), which includes transmitting the shares and proof. This cost accounts for communication with all 4 servers. The total communication cost for a server is 181.62 MB (Figure 8, *bottom*), which consists of sending complaints for 500 clients to each of the three other servers, masked shares for 50 malicious clients to each of the other servers, and aggregates of 500 clients' shares to the output party. In summary, we consider the communication costs to be minimal for clients and modest for servers.

5.3. Simulation Case Study: Safely Measuring Tor

As a case study, we use SCIF to measure the performance of nodes in a simulated Tor network [17]. Tor enables anonymous communication by forwarding its users' traffic through a series of routers (or in Tor parlance, *relays*). The use of encrypted message headers prevents relays and network eavesdroppers from learning the network locations (i.e., IP addresses) of the communicants. We chose Tor as an illustrative use-case because (1) it is a popular service with millions of daily users [18]; (2) privacy is the focal point of its ethos and thus SCIF's privacy protections make SCIF a natural fit; and (3) the network's clients and volunteer-operated relays can behave dishonestly, making input validation an important desiderata for deployment.

[5]Recall that during the offline phase, each server verifiably secret shares the PRF keys as per the replicated secret-sharing scheme. These keys are later used to locally generate a secret sharing of the masks when required for the share recovery mechanism.

The primary goal of our case study is to demonstrate the ease at which systems such as Tor can be instrumented with SCIF to provide privacy-preserving secure measurements. We opted to perform measurements on a private Tor network which we deploy in Shadow [31], [32], a high-fidelity discrete-event simulator. Shadow allows us to operate SCIF on all Tor clients and relays, which would not be possible on the live Tor network without buy-in from both its maintainers and its users. We emphasize that Shadow runs *unmodified* binaries (including Tor and SCIF) on top of a virtualized networking layer, and that no changes to Tor or SCIF were required for our case study.

We instantiated a Tor network in Shadow that consists of 100 Tor relays, 25 of which are *exit relays* that route Tor's egress traffic to the final destination. Each exit relay also operated a SCIF client. We additionally introduced six SCIF servers and one SCIF output party into the network. As a workload for our two hour experiment, we used the tgen [31] traffic generator to cause Tor clients to periodically fetch web pages through the anonymity network.

Integrating Tor and SCIF took minimal effort. We wrote a short (∼80 lines) Python script that executed on each Tor instance. The script listens to Tor's control port for system events and maintains the desired statistic (explained below). To facilitate SCIF measurements, the script constructs a binary vector where each element in the vector corresponds to bin in a histogram. This mirrors the approach of prior work on privacy-preserving measurements for Tor [33]. The script communicates this vector to the SCIF client that is running on the same node, which in turn participates in the distributed SCIF protocol with the SCIF servers.

The aggregate statistics, as computed by the SCIF output party, are shown in Figure 9. (We manually verified the results are consistent with the individual measurements from the relays.) We consider two statistics that are of interest to the Tor community: the rate of TCP connections established by exit relays (i.e., the rate of TCP flows anonymized through Tor) and the observed rate of Tor egress traffic. These are respectively depicted in the left- and right-hand sides of Figure 9. The empirically measured distribution of connections and throughput generally follow the bandwidth capacities of the exit relays; this is unsurprising since Tor employs a bandwidth-weighted relay selection strategy [17].

Our case study highlights one potential path for instrumenting other applications to use SCIF. Network simulators, such as Shadow, that execute unmodified code provide a proving ground for "glueing" applications together with SCIF. In the case of Tor, this took minimal effort and less than 100 lines of code. Our future work entails real-world experimentation with SCIF-equipped Tor—work that we believe will be critical for improving our understanding of how privacy-sensitive systems are used in practice.

6. Discussion

Improvements to implementation. We are continuing to improve SCIF, and view its development and maintenance as long-term efforts. Our most immediate plans include decreasing memory consumption, which currently presents scaling issues with very large input sizes (i.e., above 10^6). Here, we plan to leverage Ligetron [34], a recent proposal to improve the Ligero ZK system that has been shown to scale to billions of gates and run efficiently even inside of a browser. We believe that Ligetron is more memory efficient than our Ligero implementation, and its design seems compatible with SCIF.

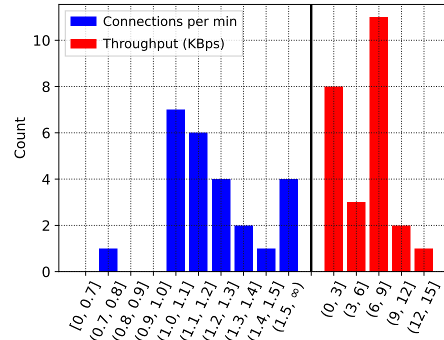

Figure 9. Histogram of the number of connections per minute observed by Tor exit relays *(left)* and the observed throughput of the exit relays *(right)*.

