# Accountable Private Set Cardinality for Distributed Measurement

ELLIS FENSKE, U.S. Naval Academy, USA
AKSHAYA MANI, Georgetown University, USA and University of Waterloo, Canada
AARON JOHNSON, U.S. Naval Research Laboratory, USA
MICAH SHERR, Georgetown University, USA

We introduce cryptographic protocols for securely and efficiently computing the cardinality of set union and set intersection. Our private set-cardinality protocols (PSC) are designed for the setting in which a large set of parties in a distributed system makes observations, and a small set of parties with more resources and higher reliability aggregates the observations. PSC allows for secure and useful statistics gathering in privacy-preserving distributed systems. For example, it allows operators of anonymity networks such as Tor to securely answer the questions: *How many unique users are using the network?* and *How many hidden services are being accessed?*

We prove the correctness and security of PSC in the Universal Composability framework against an active adversary that compromises all but one of the aggregating parties. Although successful output cannot be guaranteed in this setting, PSC either succeeds or terminates with an abort, and we furthermore make the adversary *accountable* for causing an abort by blaming at least one malicious party. We also show that PSC prevents adaptive corruption of the data parties from revealing past observations, which prevents them from being victims of targeted compromise, and we ensure safe measurements by making outputs differentially private.

We present a proof-of-concept implementation of PSC and use it to demonstrate that PSC operates with low computational overhead and reasonable bandwidth. It can count tens of thousands of unique observations from tens to hundreds of data-collecting parties while completing within hours. PSC is thus suitable for daily measurements in a distributed system.

CCS Concepts: • **Security and privacy → Cryptography**; *Network security*; **Security protocols**.

Additional Key Words and Phrases: secure computation; privacy-preserving measurement

## 1 INTRODUCTION

Measurements are essential to understanding the use of distributed systems and monitoring them for abuse. In a privacy-preserving distributed system, such as an anonymity network, measurement is complicated by the system's privacy requirements. Storing records of system activity can pose significant risks to the system's users, and consequently the ethics of such techniques [51] have been widely debated [57, 61]. Ideally, the measurements produced should satisfy strong privacy definitions, and during the measurement process the system should protect sensitive intermediate data.

For example, in an anonymous-communication system such as Tor [19], it is very helpful to measure statistics such as the number of users, the popular applications used over Tor, and the

Authors' addresses: Ellis Fenske, U.S. Naval Academy, Annapolis, USA, fenske@usna.edu; Akshaya Mani, Georgetown University, Washington, USA and University of Waterloo, Waterloo, Canada, akshaya.mani@uwaterloo.ca; Aaron Johnson, U.S. Naval Research Laboratory, Washington, USA, aaron.m.johnson@nrl.navy.mil; Micah Sherr, Georgetown University, Washington, USA, msherr@cs.georgetown.edu.


2471-2566/2022/5-ART25 $15.00
https://doi.org/10.1145/3477531

number of errors that occur. With such information, users can better understand their anonymity, policymakers can consider the value of anonymity, and network designers can identify its problems. However, the privacy goal of the network would be undermined if detailed records were collected of who uses the system, when it is used, and for what purposes. As other example applications, structured overlay networks such as Chord [63] may wish to measure network activity without exposing individual actions, and online services may want to know how many users they share without revealing their identities.

Two general privacy technologies can help solve this problem: secure multiparty computation protocols [47] (MPC) and differentially private mechanisms [24]. Generic MPC protocols can be used by a set of parties to compute any function of their private inputs without revealing those inputs. Differentially private mechanisms guarantee that the value they compute depends little on any specific input, thereby only revealing information implied by the collection of inputs. Thus to safely measure a distributed system, the members could execute a generic MPC protocol, using their observations as input, in order to compute a differentially private measurement function [58]. However, generic protocols have relatively high computational and communication costs. Moreover, this approach does not protect the observations being recorded by the parties to serve as inputs to the secure computation, which could be revealed if such parties can be corrupted or otherwise compelled to reveal their internal state during the measurement period.

Some specific protocols have been developed to more efficiently and securely compute aggregate measurements in anonymity networks, discussed in detail in the following section. We extend this line of work by presenting a protocol which is capable of privately measuring unique counts, calculating private set cardinality operations for the set union and set intersection operations. In the scenario we consider, a set of Data Parties (DPs) each observes some set, and we wish to compute the size of the union or intersection of those sets. More formally, if there are $d$ DPs with $\mathrm{DP}_k$ observing set $\mathcal{I}_k$, we want to be able to privately compute $|\bigcup_{k=1}^{d} \mathcal{I}_k|$ for set-union cardinality and $|\bigcap_{k=1}^{d} \mathcal{I}_k|$ for set-intersection cardinality. We assume that there exists a smaller set of Computation Parties (CPs) that can be used to aggregate the observations of the *DPs*, which will improve efficiency and allow us to tolerate failures of the *DPs*.

We seek to satisfy stringent privacy and security goals to make the protocol suitable even in very adversarial settings, appropriate for a distributed system in which many of the parties may be malicious. Anonymity networks are a key desired application of our protocols, and several active attacks have been observed on the Tor network. To this end, we require that the protocol is secure against an *active* adversary that can deviate from the prescribed protocol. Moreover, we desire security against a *dishonest majority* of the Computation Parties, to ensure the security of user data as long as at least one CP is honest. We seek *composable* security, so that measurement can be repeatedly and concurrently run while maintaining security. We further target *adaptive security* among the Data Parties such that corruptions during the lengthy measurement periods do not reveal past observations. Adaptive security is particularly important for DPs as they may otherwise be susceptible to a legal compulsion attack targeted at data stored about the observations they make [26]. We also require that the protocol output satisfies *differential privacy* to ensure that a securely computed function poses no privacy risk. Finally, we require that no Data Party can prevent the protocol from completing successfully, and, while for efficiency we allow a Computation Party to cause a failure, we require *accountability* of a responsible party so that the party can be excluded and the protocol restarted.

We present the PSC protocol for private set-cardinality operations. PSC satisfies all our functionality and security goals, and it does so in a way that is efficient and practical to be used for distributed-system measurements, including especially measurements of anonymity systems such

as Tor. PSC has two set-cardinality variants: one for set union and the other for set intersection. We present the set-union variant and describe the changes needed to it for set intersection in Section 8.

We prove the correctness and security of PSC in the Universally Composable (UC) framework [11]. This proof methodology allows us to easily express the main security and privacy goals via a single ideal functionality. It also guarantees that security is maintained even if the protocol is run concurrently with itself or as part of a larger cryptographic system.

We additionally introduce an implementation of PSC, released as open-source software written in memory-safe Go. To achieve greater efficiency, our implementation uses some subprotocols that are not proven UC-secure, such as a verifiable shuffle [54] that is more practical than verifiable shuffles proven secure in the UC model. With these subprotocols, our implementation can still be proven secure in the classical (*i.e.* standalone) model and, as we demonstrate via at-scale evaluation experiments, incurs only moderate bandwidth and computation costs and can be practically deployed.

This paper expands upon an earlier version of this work [27] in many ways, including in particular modifying the protocol to provide accountability, adding measures to prevent input modification, and detailing how to accomplish set-intersection cardinality.

## 2 RELATED WORK

We consider three main parallel lines of work related to our central problem: protocols designed to measure anonymity networks, protocols designed to compute secure distributed set operations, and fully generic multiparty computation protocols. Finally, we discuss related work surrounding accountability properties in cryptographic protocols.

*Privacy-Preserving Measurements of Distributed Systems.* The PrivEx system of Elahi et al. [26] uses partially-homomorphic aggregation and differential privacy [24] to privately collect statistics about the Tor network [19]. PrivCount [40], which we extend in this work, improves upon PrivEx by offering multi-phase measurements and also an optimal allocation of the $\epsilon$ privacy budget. History$\epsilon$ [50] is also inspired by PrivEx and uses histogram-based queries to provide integrity guarantees by bounding the influence of malicious data contributors. Melis et al. [53] present a system to perform distributed item-frequency counting, with one application being computing median statistics from Tor relays. It uses homomorphic encryption to securely aggregate count-min sketches.

While these systems provide significant efficiency improvements over generic protocols, they lack the ability to perform set-cardinality operations. For example, while PrivEx, PrivCount, and History$\epsilon$ can answer the question *"How many clients were observed entering the Tor network?"*, they cannot determine the number of unique clients (i.e. set-union cardinality). Similarly, while count-min sketches can help determine the median number of clients across all entry relays, they cannot determine the how many clients are seen at all relays (i.e. set-intersection cardinality).

*Private Set Operations Protocols.* Brandt suggests a protocol [8] very similar to the aggregate-shuffle-rerandomize-decrypt scheme our CPs execute. However, our construction differs in a few crucial parts: we need to include separate DP parties to provide the input that must be adaptively secure and limit as much as possible their computational work, and in our protocol the parties jointly generate noise to satisfy a differential privacy guarantee. Beyond these modifications, the bulk of our contribution is a thorough proof of security of the protocol in the UC model (to make the proof go through, we needed to add re-encryption during the re-randomization phase), a specific application for the general theoretical protocol (making measurements in privacy-preserving systems like Tor), and a functional implementation of the protocol alongside empirical data measuring computation and communication costs gathered through experiments.

Several protocols to securely compute set-union cardinality (or the related problem of set-intersection cardinality) have been proposed [17, 25, 44, 62, 66] using a variety of techniques including bloom filters, homomorphic encryption, sketches, and polynomial evaluation. However, none of these provides all of the security properties that we desire for distributed measurement in a highly-adversarial setting: malicious security against a dishonest majority, adaptive security for Data Parties, and differentially-private output. Similar protocols have been designed to securely compute set operations [29, 37, 38], but these protocols output a set rather than its cardinality.

*Secure Multiparty Computation.* General secure multiparty computation (MPC) protocols can realize any functionality including set cardinality, even in the UC model [1, 12]. Moreover, advances in efficient MPC have been made in the multi-party, dishonest-majority, active security setting that we require [16, 46, 49, 69]. However, these works assume that the same parties that have the inputs perform the computation and thus do not describe how to securely transfer inputs if these parties are different. Moreover, such protocols make use of a relatively expensive "offline" phase, while we intend to allow for measurements that are run on a continuous basis.

Wails et al. [68] have explored how to apply these generic MPC protocols in the context of measuring the Tor network. They present new protocols to transfer inputs from a large number of observing parties to a smaller number of computation parties. They show how to securely compute approximate set-union cardinality using a sketch (viz. LogLog [21]). While their construction does not provide differential privacy, Choi et al. [14] show that sketch-based counts can be made differentially private with high accuracy. Compared to PSC, this approach of using sketches and generic MPC protocols has the advantage of scaling well to large counts (e.g. billions), but it has the disadvantages of poor accuracy when the count is small and error from noise that is proportional to the count rather than constant. These disadvantages arise from the use of sketch-based counting. Furthermore, the generic MPC protocols allow an adversary to abort the protocol without accountability, which enables a continuous denial of service.

*Accountability.* To handle denial of service attacks on secure multiparty computation protocols, there has been recent work developing protocols that can identify to all honest parties a malicious party who has prevented the protocol from terminating successfully. This type of termination is called *identifiable abort*, and there are generic multiparty computation protocols with this property [2, 3, 32, 39]. Taking another approach to accountability, some protocols provide a stronger public-accountability property by producing a proof of malicious behavior on some trusted publicly readable trusted ledger or bulletin board [3, 43]. While all of these contributions using either approach give accountable methods for general private measurements of any kind, including those we wish to collect, these contributions do not provide implementations or experiments, and require that the parties providing inputs are those that perform the computations, meaning using them for the applications we intend would incur heavy computation and communication costs to our Data Parties. Additionally, these approaches do not consider the problem of adaptive corruptions intended to extract measurement data during a long collection period, a property we consider to be crucial for many typical use cases such as measuring anonymous communication systems like Tor.

## 3 BACKGROUND

Before describing the problem and our solution, we briefly review some concepts and background that are necessary for understanding the protocol, Private Set-Union Cardinality (PSC).

*Differential Privacy.* Differential privacy [24] is a privacy definition that offers provable and quantifiable privacy of database queries. Differential privacy guarantees that the query responses look nearly the same regardless of whether or not the input of any one user is included in the

database. Thus anything that is learned from the queries is learned independently of the inclusion of any single user's data [41]. Several mechanisms have been designed to provide differential privacy while maximizing accuracy [7, 23, 52, 56].

More formally, an $(\epsilon, \delta)$-differentially-private mechanism is an algorithm $\mathcal{M}$ such that, for all datasets $D_1$ and $D_2$ that differ only on the input of one user, and all $S \subseteq \text{Range}(\mathcal{M})$, the following holds:

$$Pr[\mathcal{M}(D_1) \in S] \leq e^{\epsilon} \times Pr[\mathcal{M}(D_2) \in S] + \delta. \tag{1}$$

$\epsilon$ and $\delta$ quantify the amount of privacy provided by the mechanism, where smaller values of each indicate more privacy. Dwork et al. [22] proves that binomial noise, that is, the sum of $n$ uniformly random binary values, provides $(\epsilon, \delta)$-differential privacy for queries that each user can affect by at most one when

$$n \geq \left( \frac{64 \ln(2/\delta)}{\epsilon^2} \right) \tag{2}$$

Eq. 2 presents a trade-off between privacy and utility, an issue inherent to differential privacy. Put alternatively, for the privacy level given by $\epsilon$ and $\delta$, Eq. 2 yields the amount of binomial noise that is required to be added to the output of a query that each user can change by at most one. In this paper, we use this binomial noise technique to achieve differential privacy.

*Termination and Accountability.* In any multiparty computation protocol, weak trust assumptions like a dishonest majority preclude properties like guaranteed output delivery [39]. This means even if a protocol guarantees its output is accurate, malicious parties may prevent the output from being produced at all by causing the protocol to hang indefinitely or to abort. Exacerbating this problem, dishonest majority protocols cannot guarantee *fairness*, meaning malicious parties could learn the result before choosing whether to allow a protocol to successfully conclude.

As a solution to this, many protocols introduce an accountability guarantee called *identifiable abort* [2], which guarantees that either the correct output or the identity of some number of malicious parties is given to the honest parties at protocol termination. This allows honest parties to identify a source of failure and take appropriate steps to remedy the problem: identify network failures, rerun the protocol excluding certain parties, or publicly levy accusations of malicious behavior.

This guarantee only mitigates denial of service attacks in practice if a protocol comes with a guarantee of termination within finite time, or malicious parties could indefinitely delay the accountability messages as well as the intended protocol output. This is handled often in literature by defining protocols in the *synchronous communication model*, where messages are sent in a sequence of explicit synchronized communication rounds. In every round, each party may send messages to any set of other parties, and these messages are guaranteed to be delivered to their recipient at the end of that round. Synchronous, round-based communications can be constructed explicitly using bounded delay and synchronized clocks [42].

*Zero-Knowledge Proofs.* PSC uses a few types of zero-knowledge proofs demonstrating knowledge and relationships between elements in some group $G$ of order $q$ with respect to some generator $g$. For example, we require a proof of knowledge of the discrete log of $y = g^x$, $x \in \mathbb{Z}_q$. In general, a zero-knowledge proof system [31] is a protocol between a prover $\mathcal{P}$ and verifier $\mathcal{V}$ in which the prover demonstrates the truth of some statement without revealing more than that truth, where the statement may be, for example, the existence or knowledge of a witness to membership in a language.

PSC uses many $\Sigma$-protocols [15], which are proofs of knowledge that are three-round interactive protocols starting with a commitment by the prover, followed by a random challenge from the

verifier, and ended by a response from the prover. Σ-protocols are honest-verifier zero knowledge (HVZK), that is, their transcript is simulatable assuming the verifier behaves honestly.

PSC also makes use of *verifiable shuffles* [55]. Informally, a verifiable re-encryption shuffle takes as input ciphertexts, outputs a permutation of a re-encryption of those ciphertexts, and proves that the output is a re-encryption and permutation of the input. There are two security requirements for verifiable shuffles: privacy and verifiability. Privacy requires an honest shuffle to protect its secret permutation. Verifiability requires that any attempt by a malicious shuffle to produce an incorrect output must be detectable. Several protocols for verifiable shuffling have been proposed [4, 30, 34, 54].

