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

Jianan Su
Georgetown University

Laasya Bangalore
SandboxAQ*

Harel Berger
Ariel University*

Jason Yi
Georgetown University

Sophia Castor
Georgetown University

Micah Sherr
Georgetown University

Muthuramakrishnan Venkitasubramaniam
Georgetown University

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.

KEYWORDS

secure aggregation, multiparty computation, malicious security

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 [28], vitals in fitness trackers [24], various statistics from web browsers, and, more generally, federated learning [13, 35].

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 [20], which has been deployed in various real-world scenarios by organizations such as Apple, Google, Internet Services Research Group (ISRG), and Mozilla [19]. 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.

Prio is designed with a strong focus on privacy, ensuring that the inputs of honest clients remain protected even if all but one server is malicious. However, its security assumes that servers are only semi-honestly corrupted. Moreover, if even a single server crashes, all data is lost, causing the computation to abort.

*Most work was done during the author's time at Georgetown University.

Proceedings on Privacy Enhancing Technologies 2025(3), 1–21

https://doi.org/XXXXXXX.XXXXXXX

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

(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) One-shot client participation: Input clients are required to participate in at most one round of communication, ensuring resilience against unreliable client-side networks.

(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 $P : \mathbb{F}^m \rightarrow \{0, 1\}$ 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 [15, 16, 36].

(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.

The current state of affairs for secure aggregation is that there is no system that sufficiently meets all the desiderata. Broadly, secure aggregation has been studied in two settings: (1) the single-server setting, introduced by Bonawitz et al. [8, 9, 12, 15, 30, 40, 41], involves input parties that also serve as compute parties, organized in a star network topology, and (2) the multi-server setting, explored in works such as [1, 7, 20, 38, 39], delegates computation to an external set of servers, separating the roles of input and compute parties. However, both settings have limitations in the presence of malicious servers: the multi-server setting often lacks guaranteed output delivery, while the single-server setting may compromise guaranteed input inclusion[2].

This paper addresses these shortcomings by proposing a practical secure aggregation protocol that withstands both malicious clients and servers while fulfilling the above specified desiderata (a) - (f).

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 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 [6]. Introduced in the work by Applebaum et al. [4], 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. (2) All (honest) servers sum 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. (3) To ensure correct output reconstruction even in the presence of malicious servers, the underlying secret-sharing scheme must incorporate an error-correction 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 [6]. 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 [3] sublinear zero-knowledge argument system, which in turn yields the first concretely efficient implementation of a VRS (and consequently a VSS i.e., Verifiable Secret Sharing) protocol.

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 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.

[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.

[2] Also, the single-server setting typically requires multiple rounds of client participation, even when the central server is only semi-honest.

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 [22]—a popular anonymity service [32]—and show how SCIF can be used to safely learn information about the behavior of Tor nodes.

SCIF is available for download at https://github.com/GUSecLab/smc-in-a-box.

2 RELATED WORK

Single-server setting. Initial research developed secure aggregation protocols in the single-server setting designed to protect the privacy of client inputs while tolerating potential client dropouts [9, 12, 30, 40, 41]. 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 [7, 8, 15, 17] implemented these defenses using *input validation* mechanisms[3] 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, 15]. Others go further by identifying and excluding misbehaving clients from the aggregate (full security) [7, 8, 17]. Although these protocols handle malicious adversaries, they require multiple client-server interactions, which is impractical with unreliable clients. We aim for full security with just one client round, requiring only a single message to the server.

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, see the survey by Mansouri et al. [34].

Multi-server setting. In the multi-server setting, considerable research [1, 20, 38, 39] 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 [20] 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 [1, 20, 38, 39] enhance the original protocol. Prio+, a system introduced by [1], 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 [38], 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 [39], 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.

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.

Comparison with Flag [7]. The Flag secure aggregation system by Bangalore et al. [7] shows 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.

Comparison with Prime [6]. Bangalore et al. in Prime [6], introduced dCP and VRS functionalities as abstractions for robust secure aggregation, focusing on generic theoretical instantiations for these functionalities with an end goal of achieving provable differential privacy. In contrast, our work builds upon this abstraction and building concretely efficient instantiations for dCP and VRS. Specifically, we rely on Ligero to implement dCP and Replicated Secret Sharing (RSS) for VRS, whereas Prime extends [45] via generic techniques to realize dCP. In more detail, [45] constructs a VSS scheme using a distributed polynomial commitment scheme (GKR-based), which can then be adapted to support dCP. Our choice of Ligero for dCP is motivated by its efficiency and its natural alignment with the Commit-and-Prove paradigm (due to MPC-in-the-head). Recent works have demonstrated Ligero's space efficiency by implementing it on a browser [43] and its proof sizes can be improved to polylog($|C|$) instead of $\sqrt{|C|}$ [11] where $|C|$ is the circuit size. For VRS, we opted for RSS because it requires generating only a single seed in the offline phase, independent of the number of random shares needed during the online phase. Finally, we provide an end-to-end implementation of our system while [6] provides only asymptotic guarantees.

Comparison with Mario [37]. Concurrent work recently posted on ePrint, Mario [37], considers a similar setting as ours with honest

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

majority assumption among servers[4] and aims to get robustness against faulty inputs with guaranteed output delivery. However, the protocol as described currently does not achieve guaranteed output delivery. The authors have acknowledged this (in private communication) and are in the process of actively revising their submission and benchmarks.

Others. Honest-majority MPC protocols that guarantee output delivery, such as [23], are generally unsuitable for our requirements. These protocols are not optimized for lightweight (i.e., bandwidth-limited) clients as they rely on VSS protocols. Such VSS protocols often require clients to interact in multiple rounds, rely on public-key cryptography, or involve higher communication overhead among servers, making them impractical in our setting.

In FLP [14], the communication for proofs and verification scales as $\sqrt{|C|}$ for both client-to-server and server-to-server interactions, where $|C|$ represents the circuit size. Our protocol improves on this by limiting $\sqrt{|C|}$ communication for sending the proof from the client-to-server, with only a single hash broadcasted among servers for verification. On the other hand, MVZK's [44] proof size grows linearly with the circuit size, even for a constant number of servers. For predicates larger than the input size, our use of Ligero proofs offers better scalability.

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 O. For simplicity, we assume that each client $\mathcal{U}_i$ is 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. $\mathcal{BC}(n)$ represents the communication cost, measured in the number of field elements, required to broadcast a message of length n bits.

3.1 Our Model

We consider a synchronous network model involving $\mathsf{n_c}$ clients, $\mathsf{n_s}$ servers, and an output party 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. Additionally, we assume the presence of a broadcast channel among the servers [5]. 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. We consider a static and rushing adversary. A static adversary selects its corrupted parties before the protocol begins and does not change them during execution. A rushing adversary observes the messages of honest parties in each round and then sends its own messages, enabling it to adapt based on the honest parties' actions.

3.2 Secure Aggregation

We consider the problem of adding vectors over integers captured by a large enough field $\mathbb{F}$. 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, given in Figure 1. This functionality is parameterized by $\mathsf{n_c}$, the (maximum) number of clients, and d, the length of the 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 send the result to the output party.

3.3 Replicated Secret Sharing Scheme

We adopt the notation from [21] to describe the Replicated Secret Sharing (RSS) Scheme. RSS, introduced by [25], with threshold t, is defined by the following procedures[6]. 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 an 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 [21] 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. Also, let $k = k_{T_1} + \ldots + k_{T_\lambda}$ where $k_{T_1}, \ldots, k_{T_\lambda}$ are randomly sampled and are used to form an RSS sharing of k.

To generate an RSS sharing of a fresh random value r^ℓ, we derive values $r^\ell_{T_1}, \ldots, r^\ell_{T_\lambda}$ as follows:

[4] Our work assumes a strong honest majority among the servers, with fewer than one-third of servers being malicious, while [37] assumes a weaker honest majority, with fewer than half of the servers malicious.

[5] In our setting, where fewer than one-third of the servers are maliciously compromised, a broadcast channel can be efficiently simulated using point-to-point communication.

[6] 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 $\mathcal{F}_{\text{Agg}}$

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

(1) Upon receiving input ("Input", 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 ("Output", O) from the output party O, proceed as follows:
   - Compute the aggregate $\mathsf{Y} = \sum_{\mathcal{U}_i \in \mathcal{U}} P(X_i) \cdot X_i$ (note that this is equivalent to aggregating only inputs of clients that satisfy the predicate P).
   - Send ("Output", sid, Y) to the output party O and halt.

Figure 1: Ideal Functionality for Secure Aggregation with Input Certification

$$r_T^\ell = F_{k_T}(\ell)$$

for each subset T in the RSS scheme. The resulting sharing, denoted by $(\mathsf{rsh}_1^\ell, \ldots, \mathsf{rsh}_n^\ell)$, is structured such that the i^{th} share rsh_i^ℓ consists of all $r_{T_j}^\ell$ where $i \in T_j$, as defined in $\mathsf{Enc}(\cdot)$.

3.4 Zero-Knowledge Proofs

A *Zero-Knowledge Proof (ZKP)* is a cryptographic protocol that enables a prover to convince a verifier of the truth of a statement without revealing any additional information beyond the validity of the statement itself. Formally, a ZKP for a relation $\mathcal{R}$ consists of a tuple of probabilistic polynomial-time (PPT) algorithms:

Definition 3.1 (Zero-Knowledge Proof for Distributed Relations). Let $\mathcal{P}$ be a prover and $\mathcal{V} = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$ be a set of verifiers. The prover $\mathcal{P}$ holds an input x and a witness w, while each verifier $\mathcal{V}_i$ possesses a share sh_i. The goal is for $\mathcal{P}$ to convince each $\mathcal{V}_i$ that $(\mathsf{sh}_i, x, w) \in \mathcal{R}_i$ without revealing additional information about w.