A useful enhancement of a secure aggregation protocol involves executing the protocol over multiple iterations. This approach is beneficial in federated learning, where the objective is to repeatedly run the protocol to achieve a stable solution, such as a machine learning model. This necessitates that servers retain state and implement a method to verify the consistency of client inputs across iterations, which we propose as future work.

In-passing contributions. Several components of SCIF may be of independent value to applied cryptographers and system implementers. Although Go offers a comprehensive cryptography library, not all primitives required by SCIF were available. In constructing SCIF, we developed (to our knowledge) the first Go-based implementations of packed secret sharing and replicated secret sharing. We designed these components with modularity in mind so that they are separable from SCIF. We are releasing each as independent FOSS libraries, with the hope that they may foster development of new privacy-preserving systems.

Deployment scenarios. Although there are myriad uses for private-preserving statistics collection, we conclude by highlighting exciting use-cases for SCIF. For example, hospitals may want to collaborate in analyzing patient outcomes for emerging diseases, enabling researchers to identify patterns and analyze the exposure of threats in a privacy-preserving manner. Embedded as a browser extension, SCIF could facilitate studies that examine online ad networks, the selection of content on users' social networking feeds, or users' web browsing behavior. And, as explored above, SCIF is a natural fit for performing measurements of anonymous networks and/or censorship-resistant technologies. Our future work entails exploring these and other use-cases.

References

[1] T. Jeske, “Floating car data from smartphones: What google and waze know about you and how hackers can control traffic,” 2013, Presented at BlackHat Europe.

[2] A. Hilts, C. Parsons, and J. Knockel, “Every step you fake: a comparative analysis of fitness tracker privacy and security,” 2016, Presented at Open Effect.

[3] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data,” in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, ser. Proceedings of Machine Learning Research, A. Singh and X. J. Zhu, Eds., vol. 54, 2017, pp. 1273–1282.

[4] K. A. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, “Practical secure aggregation for privacy-preserving machine learning,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[5] H. Corrigan-Gibbs and D. Boneh, “Prio: Private, robust, and scalable computation of aggregate statistics,” in *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, A. Akella and J. Howell, Eds. USENIX Association, 2017, pp. 259–282. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs

[6] ——, “Henry Corrigan-Gibbs’ web page, prio: Private, robust, and scalable computation of aggregate statistics,” https://people.csail.mit.edu/henrycg/pubs/nsdi17prio/, 2017, accessed: 01-05-2022.

[7] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, “Targeted backdoor attacks on deep learning systems using data poisoning,” *CoRR*, vol. abs/1712.05526, 2017. [Online]. Available: http://arxiv.org/abs/1712.05526

[8] L. Burkhalter, H. Lycklama, A. Viand, N. Küchler, and A. Hithnawi, “Rofl: Attestable robustness for secure federated learning,” *CoRR*, vol. abs/2107.03311, 2021. [Online]. Available: https://arxiv.org/abs/2107.03311

[9] T. D. Nguyen, P. Rieger, H. Yalame, H. Möllering, H. Fereidooni, S. Marchal, M. Miettinen, A. Mirhoseini, A. Sadeghi, T. Schneider, and S. Zeitouni, “FLGUARD: secure and private federated learning,” *CoRR*, vol. abs/2101.02281, 2021. [Online]. Available: https://arxiv.org/abs/2101.02281

[10] S. Addanki, K. Garbe, E. Jaffe, R. Ostrovsky, and A. Polychroniadou, “Prio+: Privacy preserving aggregate statistics via boolean shares,” Cryptology ePrint Archive, Report 2021/576, 2021, https://eprint.iacr.org/2021/576.

[11] L. Bangalore, M. H. F. Sereshgi, C. Hazay, and M. Venkitasubramaniam, “Flag: A framework for lightweight robust secure aggregation,” in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, ASIA CCS 2023, Melbourne, VIC, Australia, July 10-14, 2023*, J. K. Liu, Y. Xiang, S. Nepal, and G. Tsudik, Eds. ACM, 2023, pp. 14–28. [Online]. Available: https://doi.org/10.1145/3579856.3595805

[12] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, “Practical secure aggregation for privacy-preserving machine learning,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[13] J. Bell, A. Gascón, T. Lepoint, B. Li, S. Meiklejohn, M. Raykova, and C. Yun, “ACORN: input validation for secure aggregation,” *IACR Cryptol. ePrint Arch.*, p. 1461, 2022. [Online]. Available: https://eprint.iacr.org/2022/1461