The Σ-protocols we use in PSC and the verifiable shuffles with public-coin interactive proofs can be made non-interactive using the Fiat-Shamir heuristic [28], in which the random challenges are generated by the prover by applying a cryptographic hash function. Non-interactive proofs (both shuffles and Σ-protocols) can be sent to many verifiers through a broadcast, but PSC additionally uses interactive multi-verifier proofs, where a single proof is sent to multiple verifiers such that all honest verifiers agree on the statement to be proved and the correctness of the proof to provide accountability.

*Broadcasts.* PSC uses an accountable broadcast communication functionality. The security property that we require for this broadcast is that honest parties agree on the message from the broadcaster, whether it is an *equivocation*, where a broadcaster sends distinct messages to different parties, a single consistent message to all parties, or no message at all. Dolev and Strong [20] give a protocol that satisfies this property assuming synchronous point-to-point communication channels in $t + 1$ rounds, where $t$ is the number of malicious parties. For protocols with many parties the round- and communication-complexity of this primitive make it too inefficient to be practical if deployed naïvely.

*Universal Composability and Setup Assumptions.* PSC, like many complex cryptographic protocols is constructed from smaller cryptographic primitives and subprotocols. The sequential composition theorem [31] justifies constructing protocols modularly in this way, but for protocols to remain secure under general concurrent composition (i.e. simultaneously with arbitrary other protocols, even those designed specifically to interact with them) we require a stronger property. In 2002, Canetti gave a composition theorem for this case under a framework called Universal Composable (UC) security [12]. This composition theorem guarantees that compositions of UC-secure subprotocols are UC-secure.

Given impossibility results for UC secure protocols without setup assumptions, UC protocols are often defined in the Common Reference String (CRS) model, under which any function of a given collection of parties' private inputs can be securely computed [12]. In particular, there are natural constructions of UC-secure zero-knowledge proofs from Σ-protocols [44] and UC-secure verifiable shuffles in this model [70]. Protocols that meet this standard of security can be inefficient, so in practice many protocols rely on stronger setup assumptions and provide weaker composability guarantees. We highlight the Random Oracle Model, where all parties have access to a random oracle that returns random strings to all queries, but must return the same result given two identical queries. In the Random Oracle Model, the Fiat-Shamir heuristic constructs auxiliary-input zero-knowledge proofs [5], which are secure under sequential composition [33]. Random Oracles are usually instantiated with hash functions, a leap which cannot be rigorously justified for any real hash function [59] but which is widely deployed in practice.

## 4 PROBLEM AND SETTING

We consider the problem of distributed private measurement. In particular, we define a finite set of *observations*, which could be explicit identifiable events or abstract counters intended to estimate a count of these events (*e.g.*, in a hash table), to be collected across a large distributed set of $d$ Data Parties (DPs). Each DP collects its own set of observations, but we wish to learn how many distinct events has occured over a given time period across all DPs. We wish to calculate the cardinality of the set-union and set-intersection of these observations.

We collect data on these observations during a *collection period*, during which DPs record observations. After the collection period, the DPs participate in a distributed *protocol phase* that produces the cardinality of the set-union or set-intersection of the DPs' observations. The protocol is privacy-preserving, which informally means that (i) no information collected by an honest DP is ever exposed and (ii) an adversary (defined below) cannot identify any individual data from the aggregated result (i.e. the cardinality).

Additionally, we desire *accountability*, meaning that the honest parties can identify some dishonest party if the execution fails. (We provide a more formal definition of accountability later in the paper.) Importantly, our accountability property does not cover the veracity of DPs' reported observations—a malicious DP can fabricate or omit an observation. Rather, our aim is to detect parties that interfere with the correct computation of the aggregated result.

### 4.1 System Model

To solve this problem, we introduce a set of $m$ Computation Parties (CPs) whose purpose is to offload a majority of the computational- and bandwidth-work from the DPs, and to perform a multi-party computation to aggregate, process, and clean the data. We assume that $d \gg m$ so that the number of DPs is much larger than the number of CPs.

We further assume that DPs are limited in terms of computational resources and bandwidth, so we measure efficiency in this setting by aggregate computational and communication resources consumed but also by the extent to which resources consumed by the DPs for measurement purposes is minimized.

We consider the problem in the synchronous communication model and do not specifically prove results about or implement precise synchronized rounds, leaving them abstract for the purpose of exposition, reasoning about security properties, and prototype implementation.

### 4.2 Threat Model

We consider in our threat model arbitrary adversaries who corrupt any number of DPs and all but one of the CPs, and consider primarily adversaries who wish to learn private data: which observations have been recorded, and by which DPs. Therefore, it is crucial that an adversary who adaptively attacks DPs during the collection period not does learn more private data by attacking DPs who execute our measurement protocol than that same adversary could learn by attacking DPs who do not, so any satisfactory measurement protocol must resist adaptive attacks against DPs during the collection period.

Further, we expect the protocol to be run repeatedly (*e.g.*, to be run continuously with collection periods beginning and ending simultaneously). This means even simple count data can reveal a large amount of private information over time, especially against active adversaries who can insert inputs through malicious DPs and observe differences in the protocol output. This means the measurements released by the protocol must be differentially private.

Finally, DPs do not always exclusively take measurements, and often these measurements are associated with a large, complex cryptographic protocol whose execution is the primary

purpose of the DPs. Further, the protocol is intended to be run concurrently with other copies of itself (measuring different events, or over different time periods). This means the protocol should be privacy-preserving even when run concurrently with other arbitrary protocols, meaning a measurement protocol should be secure in the UC model.

CPs should be taken from a distributed set of parties not likely to collude in order to recover the sensitive private data they process, and we formalize this intuition by assuming at least one of the CPs is honest. We do not consider the problem of adaptively corrupted CPs.

While we accept that malicious DPs may report false data, we may perform repeated sampling of DPs among those possible in order to provide robust measurements over time, or exclude obviously malicious DPs (those who report having observed every event in order to produce a trivial result from a Set-Union Cardinality measurement, for example). However, since CPs participate in every round, it is important that CPs may not adjust the results or prevent the protocol from completing without consequences. This takes the form of a formal *accountability property* in the dishonest-majority setting, which guarantees honest parties agree on a set of malicious parties to blame in the case the protocol fails to terminate successfully.

### 4.3 Problem Statement

We define the problem we wish to solve as follows:

*Definition 4.1.* Fix $m$ CPs, $d$ DPs, $b$ possible observations, and a time period $T$. Represent the observations made by each $\text{DP}_j$ during $T$ as $\vec{O^j}$. We define the *Private Set-Union Cardinality Problem* (with Set-Intersection defined similarly) as calculating $\left|\bigcup \vec{O^j}\right|$ and distributing this result to all CPs, subject to the following formal and informal constraints:

- The result should be *accurate*, meaning that malicious CPs may not alter the data submitted by honest DPs or selectively exclude data submissions.
- The result should be *private*, meaning that no adversary may learn information about individual observations of the DPs exceeding that allowed by the Differential Privacy guarantee. In particular, we require that compromise of DPs during the collection period reveal no information about the observations recorded pre-compromise.
- The solution should be *efficient*, meaning that DPs involved in the measurement process do not consume excessive resources and that CPs can output a measurement within hours of the termination of the collection period in a practical, real-world setting.
- The solution should be *accountable*, so every time the protocol is invoked, it terminates successfully or all honest CPs come to consensus on the identity of the CP or CPs who prevented successful termination.
- The solution should be *secure*, meaning it satisfies all above requirements so long as there is one honest CP, and be accompanied by a thorough proof of security.
- The solution should be *practical* enough to implement and deploy in practice today, and rely only on well-tested and well-understood cryptographic assumptions.

In the following, we present, implement, evaluate, and prove secure a protocol that solves the above PSC problem.

### 4.4 Solution Overview

We provide the solution in three parts. First, we describe the protocol in a hybrid model using abstract ideal functionalities. Then we proceed along two parallel paths:

(1) A proof of security for the required primitives in the Universal Composability (UC) setting, followed by a proof of security for the hybrid-model protocol, both in Section 6.

(2) An implementation of the protocol using heuristically secure primitives (namely, the Fiat-Shamir Heuristic) for the purpose of practicality and efficiency, open-sourced and evaluated for efficiency in Section 7.

The hybrid-model proof presented in Section 6.8 then applies to both the implemented protocol and as a proof of the existence and security of an unimplemented theoretical UC-secure protocol with the same structure as our implementation.

In particular, making replacements as described below, we can convert a proof of security in the UC framework in the Common Reference String (CRS) model to a proof of security in the random oracle model under sequential composition. The replacements primarily rely on the fact that the Fiat-Shamir heuristic provides non-interactive Zero-Knowledge Proofs in the Random Oracle model [5], and we may then apply the sequential composition theorem [31] in place of the UC composition theorem. Explicitly:

- The broadcast functionality we use is realized by the Dolev-Strong protocol, which can be implemented with UC signatures, but in the implementation we use Schnorr signatures and apply optimistic optimizations to the protocol as described in Section 6.1.1.
- We require one-to-many Σ-protocols in our proof. These are constructed by compiling Σ-protocols to UC secure accountable one-to-many Zero-Knowledge Proofs as described in Section 6.3, but for implementation purposes we simply apply the Fiat-Shamir heuristic and send non-interactive zero-knowledge proofs over our broadcast channel.
- In the UC setting, we use the shuffle provided by Wikström and outline how it can be made accountable. For implementation purposes, we use an implementation of the shuffle given by Neff [54] and again apply the Fiat-Shamir heuristic. We use this shuffle because it is simple and the code is available and open source, and because additional protocols are required to construct public parameters in Wikström's shuffle that cannot be generated in a natural way by public coins. This is discussed further in Section 6.4.

## 5 PROTOCOL DESCRIPTION

At a high level, the protocol proceeds in two main sections, which we separate into seven *phases*. The goal is to construct a vector of noisy shuffled counters, containing one component for each possible event and additional components for noise. The counters are ciphertexts that are zero or nonzero, representing a binary value. These counters are split into two shares, an initial encrypted blind and a corresponding plaintext counter which DPs modify, recording observations obliviously. Once submissions are complete, the CPs insert encrypted noise counters, shuffle the counters, and clean them, removing non-binary information contained within the plaintext counters in a process called rerandomization. Finally, the counters can be decrypted, with the number of nonzero counters in the final tally giving the final result. We visually present the protocol in phases in Figure 1, and continue our high-level description in more detail as follows, noting that while we give all phases in sequence in our discussion, implementation, and proof, the data collection period provides a natural "offline phase" where CPs remain idle, meaning that Noise Generation can be trivially executed in parallel with data collection.

(1) **Key Generation**. In the Key Generation phase, the CPs generate a joint public session key for each run of the protocol and distribute it to all DPs.
(2) **Data Collection**. DPs construct and distribute to CPs an encrypted blind and corresponding proof of knowledge of cleartext for each counter, which represent the set of observable events to be measured, storing the negation of each value in plaintext. During the data collection period, DPs record an observation by replacing its corresponding counter with a random value.

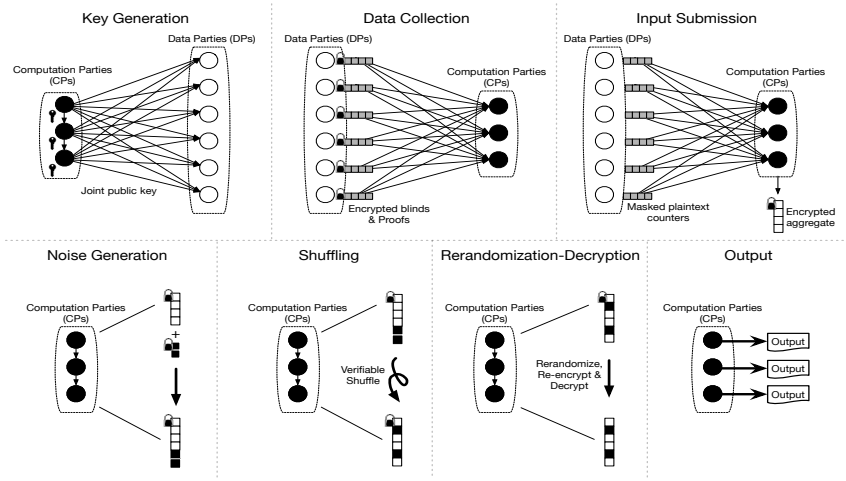


Fig. 1. An overview of the major steps of the PSC protocol — the white, gray, and black square represent plaintext, blind (or blinded data), and noise respectively. The lock represents that the underlying data (*i.e.*, the blinds, noise, or plaintext) is encrypted.

(3) **Input Submission**. When the collection period is over, the DPs broadcast their counters to the CPs, who jointly encrypt and sum these counters with the blinds submitted previously. If a DP has not modified its counter, the recorded counter and plaintext inside the corresponding blind will cancel. Otherwise, their sum will be some random nonzero value.
(4) **Noise Generation**. CPs perform a verifiable shuffle of encryptions of the pair $(0, 1)$ $n$ times and extract the first component of the resulting shuffled vector to add $n$ elements of binomial noise to the aggregate vector of ciphertext counters.
(5) **Shuffling**. CPs perform a verifiable shuffle of the noise-and-data vector, mixing the generated noise with the true counter values.
(6) **Rerandomization-Decryption**. CPs rerandomize, re-encrypt, and decrypt the shuffled vector of data and noise to ensure the plaintexts carry no information beyond the parity property they represent in the protocol.
(7) **Output**. The last CP to perform this decryption broadcasts the plaintexts and the number of nonzero elements is the noisy cardinality.

### 5.1 Primitives and Assumptions

We define two groups of parties, Data Parties or DPs, who make observations from some set of possible observations during a *collection period*, and Computation Parties or CPs who receive, aggregate, clean, and publicize the privacy-safe observation statistic. We make distinct adversarial assumptions with respect to the two groups: for DPs, we work in the erasure model with adaptive corruptions, assuming that an adversary can compromise a DP during the collection period but requiring that it learns nothing about measurements recorded and erased before the corruption. For CPs, we work in the static corruption model and make no assumptions about secure erasures. We denote a vector by $\vec{v}$, and write $\vec{v}_i$ to mean the $i$th component of $\vec{v}$, with superscript notation $\vec{v}^i$ indicating a sequence of vectors. We separate the protocol into phases, each containing some number of synchronous communication rounds. Often, these phases will be split into subphases.

Throughout, we assume $g$ is a generator of a large group $G$ of prime order $q$ for which the Decisional Diffie Hellman assumption holds. We use as our primary encryption mechanism the

exponential version of ElGamal, where rather than a group element $g^m$ as the plaintext, we consider the plaintext to be the integer $m$ itself. This scheme is additively homomorphic, but now requires calculating a discrete log for decryption as in the encryption scheme outlined by Benaloh [6]. For our application, plaintexts encode binary values so we are interested only in whether or not a plaintext $m = 0$, which is trivial to determine given $g^m$ so decryption poses no problem for us. We by convention write an ElGamal encryption of a message $m$ as $E_y(r; m) = (g^r, y^r g^m)$ and freely refer to $r$ as its randomization factor and to $g^r$ as its "first component". Throughout, we refer to three homomorphic manipulations of ElGamal ciphertexts, which can be constructed only using knowledge of the public parameter $y$:

- *Re-Encryption*, where a ciphertext has its randomization factor shifted by a constant additive factor $s$ without modifying the plaintext, converting a ciphertext $(g^r, y^r g^m) \rightarrow (g^{r+s}, y^{r+s} g^m)$.
- *Rerandomization*, where a ciphertext has its randomization factor shifted by a scalar multiplicative factor $s$, applying the same multiplication to the plaintext component, so that a ciphertext $(g^r, y^r g^m) \rightarrow (g^{rs}, y^{rs} g^{ms})$.
- *Aggregation*, where an arbitrary vector of ciphertexts encrypted to the same key are homomorphically added together by taking the componentwise product of each of the two components.