A ZKP for the distributed relations $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ consists of three algorithms $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ defined as follows:

- **Setup:** $\mathsf{Setup}(1^\kappa) \to \mathsf{pp}$: A trusted setup generates the public parameters pp based on the security parameter κ.
- **Prove:** $\mathsf{Prove}(\mathsf{pp}, x, w) \to (\mathsf{com}, \pi_1, \pi_2^1, \ldots, \pi_2^n)$: Given public parameters pp, input x, and witness w, the prover $\mathcal{P}$ generates a proof consisting of a common component π_1 and verifier-specific components π_2^i for each verifier $\mathcal{V}_i$.
  For ease of use within dCP, the proof generation is divided into two steps, $\mathsf{Prove} = (\mathsf{Prove}_0, \mathsf{Prove}_1)$:
  - **Commitment Generation:** $\mathsf{Prove}_0(\mathsf{pp}, x, w) \to (\mathsf{com}, \mathsf{st})$: The prover first computes a commitment com to its input x and witness w and produces an intermediate state st.
  - **Proof Generation:** Using st, along with x and w, the prover generates the proof:
    $$\mathsf{Prove}_1(\mathsf{pp}, \mathsf{st}, x, w) \to (\pi_1, \pi_2^1, \ldots, \pi_2^n).$$
    where π_1 is the common portion of the proof given to all verifiers and π_2^i is a verifier-specific portion given to $\mathcal{V}_i$.
- **Verify:** Each verifier $\mathcal{V}_i$ runs
  $$\mathsf{Verify}(\mathsf{pp}, \mathsf{sh}_i, \mathsf{com}, \pi_1, \pi_2^i) \to \{0, 1\}.$$
  If the output is 1, $\mathcal{V}_i$ accepts; otherwise, it rejects.

A valid ZKP for distributed relations must satisfy the following properties:

- **Completeness:** If $(\mathsf{sh}_i, x, w) \in \mathcal{R}_i$ for all $i \in [n]$, then an honest prover $\mathcal{P}$ can convince all verifiers, i.e.,
  $$\Pr\left[\mathsf{Verify}(\mathsf{pp}, \mathsf{sh}_i, \mathsf{com}, \pi_1, \pi_2^i) = 1 \quad \forall i \in [n]\right] = 1.$$
- **Soundness:** No computationally bounded adversary (cheating prover) can convince any honest verifier $\mathcal{V}_i$ of $(\mathsf{sh}_i, x, w) \in \mathcal{R}_i$ unless it is true, except with negligible probability.
- **Zero Knowledge:** There exists a simulator that, given access only to sh_i, x, com, and verification keys, can generate proofs indistinguishable from those of an honest prover, ensuring no additional information about w is leaked.

Instantiation with Ligero. Ligero is a zero-knowledge proof system designed for efficient proof generation and verification using probabilistically checkable proofs (PCPs). Unlike traditional zk-SNARKs, Ligero does not require a trusted setup. The proof size is $\tilde{O}(\sqrt{|\mathsf{C}|})$ where C is the circuit representation of the relation $\mathcal{R}$. Due to these properties, Ligero is well-suited for privacy-preserving applications such as blockchain and secure computation. We use Ligero to instantiate zero-knowledge proofs for distributed relations, with further details provided in Appendix B.

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 an RSS 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 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. [45] who rely on 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 verifier $\mathcal{V}_j \in \mathcal{V}$.

(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 present a high-level overview of our dCP protocol, which is based on the Ligero zk-SNARK proof system. First, we establish the terminology for the circuit that will be used in our dCP construction. Intuitively, we construct a circuit C for the predicate P such that the C outputs the shares of the input x if the x satisfies the predicate and otherwise outputs $\perp$[7]. We repeat the definition verbatim from [6].

Definition 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 $(\mathsf{sh}_1, \ldots, \mathsf{sh}_n) \leftarrow \mathsf{Enc}(\mathsf{w})$.

Let us begin by considering how a prover with input w can prove to a single verifier, denoted $\mathcal{V}_j$, that two specific properties hold for w. A simple approach would be to use any zero-knowledge proof to demonstrate these properties to each verifier. However, this method is insufficient because a dishonest prover could use different inputs for different verifiers that satisfy the predicate — ensuring that each proof passes verification while leading to an inconsistent sharing among the verifiers. To prevent this, we must ensure that the prover uses the same input across all verifiers when generating both the shares and the proof. In other words, the shares must remain consistent with respect to a given encoding scheme across all verifiers as well as with the proof. To enforce this, we structure the proof in two parts, with the second part specifically verifying share consistency.

- A common proof, valid for all verifiers, which shows that the input w satisfies the predicate.
- A verifier-specific proof, which ensures that the same input w used in the common proof was also used to generate the shares for each verifier, and that these shares match the shares held by the verifiers. In particular, we need to ensure that $\mathsf{sh}_j = [C(\mathsf{w})]_j$ where share sh_j is possessed by verifier $\mathcal{V}_j$, and $[C(\mathsf{w})]_j$ is the j^{th} output of the circuit on input w.

The common portion of the proof can be generated using any zero-knowledge proof. We instantiate this using the Ligero proof system and show how to modify it to efficiently prove the verifier-specific portion of the proof. At a high-level, the prover can execute the Ligero proof generation with respect to the circuit C and input w. All constraints imposed by the circuit are enforced via code, linear, and quadratic tests (described in Appendix B.1). For the verifier-specific proof, we show that it suffices to augment the Ligero proof with an additional linear check enforcing the constraint $\mathsf{sh}_j = [C(\mathsf{w})]_j$ for every verifier $\mathcal{V}_j$. Details on proof generation and Ligero are in Appendix B.

However, naively repeating this proof generation process for each of the n verifiers would violate the zero-knowledge property. The additional linear test (in the verifier-specific portion of the proof) differs across verifiers, leading to different randomness generated for different verifiers when the Fiat-Shamir transformation[8] is applied. In more detail, this will result in different columns of the encoded extended witness[9][10] are opened to each verifier. Consequently, an adversary controlling up to t verifiers could observe too many columns, potentially leaking information about the witness w and violating the zero-knowledge property. To prevent this, we ensure that all verifiers generate a common randomness when applying the Fiat-Shamir transform - ensuring they open the same columns. We achieve this by constructing a Merkle tree, where the leaves consist of the outputs of the additional linear test, augmented with nonces for privacy. This Merkle root, say com', is then used in the Fiat-Shamir transform.

In summary, the additional linear test for share-consistency and the modification of the Fiat-Shamir transform allow us to extend

[7] While a simple approach might involve a circuit that outputs 0 or 1, we instead output the shares of x, as this facilitates consistency checks to verify the validity of the shares in subsequent steps.

[8] Used to obtain the non-interactive variant of the Ligero Proof system

[9] In the Ligero proof system, the prover generates an extended witness from the witness w and arranges it as a matrix. Each row of the matrix is then encoded using the Reed-Solomon codes, forming what we refer to this as the encoded extended witness.

[10] During Ligero proof generation, a column check is performed, where a random subset of columns of the encoded extended witness are opened and checked for consistency. In the non-interactive variant, the randomness determining which columns are opened is derived via the Fiat-Shamir transform.

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 $\mathsf{C} : \mathbb{F}^{n_1} \to \mathbb{F}^{n_2}$. The circuit C takes as input $\mathsf{w} = (\mathsf{x}, r_{\mathsf{dcp}})$ and produces the output $(\mathbf{out}_1, \ldots, \mathbf{out}_n)$, where each $\mathbf{out}_j = [\mathsf{C}(\mathsf{w})]_j$ represents the j^{th} component of the circuit's output.
We employ a zero-knowledge proof (ZKP) scheme with $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ algorithms that is instantiated with a slight variant of Ligero proof system. We assume that the prover and verifiers execute the Setup procedure to generate the public parameters pp. For details on the ZKP instantiation and a self-contained description of Ligero, refer to Appendix B.1.

Input & Output The prover has input $\mathsf{w} \in \mathbb{F}^{n_1}$ and the verifiers have no inputs. Each verifier $\mathcal{V}_j \in \mathcal{V}$ outputs either $(\mathbf{out}_j, \text{accept})$ or reject. If $\mathcal{V}_j$ outputs $(\mathbf{out}_j, \text{accept})$, then it must hold that $(\mathbf{out}_j, \mathsf{w}) \in \mathcal{R}_j$.

Commit Phase: This phase proceeds as follows.

(1) The prover $\mathcal{P}$ runs the algorithm $\mathsf{Prove}_0(\mathsf{pp}, \mathsf{w})$ to obtain the commitment portion, denoted as com, and a state st.
(2) $\mathcal{P}$ sends $(\mathsf{Commit}, \mathsf{sid}, \mathcal{P}, \mathsf{com})$ to all verifiers $\mathcal{V}$, where each verifier $\mathcal{V}_j \in \mathcal{V}$ receives a corresponding commitment com_j.
(3) Each verifier $\mathcal{V}_j$ broadcasts its received commitment com_j to all other verifiers.
(4) The verifiers collectively check whether there exists a commitment $\mathsf{com}^\star$ that matches at least $n - t$ of the broadcast commitments com_j. If no such $\mathsf{com}^\star$ is found, they set $\mathsf{com}^\star = \perp$.

Prove Phase: In this phase, the prover convinces the verifiers of the validity of its input with respect to their assigned relations.

(1) The prover $\mathcal{P}$ computes the shares $(\mathsf{sh}_1, \ldots, \mathsf{sh}_n) \leftarrow \mathsf{Enc}(\mathsf{w})$. If computed correctly, these shares correspond to the circuit output $(\mathbf{out}_1, \ldots, \mathbf{out}_n)$, as defined in Definition 4.1.
(2) To prove to each verifier $\mathcal{V}_j \in \mathcal{V}$ that $(\mathsf{sh}_j, \mathsf{w}) \in \mathcal{R}_j$, the prover runs $\mathsf{Prove}_1(\mathsf{pp}, \mathsf{st}, \mathsf{w})$ to generate the proof, consisting of a common portion π_1 and verifier-specific portions π_2^j.
(3) The prover then sends the proof (π_1, π_2^j) and the share sh_j to each verifier $\mathcal{V}_j$.
(4) Each verifier $\mathcal{V}_j$ verifies the proof by running
$$\mathsf{Verify}(\mathsf{pp}, \mathsf{sh}_j, \mathsf{com}^\star, \pi_1, \pi_2^j).$$
If verification succeeds (i.e., the output is 1), $\mathcal{V}_j$ outputs $(\text{accept}, \mathbf{out}_j)$. Otherwise, it outputs reject. Note that if $\mathsf{com}^\star = \perp$, the verification automatically fails.