[14] L. Bangalore, A. Cheu, and M. Venkitasubramaniam, “PRIME: Differentially private distributed mean estimation with malicious security,” *IACR Cryptol. ePrint Arch.*, p. 1771, 2024. [Online]. Available: https://eprint.iacr.org/2024/1771

[15] B. Applebaum, E. Kachlon, and A. Patra, “Verifiable relation sharing and multi-verifier zero-knowledge in two rounds: Trading nizks with honest majority,” *IACR Cryptol. ePrint Arch.*, p. 167, 2022. [Online]. Available: https://eprint.iacr.org/2022/167

[16] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, “Ligero: Lightweight sublinear arguments without a trusted setup,” Cryptology ePrint Archive, Paper 2022/1608, 2022. [Online]. Available: https://eprint.iacr.org/2022/1608

[17] R. Dingledine, N. Mathewson, and P. Syverson, “Tor: The Second-Generation Onion Router,” in *USENIX Security Symposium (USENIX)*, Aug. 2004.

[18] A. Mani, T. W. Brown, R. Jansen, A. Johnson, and M. Sherr, “Understanding Tor Usage with Privacy-Preserving Measurement,” in *ACM SIGCOMM Conference on Internet Measurement (IMC)*, Oct. 2018.

[19] J. H. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova, “Secure single-server aggregation with (poly)logarithmic overhead,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020. [Online]. Available: https://doi.org/10.1145/3372297.3417885

[20] J. So, B. Guler, and A. S. Avestimehr, “Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning,” *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 167, 2020. [Online]. Available: https://eprint.iacr.org/2020/167

[21] S. Kadhe, N. Rajaraman, O. O. Koyluoglu, and K. Ramchandran, “Fastsecagg: Scalable secure aggregation for privacy-preserving federated learning,” *CoRR*, vol. abs/2009.11248, 2020. [Online]. Available: https://arxiv.org/abs/2009.11248

[22] J. So, C. J. Nolet, C. Yang, S. Li, Q. Yu, R. E. Ali, B. Guler, and S. Avestimehr, “Lightsecagg: a lightweight and versatile design for secure aggregation in federated learning,” in *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, D. Marculescu, Y. Chi, and C. Wu, Eds. mlsys.org, 2022. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2022/hash/6c44dc73014d66ba49b28d483a8f8b0d-Abstract.html

[23] A. R. Chowdhury, C. Guo, S. Jha, and L. van der Maaten, “Eiffel: Ensuring integrity for federated learning,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 2535–2549. [Online]. Available: https://doi.org/10.1145/3548606.3560611

[24] M. Mansouri, M. Önen, W. B. Jaballah, and M. Conti, “Sok: Secure aggregation based on cryptographic schemes for federated learning,” *Proc. Priv. Enhancing Technol.*, vol. 2023, no. 1, pp. 140–157, 2023. [Online]. Available: https://doi.org/10.56553/popets-2023-0009

[25] M. Rathee, C. Shen, S. Wagh, and R. A. Popa, “ELSA: secure aggregation for federated learning with malicious actors,” in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 1961–1979. [Online]. Available: https://doi.org/10.1109/SP46215.2023.10179468

[26] M. Rathee, Y. Zhang, H. Corrigan-Gibbs, and R. A. Popa, “Private analytics via streaming, sketching, and silently verifiable proofs,” *IACR Cryptol. ePrint Arch.*, p. 666, 2024. [Online]. Available: https://eprint.iacr.org/2024/666

[27] A. P. K. Dalskov, D. Escudero, and A. Nof, “Fast fully secure multi-party computation over any ring with two-thirds honest majority,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 653–666. [Online]. Available: https://doi.org/10.1145/3548606.3559389

[28] M. Ito, A. Saito, and T. Nishizeki, “Secret sharing scheme realizing general access structure,” *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, vol. 72, no. 9, pp. 56–64, 1989.

[29] J. Zhang, T. Xie, T. Hoang, E. Shi, and Y. Zhang, "Polynomial commitment with a one-to-many prover and applications," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 2965–2982. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-jiaheng

[30] Jinzhu, "GORM: The fantastic ORM library for Golang," available at https://gorm.io.

[31] R. Jansen and N. Hopper, "Shadow: Running Tor in a Box for Accurate and Efficient Experimentation," in *Network and Distributed System Security Symposium (NDSS)*, 2012.