We make use of the following primitives, all in the synchronous communication model:

- $\mathcal{F}_{\text{BC}}$, an accountable consensus broadcast protocol, given in Figure 2.
- $\mathcal{F}_{\text{SKGD}}$, a functionality encapsulating the sub-protocol for Session Key Generation and Distribution, producing and distributing a joint ElGamal public session key shared by the CPs to all parties. This is given in Figure 5.
- $\mathcal{F}_{\text{ZKP-DL}}$, $\mathcal{F}_{\text{ZKP-S}}$, $\mathcal{F}_{\text{ZKP-RRD}}$, one-to-many zero-knowledge proofs for knowledge of a discrete log, a shuffle, and for a combined re-encryption, rerandomization, and decryption operation. These are all implemented with a general zero-knowledge proof functionality given in Figure 3 and each of these specific proofs is realized by a different protocol, given in detail in Section 6.

We provide the security proofs for each of these functionalities including their accountability properties in Section 6 and provide implementation details in Section 7.

Beyond cryptographic assumptions, we assume all parties have permanent signing keys through a Public Key Infrastructure and use these to construct signatures in order to send authenticated messages. Keys for encryption are generated on a per-session basis for each protocol run so we describe their construction explicitly. The zero-knowledge proofs are given in a hybrid model using ideal functionalities for commitments that may be realized in the Common Reference String (CRS) model.

### 5.2 Protocol

At a high level, the protocol proceeds in a sequence of phases, where the session keys are established and blinds are set by the DPs to guard against adaptive corruptions. The data is then collected over some period, and then transmitted to the CPs. The CPs proceed in a three-step process to prepare the data for release: adding random noise to provide differential privacy, shuffling the noise and data together, and "rerandomizing" the nonzero elements that encode 1 so they do not carry any information about the exact representation selected by the DPs.

More precisely, we define the protocol $\pi_{PSC}$ in the $(\mathcal{F}_{\text{BC}}, \mathcal{F}_{\text{SKGD}}, \mathcal{F}_{\text{ZKP-S}}, \mathcal{F}_{\text{ZKP-RRD}}, \mathcal{F}_{\text{ZKP-DL}})$-hybrid model by describing each phase.

(1) **Key Generation**.

- (Round 1). CPs send (GenerateKeys) to $\mathcal{F}_{\text{SKGD}}$. CPs and DPs wait to receive the public session key $y$ from $\mathcal{F}_{\text{SKGD}}$.
- (Blame). CPs receive either a set of session keys from $\mathcal{F}_{\text{SKGD}}$ or a blame message. If a CP receives a blame message, they output it and terminate.

(2) **Data Collection**.
- (Round 1). Each $\text{DP}_j$ generates random $\beta_1^j \ldots \beta_b^j \in \mathbb{Z}_q$, encrypts each, and collects these ciphertexts into a vector $\vec{\beta}^j$. Each $\text{DP}_j$ broadcasts $\vec{\beta}^j$ to all CPs using $\mathcal{F}_{\text{BC}}$.
- (Round 2). Each $\text{DP}_j$ proves knowledge of the cleartext for each component of $\vec{\beta}^j$ by proving knowledge of the discrete log of the first component of each ciphertext, $r_i^j$ using $\mathcal{F}_{\text{ZKP-DL}}$ to all CPs. Each $\text{DP}_j$ saves a vector of plaintexts $\vec{c}^j$ by $\vec{c}_i^j = -\beta_i^j$ and erases all information about $\vec{\beta}^j$ and the proofs from memory.
- (Collection Period). While the collection period is active, $\text{DP}_j$ records observing event $i$ upon receiving a message from the environment of the form (Observation, $i$) by setting $\vec{c}_i^j \leftarrow r$ for $r$ random in $\mathbb{Z}_q$. This phase ends when the collection period is complete.
- (Blame). If a broadcast or one-to-many proof from a DP fails, the CPs exclude the input from that DP during this phase and for the remainder of the protocol. If a CP is blamed, the protocol terminates.

(3) **Noise Generation**. For every component $i$ of $\vec{n}$, all CPs set $\vec{N}^{i0}$ to be the pair of deterministic encryptions to $y$ of 0 and 1 using first component $g^1$. CPs calculate $\vec{N}^{im}$ in $m$ rounds:
- (Round $ij$, $i \in 1..n, j \in 1..m$).
  - (Subround $ij_1$). $\text{CP}_j$ shuffles $\vec{N}^{i(j-1)} = \text{Shuffle}(\vec{N}^{i(j-1)}, \vec{r}^{ij}, \pi^{ij})$ and sends $\vec{N}^{ij}$ to $\mathcal{F}_{\text{BC}}$ with $\text{CP}_j$ generating two random integers $\vec{r}^{ij} \in \mathbb{Z}_q^2$ and a random permutation $\pi^{ij}$ on two elements.
  - (Subround $ij_2$). $\text{CP}_j$ sends ((ZKP), $(\vec{N}^{i(j-1)}, \vec{N}^{ij})$, $(\vec{r}^{ij}, \pi^{ij})$) to $\mathcal{F}_{\text{ZKP-S}}$.
  - (Subround $ij_3$). CPs continue iff they receive (Proof, $\text{CP}_j$, $(\vec{N}^{i(j-1)}, \vec{N}^{ij})$).
- CPs take the first ciphertext in the resulting vector, setting $\vec{n}_i = \vec{N}_1^{im}$
- (Blame). In each round $j$ all CPs expect to begin with a known ciphertext, receive a message, and a proof that verifies. If a $\text{CP}_j$ uses the incorrect ciphertext, is blamed through $\mathcal{F}_{\text{BC}}$ sending the shuffled ciphertext, or the proof fails, CPs blame $\text{CP}_j$ and exit.

$\vec{n}$ is then a vector of length $n$.

(4) **Input Submission**.
(a) (Round 1). Each $\text{DP}_j$ sends its plaintext vector $\vec{c}^j$ to all CPs using $\mathcal{F}_{\text{BC}}$.
(b) (Round 2). CPs verify they received vectors of ciphertexts of the correct length and that the proofs from the Data Collection phase verify from each DP. For each remaining $\text{DP}_j$, CPs encrypt each component of the vector of counters $\vec{c}^j$ with the session key $y$, and componentwise homomorphically add the ciphertexts to construct a new aggregate data vector $\vec{d}$: $\vec{d}_i = \bigoplus_j \left(b_i^j \oplus (g, y \cdot g^{c_i^j})\right)$ where $\oplus$ denotes homomorphic addition of plaintexts.
(c) Blame cannot arise in this phase since DPs cannot be blamed, only excluded from the tally.

(5) **Shuffling**. All CPs set $\vec{s}^0$ as the concatenation of $\vec{n}$ and $\vec{d}$.
(a) (Round $j \in 1..m$).
(i) (Subround $j_1$). $\text{CP}_j$ shuffles $\vec{s}^j = \text{Shuffle}(\vec{s}^{(j-1)}, \vec{r}^j, \pi^j)$ and sends $\vec{s}^j$ to $\mathcal{F}_{\text{BC}}$ using random integers $\vec{r}^j \in \mathbb{Z}_q^{(b+n)}$ and a random permutation $\pi^j$ on $b+n$ elements.
(ii) (Subround $j_2$). $\text{CP}_j$ sends (ZKP, $(\vec{s}^j, \vec{s}^{(j-1)})$, $(\vec{r}^j, \pi^j)$) to $\mathcal{F}_{\text{ZKP-S}}$.
(iii) (Subround $j_3$). CPs continue iff they receive (Proof, $\text{CP}_j$, $(\vec{s}^j, \vec{s}^{(j-1)})$), otherwise they blame $\text{CP}_j$.

(b) (Blame). The phase is structured identically to Noise Generation so blame arises in exactly the same way.

(6) **Rerandomization-Decryption**. CPs set $\vec{p}^0 \leftarrow \vec{s}^m$.

(a) (Round $j \in 1..m$).

(i) (Subround $j_1$). $\text{CP}_j$ selects random $\vec{\sigma}^j$, $\vec{r}^j \in \mathbb{Z}_q^{(b+n)}$. For each $i$, denote the ciphertext $\vec{p}_i^{(j-1)}$ as $(a_i, b_i)$. Then $\text{CP}_j$ calculates for each $i$, $\alpha_i \leftarrow (a_i g^{\sigma_i})^{r_i}$ If $\alpha_i$ is the identity element, $\text{CP}_j$ restarts subround $j_1$ with fresh random values. Otherwise, $\text{CP}_j$ calculates $\beta_i = (b_i y^{\sigma_i})^{r_i} \alpha_i^{-x_j}$ and sets $\vec{p}_i^j \leftarrow (\alpha_i, \beta_i)$ and sends $\vec{p}^j$ to $\mathcal{F}_{\text{BC}}$.

(ii) (Subround $j_2$). $\text{CP}_j$ sends (ZKP, $(y_j, \vec{p}^j, \vec{p}^{(j-1)})$, $(\vec{\sigma}^j, \vec{r}^j, x_j)$) to $\mathcal{F}_{\text{ZKP-RRD}}$.

(iii) (Subround $j_3$). CPs continue iff they receive (Proof, $(y_j, \vec{p}^j, \vec{p}^{(j-1)})$), the first component of $\vec{p}_i^j$ is not the identity element, and the $y_j$ in this message is the same as the recorded $y_j$ from key exchange, otherwise they blame $\text{CP}_j$.

(b) (Blame). $\text{CP}_j$ is blamed in this round if $\text{CP}_j$ is blamed by $\mathcal{F}_{\text{BC}}$ during subround $j_1$, if the proof fails to verify or appear in subround $j_2$, or if the first element of $\vec{p}_i^j$ for any $i$ is the identity.

(7) **Output**. CPs count the number of nonzero plaintexts in the vector $\vec{p}^m$ and subtract the expected number of noise counters $n/2$ to output the final measurement.

## 6 SECURITY PROOF

### 6.1 Consensus Broadcast

While certain signed messages may be used as proof of malicious behavior, in order to provide a robust accountability property against denial of service attacks we require a method by which parties can agree on whether not a given message has or has not been sent, which means we require consensus broadcast, encapsulated in a broadcast functionality $\mathcal{F}_{\text{BC}}$ defined in Figure 2. To achieve a consensus broadcast, we use the Dolev-Strong protocol.

$\mathcal{F}_{\text{BC}}$

(1) **(Round 1)** Wait for a message from the broadcaster $B$.

(2) **(Round 2)** If $B$ sent a message $m$, send $m$ to all parties and exit. Otherwise, send (Blame, $B$) to all parties and exit.

Fig. 2. $\mathcal{F}_{\text{BC}}$, the ideal broadcast functionality.

*6.1.1 Dolev-Strong.* The Dolev-Strong protocol is thoroughly described and proved correct by the authors [20], in that all parties come to consensus on the (possibly empty) set of messages sent by $B$. For our specific application, when the size of this set exceeds 1 for any honest party it may simply send these two messages to all parties and terminate, blaming the broadcaster for equivocating. We briefly sketch the protocol below for completeness and add modifications required to include the blame messages, and finally describe optimizations that will increase performance in the case that malicious behavior does not occur.

(1) The broadcaster $B$ sends ($m$, Sign($M$)) to all parties, outputs $m$ and terminates. All other parties $P_j$ initialize a set accept$_j$.

(2) For rounds $j \in 1..m$,

- Upon receiving a message $M$ with valid signatures from $j$ distinct parties (including $B$) of $M$, add $M$ to the set accept.

- If the cardinality of ACCEPT increased this round, sign up to two new additions to ACCEPT chosen arbitrarily and send these messages with all of their signatures to every party in the next round.

(3) In round $m + 1$, if there is only one message in ACCEPT, output it. Otherwise, output (BLAME, $B$).

*Optimizations.* We observe that in the scenarios we intend to deploy the protocol, the number of CPs $m$ is small, and since the protocol has an accountability property, there is a cost to malicious behavior (e.g., being removed as a participant in the protocol in an out-of-band process) so this behavior should be uncommon. In this case, we can optimize the broadcast protocol by having each party explicitly indicate to each other party in a given round when there is no message to send with a "heartbeat" message. This optimization allows for an all-honest group of CPs to quickly complete a broadcast without waiting for m rounds to complete, at the cost of increasing the communication cost of the protocol with these heartbeat messages proportional to $m^2$ per-CP.

In addition, we can optimistically hope that the broadcaster sends the data: when a non-broadcaster party echoes the message to another party, they can simply echo the hash, and let the party respond as to whether or not it has already received the message. If so, nothing needs to be done with this message so we are done. If not, the sender sends the full message as well.

### 6.2 Rerandomization-Decryption Σ-Protocol

We outline and prove correct a Σ-protocol that proves a CP has performed a rerandomization, a re-encryption and a partial decryption.

THEOREM 6.1. *Suppose we have a ciphertext $(A, B)$, with $g, y_i, y$ publicly known with $y_i = g^{x_i}$, $y = \prod y_i$ and we wish to present $(\alpha, \beta)$ as a re-encryption rerandomization and partial decryption of $(A, B)$ so that $\alpha = (Ag^{\sigma})^r$, $\beta = (By^{\sigma})^r \alpha^{-x_i}$ for some random shift value $\sigma$ and rerandomization value $r$ and the private key $x_i$ of party $i$. Then $A, B, \alpha, \beta, g, y$ are known to both the prover and the verifier. We describe the proof:*

(1) *The Prover $P$ selects $t_1, t_2, t_3$ at random and sends $T_1 = A^{t_1} g^{t_2}$, $T_2 = B^{t_1} y^{t_2} \alpha^{t_3}$, $T_3 = g^{t_3}$*
(2) *The Verifier $V$ sends a random challenge $c$ to $P$.*
(3) *The Prover sends*

$$r_1 = rc + t_1 \qquad r_2 = \sigma rc + t_2 \qquad r_3 = -x_i c + t_3$$

(4) *The Verifier accepts the proof if and only if the following three equations hold:*

$$A^{r_1} g^{r_2} = \alpha^c T_1 \qquad B^{r_1} y^{r_2} \alpha^{r_3} = \beta^c T_2 \qquad g^{r_3} = y_i^{-c} T_3$$

*The above interactive proof is an HVZK proof that proves knowledge of $r, \sigma, x_i$ such that the equations above for $\alpha, \beta$ hold.*

PROOF. (1) **Completeness.** The proof scheme is clearly complete: $P$ knows or generates $r, \sigma, t_1, t_2, t_3, -x_i$ so that it can properly generate $r_1, r_2, r_3$. Given these values, it is easy to see the three equations hold.

(2) **Special Soundness.** Suppose the prover provides two proofs with the same commitment values $t_1, t_2, t_3$, with challenges $c_1$ and $c_2$. Then we have:

$$r_1 = rc_1 + t_1 \qquad r_1' = rc_2 + t_1$$
$$r_2 = \sigma rc_1 + t_2 \qquad r_2' = \sigma rc_2 + t_2$$
$$r_3 = -x_i c_1 + t_3 \qquad r_3' = -x_i c_2 + t_3$$

Then it is easy to see that $r = \frac{r_1 - r_1'}{c_1 - c_2}$ and then $\sigma = \frac{r_2 - r_2'}{r(c_1 - c_2)}$ and that $-x_i = \frac{r_3 - r_3'}{c_1 - c_2}$ so that special soundness is satisfied.

(3) **Honest Verifier Zero Knowledge.** We define a simulator as follows: The simulator behaves as an honest prover does until it receives $c$, then rewinds $V$, selects random $r_1, r_2, r_3$ and sets

$$T_1 = \frac{A^{r_1} g^{r_2}}{\alpha^c} \qquad T_2 = \frac{B^{r_1} y^{r_2} \alpha^{r_3}}{\beta^c} \qquad T_3 = \frac{g^{r_3}}{y_i^c}$$

Since $V$ is an honest verifier, it, given the same randomness, provides the same challenge $c$ and the equations required for $V$ to verify both hold and the simulation is successful.

□

Finally, having provided the Σ-protocol and proof, we apply the compiler from Section 6.3 to convert it to an accountable UC secure protocol which UC-realizes the functionality defined in Figure 3, parameterized by the group element $g$, joint public key $Y$, and public keys $y_i$.