Figure 2: A Distributed Commit-and-Prove Protocol

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. We provide more details of the instantiation in Appendix B.2.

Lastly, in the dCP protocol, the prover could broadcast the commitment to all verifiers during the commit phase. However, implementing such a broadcast would require the prover to participate in multiple rounds. Since our goal is to limit the prover's participation to a single round, we optimize the protocol by leveraging the broadcast channel between the verifiers. Instead of broadcasting the commitment directly, the prover sends it to each verifier individually over a point-to-point channel and then each verifier can broadcast the commitment to all other verifiers. The verifiers then collectively decide the output by verifying two conditions: (1) The proof verification passes. (2) The commitment received from the prover matches at least $n - t$ commitments broadcast by other verifiers. The verifiers output reject if these conditions fail.

The ideal functionality for 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. Next, we will establish the following definition to specify a set of relations $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ via a circuit C, which will be used in the dCP construction and formal theorem.

Definition 4.2 (Determination of Distributed Relations from Circuits). Consider a circuit $\mathsf{C} : \mathbb{F}^{\text{in}} \to \mathbb{F}^{\text{out}}$ that takes an input w and produces $(\mathbf{out}_1, \ldots, \mathbf{out}_n)$, where $\mathbf{out}_j = [C(\mathsf{w})]_j$ represents the j^{th} output of the circuit with input w. For each $j \in [n]$, we define a relation $\mathcal{R}_j$ as follows: A pair $(\mathbf{out}_j, \mathsf{w})$ *belongs to* $\mathcal{R}_j$ if and only if $[\mathsf{C}(\mathsf{w})]_j \neq \perp$. Furthermore, we say that a 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 established terminology, we now present an informal theorem for the dCP protocol. The full proof is provided in Appendix C.

Theorem 4.3. *(Informal) 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 realizes 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 in random oracle model.

The communication between the prover and verifier $\mathcal{V}_i$ is $O(|\mathbf{out}_i| + \sqrt{|\mathsf{C}| \cdot \kappa + h})$ field elements and each verifier broadcasts $\mathcal{BC}(h)$ 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 (in bits).

Communication efficiency. The prover initially sends the commitments com, the root of a Merkle tree which is h-bits long, incurring a cost of $O(h)$. 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. Then, each verifier broadcasts its received commitment to all other verifiers. Thus, the total costs match those outlined in Theorem 4.3. Additionally, we observe that the proof size scales with $\binom{n}{t}$, as the size of the circuit C depends on its output sh_j, which has a size of $\binom{n}{t}$.

4.2 Verifiable Relation Sharing (VRS) from dCP

The problem of Verifiable Relation Sharing (VRS), introduced by [4], 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 [6], which demonstrates the construction of a 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 [6], 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 [6]. The ideal VRS functionality is provided in Figure 11 of Appendix A and our VRS construction is given in Figure 3.

4.3 Secure Aggregation from VRS

This section presents our secure aggregation protocol, leveraging the VRS functionality discussed earlier. At its core, our protocol consists of two primary phases:

- Input Sharing: Clients share their input using the VRS functionality $\mathcal{F}_{\mathsf{VRS}}$. Here, each client acts as a prover and servers act as verifiers At the end of each invocation of $\mathcal{F}_{\mathsf{dCP}}$ by a client, the servers receive accept/reject along with the input share associated with this client.
- Output Reconstruction: The 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, who then error-corrects and reconstructs the final aggregate.

For completeness, we provide a detailed description of the protocol using VRS in Figure 4, as taken verbatim from [6]. The theorem statement is provided below.

Theorem 4.4. *(Informal) 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* O*, which is secure against a static, 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. 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* $\rho = \sqrt{(\mathsf{d} \cdot \binom{\mathsf{n_s}-1}{\mathsf{t_s}} + |P|) \cdot \kappa}$ *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:* $O(\mathsf{n_c} \cdot \mathsf{n_s} \cdot \rho + \mathsf{n_c} \cdot \mathsf{n_s} \cdot \mathcal{BC}(h) + \cdot \mathsf{n_c} \cdot \mathsf{n_s} \cdot \mathsf{d} \cdot \mathcal{BC}(\binom{\mathsf{n_s}-1}{\mathsf{t_s}}))$[11]
- *Online phase in the optimistic setting (when all the servers are honest) and* γ*-fraction of the clients are malicious:* $O(\mathsf{n_c} \cdot \mathsf{n_s} \cdot (\mathsf{d} \cdot \binom{\mathsf{n_s}-1}{\mathsf{t_s}} + \rho) + \mathsf{n_c} \cdot \mathsf{n_s} \cdot \mathcal{BC}(h) + \gamma \cdot \mathsf{n_c} \cdot \mathsf{n_s} \cdot \mathsf{d} \cdot \mathcal{BC}(\binom{\mathsf{n_s}-1}{\mathsf{t_s}}))$.[12]

where h *is the output length (in bits) of the hash function and* κ *is the security parameter.*

The costs in the above theorem are derived by scaling the costs in Theorem D.1 by a factor of $\mathsf{n_c}$. This theorem parallels Theorem 4.4 in [6], but incorporates replicated secret sharing and the Ligero proof system. For further details, refer to [6].

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://github.com/GUSecLab/smc-in-a-box.

We begin by describing SCIF's architecture and operation (Section 5.1). We then explore the operational costs of using SCIF via a

[11] Recall that $\mathcal{BC}(n)$ represents the communication cost, measured in the number of field elements, required to broadcast a message of length n bits.

[12] The main difference between optimistic and worst case settings is the additional γ factor in the second term of the cost expression where γ is the fraction of malicious clients, which is less than 1.)

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 $\mathcal{V} = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$ 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 $\mathsf{C} : \mathbb{F}^{t+1} \to \mathbb{F}^{n_2}$ and n relations $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ as per Definition 4.2. All the parties have access to a dCP ideal functionality $\mathcal{F}_{\mathsf{dCP}}$, which is parameterized by $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$.

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 $(\mathsf{Share}_1, \ldots, \mathsf{Share}_n) \leftarrow \mathsf{Enc}(x)$. Otherwise, all verifiers output reject.

Offline Phase. The parties interactively generate a valid RSS sharings of n keys $k^{(1)}, \ldots, k^{(n)}$ where each $k^{(i)}$ is associated with $\mathcal{V}_i$. Each verifier $\mathcal{V}_i \in \mathcal{V}$ performs the following steps:

(1) Acting as a dealer, secret share a random key, say $k^{(i)}$ among the verifiers using RSS.
(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 against whom a complaint was raised.

After completing these steps, each verifier holds RSS sharings of n keys $k^{(1)}, \ldots, k^{(n)}$, each originally shared by a different verifier. Whenever a fresh random RSS sharing is needed, the following procedure is performed:

(1) The existing RSS key shares can be used to derive n RSS sharings $\{(\mathsf{rsh}_1^{(i)}, \ldots, \mathsf{rsh}_n^{(i)}\}_{i \in n}$ (as described in Section 3.3).
(2) By summing up these n RSS shares locally (using the linearity property of RSS), we can generate the final random RSS sharing $(\mathsf{rsh}_1, \ldots, \mathsf{rsh}_n)$ non-interactively.

This process can be repeated as needed to generate an many random RSS sharings for use later during the sharing phase.

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: $(\mathsf{sh}_1, \ldots, \mathsf{sh}_n) \leftarrow \mathsf{Enc}(x; r_{\mathsf{vrs}})$. Let $(\mathsf{sh}_1, \ldots, \mathsf{sh}_n)$ 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 $(\mathsf{Masked\text{-}Share}, \mathsf{sid}, \mathcal{V}_j, \mathsf{msh}'_j)$. The verifiers perform a consistency check to ensure that the masked shares obtained from the verifiers' broadcasts, $(\mathsf{msh}'_1, \ldots, \mathsf{msh}'_n)$, form a valid encoding. This is verified by checking whether decoding succeeds on the collected shares $(\mathsf{msh}'_1, \ldots, \mathsf{msh}'_n)$.

(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 $\mathcal{V}_j$ outputs $(\mathsf{accept}, \mathsf{Share}_j)$ where Share_j is computed as follows:
(i) **Keep Existing Share:** If $\mathsf{happy}_j = \mathsf{accept}$, then set $\mathsf{Share}_j := \mathsf{sh}_j$, or
(ii) **Recover Share:** If $\mathsf{happy}_j = \mathsf{reject}$, then $\mathcal{V}_j$ needs to recover its share by computing $\mathsf{Share}_j := \mathsf{msh}''_j - \mathsf{rsh}_j$ where $(\mathsf{msh}''_1, \ldots, \mathsf{msh}''_n)$ is obtained by error-correcting $(\mathsf{msh}'_1, \ldots, \mathsf{msh}'_n)$.

Figure 3: A VRS Protocol for predicate P

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 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.

Input-Certified Secure Aggregation Protocol Π_{Agg}

Public parameters. Field $\mathbb{F}$, input vector length d, server corruption threshold t_s, predicate $P : \mathbb{F} \to \{0, 1\}$.
Parties. Clients $\mathcal{U} = \{\mathcal{U}_1, \ldots, \mathcal{U}_{n_c}\}$, servers $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_{n_s}\}$ and an output party O.
Input & Output. $\{x_1, \ldots, x_{n_c}\}$ where $x_i \in \mathbb{F}$ is client $\mathcal{U}_i$'s input. The output party 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 $\mathcal{F}_{\text{VRS}}$ with input $x_i \in \mathbb{F}$. This calls for the client $\mathcal{U}_i$ to send the tuple $(\text{Input}, \text{sid}, \mathcal{U}_i, x_i, r_{\text{vrs},i})$ to $\mathcal{F}_{\text{VRS}}$ where $r_{\text{vrs},i}$ is the randomness sampled by the $\mathcal{U}_i$.
(2) Each server $\mathcal{S}_j$ participates in $\mathcal{F}_{\text{VRS}}$ as a verifier and receives the tuple $(\text{Output}, \text{sid}, \mathcal{S}_j, \text{Share}_j^{(i)}, \text{happy}_j^{(i)})$ for all $j \in [n_s]$. As per the properties of the $\mathcal{F}_{\text{VRS}}$, all servers output the same happy bit i.e., $\text{happy}_j^{(i)} = \text{happy}_{j'}^{(i)}$ for $j, j' \in [n_s]$. So, we drop the subscript j while referring to the happy bit.