[32] R. Jansen, J. Newsome, and R. Wails, "Co-opting Linux Processes for High-Performance Network Simulation," in *USENIX Annual Technical Conference*, 2022.

[33] A. Mani and M. Sherr, "Historϵ: Differentially Private and Robust Statistics Collection for Tor," in *Network and Distributed System Security Symposium (NDSS)*, Feb. 2017.

[34] R. Wang, C. Hazay, and M. Venkitasubramaniam, "Ligetron: Lightweight scalable end-to-end zero-knowledge proofs. post-quantum zk-snarks on a browser," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2023, pp. 86–86.

[35] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, "Ligero: Lightweight sublinear arguments without a trusted setup," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[36] E. Ben-Sasson, A. Chiesa, and N. Spooner, "Interactive oracle proofs," in *TCC*, 2016, pp. 31–60.

[37] J. Kilian, "A note on efficient zero-knowledge proofs and arguments (extended abstract)," in *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, 1992, pp. 723–732.

[38] M. Backes, A. Kate, and A. Patra, "Computational verifiable secret sharing revisited," in *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, ser. Lecture Notes in Computer Science, D. H. Lee and X. Wang, Eds., vol. 7073. Springer, 2011, pp. 590–609. [Online]. Available: https://doi.org/10.1007/978-3-642-25385-0_32

Appendix A. Ideal Functionalities for dCP and VRS

We present the ideal functionality of dCP in Figure 10 and VRS in Figure 11, as described in [14].

Appendix B. Building Block: Ligero Proof System

In this appendix, we give a self-contained description of the Ligero system [35]. This description is reused from FLAG [11]. For ease of exposition, we will describe it in the Interactive Oracle Proofs (IOP) model [36]. First, we recall some basic notation definitions of the codes used in the Ligero system.

Coding notation. For a code $C \subseteq \Sigma^n$ and vector $v \in \Sigma^n$, denote by $d(v, C)$ the minimal distance of v from C, namely the number of positions in which v differs from the closest codeword in C, and by $\Delta(v, C)$ the set of positions in which v differs from such a closest codeword (in case of ties, take the lexicographically first closest codeword), and by $\Delta(V, C) = \bigcup_{v \in V} \{\Delta(v, C)\}$. We further denote by $d(V, C)$ the minimal distance between a vector set V and a code C, namely $d(V, C) = \min_{v \in V}\{d(v, C)\}$. Our IOP protocol uses Reed-Solomon (RS) codes, defined next.

Definition B.1 (Reed-Solomon Code). For positive integers n, k, finite field $\mathbb{F}$, and a vector $\eta = (\eta_1, \ldots, \eta_n) \in \mathbb{F}^n$ of distinct field elements, the code $\mathsf{RS}_{\mathbb{F},n,k,\eta}$ is the $[n, k, n-k+1]$ linear code over $\mathbb{F}$ that consists of all n-tuples $(p(\eta_1), \ldots, p(\eta_n))$ where p is a polynomial of degree $< k$ over $\mathbb{F}$.

Definition B.2 (Encoded message). Let $L = \mathsf{RS}_{\mathbb{F},n,k,\eta}$ be an RS code and $\zeta = (\zeta_1, \ldots, \zeta_\ell)$ be a sequence of distinct elements of $\mathbb{F}$ for $\ell \leq k$. For $u \in L$ we define the message $\mathsf{Decode}_{L,\zeta}(u)$ to be $(p_u(\zeta_1), \ldots, p_u(\zeta_\ell))$, where p_u is the polynomial (of degree $< k$) corresponding to u. For $U \in L^m$ with rows $u^1, \ldots, u^m \in L$, we let $\mathsf{Decode}_{L^m,\zeta}(U)$ be the length-$m\ell$ vector $x = (x_{11}, \ldots, x_{1\ell}, \ldots, x_{m1}, \ldots, x_{m\ell})$ such that $(x_{i1}, \ldots, x_{i\ell}) = \mathsf{Decode}_{L,\zeta}(u^i)$ for $i \in [m]$. Finally, when ζ is clear from the context, we say that U *encodes* x if $x = \mathsf{Decode}_{L^m\zeta}(U)$. All our codes will employ the same $\mathbb{F}, n, \eta$ and we will simply refer the code by RS_k.

At a very high level, the Ligero IOP protocol proves the satisfiability of an arithmetic circuit C of size s in the following way. The prover arranges (a slightly redundant representation of) the s wire values of C on a satisfying assignment in a matrix, and encodes each row of this matrix using the Reed-Solomon code. The verifier challenges the prover to reveal linear combinations of the entries of the codeword matrix and checks their consistency with n_{open} randomly selected columns of this matrix.