### 6.3 Accountable Σ-Protocols

In order to satisfy the desired accountability property for the CPs we provide accountable zero-knowledge proofs. A natural way to solve this problem is with a non-interactive zero-knowledge proof. While there are general non-interactive zero-knowledge proof constructions for any relation secure in the UC model [35] without a random oracle, they are theoretical. Instead, we provide one-to-many zero-knowledge proofs for the computations in the protocol which provide an accountability property with respect to the other participants in the protocol, but not public verifiability. Hazay and Nissim present a compiler which converts any Σ-protocol into a UC-secure zero-knowledge proof. Given an authenticated broadcast channel, we follow this approach and provide a similar compiler which converts any Σ-protocol into an accountable UC-secure one-to-many zero-knowledge proof. The two pieces required for this are a coin flipping protocol and a commitment protocol, both of which must be made accountable against faulty messages from CPs as well as instances where CPs prevent the protocol from completing by simply not sending messages required for the protocol to continue.

We obtain termination through a broadcast channel realized by the Dolev-Strong protocol [20], so that in every round all honest parties agree on the message sent by a broadcaster. This message may be empty or the broadcaster may equivocate, but in either case all honest parties agree that the broadcaster is faulty and the honest parties blame the broadcaster and exit.

The one-to-many commitments outlined in Canetti [12], $\mathcal{F}_{\text{MCOM}}^{1:M}$, are non-interactive and so can be sent over the broadcast channel. We use these commitments over the broadcast channel to construct an accountable multi-party coin flipping protocol, $\mathcal{F}_{\text{COIN}}$. The construction, functionality, and proof are natural and straightforward, and so we present these details in Appendix A.

Finally, we outline how the UC-secure zero-knowledge proof of a shuffle given by Wikström [70] can be constructed with these two accountable primitives as well, providing an accountable UC-secure verifiable shuffle.

We realize the following zero-knowledge proof functionality given in Figure 3 with the protocol provided in Figure 4, all parameterized by a relation $R$. We assume knowledge of the statement $x$ to

be proved by all parties when the functionality is invoked, and that the identity of the prover is known and fixed.

$\mathcal{F}_{\text{ZK}}^{\mathcal{R}}$

- **(Round 1)**. Wait to receive the message (PARTICIPATE) from each non-prover party, and a single message from the prover.
- Drop messages from $P$ that are not of the form (ZKP, $x$, $w$) with $R(x, w)$ and any other message from any non-prover CP not of the form (PARTICIPATE).
- **(Round 2)**. If the set $L$ of CPs from which the functionality has not received a valid message is non-empty, broadcast (BLAME,$L$). Otherwise, broadcast (PROOF, $x$).

Fig. 3. $\mathcal{F}_{\text{ZK}}^{\mathcal{R}}$, the ideal functionality for an interactive one-to-many zero-knowledge proof

$\pi_R$

Given a security parameter $n$, a statement $x$ known by all parties, and a $\Sigma$-protocol $\pi$ for the relation $R$. For clarity, we split the protocol into four rounds, but note that Rounds 1 and 3 consist of multiple formal communication rounds.

- The prover runs $\pi$ with the statement $x$ and its witness $w$, selecting $n$ random commitments and $n$ random challenges, then calculating the response for each of these pairs. Then the prover rewinds each of these execution and uses fresh random challenges for each, calculating the correct response to each of these rewound executions as well. From these executions it recovers:
  - $n$ distinct (commitment) messages as $\alpha_1, \ldots, \alpha_n$
  - $n$ pairs of random challenges $(\beta_1^0, \beta_1^1), \ldots, (\beta_n^0, \beta_n^1)$ with $\beta_i^0 \neq \beta_i^1$ for each $i$
  - $n$ pairs of responses $(\gamma_1^0, \gamma_1^1), \ldots, (\gamma_n^0, \gamma_n^1)$ such that for each $i$, $(\alpha_i, \beta_i^0, \gamma_i^0)$ and $(\alpha_i, \beta_i^1, \gamma_i^1)$ both form valid transcripts for $\pi$.
- **(Round 1)** The prover sends vectors $(\vec{\alpha}, \vec{\beta}^0, \vec{\beta}^1)$ to $\mathcal{F}_{\text{BC}}$ and commits individually to each component of the vectors $\vec{\gamma}^0, \vec{\gamma}^1$ using $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$.
- **(Round 2)** After receiving a message of the form $(\vec{\alpha}, \vec{\beta}^0, \vec{\beta}^1)$ from $\mathcal{F}_{\text{BC}}$ on behalf of $P$ and $2n$ receipts from $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$, each verifier confirms that there are no duplicate values in the vector $\vec{\alpha}$ and that for every $i$, $\beta_i^0 \neq \beta_i^1$. If any of these checks fail, output (BLAME, $P$) and exit. Otherwise, every party sends (GENERATECOINS) to $\mathcal{F}_{\text{COIN}}$.
- **(Round 3)** Upon receiving $\vec{b}$ from $\mathcal{F}_{\text{COIN}}$, P opens for each $i$, $\gamma_i^{b_i}$ to all participants.
- **(Round 4)** Verifiers check that for each $i$, $(\alpha_i, \beta_i^{b_i}, \gamma_i^{b_i})$ is a valid transcript for $\pi$. If the prover does not open a commitment or any of the transcripts are not valid, verifiers output (BLAME, $P$). Otherwise, verifiers output (PROOF, $x$).
- If $\mathcal{F}_{\text{COIN}}$ or $\mathcal{F}_{\text{BC}}$ outputs a blame message, all parties output this message and exit.

Fig. 4. $\pi_R$, the hybrid-model protocol for an interactive one-to-many zero-knowledge proof

Theorem 6.2. *If $\pi$ is a $\Sigma$-protocol for $R$, $\pi_R$ UC-realizes $\mathcal{F}_{ZK}^{\mathcal{R}}$ in the static-corruption, synchronous, $(\mathcal{F}_{MCOM}^{1:M}, \mathcal{F}_{BC}, \mathcal{F}_{COIN})$-hybrid model if there is at least one honest CP.*

Proof. We follow the approach in [37], but we must consider modifications required to construct the proof in a a one-to-many setting, and we must account for blame messages required for accountability. We follow the definition of $\Sigma$-protocol for $\pi$ given by Damgård in [15] which defines two algorithms: a simulator $\pi_s$ for $\pi$ which accepts a uniformly random challenge value (i.e. one provided by an honest verifier) and outputs a transcript-triple indistinguishable from a real execution of $\pi$, and a witness extractor $\pi_e$ which accepts two transcripts for $\pi$ with identical first (commitment) values but distinct second (challenge) values, and outputs a valid witness $w$ for $\pi$. We define the simulator $S$ as follows. $S$ runs a copy of $A$ and runs every party honestly in the simulated real-model for the benefit of the environment $Z$ except where specified below:

**Honest Prover**.

- Simulating Round 1. $S$ selects $2n$ challenge values, $\vec{\beta}^0$, $\vec{\beta}^1$ and flips $n$ coins to generate a coin vector $\vec{b}$. $S$ invokes the simulator $\pi_s$ with $(x, \beta_i^{b_i})$ as statement and challenge. $S$ recovers the commitment, $\alpha_i$, and response, $\gamma_i^{b_i}$ from each transcript produced by $\pi_s$, generates $n$ random values $\gamma_i^{\neg b_i}$ and collects these into vectors $\vec{\alpha}$, $\vec{\beta}^0$, $\vec{\beta}^1$, $\vec{\gamma}^0$, $\vec{\gamma}^1$, and broadcasts and commits to these values as appropriate in the protocol $\pi_R$.
- Simulating Round 2. $S$ sends (GenerateCoins) on behalf of the honest parties, and sends (Participate) to $\mathcal{F}_{\text{ZK}}^{\mathcal{R}}$ for dishonest verifiers who sent (GenerateCoins) to $\mathcal{F}_{\text{COIN}}$ in this round.
- Simulating Round 3. This round only takes place if every dishonest verifier has sent GenerateCoins and $S$ has subsequently sent (Participate) on their behalfs. $S$ sends the vector $\vec{b}$ from Round 1 on behalf of $\mathcal{F}_{\text{COIN}}$ as a response to the verifiers. The prover is simulated honestly.
- Simulating Round 4. Honest verifiers are simulated honestly.

**Dishonest Prover**

- Simulating Round 1. $S$ recovers the vectors $\vec{\alpha}$, $\vec{\beta}^0$, $\vec{\beta}^1$, $\vec{\gamma}^0$, $\vec{\gamma}^1$ triples from the initial messages and commitments sent by $Z$ on behalf of $P$.
- Simulating Round 2. $S$ sends (Participate) to $\mathcal{F}_{\text{ZK}}^{\mathcal{R}}$ on behalf of the honest parties, and on behalf of the dishonest parties that send (GenerateCoins) to $\mathcal{F}_{\text{COIN}}$.
- Simulating Round 3. $S$ simulates $\mathcal{F}_{\text{COIN}}$ honestly, and if $\mathcal{F}_{\text{COIN}}$ sends a blame message instead of a bit vector, the protocol execution terminates. Otherwise, $S$ waits for $P$ to open its commitments. If $P$ opens the $n$ commitments and each of the $n$ transcripts is valid, $S$ checks the remaining $n$ unopened commitments. If any of these also forms a valid transcript, $S$ runs $\pi_e$ on one of these valid transcript pairs, recovering a witness $w$ and sends (ZKP, $x$, $w$) to $\mathcal{F}_{\text{ZK}}^{\mathcal{R}}$. If all of the remaining $n$ unopened commitments are invalid, $S$ halts and the simulation fails. If $P$ fails to open the $n$ commitments as required or any opened commitment generates an invalid transcript for $\pi$, $S$ sends $\perp$ to $\mathcal{F}_{\text{ZK}}^{\mathcal{R}}$ on behalf of $P$.
- Simulating Round 4. Honest verifiers are simulated honestly.

We claim that the view of $Z$ in the ideal model as it interacts with $S$ and $\mathcal{F}_{\text{ZK}}^{\mathcal{R}}$ is computationally indistinguishable from the view of $Z$ as it interacts with an adversary $A$ in the execution of $\pi_R$.

**Honest Prover**. We define a sequence of hybrid executions, beginning with the real execution of $\pi_R$ (which we note is formally an execution in a hybrid model, but which we denote "real execution" here to avoid ambiguity) which we label $H_0$. In this case, $P$ has a witness $w$, constructs $n$ valid transcript pairs, opens one set of commitments, and the honest verifiers accept the proof. We note that any partial execution terminated by malicious verifiers is a prefix of a full execution of the

protocol, with a blame component. In the case the protocol is aborted early, the set of CPs to blame is identical in all cases: $S$ simply sends participation messages on behalf of dishonest verifiers exactly when these verifiers participate in $\mathcal{F}_{\text{COIN}}$. We define, for each $i > 0$, $H_i$ to be a hybrid where for each $j \leq i$, the execution is modified as follows:

- $P$ constructs transcripts and behaves in round 1 for each value with index $j > i$ as in the real execution $H_0$. For $\alpha_j, \beta_j^0, \beta_j^1, \gamma_j^0, \gamma_j^1$ with $j \leq i$, $P$ runs the protocol as $S$ does, generating these values using $\pi_s$ and without $w$.
- $\mathcal{F}_{\text{COIN}}$ behaves honestly except for $j \leq i$. For these values, $\mathcal{F}_{\text{COIN}}$ learns the value $b_j$ from $P$ determined in the simulation of Round 1 and if all parties send (GenerateCoins), $\mathcal{F}_{\text{COIN}}$ sends this mixed vector of $n$ values.

Lemma 6.3. *No PPT environment $Z$ can distinguish between $H_0$ and $H_n$ with non-negligible probability.*

Proof. It suffices to show that for $1 \leq i \leq n$, there is no PPT environment $Z$ that can distinguish with non-negligible probability between $H_i$ and $H_{i-1}$. Every message of the transcripts from any execution of these two adjacent hybrids are identical except for $\alpha_i, \beta_i^0, \beta_i^1, \gamma_i^0, \gamma_i^1$, and $b_i$. $b_i$, $\beta_i^1$, and $\beta_i^0$ are identically distributed in each hybrid, as they are generated according to the same distribution (uniform in each case) by a party not controlled by $Z$. $\gamma_i^{\neg b_i}$ is never observed by $Z$ in any execution of any hybrid. Then if $Z$ can with non-negligible probability distinguish between $H_i$, $H_{i-1}$, then $Z$ can distinguish between the transcript $(\alpha_i, \beta_i^{b_i}, \gamma_i^{b_i})$ constructed by the simulator $\pi_s$ (in $H_i$) and a real execution of $\pi$ (in $H_{i-1}$). Since $\pi$ is computational zero knowledge, these transcripts are computationally indistinguishable so no such $Z$ can exist. □

Then it suffices to show that the difference between $H_n$ and the ideal model $I$ is a view change. We begin with $H_n$ and introduce the simulator $S$, noting that $P$ and $\mathcal{F}_{\text{COIN}}$ in $H_n$ and $S$ in $I$ behave identically by construction. In $H_n$, $P$ does not use its witness $w$ so all of its messages can be constructed by the simulator, which also simulates $\mathcal{F}_{\text{COIN}}$, so that the coordination of the bits sent by $\mathcal{F}_{\text{COIN}}$ and the vector $\vec{b}$ generated by $P$ happens inside the execution of $S$. Honest CPs have no inputs and are simulated honestly so that their outputs always match the output of $\mathcal{F}_{\text{ZK}}^{\mathcal{R}}$. Finally, blame is determined based on the output of $\mathcal{F}_{\text{ZK}}^{\mathcal{R}}$, but in the ideal functionality makes this determination by simply determining whether the appropriate participation messages were sent by CPs and if the statement and witness $(x, w)$ from the prover are valid. We note that we assume the protocol with an honest prover is run with valid input, meaning that an honest prover always sends a valid witness to $\mathcal{F}_{\text{ZK}}^{\mathcal{R}}$.

**Dishonest Prover**

- In Round 1, 2, and 4 the simulator behaves exactly as instructed by $Z$. In round 3, the we observe that the probability that the simulation fails is the probability that the uniformly generated $n$ coins all land in the prover's favor, which is $\frac{1}{2^n} \in \text{Negl}(n)$.

□

### 6.4 Accountable Shuffles

We sketch an accountable construction of Wikström's proof of a shuffle in the UC model, arguing that the abov techniques can be applied to construct a multi-verifier shuffle proof.

The proof consists of a sequence of six messages. Two are commitments, two are simple values that can be broadcast, one consists of a set of primes that may be generated by public coins, outlined in detail in [70]. The final value that is missing is an RSA modulus which the prover cannot know. In the one-to-one case the verifier may trivially generate this value, but for our application it must

be jointly generated by a group of verifiers, with no verifier knowing the factorization. We note that accountable efficient distributed protocols [36] to construct a public RSA modulus have been proved secure in the UC model since this proof of a shuffle was originally presented, and may be used in our setting to generate the required parameters in a distributed verifier setting with a dishonest majority and with an accountability property. Finally, we note that extraction of a witness in the proof of the shuffle is done by extracting the witness directly from the ideal zero-knowledge proof of knowledge of the cleartext functionality $\mathcal{F}_{\text{ZK}}^{\mathcal{R}_C}$. While this functionality is realized in the original proof by a version of verifiable secret sharing, we note this can also be realized by our accountable proof $\mathcal{F}_{\text{ZKP-DL}}$, which we have realized above.

### 6.5 Session Key Generation and Distribution

While we assume a fixed PKI that distributes and certifies signing keys for every party, a new session key must be constructed for each invocation of the protocol, and it must be jointly constructed by all CPs in an accountable way and then distributed to the DPs so that all DPs agree on the joint key. We sketch how this may be done in the UC setting and outline a natural functionality to accomplish this, $\mathcal{F}_{\text{SKGD}}$. This functionality may be realized by letting each CP broadcast its public keys and

$\mathcal{F}_{\text{SKGD}}$

(1) **Round 1**. Upon receiving a message (GenerateKeys) from an honest party $P_j$, generate a new ElGamal key: $(x_j, g^{x_j})$ and send $(P_j, g^{x_j})$ to $A$ and all parties, and send $x_j$ to $P_j$. Upon receiving private keys $x_j$ from a malicious party $P_j$, send $(P_j, g^{x_j})$ to all parties. At the termination of Round 1, if any CP has not generated a key, collect those parties into a set $B$ and broadcast (Blame, $B$) to all CPs. If all CPs have generated keys, send the joint public key $y = \prod_j g^{x_j}$ to all DPs.