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

(1) At the end of all the invocations to $\mathcal{F}_{\text{VRS}}$, the servers define a set Valid to comprise of all clients $\mathcal{U}_i$ such that $\text{happy}^{(i)} = \text{accept}$.
(2) Each server $\mathcal{S}_j$ sums the shares it received from all clients in the set Valid i.e., $\text{osh}_j = \sum_{\mathcal{U}_i \in \text{Valid}} \text{Share}_j^{(i)}$ 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 $\text{osh}_j := \perp$. Finally, the output party error-corrects the vector $(\text{osh}_1, \ldots, \text{osh}_{n_s})$ to reconstruct Y and sets Y as the output.

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

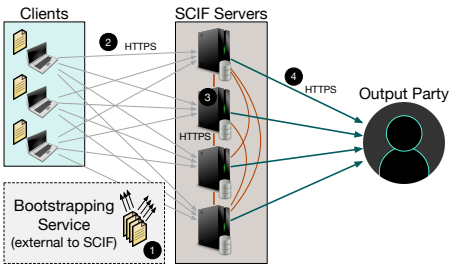


Figure 5: High-level architecture of SCIF. After bootstrapping (❶), clients send secret shares and proofs to each server (❷). Servers then construct their outputs (❸) and communicate their outputs to the output party (❹).

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 [29]; 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, $\top/\perp \leftarrow \text{auth}(\text{exp}, \text{cred})$, 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[13], 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.

[13]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.

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 an 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. The input validity predicate used in evaluation ensures that each input elements is a bit (i.e., 0 or 1). The implementation can be extended to arbitrary predicates represented as a circuit such as range checks or bounded norms. 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 E.

Proof generation and verification. The operations for generating and verifying proofs occur at the client and the servers respectively. 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 F, 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. We simulate malicious clients that send a malformed message, such as an input share or proof, to the server. Since our system's performance depends on the number of the clients for which the share recovery mechanism is triggered, we study the performance of our system by varying the percentage of clients involved and we trigger the share recovery by having malicious clients send malformed proofs to a server. We simulate various malicious server behaviors, such as dropout, sending malformed shares, failing to complain, and complaining unnecessarily. While we simulated these scenarios, we did not report them since they did not add overhead beyond what malicious clients caused, though they demonstrate our system can handle them.

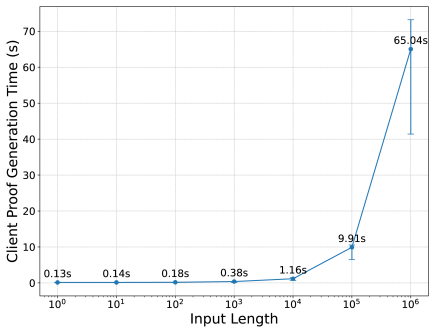
To evaluate SCIF's performance when a fraction of the clients behave maliciously and transmit malformed proofs to one of the servers, our experiments use four servers and 500 clients, the latter of which were distributed across six machines. Each of the client machines was provisioned with 16 cores and 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 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[14] 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.

[14] 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.



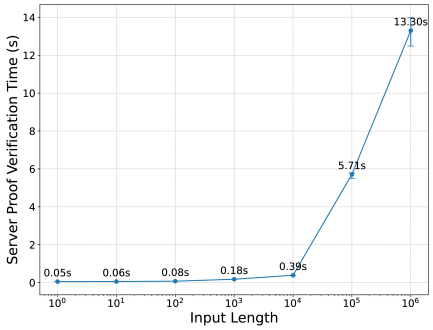
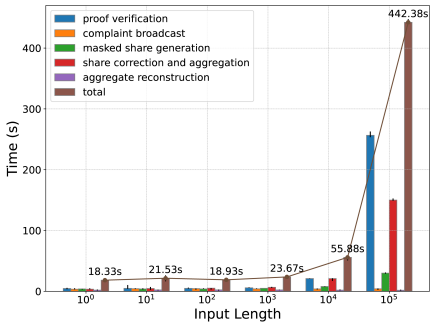


Figure 6: The time required for a client to generate a proof *(top)* and for a server to verify a client's proof *(bottom)* for various sized client inputs (in log-scale).



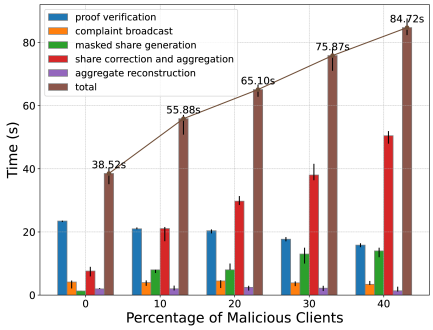
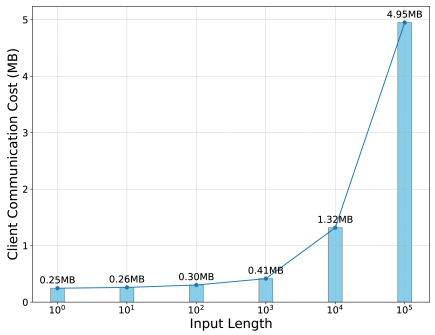


Figure 7: *Top:* Execution time (log-scale) with 500 clients and varying client input lengths, when 50 clients (10%) submit invalid proofs. *Bottom:* Execution time for various percentages of malicious clients, and a client input length of 10^4.



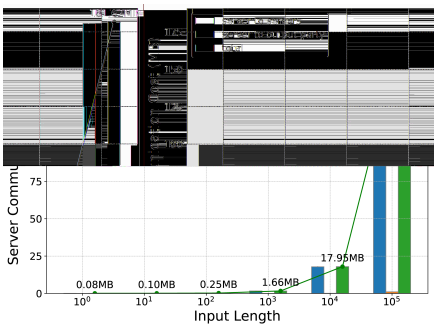


Figure 8: *Top:* Client communication cost as the input length varies (log-scale). *Bottom:* Server communication cost for 500 clients with varied input lengths (log-scale). Fifty (10%) of the clients are malicious.

	Client		Server	
	Runtime	Comm.	Runtime	Comm.
Prio [20]	3.64s	1.9MB	14s	0.45MB
SCIF	1.16s	1.32MB	38.52s (55.88s)	0.2MB (17.95MB)

Table 1: Comparison of Prio with SCIF. Result for Prio shows semi-honest performance with input validation, while the result for SCIF is malicious performance with input validation. For both systems, we set $n_s = 4$, $n_c = 500$ and $d = 10^4$. For SCIF, values within the parentheses are reflective of when 50 clients (10%) submit invalid inputs; otherwise, it shows when all clients submit valid inputs.

Comparison with Prio. Previous multi-server protocols such as Prio [20] and Elsa [38] validate inputs in the presence of semi-honest servers. We compare SCIF to Prio, as both implementations support verifying whether inputs are 0 or 1, but not to Elsa, which does not support such input validation. We use the Prio [18] prototype in Go from the Prio authors and configure it to evaluate runtime and communication cost of the clients and servers.

Table 1 shows the results of our comparison. The client runtime in SCIF is about 3× faster than Prio, and each client sends slightly less data to the servers in SCIF compared to Prio. However, SCIF's server runtime is slower due to additional procedures for input validation in the presence of malicious servers, while Prio only handles semi-honest servers. Since Prio's implementation does not simulate malicious client behavior, we compare server communication cost assuming all clients behave honestly. In this scenario, the data sent by each server in Prio is about 2× more than SCIF. In SCIF, the server communication cost increases with the number of clients for which share recovery is triggered, during which all servers need to broadcast the masked shares associated with a specific client. Assuming semi-honest corruptions of the servers, then the server communication cost increases with the number of malicious clients.

5.3 Simulation Study: Safely Measuring Tor

As a case study, we use SCIF to measure the performance of nodes in a simulated Tor network [22]. 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 its adversarial model assumes both malicious users and relays [42]. Tor's Directory Authorities are not assumed to be honest and Tor uses a consensus protocol to handle potential misbehavior. Adding measurement

capabilities to Tor requires robust security protections and the avoidance of trusted parties. SCIF fits this by tolerating misbehavior from users, as well as from relays or Directory Authorities acting as the parties to conduct the measurements, while ensuring output delivery despite such misbehavior.

We perform measurements on a private Tor network which we deploy in Shadow [26, 27], 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. Notably, Shadow runs unmodified binaries, including Tor and SCIF, on a virtualized networking layer without requiring any modifications to either.

As a proof of concept, 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 [26] 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 [22].

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 AND FUTURE WORK

In this work, we presented a practical secure aggregation protocol with input validation that ensures output delivery and input inclusion in the presence of malicious servers and malicious clients. Our end-to-end system implementation is lightweight, and the evaluation in a real-world setting demonstrates client overhead remains

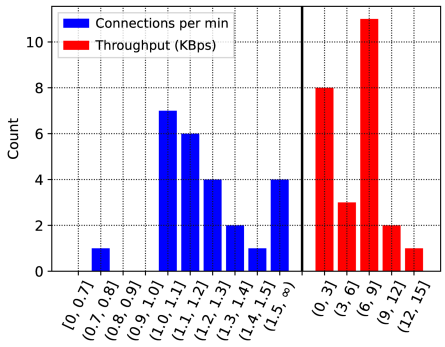


Figure 9: Histogram of the number of connections per minute observed by Tor exit relays *(left)* and the observed throughput of the exit relays *(right)*.

minimal, a critical factor for practical applications. Although the overhead of servers rises to ensure full security, it remains moderate and manageable in practice. Here are a few directions for improving and extending our work in the future.

Improving implementation scalability. Our current implementation is not memory-optimized and stores all shares and proofs in memory, which imposes a limitation on the server's computational resource, resulting in reduced efficiency. One possible solution is using in-memory (but disk-backed) solutions (e.g.,via Redis) to enable larger testing setups, and better memory management to reduce the state that must be maintained. The other solution is to leverage improved Ligero ZK system such as Ligetron [43], which scales to billions of gates and run efficiently even inside of a browser.