For convenience, we provide a list of our parameters in Table 1.

TABLE 1. DESCRIPTION OF OUR PARAMETERS.

Parameter	Description
w_{ext}	Extended witness
U	Encoded extended witness
m	# of rows in the extended witness
ℓ	# of columns in the extended witness
s	Circuit size
n	Codeword length
n_{open}	# of queries on U
κ	Security parameter

Formal description of the Ligero IOP$(\mathrm{C}, \mathbb{F})$. This section provides a self-contained description of the Ligero IOP for an arithmetic circuit over a (sufficiently large) field $\mathbb{F}$. We remark that the exposition here is a variant of the system described in [35] that is optimized for a proof length and prover's computation.

- **Input:** The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ share a common input arithmetic circuit $\mathrm{C} : \mathbb{F}^N \to \mathbb{F}$ and input statement x. $\mathcal{P}$ additionally has input $\mathsf{w} = (\mathsf{w}_1, \ldots, \mathsf{w}_N)$ such that $\mathrm{C}(\mathsf{w}) = 1$. $\mathcal{P}$ and $\mathcal{V}$ agree on an encoding $\mathsf{RS}_{\mathbb{F},n,k,\eta}$ and ζ. In fact, we will assume there are public algorithms that can generate ζ and η given $\mathbb{F}, n$ and k.

Functionality $\mathcal{F}_{\mathsf{dCP}}$

$\mathcal{F}_{\mathsf{dCP}}$ runs among the Prover $\mathcal{P}$ and n Verifiers $\mathcal{V} = \{\mathcal{V}_1, \dots, \mathcal{V}_n\}$ and an adversary Sim. It is parameterized by n relations $(\mathcal{R}_1, \dots, \mathcal{R}_n)$. $\mathcal{F}_{\mathsf{dCP}}$ proceeds as follows.

Commit Phase : Upon receiving a message $(\mathsf{Commit}, \mathsf{sid}, \mathcal{P}, \mathsf{w})$ from the prover $\mathcal{P}$, record the values w and $\mathcal{P}$, and send the message $(\mathsf{receipt}, \mathsf{sid}, \mathcal{P})$ to the verifiers in $\mathcal{V}$ and Sim. (If a commit message has already been received, then ignore any other messages with the same sid.)

Prove Phase : Upon receiving a message $(\mathsf{Prove}, \mathsf{sid}, \mathcal{P}, \mathcal{V}_j, \mathsf{sh}_j)$ from the prover $\mathcal{P}$, then proceed as follows:

- If $(\mathsf{sh}_j, \mathsf{w}) \in \mathcal{R}_j$, send the message $(\mathsf{Proof}, \mathsf{sid}, \mathcal{P}, \mathcal{V}_j, \mathsf{sh}_j, \mathsf{accept})$ to the verifier $\mathcal{V}_j$ and Sim. $\mathcal{V}_j$ outputs $(\mathsf{accept}, \mathsf{sh}_j)$.
- Otherwise, send $(\mathsf{Proof}, \mathsf{sid}, \mathcal{P}, \mathcal{V}_j, \perp, \mathsf{reject})$ to the verifier $\mathcal{V}_j$ and Sim. $\mathcal{V}_j$ outputs reject.

Figure 10. Ideal Functionality for Distributed Commit-and-Prove

Functionality $\mathcal{F}_{\mathsf{VRS}}$

The functionality $\mathcal{F}_{\mathsf{VRS}}$ communicates with a dealer $\mathcal{D}$, a set of n verifiers $\mathcal{V} = \{\mathcal{V}_1, \dots, \mathcal{V}_n\}$, and an adversary $\mathcal{A}$. It is parameterized by a verifier corruption-threshold t, d is the length of input vector, and predicate $P : \mathbb{F}^d \to \{0, 1\}$.

Inputs. The Dealer $\mathcal{D}$ has input $x \in \mathbb{F}^d$ and randomness r_{vrs}. The verifiers $\mathcal{V}$ do not have any inputs. The dealer sends the message $(\mathsf{Input}, \mathsf{sid}, \mathcal{D}, x, r_{\mathsf{vrs}})$ to $\mathcal{F}_{\mathsf{VRS}}$.

Output. Upon receiving the input from $\mathcal{D}$, $\mathcal{F}_{\mathsf{VRS}}$ proceeds as follows.