Fig. 5. $\mathcal{F}_{\text{SKGD}}$, an ideal functionality for ElGamal Key Generation and Distribution

accountably prove knowledge of the corresponding private keys (including the DPs) but this is prohibitively inefficient even in theory with the expected thousands of DPs. Rather, we sketch a more efficient method that mirrors our implementation but may be realized in the UC setting:

(1) CPs generate a private key $x_j$.
(2) CPs broadcast the corresponding public key $g^{x_j}$ to all other CPs using $\mathcal{F}_{\text{BC}}$.
(3) CPs prove knowledge of the discrete log $x_j$ accountably *to each other only* using $\mathcal{F}_{\text{ZKP-DL}}$.
(4) CPs each broadcast a signature of the joint public key $y = \prod_j g^{x_j}$, the product of the public keys of all CPs with signature $\sigma_j = \text{Sign}(\text{Session-Key}|sid|j|y)$ using a UC signature functionality.
(5) CPs each send $(y, \vec{\sigma})$ to every DP via a point-to-point authenticated channel $\mathcal{F}_{\text{AUTH}}$.
(6) DPs wait to receive a public key signed by all CPs. If they receive zero or more than one distinct signed key, the DP halts. Otherwise, they accept this single key and move forward with the protocol.
(7) Blame is assigned by all CPs if a CP does not send a message in one of the above phases as expected, or if they send an invalid message: the ZKP is not valid, the signature is not valid, or the signature is on the wrong value.

In this construction, each CP sends each DP a single message. We sketch the security proof: all CPs agree on the keys of each CP or one is blamed in the first phase. If one is blamed, the protocol

terminates. If not, the honest CP sends a valid key pair and vector of signatures to each DP, ensuring each DP has at least one valid message. Since one CP is honest, let it be $CP_h$, the UC signature functionality constructs at most one $\sigma_h$, so no DP receives more than one valid public key pair.

### 6.6 PSC Ideal Functionality

We give the ideal functionality for PSC in Figure 6.

$\mathcal{F}_{\text{PSC}}$

(1) **(Corruptions)** Wait until (CORRUPT-CP, $C$) is received from $A$. For each $j \in C$, add $\text{CP}_j$ to the list of corrupted parties.
(2) **(Collection Period)**. Initialize, for every $\text{DP}_j$, a bit vector $\vec{v}^j$ of $b$ zeros. Upon receiving a message of the form (OBSERVATION, $i$) from some $\text{DP}_j$, set $v_i^j \leftarrow 1$. Upon receiving (CORRUPT-DP, $j$) from $A$ during this period, add $\text{DP}_j$ to the list of corrupted parties. Note that past inputs to $\text{DP}_j$ are not revealed to $A$ upon corruption.
(3) **(Round 1)**. Sample the noise value as $N \sim \text{Bin}(n, 1/2)$, where $\text{Bin}(n, p)$ is the binomial distribution with $n$ trials and success probability $p$. Let $w_i = \max\left(v_i^1, \ldots, v_i^d\right)$, $1 \le i \le b$. Compute output $z \leftarrow \left(\sum_{i=1}^{b} w_i\right) + N - n/2$. Send $z$ to $A$.
(4) **(Round 2)**. Send $z$ to all CPs.
(5) **(Blame)**. If at any point, $\mathcal{F}_{\text{PSC}}$ receives a message of the form (BLAME, $B$) from $A$ where $B$ is a set of corrupted CPs, replay that message to every CP and terminate the protocol.

Fig. 6. $\mathcal{F}_{\text{PSC}}$, the Private Set-Union Cardinality ideal functionality

### 6.7 Simulator Definition

We define $S$, the ideal model adversary parametrized by a real-world adversary $A$, which interacts with $\mathcal{F}_{\text{PSC}}$ and an arbitrary environment machine $Z$. We assume a dummy adversary $A$ that simply relays messages to and from $Z$ since this generalizes all adversaries [12]. The primary challenges here are that $S$ must extract inputs of the corrupted parties before the result is calculated by $\mathcal{F}_{\text{PSC}}$ since these values are its inputs, and must fool $Z$ into believing the honest parties are executing the protocol faithfully on input given by $Z$. This means the behavior of honest parties with the "dummy" inputs from $S$ must be indistinguishable from honest parties executing the protocol with the true input from $Z$, and also that the final output of $\mathcal{F}_{\text{PSC}}$ is distributed as dictated by $Z$. Finally, $S$ must note observable malicious behavior that generates a blame message and forward these onto $\mathcal{F}_{\text{PSC}}$ accordingly. We define $CP_h$ as the last honest CP in the CP ordering.

### 6.8 Proof

Fix a security parameter $k \in \mathbb{N}$ and consider any PPT environment machine $Z$. $Z$ interacts with a given protocol $\pi$ by submitting environmental inputs to honest parties and reading their outputs, communicating with an adversary $A$, and corrupting parties at will according to our assumptions: DPs any time during the collection period, and CPs statically at the beginning of the protocol. We define the output of $Z \in \{0, 1\}$ in the real execution of $\pi_{\text{PSC}}$ as $\text{REAL}_{\pi_{\text{PSC}}, A, Z}(k)$ and the output of $Z$ in the ideal execution as $\text{IDEAL}_{\mathcal{F}_{\text{PSC}}, S, Z}(k)$, with the ideal functionality $\mathcal{F}_{\text{PSC}}$, simulator $S$, defined above. For each $k$ each of the above outputs is a random variable, taken over the random input tapes provided to the parties and functionalities in each execution. If we fix the random input to $Z$, and then observe that the output of $Z$ is a function of the execution trace it observes, so if

The ideal model simulator $S$

(1) **Key Generation**. $S$ simulates all parties honestly, extracting the private keys of each malicious CP as they are sent to $\mathcal{F}_{\text{SKGD}}$.
(2) **Data Collection**. $S$ simulates honest $\text{DP}_j$ by selecting a random value $r_{ij}$ for every plaintext counter $i$, and encrypting zero as its blind. $S$ provides these plaintext counters for $Z$ if $\text{DP}_j$ is corrupted adaptively. For malicious $\text{DP}_j$, $S$ extracts the plaintext of each blind for each bin $i$ using the keys extracted from $\mathcal{F}_{\text{SKGD}}$, saving these plaintexts as $r_{ij}$.
(3) **Noise Generation**. During the Noise Generation phase, when $\text{CP}_h$ submits its shuffle, $S$ instead constructs two fresh encryptions of zero and submits these on behalf of $\text{CP}_h$. $S$ simulates $\mathcal{F}_{\text{ZKP-S}}$ to tell all CPs that this "shuffle" is valid.
(4) **Input Submission**. $S$ sends $-r_{ij}$ for every counter on behalf of every honest $\text{DP}_j$, submitting zero for every counter. $S$ collects the counters from every $\text{DP}_j$ as $c_{ij}$. For each bin $i$, $S$ checks if $\sum_j r_{ij} + c_{ij}$ is the identity. If not, $S$ submits (Observation, $i$) to $\mathcal{F}_{\text{PSC}}$ on behalf of $\text{DP}_j$.
(5) **Shuffling**. $S$ simulates honest CPs honestly except for $\text{CP}_h$, replacing its submission during this phase with a vector of fresh encryptions of 0.
(6) **Rerandomization-Decryption**. $S$ simulates all honest CPs honestly except for $CP_h$. $CP_h$ is simulated as follows: Once $S$ receives $z$ from $\mathcal{F}_{\text{PSC}}$, $S$ encrypts a vector containing $z + n/2$ random non-identity group elements and $(n + b) - z - n/2$ encryptions of 0, where encryption is done for the remaining CPs: for a value $k$, $S$ encrypts $E_{(h+1),(h+2)\ldots,m}(k; r)$ using the convention that if $m = h$, no encryption is done and this is a plaintext element $g^k$. $S$ then shuffles this vector of encrypted values (or plaintexts, in the case that $h = m$), and submits it on behalf of $CP_h$, simulating $\mathcal{F}_{\text{ZKP-RRD}}$ so that it attests to the validity of this vector as a correct rerandomization-decryption of the previous vector.
(7) **Conclusion and Blame**. If any of the accountable functionalities produce a blame message for a set of CPs $B$, or if some set $B$ of malicious parties do not send a valid message in some round, $S$ sends a message (blame, $B$) to $\mathcal{F}_{\text{PSC}}$ where $B$ is the set of blamed CPs.

Fig. 7. The ideal model simulator $S$

the messages sent by other parties in two distinct protocol runs with the same $Z$ are identically distributed, the output of $Z$ in each case is identically distributed as well.

We define the following sequence of (sequences of) hybrid executions which are protocols that interact with $Z$ defined by incremental modifications from $\text{REAL}_{\pi_{\text{PSC}}, A, Z}(k, z)$, where each time the behavior of honest parties is adjusted, using information collected by the hybrid itself from its interaction with $Z$. These hybrids "simulate" the execution for the benefit of $Z$ as the simulator does, but generally can recover inputs that $S$ does not have access to like the direct inputs from $Z$ to the honest parties, the goal being that the behavior of the last hybrid in this sequence is the behavior of $S$. Each incremental transition is a transition between two executions that are identical (meaning the distribution of messages sent by honest parties is constructed identically before and after the change in hybrid executions) or are computationally indistinguishable assuming the Decisional Diffie-Helman assumption for our group $G$. We denote a hybrid execution with a single letter (e.g. $H$) with the fixed $k$, $Z$ implicit.

Since the ideal-model simulator $S$ does not know the inputs of honest parties, we transition from $\text{REAL}_{\pi_{\text{PSC}},A,Z}(k,z)$ by replacing the messages sent by honest parties with "dummy" values, which will be produced by $S$ in the ideal model. We reduce the environment $Z$'s ability to distinguish between two hybrid executions to winning the IND-CPA game for ElGamal, but in order to do this the hybrid must correctly "decrypt" a value encrypted to the public key of the IND-CPA challenger, which means we must recover these submitted secrets by other means to faithfully reproduce and swap-in the final result expected by $Z$ based on the inputs it has submitted.

The hybrid executions record and reproduce these final results in order to provide messages sent by honest parties that replace decryption of ciphertexts for which they do not have the keys. We formalize this process in a "Ghost Execution".

*6.8.1 The Ghost Execution and "Fixing" the Simulation.* In the last phase of the protocol, Rerandomization-Decryption, we observe that the last honest $\text{CP}_h$ produces a vector of ciphertexts (or plaintexts, if $m = h$) as its output, but if we fix the plaintexts in the vector, then the output of $\text{CP}_h$ is a vector where each component is a uniformly random fresh encryption of its corresponding plaintext multiplied by a uniformly random nonzero Rerandomization factor. This is because $\text{CP}_h$ re-encrypts the ciphertext alongside the rerandomization-decryption. This means if the plaintexts are known to $\text{CP}_h$ in this phase, the output it constructs by honestly decrypting the vector it received to Rerandomize-Decrypt is identical in distribution to the output it constructs by constructing a fresh encryption of the decryption of that plaintext times a uniformly random nonzero value. We call this replacement process in the final phase "fixing the execution".

Any hybrid execution can extract parameters used for each operation in every phase by all parties, and use them to reconstruct the plaintext in the last phase as required, since each component that goes into the calculation of these plaintexts is constructed by an honest party (and therefore run by the hybrid directly) or for malicious parties, extracted as follows:

(1) Blinds from malicious parties are extracted from $\mathcal{F}_{\text{ZKP-DL}}$.
(2) Malicious parties' shuffles in noise generation and shuffling are extracted from $\mathcal{F}_{\text{ZKP-S}}$.
(3) The Rerandomization-Decryption factors are extracted from $\mathcal{F}_{\text{ZKP-RRD}}$.

Hybrid executions keep track of these values, extracting the values above selected by $Z$, and recording the values selected by honest parties. The values are stored in a parallel execution called the ghost execution, and in the last phase of the protocol, Rerandomization-Decryption, $\text{CP}_h$ uses these plaintexts constructed through direct extraction rather than its input. $\mathcal{F}_{\text{ZKP-RRD}}$ indicates to all other CPs that these values are a correct Rerandomization-Decryption as expected, and in this way we "fix the execution".

*6.8.2 Blind Submission.* We set $B_{10} = \text{REAL}_{\pi_{\text{PSC}},A,Z}(k)$. We define a sequence of $db$ executions:

$$B_{\text{SEQ}} = \langle B_{10}, B_{11}, B_{12}, \ldots, B_{1d}, \ldots, B_{2d}, \ldots, B_{bd} \rangle$$

where each $B_{ij}$ with $i \in 1..b$, $j \in 1..d$. Define $B_{ij}$ by:

- Run $B_{10}$ honestly, except that for counter $t$ and dp $D$, during blind submission, if $D$ is honest and either $t < i$ or $t = i$ and $D \leq j$, submit a fresh encryption of zero on behalf of $D$ for its blind in bin $t$. Record a random blind $b$ in the Ghost Execution for each of these honest DPs, and save as their corresponding plaintext counter $-b$.
- $\text{CP}_h$ fixes the simulation as described in 6.8.1, submitting values from the Ghost Execution as its message during the Rerandomization-Decryption phase.

Lemma 6.4. *Every adjacent pair of hybrid executions in $B_{SEQ}$ is computationally indistinguishable.*

Proof. Suppose we have two adjacent hybrid executions $B_l$, $B_r$. We note that by construction, in each adjacent pair of hybrid executions the behavior of honest parties differs by exactly one ciphertext submitted by one DP. Assume there exists a PPT environment $Z$ that can distinguish between these executions. We define an algorithm $A$ that runs $Z$ and claim that the advantage of $Z$ distinguishing $B_l$ and $B_r$ is the same as the advantage of $A$ in the IND-CPA game for ElGamal. $A$ plays the IND-CPA game for ElGamal:

(1) A accepts a public key $y_c$ from the challenger in the IND-CPA game. A runs an execution of $B_l$ and replaces $y_h$ with $y_c$ in the protocol. Each time a functionality (more precisely, ZKPs in the Rerandomization-Decryption phase and $\mathcal{F}_{\text{SKGD}}$) requires $\text{CP}_h$ prove knowledge of the secret key, the ZKP functionality accepts the "proof" provided by $\text{CP}_h$.
(2) For the blind in bin $t$ submitted by DP $D$, $A$ calculates a random blind $b$ value as an honest DP would and sends the pair of plaintexts $(b, 0)$ to the challenger in the IND-CPA game and receives a ciphertext $C$.
(3) A sends $C$ as the ciphertext blind for bin $t$ on behalf of DP $D$, further encrypting $C$ to the public keys of all other CPs.
(4) A sets $-b$ as the plaintext counter for bin $t$ and simulates the DP exactly as in $B_r$.
(5) Instead of decrypting, $A$ replaces each output of $\text{CP}_h$ during Rerandomization-Decryption as described in 6.8.1 and gives to the IND-CPA challenger the output of $Z$.

If the challenger encrypts $b$, the blind for $t$ is random and the plaintext is its opposite and this is an execution of $B_l$. If the challenger encrypts 0, the "fixed" value output by $\text{CP}_h$ is the same as before, but this is an execution of $B_r$.

Therefore $A$ has the same nonzero advantage in the IND-CPA game for ElGamal that $Z$ does distinguishing between $B_l$, $B_r$ which is non-negligible, violating the DDH assumption.

□

*6.8.3 Data Collection.* We set $D_{10} = B_{bd}$. We define a sequence of $db$ hybrid executions:

$$D_{\text{SEQ}} = D_{10}, D_{11}, D_{12}, \ldots, D_{1d}, \ldots, D_{2d}, \ldots, D_{bd}$$

where each $D_{ij}$ with $i \in 1..b$, $j \in 1..d$. Define $D_{ij}$ by:

- Run $D_{10}$ honestly, except that when a currently-honest DP $D$ is instructed to record an observation for counter $t$, if $t < i$ or $t = i$ and $D \leq j$, ignore it. Add a random value $r_t$ to the sum for counter $t$ in the ghost execution.
- $\text{CP}_h$ fixes the simulation as described in 6.8.1.