Deployment for real-world applications. While privacy preserving statistics collection have many uses, we focus on two for SCIF. 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 will explore these and other use cases.

ACKNOWLEDGMENTS

This work was partially funded by Fritz Family Fellowships, the Callahan Family Chair Fund, the Farr Faculty Excellence Award, and the Google Cloud Research Credit program. The opinions and findings expressed in this paper are those of the authors and do not necessarily those of any employer or funding agency.

REFERENCES

[1] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. 2021. Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares. Cryptology ePrint Archive, Report 2021/576. https://eprint.iacr.org/2021/576.

[2] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[3] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2022. Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. Cryptology ePrint Archive, Paper 2022/1608. https://doi.org/10.1145/3133956

[4] Benny Applebaum, Eliran Kachlon, and Arpita Patra. 2022. Verifiable Relation Sharing and Multi-Verifier Zero-Knowledge in Two Rounds: Trading NIZKs with Honest Majority. *IACR Cryptol. ePrint Arch.* (2022), 167. https://eprint.iacr.org/2022/167

[5] Michael Backes, Aniket Kate, and Arpita Patra. 2011. 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 (Lecture Notes in Computer Science, Vol. 7073)*, Dong Hoon Lee and Xiaoyun Wang (Eds.). Springer, 590–609. https://doi.org/10.1007/978-3-642-25385-0_32

[6] Laasya Bangalore, Albert Cheu, and Muthuramakrishnan Venkitasubramaniam. 2024. PRIME: Differentially Private Distributed Mean Estimation with Malicious Security. *IACR Cryptol. ePrint Arch.* (2024), 1771. https://eprint.iacr.org/2024/1771

[7] Laasya Bangalore, Mohammad Hossein Faghihi Sereshgi, Carmit Hazay, and Muthuramakrishnan Venkitasubramaniam. 2023. 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*, Joseph K. Liu, Yang Xiang, Surya Nepal, and Gene Tsudik (Eds.). ACM, 14–28. https://doi.org/10.1145/3579856.3595805

[8] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. 2022. ACORN: Input Validation for Secure Aggregation. *IACR Cryptol. ePrint Arch.* (2022), 1461. https://eprint.iacr.org/2022/1461

[9] James Henry Bell, Kallista A. Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. 2020. Secure Single-Server Aggregation with (Poly)Logarithmic Overhead. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3372297.3417885

[10] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. 2016. Interactive Oracle Proofs. In *TCC*. 31–60.

[11] Rishabh Bhadauria, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Wenxuan Wu, and Yupeng Zhang. 2023. Private Polynomial Commitments and Applications to MPC. In *Public-Key Cryptography - PKC 2023 - 26th IACR International Conference on Practice and Theory of Public-Key Cryptography, Atlanta, GA, USA, May 7-10, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13941)*, Alexandra Boldyreva and Vladimir Kolesnikov (Eds.). Springer, 127–158. https://doi.org/10.1007/978-3-031-31371-4_5

[12] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[13] Kallista A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3133956.3133982

[14] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2019. Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 11694)*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer, 67–97. https://doi.org/10.1007/978-3-030-26954-8_3

[15] Lukas Burkhalter, Hidde Lycklama, Alexander Viand, Nicolas Küchler, and Anwar Hithnawi. 2021. RoFL: Attestable Robustness for Secure Federated Learning. *CoRR* abs/2107.03311 (2021). arXiv:2107.03311 https://arxiv.org/abs/2107.03311

[16] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. 2017. Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning. *CoRR* abs/1712.05526 (2017). arXiv:1712.05526 http://arxiv.org/abs/1712.05526

[17] Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. 2022. 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*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 2535–2549. https://doi.org/10.1145/3548606.3560611

[18] Henry Corrigan-Gibbs. 2017. Prototype implementation of Prio. Available at https://github.com/henrycg/prio/tree/master.

[19] Henry Corrigan-Gibbs and Dan Boneh. 2017. Henry Corrigan-Gibbs' web page, Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. https://people.csail.mit.edu/henrycg/pubs/nsdi17prio/. Accessed: 01-05-2022.

[20] Henry Corrigan-Gibbs and Dan Boneh. 2017. 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*, Aditya Akella and Jon Howell (Eds.). USENIX Association, 259–282. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs

[21] Anders P. K. Dalskov, Daniel Escudero, and Ariel Nof. 2022. 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*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 653–666. https://doi.org/10.1145/3548606.3559389

[22] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium (USENIX)*.

[23] Vipul Goyal, Yifan Song, and Chenzhi Zhu. 2020. Guaranteed Output Delivery Comes Free in Honest Majority MPC. In *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12171)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, 618–646. https://doi.org/10.1007/978-3-030-56880-1_22

[24] Andrew Hilts, Christopher Parsons, and Jeffrey Knockel. 2016. Every step you fake: a comparative analysis of fitness tracker privacy and security. Presented at Open Effect.

[25] Mitsuru Ito, Akira Saito, and Takao Nishizeki. 1989. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)* 72, 9 (1989), 56–64.

[26] Rob Jansen and Nicholas Hopper. 2012. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Network and Distributed System Security Symposium (NDSS)*.

[27] Rob Jansen, Jim Newsome, and Ryan Wails. 2022. Co-opting Linux Processes for High-Performance Network Simulation. In *USENIX Annual Technical Conference*.

[28] Tobias Jeske. 2013. Floating Car Data from Smartphones: What Google and Waze Know About You and How Hackers Can Control Traffic. Presented at BlackHat Europe.

[29] Jinzhu. . GORM: The fantastic ORM library for Golang. Available at https://gorm.io.

[30] Swanand Kadhe, Nived Rajaraman, Onur Ozan Koyluoglu, and Kannan Ramchandran. 2020. FastSecAgg: Scalable Secure Aggregation for Privacy-Preserving Federated Learning. *CoRR* abs/2009.11248 (2020). arXiv:2009.11248 https://arxiv.org/abs/2009.11248

[31] Joe Kilian. 1992. A Note on Efficient Zero-Knowledge Proofs and Arguments (Extended Abstract). In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*. 723–732.

[32] Akshaya Mani, T. Wilson Brown, Rob Jansen, Aaron Johnson, and Micah Sherr. 2018. Understanding Tor Usage with Privacy-Preserving Measurement. In *ACM SIGCOMM Conference on Internet Measurement (IMC)*.

[33] Akshaya Mani and Micah Sherr. 2017. Historeː Differentially Private and Robust Statistics Collection for Tor. In *Network and Distributed System Security Symposium (NDSS)*.

[34] Mohamad Mansouri, Melek Önen, Wafa Ben Jaballah, and Mauro Conti. 2023. SoK: Secure Aggregation Based on Cryptographic Schemes for Federated Learning. *Proc. Priv. Enhancing Technol.* 2023, 1 (2023), 140–157. https://doi.org/10.56553/POPETS-2023-0009

[35] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *International Conference on Artificial Intelligence and Statistics (AISTATS) (Proceedings of Machine Learning Research, Vol. 54)*, Aarti Singh and Xiaojin (Jerry) Zhu (Eds.). 1273–1282.

[36] Thien Duc Nguyen, Phillip Rieger, Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. 2021. FLGUARD: Secure and Private Federated Learning. *CoRR* abs/2101.02281 (2021). arXiv:2101.02281 https://arxiv.org/abs/2101.02281

[37] Truong Son Nguyen, Tancrède Lepoint, and Ni Trieu. 2024. Mario: Multi-round Multiple-Aggregator Secure Aggregation with Robustness against Malicious Actors. *IACR Cryptol. ePrint Arch.* (2024), 1428. https://eprint.iacr.org/2024/1428

[38] Mayank Rathee, Conghao Shen, Sameer Wagh, and Raluca Ada Popa. 2023. 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, 1961–1979. https://doi.org/10.1109/SP46215.2023.10179468

[39] Mayank Rathee, Yuwen Zhang, Henry Corrigan-Gibbs, and Raluca Ada Popa. 2024. Private Analytics via Streaming, Sketching, and Silently Verifiable Proofs. *IACR Cryptol. ePrint Arch.* (2024), 666. https://eprint.iacr.org/2024/666

[40] Jinhyun So, Basak Guler, and Amir Salman Avestimehr. 2020. Turbo-Aggregate: Breaking the Quadratic Aggregation Barrier in Secure Federated Learning. *IACR Cryptol. ePrint Arch.* 2020 (2020), 167. https://eprint.iacr.org/2020/167

[41] Jinhyun So, Corey J. Nolet, Chien-Sheng Yang, Songze Li, Qian Yu, Ramy E. Ali, Basak Guler, and Salman Avestimehr. 2022. 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*, Diana Marculescu, Yuejie Chi, and Carole-Jean Wu (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2022/hash/6c44dc73014d66ba49b28d483a8f8b0d-Abstract.html

[42] Ryan Wails, Aaron Johnson, Daniel Starin, Arkady Yerukhimovich, and S. Dov Gordon. 2019. Stormy: Statistics in Tor by Measuring Securely. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. https://doi.org/10.1145/3319535.3345650

[43] Ruihan Wang, Carmit Hazay, and Muthuramakrishnan Venkitasubramaniam. 2023. 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, 86–86.

[44] Kang Yang and Xiao Wang. 2022. Non-interactive Zero-Knowledge Proofs to Multiple Verifiers. In *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 13793)*, Shweta Agrawal and Dongdai Lin (Eds.). Springer, 517–546. https://doi.org/10.1007/978-3-031-22969-5_18

[45] Jiaheng Zhang, Tiancheng Xie, Thang Hoang, Elaine Shi, and Yupeng Zhang. 2022. 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*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 2965–2982. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-jiaheng

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 [6].

B LIGERO PROOF SYSTEM OVERVIEW AND INSTANTIATIONS

B.1 Ligero Proof System Overview

In this appendix, we give a self-contained description of the Ligero system [2]. This description is reused from FLAG [7]. For ease of exposition, we will describe it in the Interactive Oracle Proofs (IOP) model [10]. 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 2.