- Compute the shares $(\mathsf{Share}_1, \dots, \mathsf{Share}_n) \leftarrow \mathsf{Enc}(x; r_{\mathsf{vrs}})$
- If $P(x) = 1$ holds, then send $(\mathsf{Output}, \mathsf{sid}, \mathcal{D}, \mathcal{V}_j, \mathsf{Share}_j, \ \mathsf{accept})$ to each verifier $\mathcal{V}_j$, who then outputs $(\mathsf{accept}, \ \mathsf{Share}_j)$, for all $i \in [n]$.
- Otherwise, send $(\mathsf{Output}, \mathsf{sid}, \mathcal{D}, \mathcal{V}_j, \perp, \ \mathsf{reject})$ to all the verifiers and the verifiers output $(\mathsf{reject}, \ \perp)$.

Figure 11. Ideal $\mathcal{F}_{\mathsf{VRS}}$ Functionality for Reed Solomon encoding

- **Oracle π:** Let m, ℓ be integers such that $m \cdot \ell > N + s$ where s is the number of multiplication gates in the circuit. For simplicity, we will assume N and s are multiples of ℓ. Then $\mathcal{P}$ generates an extended witness $\mathsf{w}_{\mathsf{ext}} \in \mathbb{F}^{m\ell}$ to be w concatenated with the internal wire values, namely $\mathsf{w}_1, \dots, \mathsf{w}_N, \alpha_1, \dots, \alpha_s, \beta_1, \dots, \beta_s, \gamma_1, \dots, \gamma_s$ where $(\alpha_i, \beta_i, \gamma_i)$ are the left input, right input and output values of the ith multiplication gate when evaluating $\mathrm{C}(\mathsf{w})$. All affine constraints on the wire values can be encoded via (A, b) where $A \in \mathbb{F}^{m\ell \times m\ell}, b \in \mathbb{F}^{m\ell}$ such that for any w that satisfies C, we have $A \cdot \mathsf{w} = b$.

  The prover samples a random codeword $U \in L^m$ where $L = \mathsf{RS}_k$ subject to $\mathsf{w} = \mathsf{Decode}_{L^m,\zeta}(U)$ where $\zeta = (\zeta_1, \dots, \zeta_\ell)$ is a sequence of distinct elements disjoint from $(\eta_1, \dots, \eta_n)$. $\mathcal{P}$ sets the oracle as $U \in L^m$. Depending on the context, we may view U either as a matrix in $\mathbb{F}^{m \times n}$ in which each row U_i is a purported L-codeword, or as a sequence of n symbols $(U[1], \dots, U[n])$, $U[j] \in \mathbb{F}^m$.
- **Interactive Protocol:**
  1) $\mathcal{V}$ picks randomness:
     a) **[Code test:]** $r_1 \in \mathbb{F}^m$,
     b) **[Linear test:]** $r_2 \in \mathbb{F}^{m\ell}$,
     c) **[Quadratic test:]** $r_3 \in \mathbb{F}^{s/\ell}$.

     and sends (r_1, r_2, r_3) to $\mathcal{P}$.
  2) $\mathcal{P}$ responds with $(q_{code}, q_{lin}, q_{quad})$ where:
     a) **[Code test:]** $q_{code} \in \mathbb{F}^n$ is computed as
     $$q_{code} = r_1^T \cdot U, \quad (1)$$
     b) **[Linear test:]** $q_{lin} \in \mathbb{F}^n$ is computed as
     $$q_{lin}[j] = (R_2[j])^T \cdot U[j] \quad (2)$$
     for $j \in [n]$ where R_2 is the unique matrix such that
     $$\mathsf{Decode}_{L_1^m,\zeta}(R_2) = r_2^T \cdot A \quad (3)$$
     where $L_1 = \mathsf{RS}_\ell$.
     c) **[Quadratic test:]** $q_{quad} \in \mathbb{F}^n$ is computed as
     $$q_{quad} = \sum_{i=1}^{s/\ell} (r_3)_i \cdot (U_{\mathsf{left}_i} \odot U_{\mathsf{right}_i} - U_{\mathsf{out}_i}) \quad (4)$$
     Recall that $\mathsf{Decode}_{L^m,\zeta}(U) = (\mathsf{w}_1, \dots, \mathsf{w}_N, \alpha_1, \dots, \alpha_s, \beta_1, \dots, \beta_s, \gamma_1, \dots, \gamma_s)$. Setting $(\mathsf{left}_i, \mathsf{right}_i, \mathsf{out}_i) = (\frac{N}{\ell} + i, \frac{N+s}{\ell} + i, \frac{N+2\cdot s}{\ell} + i)$ we have
     $$\mathsf{Decode}_{L,\zeta}(U_{\mathsf{left}_i}) = (\alpha_{\ell\cdot(i-1)+1}, \dots, \alpha_{\ell\cdot i})$$
     $$\mathsf{Decode}_{L,\zeta}(U_{\mathsf{right}_i}) = (\beta_{\ell\cdot(i-1)+1}, \dots, \beta_{\ell\cdot i})$$
     $$\mathsf{Decode}_{L,\zeta}(U_{\mathsf{out}_i}) = (\gamma_{\ell\cdot(i-1)+1}, \dots, \gamma_{\ell\cdot i})$$
  3) $\mathcal{V}$ queries a set $Q \subset [n]$ of n_{open} random symbols $U[j]$, $j \in Q$ and accepts iff the following conditions hold:
     a) **[Code test:]** q_{code} is a valid codeword, i.e. $q_{code} \in L$ and for every $j \in Q$, $q_{code}[j] = \sum_{i=1}^{m} (r_1)_i \cdot U_i[j]$.