Lemma 6.5. *Every adjacent pair of hybrid executions in $D_{\text{SEQ}}$ is identically distributed.*

Proof. Suppose we have two adjacent hybrid executions $D_l$, $D_r$. We note that by construction each adjacent pair of hybrid executions differs by at most one counter $t$ submitted by one honest $\text{DP}_j$. If no instruction of the form (Observation, $t$) is sent to $\text{DP}_j$ while $\text{DP}_j$ is honest, the two executions do not differ at all. So assume such a message is received. Before this occurs, $\text{DP}_j$ stores a value $-b$ for counter $t$, and the ghost execution has recorded $b$ as the corresponding blind. Then, in $D_l$, when $\text{DP}_j$ is sent an Observation message, $\text{DP}_j$ replaces $-b$ with a fresh random value $b'$. In $D_r$, $\text{DP}_j$ sets $-b$, but the blind for $\text{DP}_j$ for counter $t$ is replaced with a fresh random value $r_t$ in ghost execution. We observe now that $\text{DP}_j$ may be adaptively corrupted at this point, or it may not. Either way, crucially the plaintext counter stored for $\text{DP}_j$ for counter $t$ is revealed to $Z$ *after* the observation has taken place, and the changes described above have been made. We describe the differences between these executions, which consist of the value of the plaintext counter for $t$ held by $\text{DP}_j$ and the sum of the inputs for counter $t$ by $\text{DP}_j$ in the ghost execution after the Input Submission phase.

- In $D_l$, $\mathrm{DP}_j$'s plaintext counter is $b'$, a uniformly random value unrelated to any other value. The sum in the ghost execution for the submissions on behalf of $\mathrm{DP}_j$ is $b$ and a value selected by $Z$ with no knowledge of $b$.
- In $D_r$, $\mathrm{DP}_j$'s plaintext counter is $-b$, where $b$ is selected uniformly at random. The sum in the ghost execution for the submissions on behalf of $\mathrm{DP}_j$ is $b + r_t$ and an element selected by $Z$ with knowledge of $-b$.

The plaintext counters $b'$ and $-b$ are both uniformly random values ($b'$ is chosen this way, $-b$ is random since $b$ is). The sums in the ghost execution are uniformly random as well, since they both contain a summand which is uniformly random ($b$ in $D_l$ and $r_t$ in $D_r$), and about which $Z$ never learns any information. □

*6.8.4 Noise Generation.* We set $D_{bd} = N_0$ and define a sequence of hybrid executions beginning with the final execution in the previous step:

$$N_{\mathrm{SEQ}} = \langle N_0, N_1, \ldots, N_n \rangle$$

- $N_j = N_{j-1}$ except that during the Noise Generation phase, for rounds $i \leq j$, when $\mathrm{CP}_h$ calculates $\vec{N}^{ih}$, it instead replaces this vector with a vector of two fresh distinct encryptions of zero and $\mathcal{F}_{\text{ZKP-S}}$ indicates to all other CPs that these ciphertexts are a shuffle of the previous ciphertexts.
- $\mathrm{CP}_h$ fixes the simulation as described in 6.8.1.

Lemma 6.6. *Every adjacent pair of hybrid executions $N_j$, $N_{j+1}$ is computationally indistinguishable.*

Proof. Suppose an environment $Z$ can distinguish between the two adjacent executions. We construct an IND-CPA adversary $A$ that has non-negligible advantage in the IND-CPA game for ElGamal. We play the IND-CPA game for ElGamal:

(1) $A$ accepts a public key $y_c$ from the challenger in the IND-CPA game. $A$ runs $N_j$ with $y_c$ as the public key for $\mathrm{CP}_h$ until the Noise Generation phase. Proof functionalities that require knowledge of the secret key for $y_c$ accept $\mathrm{CP}_h$'s proofs as correct.
(2) $A$ sends the plaintexts (0, 1) to the challenger and receives a ciphertext $C$.
(3) During Noise Generation, $\mathrm{CP}_h$ performs the shuffle as before but replaces the ciphertext that is an encryption of 1 with $C$, encrypted with the keys of the remaining CPs.
(4) $A$ fixes these ciphertexts during Rerandomization-Decryption as described in Section 6.8.1.
(5) $A$ submits the output of $Z$ to the challenger in the IND-CPA game.

Then we observe that if the challenger encrypts 1, we have a fresh encryption of 1 and our new re-encryption of 0 in a random order, which is a valid shuffle of the previous ciphertexts and we are in execution $N_j$. If the challenger encrypts 0, we submit two fresh encryptions of 0 on behalf of $\mathrm{CP}_h$ and this is $N_{j+1}$. Then our advantage in the IND-CPA game is the same as the advantage of $Z$ distinguishing between $N_j$, $N_{j+1}$ which is negligible. □

*6.8.5 Shuffling.* We define a sequence of hybrid executions beginning with the previous transition:

$$N_n = S_0, S_1, \ldots S_{n+b}$$

- $S_j = S_{j-1}$ except that during the Shuffling phase, when $\mathrm{CP}_h$ calculates $\vec{d}^h$, for components $i \leq j$, $\mathrm{CP}_h$ broadcasts an encryption of zero and $\mathcal{F}_{\text{ZKP-S}}$ indicates to all other CPs that these ciphertexts are a shuffle of the previous ciphertexts.
- $\mathrm{CP}_h$ fixes the simulation as described in 6.8.1.

Lemma 6.7. *Every adjacent pair of hybrid executions $S_j$, $S_{j+1}$ is computationally indistinguishable.*

Proof. Suppose an environment $Z$ can distinguish between the two adjacent executions. We construct an IND-CPA adversary $A$ that has non-negligible advantage in the IND-CPA game for ElGamal. We play the IND-CPA game for ElGamal:

(1) $A$ accepts a public key $y_c$ from the challenger in the IND-CPA game. $A$ runs $S_j$ with $Z$ with $y_c$ as the public key for $\mathrm{CP}_h$ until the Shuffling phase. ZKP functionalities that require the secret key for $\mathrm{CP}_h$ instead simply accept all proofs from $\mathrm{CP}_h$ without it.
(2) $A$ performs the shuffle and recovers the plaintext $t$ for component $j+1$. $A$ sends the plaintexts $(0, t)$ to the challenger and receives a ciphertext $C$.
(3) $A$ during Shuffling, on behalf of $\mathrm{CP}_h$, performs the shuffle as before but replaces the ciphertext for component $j+1$ with $C$, further encrypted with the keys of the remaining CPs.
(4) $A$ fixes the output during Rerandomization-Decryption as described in Section 6.8.1.
(5) $A$ submits the output of $Z$ to the challenger in the IND-CPA game.

Then we observe that if the challenger encrypts $t$ this is exactly $S_j$, while if the challenger encrypts 0 this is $S_{j+1}$. Then our advantage in the IND-CPA game is the same as the advantage of $Z$ distinguishing between $S_j$, $S_{j+1}$ which is negligible. □

*6.8.6 Introduction of the Ideal Functionality.* In $S_{n+b}$, during the Rerandomization-Decryption phase $\mathrm{CP}_h$ outputs a vector of encryptions of the values recorded in the Ghost Execution. This vector is completely characterized by three values: a number of nonzero elements $o \in \mathbb{N}$, a vector of plaintexts $\vec{o}$ of length $o$, and an permutation $\pi$ on $b+n$ elements that selects an ordering of the $o$ nonzero plaintexts and the remaining zeros. Any specific execution of any hybrid selects a precise value for each of these variables. We define the final hybrid $F$:

- Execute identically to $S_{n+b}$ except instead of "fixing" the final vector of outputs during Rerandomization-Decryption on behalf of $\mathrm{CP}_h$, extract only the number of nonzero counters from the Ghost Execution and save this value $o$.
- Construct a vector as follows: take $o$ nonzero elements uniformly at random, append zero $n+b-o$ times, apply a uniformly random shuffle to the vector then encrypt each component to the (possibly empty) joint public key $\prod_{j=h+1}^{m} y_j$. Send this as the output for $\mathrm{CP}_h$.

Lemma 6.8. *$F$ and $S_{n+b}$ are identically distributed.*

Proof. The executions are identical to the point where $\mathrm{CP}_h$ sends its output in the Rerandomization-Decryption phase by construction. Therefore, it suffices to show that the distribution of the output, characterized as $o$, $\vec{o}$, $\pi$ of $\mathrm{CP}_h$ in the two executions $F$, $S_{n+b}$ is identical.

We observe that $o$ is determined identically in both executions since it is selected before the processes formally diverge. In the execution $F$, $\vec{o}$ and $\pi$ are drawn uniformly at random among nonzero elements of $\mathbb{Z}_p$ (we refer to elements by their discrete log with respect to the generator $g$), and uniformly among permutations, respectively. We show this is the case in $S_{n+b}$ as well.

Consider a given element of $\vec{o}$. In $S_{n+b}$ it is constructed by application of a sequence of $h$ multiplications of nonzero elements $\mu_1 \ldots \mu_h$ to the original input, whether that input is generated at random by an honest party, constructed by noise, or submitted by malicious parties. Call the input $\alpha$. Then the output of $\mathrm{CP}_h$ for this component is $\beta = \alpha\mu_1 \ldots \mu_h$ and we observe that $\mu_h$ is selected by the honest $\mathrm{CP}_h$ uniformly at random when the value is produced, and that each $\mu$ is nonzero since CPs cannot submit zero rerandomization factors, and $\alpha$ is nonzero by the definition of $\vec{o}$. Therefore $\alpha\mu_1 \ldots \mu_{h-1}$ is nonzero hence invertible in a group of prime order, so a uniformly random selection of $\mu_h$ gives a uniformly random distribution among nonzero elements for $\beta$.

Second, consider $\pi$: $\pi$ is determined by application of the $m$ permutations $\pi = \pi_h \circ \cdots \circ \pi_1$ where each $\pi_i$ is the permutation selected by $\mathrm{CP}_i$ during the Shuffling phase and we observe that since every permutation is invertible, selection of $\pi_h$ uniquely determines $\pi$. $\pi_h$ is chosen

during shuffling but no information about the choice is revealed until $\text{CP}_h$ submits its values during Rerandomization-Decryption, so $\text{CP}_h$ may without altering the execution at all make this choice when its final output is constructed, selecting $\pi_h$ at random and ensuring $\pi$ is as well.

□

Proposition 6.9. *$F$ and the ideal model execution $\text{IDEAL}_{\mathcal{F}_{\text{PSC},S,Z}}(k)$ are statistically indistinguishable.*

Proof. After the sequence of transitions above, the honest parties in $F$ behave identically to how they are simulated by $S$ in Figure 7 except for the calculation of the value $o$, which is recovered directly from $\mathcal{F}_{\text{PSC}}$ in $\text{IDEAL}_{\mathcal{F}_{\text{PSC},S,Z}}(k)$, and extracted from the aggregate in the ghost execution during Rerandomization-Decryption in $F$. We need to show that the value $o$ in both executions is the same. We split the value $o$ (from either execution) into two nonnegative summands since components of the final vector consist of two disjoint sets: data counters, determined by aggregating inputs, and noise counters, determined by shuffling ciphertexts in the Noise Generation phase.

**Nonzero Counters from Data**.

Fix a data counter $t$. We show that $t$ contributes to the count $o$ (i.e., is nonzero) in $F$ if and only if it contributes in $\text{IDEAL}_{\mathcal{F}_{\text{PSC},S,Z}}(k)$. Throughout this section, we consider "plaintexts" to be the discrete log of these encrypted values with respect to $g$, and call a plaintext 0 if it is $g^0$ i.e. the group identity element.

For data counter $t$, we write $\sigma_t^F$ as the sum of $3j$ values in the execution of $F$: each $\text{DP}_j$ submits a blind during the beginning of the Data Collection phase. Following notation in the simulator definition, call the plaintext of this blind $r_{tj}$. Each $\text{DP}_j$ also submits a plaintext counter $c_{tj}$. Finally, the Ghost Execution has recorded some number of random increments for each honest-at-the-time $\text{DP}_j$ that receives Observation from $Z$, call this sum $s_{tj}$. Then the plaintext that is used by $\text{CP}_h$ in $F$ for counter $t$ is zero if and only if $\sigma_t^F = \sum_j (r_{tj} + c_{tj} + s_{tj})$. Further, since the processes do not formally diverge until after $\sigma_t$ is determined, the equivalent value that is used in $\text{IDEAL}_{\mathcal{F}_{\text{PSC},S,Z}}(k)$ by $S$ to determine whether to send an observation message to $\mathcal{F}_{\text{PSC}}$ is: $\sigma_t^I = \sum_j (r_{tj} + c_{tj})$ since $S$ has no Ghost Execution. We first observe that if a $\text{DP}_j$ is honest through the input submission phase, $r_{tj} + c_{tj} = 0$ since honest DPs submit 0. We note that if either of these sums is nonzero, it guarantees that counter $t$ will be nonzero in the final tally in that execution (in $\text{IDEAL}_{\mathcal{F}_{\text{PSC},S,Z}}(k)$, the observation is sent directly to $\mathcal{F}_{\text{PSC}}$ by $S$. In $F$, the value is inserted in the output of $\text{CP}_h$ during Rerandomization-Decryption directly).

(1) $\sigma_t^I = \sigma_t^F = 0$. This means no $c_{jt}$ has been incremented, meaning that no honest DP has been sent an Observation message, and the value of $t$ is the same in both executions, or some nonzero number of uniformly randomly selected values sum to zero, which is an event with negligible probability $\frac{1}{|G|}$.
(2) $\sigma_t^I \neq 0$, $\sigma_t^F \neq 0$. In this case, $t$ is forced to count in the tallies of both executions, as the observation is directly sent to $\mathcal{F}_{\text{PSC}}$ in $\text{IDEAL}_{\mathcal{F}_{\text{PSC},S,Z}}(k)$, and remains nonzero in $F$ since it is multiplied by a sequence of nonzero values.
(3) $\sigma_t^I \neq \sigma_t^F$, with one of these equal to zero. This means $\sum_j s_{tj} \neq 0$. If $\sigma_t^F = 0$, this means randomly selected nonzero $\sum_j s_{tj}$ is exactly $-\sigma_t^I$, a sum calculated with no knowledge of $\sum_j s_{tj}$, which occurs with negligible probability $\frac{1}{|G|}$. If $\sigma_t^I = 0$, then a then-honest DP has received some Observation message, so in $F$ $t$ is included in the tally since $\sigma_t^F \neq 0$, and in $\text{IDEAL}_{\mathcal{F}_{\text{PSC},S,Z}}(k)$ $t$ is included in the tally because (Observation, $t$) has been sent to some honest DP who has forwarded it directly to $\mathcal{F}_{\text{PSC}}$.

**Nonzero Counters from Noise**. The second portion of $o$ comes from the noise counters. In $\text{IDEAL}_{\mathcal{F}_{\text{PSC},S,Z}}(k)$, this is a number drawn by an ideal functionality directly from $\texttt{Bin}(n; \frac{1}{2})$. In $F$, it is

determined by the composition of permutations in each round $i$ of the Noise Generation phase, but the ciphertexts being shuffled in $F$ after $\text{CP}_h$ applies its shuffle for each bin of Noise Generation are both encryptions of zero, so all plaintexts for these ciphertexts are encryptions of 0 and carry no information about the honest permutation $\pi^{ih}$. This means $\pi^{ih}$ for each noise generation round $i$ in $F$ is selected external to the execution during Noise Generation, and no messages in the execution depend on $\pi^{ih}$ until Rerandomization-Decryption, so the $\pi^{ih}$ may be chosen then. For each round one of the two permutations on two elements provides a result of 1 and the other of 0. We draw all $n$ of them from these randomly, producing exactly the same distribution $\texttt{Bin}(n; \frac{1}{2})$.