Table 2: 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 [2] 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.
- **Oracle:** 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 $\mathsf{w}_{\text{ext}} \in \mathbb{F}^{m\ell}$ to be w concatenated with the internal wire values, namely $\mathsf{w}_1, \ldots, \mathsf{w}_N, \alpha_1, \ldots, \alpha_s, \beta_1, \ldots, \beta_s, \gamma_1, \ldots, \gamma_s$ where $(\alpha_i, \beta_i, \gamma_i)$ are the left input, right input and output values of the i^{th} 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 $\zeta = (\zeta_1, \ldots, \zeta_\ell)$ is a sequence of distinct elements disjoint from $(\eta_1, \ldots, \eta_n)$. $\mathcal{P}$ 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 $(U[1], \ldots, U[n])$, $U[j] \in \mathbb{F}^m$.
- **Interactive Protocol:**

(1) $\mathcal{V}$ picks randomness:

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

$\mathcal{F}_{\mathsf{dCP}}$ runs among the Prover $\mathcal{P}$ and n Verifiers $\mathcal{V} = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$ and an adversary Sim. It is parameterized by n relations $(\mathcal{R}_1, \ldots, \mathcal{R}_n)$. $\mathcal{F}_{\mathsf{dCP}}$ 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 $\mathcal{V} = \{\mathcal{V}_1, \ldots, \mathcal{V}_n\}$, 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 $(\mathsf{Share}_1, \ldots, \mathsf{Share}_n) \leftarrow \mathsf{Enc}(x; r_{\mathsf{vrs}})$
- 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

(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 $(q_{\mathsf{code}}, q_{\mathsf{lin}}, q_{\mathsf{quad}})$ where:

(a) **[Code test:]** $q_{\mathsf{code}} \in \mathbb{F}^n$ is computed as

$$q_{\mathsf{code}} = r_1^T \cdot U, \tag{1}$$

(b) **[Linear test:]** $q_{\mathsf{lin}} \in \mathbb{F}^n$ is computed as

$$q_{\mathsf{lin}}[j] = (R_2[j])^T \cdot U[j] \tag{2}$$

for $j \in [n]$ where R_2 is the unique matrix such that

$$\mathsf{Decode}_{L_1^m, \zeta}(R_2) = r_2^T \cdot A \tag{3}$$

where $L_1 = \mathsf{RS}_\ell$.

(c) **[Quadratic test:]** $q_{\mathsf{quad}} \in \mathbb{F}^n$ is computed as

$$q_{\mathsf{quad}} = \sum_{i=1}^{s/\ell} (r_3)_i \cdot (U_{\mathsf{left}_i} \odot U_{\mathsf{right}_i} - U_{\mathsf{out}_i}) \tag{4}$$

Recall that $\mathsf{Decode}_{L^m, \zeta}(U) = (\mathsf{w}_1, \ldots, \mathsf{w}_N, \alpha_1, \ldots, \alpha_s, \beta_1, \ldots, \beta_s, \gamma_1, \ldots, \gamma_s)$. 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

$$\mathsf{Decode}_{L,\zeta}(U_{\mathsf{left}_i}) = (\alpha_{\ell\cdot(i-1)+1}, \ldots, \alpha_{\ell\cdot i})$$
$$\mathsf{Decode}_{L,\zeta}(U_{\mathsf{right}_i}) = (\beta_{\ell\cdot(i-1)+1}, \ldots, \beta_{\ell\cdot i})$$
$$\mathsf{Decode}_{L,\zeta}(U_{\mathsf{out}_i}) = (\gamma_{\ell\cdot(i-1)+1}, \ldots, \gamma_{\ell\cdot i})$$

(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. $q_{\mathsf{code}} \in L$ and for every $j \in Q$, $q_{\mathsf{code}}[j] = \sum_{i=1}^m (r_1)_i \cdot U_i[j]$.

(b) **Linear test:** Let $v = \mathsf{Decode}_{L_2,\zeta}(q_{\mathsf{lin}})$ 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 $j \in Q$, $q_{\mathsf{lin}}[j] = (R_2[j])^T \cdot U[j]$ where R_2 is as defined above (noting here that the verifier can locally compute R_2).

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

The soundness analysis has been argued in [3] 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 $(d/|\mathbb{F}|)^\sigma + 2/|\mathbb{F}|^{\sigma'} + (1 - e/n)^{n_{open}} + 2((e+2k)/n)^{n_{open}}$ 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 [10, 31] using commitments and Fiat-Shamir heuristic. In slightly more detail, generating a commitment 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 $j \in [n]$[15]. 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.

B.2 Instantiating ZKP for Distributed Relation

We instantiate the zero-knowledge proof (ZKP) for distributed relations using the Ligero proof system. Below, we describe the setup, proof generation, and verification procedures.

- **Setup:** The public parameters pp comprise the hash function and error-correcting codes used in proof generation, following the Ligero framework.
- **Proof Generation:** The proof consists of three components:
  (1) com: Commitment to the proof oracle.
  (2) π_1: Common proof shared among all verifiers.
  (3) $(\pi_2^1, \ldots, \pi_2^n)$: Verifier-specific proof whre π_2^j is sent to verifier $\mathcal{V}_j$

  We now describe the algorithms $(\mathsf{Prove}_0, \mathsf{Prove}_1)$ that generate the above mentioned three components.
  - Prove_0 generates the commitment to the proof oracle by using a standard transformation of an IOP to SNARKs (briefly described in Appendix B.1). An overview is provided below:
    * Compute the extended witness $\mathsf{w}_{\mathsf{ext}}$.
    * Encode $\mathsf{w}_{\mathsf{ext}}$ row-wise using Reed-Solomon coding.
    * Commit to the encoded witness $\mathsf{w}_{\mathsf{ext}}$ column-wise using a Merkle-tree-based hash. Each column forms a leaf node, and the root com serves as the proof oracle commitment.

    Note that Prove_0 also outputs state information st for use by Prove_1 which includes the randomness used to generate the Merkle-tree based commitment.
  - Prove_1 generates the common portion π_1. This portion is shared by all verifiers and ensures that the input x satisfies the predicate P and the standard Ligero proof system can be used as is to prove this. As per the Ligero specification, π_1 comprises of:
    * Standard Ligero checks for the circuit C: code, linear, quadratic, and linear-share tests.
    * Column consistency check and authentication paths for opening committed columns.
  - Prove_1 also generates the Verifier-specific portion $((\pi_2^1, \ldots, \pi_2^n))$ of the proof. Individual proofs π_2^j are sent to each verifier $\mathcal{V}_j$ and ensure share consistency between sh_j (received by verifier $\mathcal{V}_j$) and the computed output $\mathbf{out}_j = [C(\mathsf{w})]_j$, which is included in the extended witness. Broadly, π_2^j is generated using the following steps (and deviates slightly from Ligero):
    * Conduct an additional linear test to verify that $\mathsf{sh}_j = \mathbf{out}_j$, where sh_j is the received share and $\mathbf{out}_j$ is the circuit output.
    * Construct a second Merkle tree with leaves corresponding to the outputs of the additional linear test (augmented with a nonce for privacy), generating a new root com′.
    * Provide authentication paths from the new root com′ to each verifier, ensuring the integrity of their respective proof components.
- **Verification:** Each verifier $\mathcal{V}_j$ runs:

  $$\mathsf{Verify}(\mathsf{pp}, \mathsf{sh}_j, \mathsf{com}, \pi_1, \pi_2^j) \to \{0, 1\}$$

  Verification follows Ligero to verify both the common and verifier-specific portions of the proof. All the tests (including the additional linear check in π_2^j) is verified by following the Ligero verification procedure. Additionally, the authentication paths provided for the additional linear test with respect to commitment com′ is also verifier.

C PROOF OF THEOREM 4.3

Let $\mathcal{A}$ represent a malicious probabilistic polynomial-time adversary in the real model. We describe an ideal model adversary $\mathcal{S}im$ which simulates the real execution of the protocols Π_{dCP} with $\mathcal{A}$ such that no environment $\mathcal{Z}$ can distinguish Π_{dCP} with $\mathcal{A}$ from the ideal model with $\mathcal{S}im$ and $\mathcal{F}_{\mathsf{dCP}}$. $\mathcal{S}im$ invokes a copy of $\mathcal{A}$ internally and then simulates the interaction between $\mathcal{A}$ and Z. We provide a detailed description of $\mathcal{S}im$ below, covering both the honest prover and corrupted prover scenarios. Let C denote the set of verifiers corrupted by $\mathcal{A}$.

C.1 Case I: Prover is Honest

Simulating the communication with $\mathcal{Z}$: When $\mathcal{Z}$ writes a value on $\mathcal{S}im$'s input tape, $\mathcal{S}im$ writes this value on $\mathcal{A}$'s input tape. The values on $\mathcal{A}$'s output tape are copied to $\mathcal{S}im$'s output tape.

Finally, Sim sends $\widetilde{\mathsf{com}}$[16] to the verifiers on behalf of the honest prover $\mathcal{P}$.

Simulating the Prove Phase: Whenever an honest $\mathcal{P}$ sends the $(\text{Prove}, sid, \mathcal{P}, \mathcal{V}_j, \mathsf{sh}_j)$ message to $\mathcal{F}_{\mathsf{dCP}}$, Sim receives the message $(\text{Proof}, sid, \mathcal{P}, \mathcal{V}_j, \mathsf{sh}_j, \text{accept})$ from $\mathcal{F}_{\mathsf{dCP}}$ on behalf of $\mathcal{V}_j \in C$. For each $\mathcal{V}_j \in C$, then Sim internally sends the message $(\mathsf{sh}_j, \widetilde{\pi}_j)$ to

[15] 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.

[16] The tilde notation indicates the simulated versions of the variable.

$\mathcal{A}$ on behalf of $\mathcal{P}$, where $\widetilde{\pi}_j$ is the simulated proof associated with verifier $\mathcal{V}_j \in C$. Since the dCP is built upon the Ligero proof system, we first outline how to adapt the Ligero simulator to ensure zero-knowledge in our setting. Here, the simulator must generate proofs for each corrupt verifier.