b) **[Linear test:]** Let $v = \mathsf{Decode}_{L_2,\zeta}(q_{lin})$ where $L_2 = \mathsf{RS}_{k+\ell}$. Then the verifier checks if the values in v add up to $r_2^T \cdot b$, i.e. $\sum_{i=1}^{\ell} v_i = r_2^T \cdot b$ and for every $j \in Q$, $q_{lin}[j] = (R_2[j])^T \cdot U[j]$ where R_2 is as defined above (noting here that the verifier can locally compute R_2).

c) **[Quadratic test:]** Let $v' = \mathsf{Decode}_{L_3,\zeta}(q_{quad})$ where $L_3 = \mathsf{RS}_{2\cdot k}$. The verifier checks that every entry of v' is 0 and it holds that $q_{quad}[j] = \sum_{i=1}^{\widetilde{m}} (r_3)_i \cdot (U_{\mathsf{left}_i}[j] \cdot U_{\mathsf{right}_i}[j] - U_{\mathsf{out}_i}[j])$.

The soundness analysis has been argued in [16] and is formally stated in the following lemma,

Lemma B.3. Let e be a positive integer such that $e < d/3$ and suppose that there exists no $\overline{\alpha}$ such that $\mathrm{C}(\overline{\alpha}) = 1$. Then, for any maliciously formed oracle U^* and any malicious prover strategy, the verifier rejects except with at most $(d/|\mathbb{F}|)^{\sigma} + 2/|\mathbb{F}|^{\sigma'} + (1 - e/n)^{n_{\mathrm{open}}} + 2((e + 2k)/n)^{n_{\mathrm{open}}}$ probability where σ is the number of times the code test is repeated and σ' is the number of times the linear and quadratic tests are repeated.

Achieving Zero-knowledge. Note first that the verifier obtains two types of information in two different building blocks of the IPCP. First, it obtains linear combinations of codewords in a linear code L. Second, it probes a small number of symbols from each codeword. Since codewords are used to encode the NP witness, both types of information give the verifier partial information about the NP witness, and thus the basic IOP we described is not zero-knowledge. Fortunately, ensuring zero-knowledge only requires introducing small modifications to the construction and analysis. Specifically, the second type of "local" information about the codewords is made harmless by making the encoding randomized, so that probing just a few symbols in each codeword reveals no information about the encoded message. The high level idea for making the first type of information harmless is to use an additional random codeword for blinding the linear combination of codewords revealed to the verifier. However, this needs to be done in a way that does not compromise soundness.

Compiling the Ligero IOP to a SNARK. Compiling the IOP to a SNARK follows a standard compilation [37], [36] using commitments and Fiat-Shamir heuristic. In slightly more detail, generating a committing to the proof oracle proceeds as follows: (1) Compute the U matrix that is an encoding of $\mathsf{w}_{\mathsf{ext}}$ and (2) Compute a commitment to $U[j]$ for all $j \in [n]$[6]. The prover can commit and reveal specific locations of the proof oracle and the random oracle instantiated via a hash function to generate the verifier's random challenges and queries to the oracle and complete the execution (computing its own messages) based on the emulated verifier's messages. Namely, relying on Merkle-tree commitment for the proof oracle, including a challenge-response round at the end to reveal $U[j]$ $(j \in Q)$ using Merkle decommitments and Fiat-Shamir to generate the verifier's challenges in Round 1 and generate the set Q at the end.

[6] In Ligero, the n commitments are further used to build a Merkle tree and the root of the Merkle tree is used as the commitment of the proof oracle.