Then we conclude the number $o$ is selected from the same distribution in both executions except with negligible probability, as the sum of a fixed number of nonzero data counters (a number which may differ in both executions but only with negligible probability) and a number which is drawn identically from the same distribution by honest parties in each case. □

Theorem 6.10. *The protocol $\pi_{PSC}$ $\pi_{PSC}$ UC-realizes $\mathcal{F}_{PSC}$ in the ($\mathcal{F}_{BC}$, $\mathcal{F}_{SKGD}$, $\mathcal{F}_{ZKP\text{-}S}$, $\mathcal{F}_{ZKP\text{-}RRD}$, $\mathcal{F}_{ZKP\text{-}DL}$)-hybrid model if there is at least one honest CP and all CP corruptions are static and the Decisional Diffie Hellman assumption holds for the group G.*

Proof. We have the sequence of computationally indistinguishable hybrid executions, beginning with $\text{REAL}_{\pi_{\text{PSC}}, A, Z}(k)$ and followed by the sequences $B_{\text{SEQ}}$, $D_{\text{SEQ}}$, $N_{\text{SEQ}}$, $S_{\text{SEQ}}$, with the last element of this last sequence indistinguihable from $\text{IDEAL}_{\mathcal{F}_{\text{PSC}, S, Z}}(k)$. Since the number of transitions is finite, and the distinguishing advantage for any PPT environment between any adjacent two is bounded by a negligible function, we arrive at the statement of the theorem.

□

## 7 IMPLEMENTATION AND EVALUATION

We constructed an implementation of PSC in Go to verify our protocol's correctness and to measure the system's computation and communication overheads. We run experiments over large synthetic datasets and measure our implementation's performance. We describe the implementation details and design choices in Section 7.1 and then present our performance evaluation in Section 7.2.

### 7.1 Implementation

We built an implementation of PSC in Go using the Kyber [45] advanced cryptographic library. ElGamal encryption is implemented in the Edwards 25519 elliptic curve group [64] and the CPs use Neff's verifiable shuffle [54] to shuffle the ElGamal ciphertexts.

The CPs perform secure broadcast using the optimized "heartbeat" version of the Dolev-Strong protocol described in Section 6.1.1. We do not implement time-outs that explicitly signal the end of a Dolev-Strong round. Therefore, termination is not guaranteed in our implementation *i.e.* honest CPs may wait forever on malicious CPs that do not send their response. However, this does not affect our results as we analyze the performance of our protocol in an honest setting *i.e.*, all CPs and DPs are considered honest (refer Section 7.2).

The CPs use the Schnorr signature algorithm [60] over the Edwards 25519 elliptic curve for signing and SHA-256 for computing digests. We use TLS 1.2 for secure point-to-point communication.

We use "Biffle" in the Kyber library for shuffling the noise vectors during the noise generation phase. Biffle is a fast binary shuffle for two ciphertexts based on generic zero-knowledge proofs.

For zero-knowledge proofs, we use Schnorr-type proofs [10, 60] for the combined re-encryption, re-randomization and decryption of ElGamal ciphertexts and knowledge of discrete log of the Elgamal public key and blinding factors. Non-interactive versions of all these proofs are produced using the Fiat-Shamir heuristic [28].

Table 1. Default values and descriptions for system parameters.

| Param. | Description | Default |
|---|---|---|
| $b$ | # of counters | 300,000 |
| $m$ | # of Computation Parties | 5 |
| $d$ | number of Data Parties | 30 |
| $\epsilon$ | privacy parameter | 0.3 |
| $\delta$ | privacy parameter | $10^{-12}$ |

Table 2. Actual, noisy aggregates, and standard deviation for various values of $\epsilon$.

| $\epsilon$ | Actual Agg. | Noisy Agg. | Standard Dev. |
|---|---|---|---|
| 0.15 | 8927 | 8752 | 141.92 |
| 0.30 | 8927 | 8811 | 70.96 |
| 0.45 | 8927 | 8887 | 47.31 |
| 0.60 | 8927 | 8873 | 35.48 |
| 0.75 | 8927 | 8917 | 28.39 |

A single DP program emulates all the DPs in our implementation *i.e.*, each operation (*e.g.*, blind submission, data collection, *etc.*) for every DP is performed sequentially by a single DP program. However, in a real deployment, the DPs would be distributed.

To encourage the use of PSC by privacy researchers and practitioners, we are releasing PSC as free, open-source software, available for download at https://github.com/GUSecLab/psc.

**Optimization.** While computing the aggregate, as an optimization the CPs first perform a secure broadcast of the aggregate received from the DPs. If a consensus can be reached among the CPs, then the agreed-upon aggregate is used. Otherwise, in the case where the CPs cannot reach a consensus (on the aggregate), they opt for the more communication-intense process of performing a secure broadcast (using the heartbeat Dolev-Strong protocol) for each individual DP blind and plaintext responses.

### 7.2 Evaluation

Experiments are carried out on 10 core Intel Xeon machines with 32GB to 64GB of RAM running Linux kernel 4.4.0. Our implementation of PSC is currently single-threaded. Although the computational cost of PSC's noise generation is significant and may be done by the CPs before the inputs are received, we parallelize it so that it can be done on multi-core machines after the inputs are received. We use "parallel for" from the Golang par package [18] for this parallel noise shuffling.

We instantiate all CPs and DPs on our 10 core servers. Google Protocol Buffers [67] is used for serializing messages, which are communicated over TLS connections between PSC's parties. We use Go's default crypto/tls package to implement TLS.

*Query and dataset.* We evaluate PSC by considering the query: what is the number of unique IP connections as observed by the nodes in an anonymity network? Although our implementation tolerates malicious CPs and dishonest majority of DPs, we analyze the performance of our protocol in an honest setting (*i.e.*, all CPs and DPs are considered honest) for experimental purposes.

Rather than store $2^{32}$ (or, for IPv6, $2^{128}$) counters, we assume $b$ counters (where $b \ll 2^{32}$) and map IP addresses to a $(\lg b)$-bit digest by considering the first $\lg b$ bits of a hash function; this results in some loss of accuracy due to collisions. For each experiment, we chose an integer uniformly at random from the range $[0, 30000]$. Then for each DP, we choose from its counters a random subset of that size to set to 1; the remaining counters are set to 0.

We note that the performance of PSC is independent of the number of unique IPs. Therefore, in our performance evaluation, we are interested in how the number of counters $b$ affects the operation of PSC, rather than the number of unique IPs.

*Experimental setup.* The default values for the number of bins $b$, the number of CPs $m$, the number of DPs $d$, $\epsilon$, and $\delta$ are listed in Table 1.

We determine these default values by considering which values would be appropriate for an anonymity network such as Tor. The Tor Project reports approximately 2.8 million user connections

(using simple statistical techniques to roughly estimate the number of Tor users) and over 3,000 guard nodes [65]. Assuming that 1% of Tor guards deploy PSC, we have $d = 30$. Also, given this level of PSC deployment, we would expect a Tor guard to see approximately 30,000 unique IPs (*i.e.*, 1% of ~3 million user connections).

To measure the aggregate with high accuracy, we limit hash-table collisions to at most a fraction $f$ of inputs in expectation by using a hash table of size $1/f$ times the number of inputs. Therefore, for $f = 10\%$, we set (unless otherwise specified) $b = 300,000$ in all our experiments. We set $\epsilon = 0.3$ as this is currently recommended for safe measurements in anonymity networks such as Tor [40]. To limit to $10^{-6}$ the chance of a privacy "failure" affecting any of $10^6$ users, we set $\delta$ to $10^{-12}$ [24]. We set these default values as system-wide parameters, unless otherwise indicated.

*Accuracy.* The trade-off between accuracy and privacy is governed by the choice of $\epsilon$ and $\delta$. We vary $\epsilon$ from 0.15 to 0.75, keeping the number of bins at $b = 300,000$. We found that values below 0.15 produced too much noise and offered low utility. Values of $\epsilon$ greater than 0.75 would not provide a reasonable level of privacy.

The actual and the noisy aggregate values along with the standard deviation for the noisy aggregates (the noise follows a binomial distribution) for different values of $\epsilon$ is shown in Table 2. We observe that the standard deviation of the noisy value is at most 141.92 even for smaller values of $\epsilon$, such as 0.15. Therefore, as expected the noisy aggregates are very close to the actual aggregates. In summary, PSC gives highly accurate results for all tested privacy levels.

**Communication cost.** To be practical, a statistics gathering system should impose a low communication overhead for the DPs, which can have limited bandwidth. However, we envision the CPs to be well-resourced dedicated servers that can sustain at least moderate communication costs.

PSC incurs communication overhead by transmitting ElGamal encrypted blinds, a zero-knowledge proof (for the knowledge of discrete log of the blinds), and *masked* plaintext counters between the DPs and CPs and the ElGamal encrypted counters among the CPs.

We explore PSC's communication costs by varying the number of bins $b$, the number of CPs $m$, the number of DPs $d$, $\epsilon$, and $\delta$. The average communication cost for the CPs and DPs are plotted in Figure 8 and Figure 9. We omit the error bars as PSC has a deterministic communication cost—there is no variance in the communication cost incurred among the CPs (and similarly, among the DPs).

We first consider how the number of bins influences the communication cost. We run PSC, varying $b$ from 100,000 to 500,000, and plot the results in Figure 8 *(left)*. The values of the bins (*i.e.*, `0` or `1`) do not affect the outbound communication cost of the DPs, as the DPs invariably transmit an encrypted blind, a zero-knowledge proof and a plaintext value for either `0` or `1`. For up to 300,000 bins, the outbound communication cost for each DP is fairly modest. For example, if PSC is run once an hour, then the outbound communication cost is approximately 252 MB/hr (70 KBps). We also find that the inbound communication cost for each DP is constant (~1.8 KB), as the DPs only receive the signed ElGamal joint public key from every CP, irrespective of the number of bins.

The communication costs are more significant for the CPs, which we envision are dedicated machines for PSC. With 300,000 bins, each CP requires a bandwidth of 962.6 MB for sending and 2.5 GB for receiving (less than 700 KBps if executed once per hour). We observe that the difference between the CP outbound and inbound communication cost is large and roughly equal to the sum of the DP outbound communication cost per CP (*i.e.*, $252/5 \approx 50.4$ MB) across all 30 DPs. This is because each CP sends the signed ElGamal joint public key (*i.e.*, a single group element) to every DP, whereas each CP receives $b$ ElGamal encrypted blinds, zero-knowledge proofs, and plaintext counters from every DP.

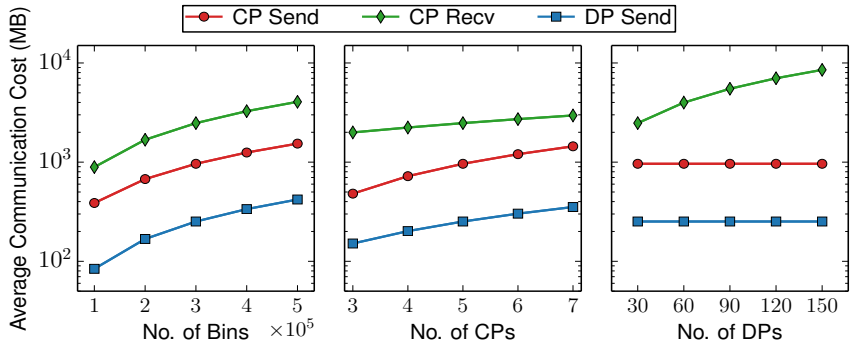


Fig. 8. The communication cost incurred by the CPs and DPs varying the number of bins (*left*), the number of CPs (*center*) and the number of DPs (*right*).

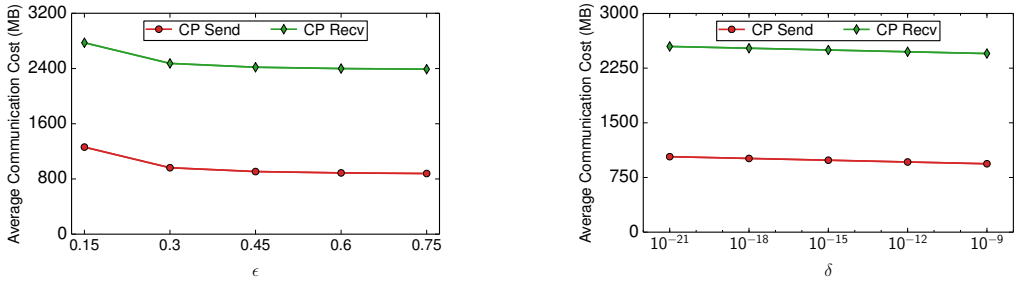


Fig. 9. The communication cost incurred by the CPs varying $\epsilon$ (*left*) and $\delta$ (*right*).

We next consider how $m$, the number of CPs, affects the communication cost. We vary $m$ from 3 to 7 and plot the results in Figure 8 *(center)*. The outbound communication cost for the DPs increases at a much slower pace when varying $m$ (as opposed to $b$), as the DPs invariably send $b$ (= 300, 000) ElGamal encrypted blinds, zero-knowledge proofs and plaintext counters to each CP. Therefore, even up to seven CPs, the outbound communication cost for each DP is fairly modest — approximately 1.4 GB (or 98 KBps, if run every hour). We note that the inbound communication cost per DP increases by a constant factor (~0.93 KB) as the number of CPs increases. This increase is due to each DP receiving a copy of the signed ElGamal joint public key from every CP.

The inbound and outbound communication costs for each CP increase *almost* linearly with the number of CPs. Recall from Section 6.1.1 that each CP participates in $O(m)$ broadcasts, $O(m^2)$ *echoes*, and $O(m^3)$ *heartbeats* in the optimized heartbeat version of the Dolev-Strong protocol. Although the total communication cost of the CPs is $O(m^3)$ (*i.e.* cubic in $m$), the communication cost for the echoes (32 bytes) and heartbeats (~100 bytes) is significantly less than that for the broadcasts (which involve transmitting zero-knowledge proofs and ElGamal ciphertexts for all counters and noise). Therefore, the communication cost per CP roughly increases by a constant factor (~240 MB) as we increase $m$.

We rerun PSC with different numbers of DPs. Figure 8 (*right*) shows that varying the number of DPs has no effect on the average communication cost for the DPs and the average outbound communication cost for the CPs. This is because the DPs invariably send $b$ ElGamal encrypted blinds, zero-knowledge proofs and plaintext counters to each CP and receive a copy of the signed joint public key from each CP. Likewise, the number of ElGamal ciphertexts and proofs transmitted by the CPs also remains the same across these experiments. However, there is a large linear increase

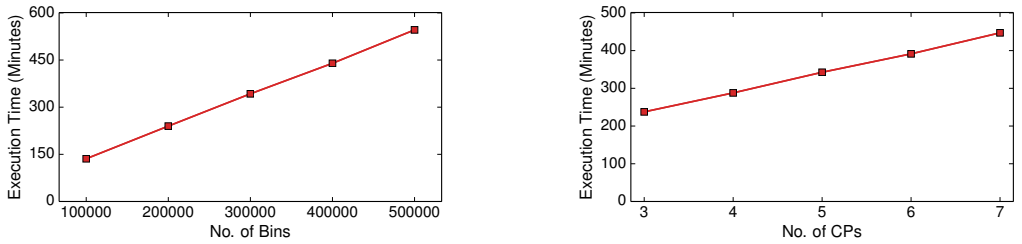


Fig. 10. The overall execution time as a function of the number of bins (*left*) and the number of CPs (*right*).

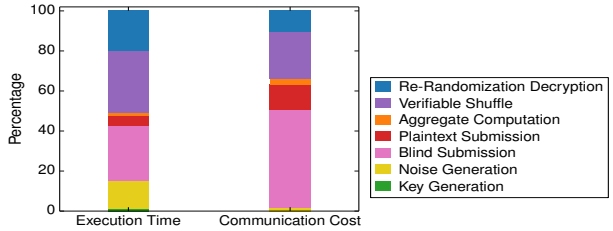


Fig. 11. The overall execution time *(right) and the communication cost (*left) incurred by the CPs for different operations, varying the number of bins.

in the inbound communication cost of the CPs as each CP receives $b$ blinds, zero-knowledge proofs and plaintext counters from each DP. The variation in the inbound communication cost per CP is approximately 1.5 GB.

Next, we run PSC with different values of $\epsilon$ to determine how the choice of privacy parameter affects the communication costs. Figure 9 *(left)* shows that the average communication costs for the CPs decrease when $\epsilon$ is increased. The communication costs for the CPs are reasonable even for a low value of $\epsilon = 0.15$. On average, each CP requires a bandwidth of at most 1.3 GB for sending and 2.8 GB for receiving (less than 800 KBps, if performed once an hour).

Lastly, we consider how the choice of $\delta$ affects the communication cost. Figure 9 *(right)* shows that the average communication costs for the CPs decrease as $\delta$ increases. The communication costs for the CPs are reasonable even for a low value of $\delta = 10^{-21}$. On average, each CP requires at most 1 GB for sending and 2.5 GB for receiving (less than 700 KBps, if performed once an hour).