Recall that our approach deviates from the standard Ligero proof system in two key ways. First, the prover sends a distinct proof to each verifier, differing only in the output of an additional linear test that we introduce. The common portion of the proof, shared across all verifiers, is simulated as in Ligero and remains unchanged for all verifiers. Before simulating the additional linear test, we first generate simulated shares for the malicious verifiers, denoted as $\{\widetilde{\mathsf{sh}}_i\}_{i\in C}$. These shares are chosen randomly in a way that ensures consistency under the sharing scheme (RSS, in our case) but remains independent of the actual input. The additional linear test for each corrupt verifier is then simulated similarly to Ligero's linear test, ensuring consistency with the newly generated shares.

Second, the outputs of these additional linear tests are committed using a Merkle tree. Specifically, the Merkle root, the linear test outputs, and their corresponding authentication paths are revealed to their respective verifiers. To simulate the Merkle root, denoted as $\widetilde{\mathsf{com}}'$, we proceed as follows: the leaf nodes corresponding to malicious verifiers are computed correctly with respect to the simulated shares and their associated linear test outputs, with nonces added. Meanwhile, the leaf nodes corresponding to honest verifiers are chosen randomly. The authentication paths are then derived from this constructed Merkle tree. Importantly, the adversary only learns the authentication paths associated with the malicious verifiers.

We now describe the simulation of different components of the Ligero proof system, including the outputs of the correctness tests and the columns revealed during the column check.

- *Simulating the correctness tests:* We simulate the outputs of the following three correctness tests as follows:
  - Code test: The encoding at the end of the degree test is simulated so that it is independent of the $\mathcal{P}$'s inputs while being consistent with the view of the adversary up to this point as well as the columns revealed during the proof. More specifically, $\mathcal{S}im$ chooses the encoding at the end of the degree test to be a random L-encoding $\widetilde{q_{\mathsf{code}}} = (q^1_{\mathsf{code}}, \ldots, q^n_{\mathsf{code}})$ under the constraint that $q^i_{\mathsf{code}} = r^T U[i] + \beta_{\mathsf{code}}[i]$ for each $i \in C \cup Q$ where Q is the set of revealed columns and β_{code} is the blinding factor added to the output of the code test.
  - Linear test: This is similar to the previous test where the encoding at the end of the linear test is made independent of the protocol execution such that it is consistent with the view of the adversary as well as the columns revealed during the proof. The output of the linear tests $\widetilde{q_{\mathsf{lin}}} = (q^1_{\mathsf{lin}}, \ldots, q^n_{\mathsf{lin}})$ under the constraint that q^i_{lin} for $i \in C_s \cup Q$ is consistent with the adversary's view i.e., $q^i_{\mathsf{lin}} = (r_1(\zeta_s), ..., r_b(\zeta_s))^T \mathbb{C}^{\mathsf{linear}}[i] + \beta_{\mathsf{lin}}[i]$ for each $i \in C_s \cup Q$ and the $\widetilde{q_{\mathsf{lin}}}$ is an L-encoding of a block of random values that sum up to 0. Here, β_{lin} is the blinding factor added to the output of the linear test.
  - Quadratic test: This test is also similar to the previous test where the encoding at the end of the quadratic test $\widetilde{q_{\mathsf{quad}}}$ is chosen to be a random L'-encoding of a block of $2l$ zeros such that it is consistent with the view of the adversary up to this point as well as the columns revealed during the proof.
- *Simulating the revealed columns and column hashes:* $\mathcal{S}im$ randomly samples the set $Q \subseteq [n]$ of size t' to represent the revealed columns. The simulator randomly samples elements and reveals them as columns of the encoded extended witness, i.e., $\{U[i]\}_{i\in C\cup Q}$, which are sent to the adversary $\mathcal{A}$. Finally, Sim programs the random oracle such that (i) the challenge set Q aligns exactly with the revealed columns in the proof transcript and (ii) the authentication paths used to verify these columns correctly are consistent with with commitment $\widetilde{\mathsf{com}}$ revealed during the commit phase.

Simulating the Commit Phase: When an honest prover $\mathcal{P}$ commits to a value w, Sim receives a message (receipt, *sid*, $\mathcal{P}$). Upon receiving this message, Sim simulates th commitment $\widetilde{\mathsf{com}}$ by constructing a Merkle tree. The leaf nodes for the revealed columns were computed from those sampled during the proof simulation, while the remaining leaf nodes are chosen randomly. The Merkle root is then computed from these leaves, using a hash function modeled as a random oracle.

We now prove that the real world view is computationally indistinguishable from the ideal world view. At a high-level, when the prover is honest, the key difference between the ideal and real executions is that the commitment $\widetilde{\mathsf{com}}$ and proof $\{\widetilde{\pi}_j\}\mathcal{V}_{j\in C}$ 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 views of the environment in the real and ideal worlds are indistinguishable. We denote $\mathbf{REAL}_{\Pi_{\mathsf{dCP}},\mathcal{A},\mathcal{Z}}(n)$ to be the output of the environment $\mathcal{Z}$ after the real execution of the protocol Π_{dCP} with adversary $\mathcal{A}$, with security parameter n. We denote $\mathbf{IDEAL}_{\mathcal{F}_{\mathsf{dCP}},\mathsf{Sim},\mathcal{Z}}(n)$ to be the output of the environment $\mathcal{Z}$ after an ideal execution with the simulator Sim (i.e., ideal adversary) and ideal functionality $\mathcal{F}_{\mathsf{dCP}}$, with security parameter n.

We state correctness of the simulation in the following lemma.

Lemma C.1. *The following two distribution ensembles are computationally indistinguishable,*

$$\left\{\mathbf{REAL}_{\Pi_{\mathsf{dCP}},\mathcal{A},\mathcal{Z}}(n)\right\}_{n\in\mathbb{N}} \approx \left\{\mathbf{IDEAL}_{\mathcal{F}_{\mathsf{dCP}},\mathsf{Sim},\mathcal{Z}}(n)\right\}_{n\in\mathbb{N}}.$$

Towards proving this indistinguishability, we consider a sequence of intermediate hybrid experiments and apply a standard hybrid argument. For each hybrid experiment **Hybrid** H_i, we define the random variable $\mathsf{hyb}_i(n)$ that denotes the output of the experiment.

Hybrid H_0**:** This hybrid is the real world execution of the protocol Π_{dCP}. By construction, we have $\mathsf{hyb}_0(n) \equiv \mathbf{REAL}_{\Pi_{\mathsf{dCP}},\mathcal{A},\mathcal{Z}}(n)$.

Hybrid H_1**:** This hybrid is similar to the previous hybrid with the exception that the corrupted verifier shares are generated independently of the real input. Specifically:

- The shares $\{\widetilde{\mathsf{sh}}_i\}_{i\in C}$ received by the corrupted verifiers are chosen randomly in a way that ensures consistency under the sharing scheme, which is RSS in our dCP construction.
- The shares for honest verifiers are then computed given the real input and the corrupted verifiers' shares.

This guarantees that the shares of corrupted verifiers do not depend on the real input.

Lemma C.2. *$\{\mathsf{hyb}_0(n)\}_{n\in\mathbb{N}}$ and $\{\mathsf{hyb}_1(n)\}_{n\in\mathbb{N}}$ are statistically indistinguishable in the random oracle model.*

Proof. The key observation is that secret sharing ensures privacy against adversaries controlling up to t parties, where t is the corruption threshold. This means that the adversary cannot distinguish whether:

- The real input was secret-shared first, and shares were then assigned to corrupted verifiers.
- The shares for corrupted verifiers were sampled independently, and the shares for honest verifiers were derived accordingly to be consistent with the real inputs.

Since the privacy threshold of the secret-sharing scheme exceeds t (i.e., the number of corrupted verifiers), the adversary's view remains statistically indistinguishable between the two hybrids.

□

Hybrid H_2**:** In this hybrid, $\mathcal{S}im_2$ is similar to **Hybrid** H_1 with that exception the $\mathcal{S}im_1$ simulates the the subset Q of columns of U that are revealed as part of the proof; this simulation is done as described in the simulator. Also, the $\widetilde{\mathsf{com}}$ (and the authentication paths for the revealed columns) is generated as described in the simulation of the commit phase.

Lemma C.3. *$\{\mathsf{hyb}_1(n)\}_{n\in\mathbb{N}}$ and $\{\mathsf{hyb}_2(n)\}_{n\in\mathbb{N}}$ are statistically indistinguishable in the random oracle model.*

Proof. Note that computing row-wise random encoding of the extended witness first and then revealing the columns is equivalent to choosing the columns of U first and then choosing a polynomial consistent with the extended with witness and the revealed columns as the degree of the polynomial used for row-wise encoding $\deg > t + t' + \ell$ where $t = |C|$, $t' = |Q|$ and ℓ is the packing factor per row.
Furthermore, the Merkle-based commitment $\widetilde{\mathsf{com}}$ is computed using a hash function modeled as a random oracle. The authentication paths are revealed for columns in set Q are computed similarly in both the hybrids and hence the adversary's view remains statistically indistinguishable between the two hybrids in the random oracle model.

□

Hybrid H_3**:** In this hybrid, $\mathcal{S}im_3$ simulates the prover similar to **Hybrid** H_2 with the exception that $\mathcal{S}im_3$ simulates the encoding q_{code} at the end of the code test sent to all the verifiers such that it is independent of the rest of protocol execution while being consistent with adversary's view up to this point (as described in the simulator). More specifically, $\mathcal{S}im_3$ chooses q_{code} to be a random L-encoding under the constraint that the shares corresponding to columns $i \in C \cup Q$ are consistent with the adversary's view i.e. $q_{\mathsf{code}} = r^T U[i] + \beta_{\mathsf{code}}[i]$.