Appendix C. Verifiable Relation Sharing (VRS) Theorem

Theorem C.1. Let $t, n \in \mathbb{N}$ such that $t < n/3$ and P is a predicate. Then, the protocol Π_{VRS} between a dealer $\mathcal{D}$ and n verifiers $\mathcal{V}_1, \ldots, \mathcal{V}_n$ described in Figure 3 securely realizes $\mathcal{F}_{\mathsf{VRS}}$ functionality in the $\mathcal{F}_{\mathsf{dCP}}$-hybrid model where we instantiate the encoding scheme $\mathsf{Enc}(\cdot)$ via replicated secret sharing scheme parameterized by (n, t). The communication between the prover and each of the verifiers is $O\big(\mathsf{d} \cdot \binom{n-1}{t} + \rho + h\big)$ field elements. The total communication of all the servers in each phase is as follows:

- Offline phase: $O(n^3 + n \cdot \mathcal{BC}(n^2))$ field elements
- Online phase: $O\big(n \cdot \rho + n \cdot \mathcal{BC}(h) + n \cdot \mathsf{d} \cdot \mathcal{BC}(\binom{n-1}{t})\big)$ field elements
- Online phase in the optimistic setting[7] (where prover and all verifiers are honest): $O(n \cdot \mathsf{d} \cdot \binom{n-1}{t} + n \cdot \rho + n \cdot \mathcal{BC}(h))$ field elements

where κ is the security parameter, $\rho = \sqrt{(\mathsf{d} \cdot \binom{n-1}{t} + |P|) \cdot \kappa}$, $|P|$ is the number of gates in the circuit associated with predicate P and h is the output length of the hash function.

Communication efficiency. The prover sends to each of the verifiers the dCP proof, which costs $O(\sqrt{(\mathsf{d} \cdot \binom{n-1}{t} + |P|) \cdot \kappa} + h)$ and an input share of size $O(\mathsf{d} \cdot \binom{n-1}{t})$. The offline phase, run among the verifiers, involves n parallel invocations of VSS to secret-share one field element. We use the VSS scheme from [38], which has a communication cost of $O(n^2 + \mathcal{BC}(n^2))$ field elements to secret-share a single field element among n parties. Thus, the total offline communication is n times the cost of a single VSS. In the online phase, each verifier receives an input share and proof, then broadcasts the commitment in the optimistic case. In the worst case, share recovery will be triggered, and the verifiers will additionally broadcast the masked shares. The costs in the theorem are computed by summing these costs.

Appendix D. Experiment Parameters

The default values of the parameters, as used in our experiments (see Section 5.2) are presented in Table 2 and Table 3.

[7] Offline phase communication costs are independent of whether the setting is optimistic or not.

TABLE 2. EXPERIMENT PARAMETERS

Parameter	Description	Value
$\mathsf{n_s}$	# of servers	4
$\mathsf{t_s}$	# of malicious server	1
n_{open}	# of queries on U [a]	240
d	input length	see Table 3
m	# of rows in the extended witness[a]	see Table 3

[a] refer to Ligero proof generation in Appendix B

TABLE 3. NUMBER OF ROWS IN THE EXTENDED WITNESS (m) FOR VARIOUS INPUT LENGTHS (d)

d	10^0	10^1	10^2	10^3	10^4	10^5	10^6
m	1	2	4	8	20	100	2000

Appendix E. Additional Evaluation Results

To understand the computation cost when the number of servers increases, we measured the cost of generating and verifying proofs using 7 servers. Figure 12 presents the results. When input length is within 10,000, generating and verifying a proof is completed in 5 seconds, compared to 2 seconds when using four servers. Even with an input length of 10^5, the median time for a client to generate a proof is 35 seconds and for a server to verify it is 12 seconds, compared to 10 seconds and 6 seconds when using four servers. Overall, SCIF is adaptable to different numbers of servers. When the total number of servers increases, more malicious servers could be tolerated as long as $\mathsf{t_s} < \mathsf{n_s}/3$.

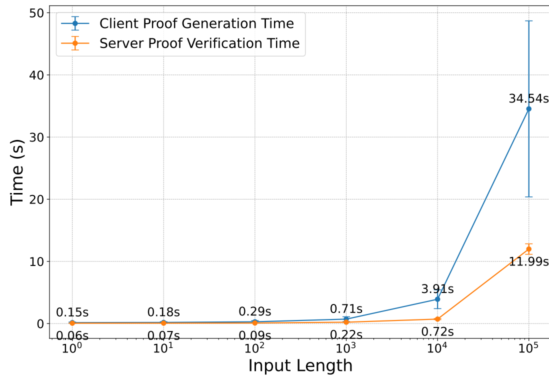

Figure 12. The time required for a client to generate a proof and for a server to verify a client's proof for various sized client inputs (in log-scale).