Lemma C.4. *$\{\mathsf{hyb}_2(n)\}_{n\in\mathbb{N}}$ and $\{\mathsf{hyb}_3(n)\}_{n\in\mathbb{N}}$ are statistically indistinguishable.*

Proof. The only difference between the two hybrids is the way in which the encoding at the end of the code test on an honest prover's input is generated. In **Hybrid** H_3, the encoding q_{code} is generated first as random encoding that is consistent with the view of the adversary. Then the encoding β_{code} is determined using q_{code} as follows $\beta_{\mathsf{code}} = q_{\mathsf{code}} - r^T U$. This results in the same view of the adversary as sampling β_{code} to be random L-encoding and then computing q_{code} as in **Hybrid** H_2. Therefore the two encodings are statistically indistinguishable. □

Hybrid H_4**:** This hybrid is similar to the previous hybrid with the exception that the outputs of the linear test (both from the common portion of the proof i.e., π_1 and verifier-specific portions $(\pi_2^1, \ldots, \pi_2^n)$) are made independent of the rest of protocol execution while being consistent with adversary's view up to this point. Further, the commitment $\widetilde{\mathsf{com}}'$ is simulated as described in the simulation.

Hybrid H_5**:** This hybrid is similar to the previous hybrid with the exception that the output of the quadratic test is made independent of the rest of protocol execution while being consistent with adversary's view up to this point.

Lemma C.5. *$\{\mathsf{hyb}_3(n)\}_{n\in\mathbb{N}}, \{\mathsf{hyb}_4(n)\}_{n\in\mathbb{N}}$ and $\{\mathsf{hyb}_5(n)\}_{n\in\mathbb{N}}$ are statistically indistinguishable.*

This lemma can be proven using reasoning similar to that of Lemma C.4.

Hybrid H_6**:** This hybrid is similar to the previous hybrid with the exception that the commitment $\widetilde{\mathsf{com}}'$ are simulated as described in the simulation.

Lemma C.6. *$\{\mathsf{hyb}_6(n)\}_{n\in\mathbb{N}}, \{\mathsf{hyb}_5(n)\}_{n\in\mathbb{N}}$ and $\{\mathsf{hyb}_5(n)\}_{n\in\mathbb{N}}$ are statistically indistinguishable in the random oracle model.*

This follows from the fact that the Merkle-based commitment $\widetilde{\mathsf{com}}'$ is computed using a random oracle with appropriately randomized leaf nodes (by adding nonces).

Hybrid H_7**:** This hybrid is similar to the previous hybrid except that prover's input is to set to default value that satisfies the predicate (e.g. a vector of zeros). This hybrid experiment identical to the $\mathcal{S}im$ where the environment $\mathcal{Z}$ interacts with $\mathcal{S}im$ and the functionality $\mathcal{F}_{\mathsf{dCP}}$. By construction, we have that $\mathsf{hyb}_7(n) \equiv \mathbf{IDEAL}_{\mathcal{F}_{\mathsf{dCP}},\mathcal{A},\mathcal{Z}}(n)$.

Lemma C.7. *$\{\mathsf{hyb}_6(n)\}_{n\in\mathbb{N}}$ and $\{\mathsf{hyb}_7(n)\}_{n\in\mathbb{N}}$ are statistically indistinguishable.*

Proof. During the course of an execution of the protocol, the adversary corrupting at most t verifiers learns at most t shares $\{\widetilde{\mathsf{sh}}_i\}_{i\in C}$, the proof $\widetilde{\pi}_1$ and $\{\widetilde{\pi}_2^j\}_{j\in C}$, commitments $\widetilde{\mathsf{com}}$ and $\widetilde{\mathsf{com}}'$, and authentication paths revealed as part of the proof. It follows from the privacy property of the secret sharing scheme that the adversary cannot learn any information about the prover's input from these t shares. The remaining components also do not reveal any information as they have been simulated independent of the prover's inputs. Thus, the hybrids $\{\mathsf{hyb}_6(n)\}_{n\in\mathbb{N}}$ and $\{\mathsf{hyb}_7(n)\}_{n\in\mathbb{N}}$ are indistinguishable. □

C.2 Case II: Prover is Corrupted

Simulating the Commit Phase: Whenever $\mathcal{A}$ (controlling $\mathcal{P}$) wants to commit to a value, Sim obtains the commitment com that

$\mathcal{A}$ sent 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 $\mathcal{P}$ makes to the random oracle while constructing the proof, and uses it to extract the witness w. Then, Sim externally sends the message (commit, sid, $\mathcal{P}$, w) to $\mathcal{F}_{\mathsf{dCP}}$ and keeps track of the value w.

Simulating the Prove Phase: 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_1, π_2^j) on behalf of the honest verifier $\mathcal{V}_j$. Sim verifies (π_1, π_2^j) as per the verification steps described in Appendix B and proceeds as follows:

- If the proof verification fails, Sim aborts.
- If the proof verification passes but $\mathcal{R}_j(\mathsf{sh}_j, \mathsf{w}) \neq 1$ for any of the honest verifiers $\mathcal{V}_j$, meaning that the revealed share is inconsistent with the extracted witness, then Sim aborts.
- If Sim does not abort, it externally sends the message (Prove, *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.

We state correctness of the simulation in the following lemma.

Lemma C.8. *The following two distribution ensembles are computationally indistinguishable,*

$$\{\mathbf{REAL}_{\Pi_{Agg},\mathcal{A},\mathcal{Z}}(n)\}_{n\in\mathbb{N}} \approx \{\mathbf{IDEAL}_{\mathcal{F}_{Agg},\mathsf{Sim},\mathcal{Z}}(n)\}_{n\in\mathbb{N}}.$$

Towards proving this indistinguishability, we consider a sequence of intermediate hybrid experiments and apply a standard hybrid argument. For each hybrid experiment **Hybrid** H_i, we define the random variable $\mathsf{hyb}_i(n)$ that denotes the output of the experiment.

Hybrid H_0**:** This hybrid is the real world execution of the protocol Π_{Agg}. By construction, we have that $\mathsf{hyb}_0(n) \equiv_{\Pi_{Agg},\mathcal{A},\mathcal{Z}} (n)$.

Hybrid H_1**:** In this experiment, a simulator $\mathcal{S}im_1$ is similar to the previous hybrid **Hybrid** H_0, except the Sim_1 additionally extracts the input w as described in the simulation of the commit phase. Additionally, $\mathcal{S}im$ checks the following: If the proof verification passes for some verifier $\mathcal{V}_j$, but the $(\mathsf{sh}_j, w) \notin \mathcal{R}_j$, the Sim_1 aborts.

Lemma C.9. $\{\mathsf{hyb}_0(n)\}_{n\in\mathbb{N}}$ *and* $\{\mathsf{hyb}_1(n)\}_{n\in\mathbb{N}}$ *are indistinguishable except with negligible probability by appropriately setting the parameters.*

Proof. When the prover is corrupted, we first claim that Sim_1 aborts with negligible probability: if $\mathcal{R}_j(\mathsf{sh}_j, \mathsf{w}) \neq 1$, then proof verification on input $(\mathsf{sh}_j, \mathsf{com}, \pi_1, \pi_2^j)$ corresponding to an honest verifier $\mathcal{V}_j$ fails, except with negligible probability. This follows from the soundness of Ligero proof system (see Theorem 4.6, [3]).

□

Hybrid H_2**:** This hybrid experiment identical to the $\mathcal{S}im$ where the environment $\mathcal{Z}$ interacts with $\mathcal{S}im$ and the functionality $\mathcal{F}_{Agg}$. By construction, we have that $\mathsf{hyb}_9(n) \equiv \mathbf{IDEAL}_{\mathcal{F}_{Agg},\mathcal{A},\mathcal{Z}}(n)$.

Lemma C.10. $\{\mathsf{hyb}_2(n)\}_{n\in\mathbb{N}}$ *and* $\{\mathsf{hyb}_1(n)\}_{n\in\mathbb{N}}$ *are statistically indistinguishable.*

Proof. 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 (dCP-Prove, 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 (Proof, sid, $\mathcal{P}, \mathcal{V}_j, \mathsf{sh}_j$, accept) whenever $\mathcal{R}_j(\mathsf{sh}_j, \mathsf{w}) = 1$ (equivalently, when proof verification passes); otherwise sends (Proof, sid, $\mathcal{P}, \mathcal{V}_j, \perp$, 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. □

D VERIFIABLE RELATION SHARING (VRS) THEOREM

Theorem D.1. *(Informal) 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 $O(\mathsf{d} \cdot \binom{n-1}{t} + \rho + h)$ 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: $O(n \cdot \rho + n \cdot \mathcal{BC}(h) + n \cdot \mathsf{d} \cdot \mathcal{BC}(\binom{n-1}{t}))$ field elements*
- *Online phase in the optimistic setting[17] (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 (in bits) of the hash function.

The above theorem is similar to Theorem 4.3 in [6] but with the costs adapted to using replicated secret sharing and Ligero proof system to instantiate the dCP protocol. We have analysed the communication efficiency in more detail below. For detailed proof, we refer to [6], specifically Appendix *K*.

The above theorem closely resembles Theorem 4.3 in [6], but with costs adapted for the use of replicated secret sharing and the Ligero proof system to instantiate the dCP protocol. Below, we provide a more detailed analysis of the communication efficiency. For the full proof, see Appendix K of [6].

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

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

field element. We use the VSS scheme from [5], 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.

E EXPERIMENT PARAMETERS

The default values of the parameters, as used in our experiments (see Section 5.2) are presented in Table 3 and Table 4.

Table 3: 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 4
m	# of rows in the extended witness[a]	see Table 4

[a] refer to Ligero proof generation in Appendix B.1

Table 4: 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

F 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 $\mathsf{t_s} < \mathsf{n_s}/3$.

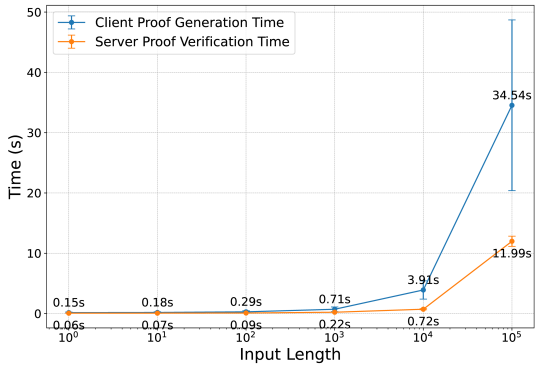


Figure 12: The time required for a client to generate a proof and for a server to verify a client's proof for various sized client inputs (in log-scale).