In summary, we find that PSC incurs reasonable communication overhead: the costs to DPs are moderate, and, while slightly higher for CPs, they remain practical.

**Overall runtime.** We explore the overall running time (including the time required for network communication) of PSC by varying the number of bins $b$ and the number of CPs $m$. The average overall running as a function of $b$ and $m$ is plotted in Figure 10.

We first consider how the number of bins $b$ affects the computation cost (note that more data must be communicated using the optimized heartbeat DS protocol as $b$ increases). The overall runtime is moderate. It takes approximately 9 hr 6 min even for an experiment with 500, 000 bins.

We next consider how the number of CPs $m$ affects the computation cost. The computation cost increases with the CPs at a slower pace than with the bins. Even up to seven CPs, the average computation cost for each CP is fairly modest — approximately 7 hr 27 min.

**Microbenchmarks.** To better understand the computational and communication overhead of PSC, we measure the execution time (including the network latency) and communication cost for

different operations (using the default values for all system parameters). The results are plotted in Figure 11. We observe that the time taken for re-randomization decryption, blind submission, and verifiable shuffles account for 19.8%, 27.8%, and 31% of the total runtime of the protocol respectively. We note that the time taken for blind submission is the total time taken across all $d$ (= 30) DPs (recall that our implementation is currently single-threaded with a single DP program emulating all the DPs). Therefore, as expected the verifiable shuffle followed by the re-randomization decryption are the most time-consuming operations.

We find that the communication cost for re-randomization decryption, plaintext submission, verifiable shuffles, and blind submission account for 10.3%, 12.4%, 23.4%, and 48.7% of the overall communication cost of the protocol. Here again, the communication cost for the blind and plaintext submission is the total cost across all $d$ (= 30) DPs. Therefore, as expected, the verifiable shuffle and the re-randomization decryption are the most communication-intense operations.

## 8 PRIVATE SET-INTERSECTION CARDINALITY

We outline how our protocol can be modified to calculate a private set-intersection rather than a private set-union. To turn the homomorphic operation on ciphertexts from performing "or" to performing "and", we invert the logical bit representation by encoding a logical 0 with any nonzero value and encoding a logical 1 with a zero. However, under this change of representation it would not be possible to record an observation obliviously (i.e. flip a logical 0 to a 1) without further protocol changes, since to record an observation each DP would need to know the original blind it submitted in order to provide that blind's negation. A DP cannot store in plaintext both the blind and the current counter value, or it would reveal its counter value to the adversary upon adaptive corruption. We can solve this problem by encrypting the counter value, but simply submitting the encrypted counter would enable a related-ciphertext attack. Moreover, the same type of proof arguments would not work as the simulator would not be able to extract the adversary's counter values during input submission. We solve these problems by adding a non-interactive Zero Knowledge Proof of Knowledge for each encrypted counter, which enables extraction during the simulation proof. We note, however, that while such ZKPs exist in the UC model, their constructions either require stronger setup assumptions than a CRS [48] or are not structured similarly to the many-verifier interactive $\Sigma$-protocols we use for the protocol for set-union [35]. We give some details of this construction and then discuss resulting changes in the protocol's security and efficiency.

**Construction**. To perform set-intersection cardinality, we modify the protocol (Sec. 5) as follows:

(1) Blinds are submitted as before: for a counter $t$, $\mathrm{DP}_j$ submits $E(b_{tj})$ and the corresponding proof of knowledge of the plaintext. DPs save two values: (1) $-b_{tj}$ as a "zero" value; and (2) the pair $(E_y(r), P(E_y(r)))$ and as a "submission" value, where $E_y(r)$ is an encryption of a random value $r$ and $P(E_y(r))$ is a non-interactive zero-knowledge proof of knowledge of $r$.
(2) To record an observation for counter $t$, $\mathrm{DP}_j$ replaces its submission value with $(E(-b_{tj}), P(E(-b_{tj})))$ and erases from memory information about this operation.
(3) When the collection period is complete, each DP submits for every counter its submission value for that counter. The protocol proceeds identically as before, but in the final result we count the number of plaintext counters that are equal to 0.

**Security**. Upon corrupting a $\mathrm{DP}_j$, an adaptive adversary now learns the original blind values for all counters. This means adaptive adversaries are able not only to set the submission of corrupt DPs to 1, as before, but may also force this value to be 0 as well regardless of previous observations by that adaptively corrupted DP. For set-union, adaptive adversaries could not revert previously recorded observations. We note however, that in both cases when the adversary submits malicious values on behalf of adaptively corrupted DPs, it knows nothing about the previous value for each

counter. Thus the intersection protocol preserves the most important adaptive security property: that no private data is revealed upon an adaptive corruption.

**Efficiency**. With respect to efficiency, we expect the additional computation and communication costs to be substantial but small enough that the protocol remains usable in practice. More precisely, the adjustments require each DP to perform a public-key encryption and non-interactive zero-knowledge proof to record an observation, whereas for set union, they simply randomize a plaintext counter. For communication cost, using the Fiat-Shamir heuristic and assuming group elements and plaintext elements of $\mathbb{Z}_q$ are roughly equal size, the total cost during input submission should increase from one plaintext counter (for each counter held by each DP) to 4 (two each for the ciphertext and proof) with other phases of the protocol unchanged.

## 9 CONCLUSION AND FUTURE WORK

We present the PSC protocols for securely computing the cardinality of set union and set intersection across multiple parties. PSC is secure against an active adversary controlling all but one honest party, provides adaptive security for the Data Parties, makes the adversary accountable for disrupting the protocol, and produces differentially-private outputs. Our implementation of PSC is available for download at https://github.com/GUSecLab/psc. Future work includes developing the sketch-based approach [14] by modifying generic MPC protocols to provide all our security properties, applying efficiency improvements in zero-knowledge proofs [4, 9], and investigating batching broadcasts [2] to improve performance while mainting our termination guarantees.

## ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation under grant CNS-1718498, the Canada Research Chairs program, the Office of Naval Research, and the Callahan Family Professor endowment. The views expressed in this work are those of the authors and do not necessarily reflect those of the funding agencies or organizations.

## REFERENCES

[1] Gilad Asharov and Yehuda Lindell. 2017. A full proof of the BGW protocol for perfectly secure multiparty computation. *Journal of Cryptology* 30, 1 (2017).
[2] Carsten Baum, Emmanuela Orsini, and Peter Scholl. 2016. Efficient secure multiparty computation with identifiable abort. In *Theory of Cryptography Conference (TCC)*.
[3] Carsten Baum, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. 2020. Efficient Constant-Round MPC with Identifiable Abort and Public Verifiability. In *Annual International Cryptology Conference (Crypto)*.
[4] Stephanie Bayer and Jens Groth. 2012. Efficient Zero-knowledge Argument for Correctness of a Shuffle. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*.
[5] Mihir Bellare and Phillip Rogaway. 1993. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security (CCS)*.
[6] Josh Benaloh. 1994. Dense probabilistic encryption. In *Workshop on selected areas of cryptography (SAC)*.
[7] Avrim Blum, Katrina Ligett, and Aaron Roth. 2013. A learning theory approach to noninteractive database privacy. *Journal of the ACM (JACM)* 60, 2 (2013).
[8] Felix Brandt. 2005. Efficient cryptographic protocol design based on distributed El Gamal encryption. In *International Conference on Information Security and Cryptology (ICISC)*.
[9] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy (S&P)*.
[10] Jan Camenisch, Manu Drijvers, and Maria Dubovitskaya. 2017. Practical UC-secure delegatable credentials with attributes and their application to blockchain. In *ACM Conference on Computer and Communications Security (CCS)*.
[11] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *Foundations of Computer Science (FOCS)*.
[12] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. 2002. Universally composable two-party and multi-party secure computation. In *Symposium on Theory of Computing (STOC)*.

[13] Ran Canetti and Tal Rabin. 2002. Universal Composition with Joint State. Cryptology ePrint Archive, Report 2002/047. https://eprint.iacr.org/2002/047.
[14] Seung Geol Choi, Dana Dachman-Soled, Mukul Kulkarni, and Arkady Yerukhimovich. 2020. Differentially-Private Multi-Party Sketching for Large-Scale Statistics. *Proceedings on Privacy Enhancing Technologies* 3 (2020).
[15] Ivan Damgård. 2010. On Σ-protocols. Lecture Notes on Cryptologic Protocol Theory, v.2.
[16] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *Annual International Cryptology Conference (Crypto)*.
[17] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. 2012. Fast and private computation of cardinality of set intersection and union. In *International Conference on Cryptology and Network Security (CANS)*.
[18] Daniël de Kok. 2020. Go par package for parallel for-loops. https://github.com/danieldk/par.
[19] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium (USENIX)*.
[20] Danny Dolev and H. Raymond Strong. 1983. Authenticated Algorithms for Byzantine Agreement. *SIAM J. Comput.* 12, 4 (1983).
[21] Marianne Durand and Philippe Flajolet. 2003. Loglog counting of large cardinalities. In *European Symposium on Algorithms (ESA)*.
[22] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy via Distributed Noise Generation. In *Advances in Cryptology (Eurocrypt)*.
[23] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography Conference (TCC)*.
[24] Cynthia Dwork, Aaron Roth, et al. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science* 9, 3-4 (2014).
[25] Rolf Egert, Marc Fischlin, David Gens, Sven Jacob, Matthias Senker, and Jörn Tillmanns. 2015. Privately Computing Set-Union and Set-Intersection Cardinality via Bloom Filters. In *Australasian Conference on Information Security and Privacy*.
[26] Tariq Elahi, George Danezis, and Ian Goldberg. 2014. PrivEx: Private Collection of Traffic Statistics for Anonymous Communication Networks. In *ACM Conference on Computer and Communications Security (CCS)*.
[27] Ellis Fenske, Akshaya Mani, Aaron Johnson, and Micah Sherr. 2017. Distributed Measurement with Private Set-Union Cardinality. In *ACM Conference on Computer and Communications Security (CCS)*. ACM.
[28] Amos Fiat and Adi Shamir. 1987. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *Advances in Cryptology (CRYPTO '86)*.
[29] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. 2004. Efficient Private Matching and Set Intersection. In *Advances in Cryptology (Eurocrypt)*.
[30] Jun Furukawa, Hiroshi Miyauchi, Kengo Mori, Satoshi Obana, and Kazue Sako. 2003. An Implementation of a Universally Verifiable Electronic Voting Scheme Based on Shuffling. In *Financial Cryptography (FC'02)*.
[31] O. Goldreich. 2001. *Foundations of Cryptography: Volume 2, Basic Applications.* Cambridge University Press.
[32] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to play ANY mental game. In *ACM Symposium on Theory of Computing (STOC)*.
[33] Oded Goldreich and Yair Oren. 1994. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology* 7, 1 (1994).
[34] Jens Groth. 2003. A Verifiable Secret Shuffle of Homomorphic Encryptions. In *Theory and Practice in Public Key Cryptography (PKC)*.
[35] Jens Groth, Rafail Ostrovsky, and Amit Sahai. 2006. Perfect non-interactive zero knowledge for NP. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*.
[36] Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, Tomas Toft, and Angelo Agatino Nicolosi. 2012. Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting. In *Topics in Cryptology – CT-RSA*.
[37] Carmit Hazay and Kobbi Nissim. 2012. Efficient Set Operations in the Presence of Malicious Adversaries. *Journal of Cryptology* 25, 3 (2012).
[38] Ali Inan, Murat Kantarcioglu, Gabriel Ghinita, and Elisa Bertino. 2010. Private record matching using differential privacy. In *International Conference on Extending Database Technology*.
[39] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. 2014. Secure multi-party computation with identifiable abort. In *Annual Cryptology Conference (CRYPTO)*.
[40] Rob Jansen and Aaron Johnson. 2016. Safely Measuring Tor. In *ACM Conference on Computer and Communications Security (CCS)*.
[41] Shiva P Kasiviswanathan and Adam Smith. 2014. On the 'Semantics' of Differential Privacy: A Bayesian Formulation. *Journal of Privacy and Confidentiality* 6, 1 (2014).

[42] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Universally composable synchronous computation. In *Theory of Cryptography Conference (TCC)*.
[43] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. 2016. Fair and Robust Multi-party Computation Using a Global Transaction Ledger. In *Advances in Cryptology (EUROCRYPT)*.
[44] Lea Kissner and Dawn Song. 2005. Privacy-Preserving Set Operations. In *Annual International Cryptology Conference (Crypto)*.
[45] kyber. 2020. kyber: DEDIS Advanced Crypto Library for Go. https://godoc.org/go.dedis.ch/kyber.
[46] Enrique Larraia, Emmanuela Orsini, and Nigel P Smart. 2014. Dishonest Majority Multi-Party Computation for Binary Circuits. In *Annual International Cryptology Conference (Crypto)*.
[47] Yehida Lindell. 2005. Secure multiparty computation for privacy preserving data mining. In *Encyclopedia of Data Warehousing and Mining*. 1005–1009.
[48] Yehuda Lindell. 2015. An Efficient Transform from Sigma Protocols to NIZK with a CRS and Non-programmable Random Oracle. In *Theory of Cryptography (TCC)*.
[49] Yehuda Lindell, Benny Pinkas, Nigel P Smart, and Avishay Yanai. 2015. Efficient constant round multi-party computation combining BMR and SPDZ. In *Annual Cryptology Conference (Crypto)*.
[50] Akshaya Mani and M. Sherr. 2017. Histor$\epsilon$: Differentially Private and Robust Statistics Collection for Tor. In *Network and Distributed System Security Symposium (NDSS)*.
[51] Damon McCoy, Kevin Bauer, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. 2008. Shining Light in Dark Places: Understanding the Tor Network. In *Privacy Enhancing Technologies Symposium (PETS)*.
[52] Frank McSherry and Kunal Talwar. 2007. Mechanism design via differential privacy. In *Foundations of Computer Science (FOCS)*.
[53] Luca Melis, George Danezis, and Emiliano De Cristofaro. 2016. Efficient Private Statistics with Succinct Sketches. In *Network and Distributed System Security Symposium (NDSS)*.
[54] C. Andrew Neff. 2001. A Verifiable Secret Shuffle and its Application to E-voting. In *ACM Conference on Computer and Communications Security (CCS)*.
[55] Lan Nguyen, Rei Safavi-Naini, and Kaoru Kurosawa. 2004. Verifiable Shuffles: A Formal Model and a Paillier-based Efficient Construction with Provable Security. In *Applied Cryptography and Network Security (ACNS)*.
[56] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. 2007. Smooth Sensitivity and Sampling in Private Data Analysis. In *Symposium on Theory of Computing (STOC)*.
[57] Craig Partridge and Mark Allman. 2016. Ethical considerations in network measurement papers. *Commun. ACM* 59, 10 (2016).
[58] Martin Pettai and Peeter Laud. 2015. Combining differential privacy and secure multiparty computation. In *Annual Computer Security Applications Conference (ACSAC)*.
[59] Shai Halevi Ran Canetti, Oded Goldreich. 2004. The Random Oracle Methodology, Revisited. *Journal of the ACM (JACM)* 51, 4 (2004).
[60] Claus-Peter Schnorr. 1991. Efficient Signature Generation by Smart Cards. *Journal of Cryptology* 4, 3 (1991).
[61] Christopher Soghoian. 2011. Enforced Community Standards For Research on Users of the Tor Anonymity Network. In *Workshop on Ethics in Computer Security Research (WECSR)*.
[62] Rade Stanojevic, Mohamed Nabeel, and Ting Yu. 2017. Distributed Cardinality Estimation of Set Operations with Differential Privacy. In *IEEE Symposium on Privacy-Aware Computing (PAC)*.
[63] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*.
[64] Rene Struik. 2021. *Alternative Elliptic Curve Representations*. Internet-Draft draft-ietf-lwig-curve-representations-20. Internet Engineering Task Force. Work in Progress.
[65] Tor Metrics. 2020. https://metrics.torproject.org/.
[66] Jaideep Vaidya and Chris Clifton. 2005. Secure Set Intersection Cardinality with Application to Association Rule Mining. *Journal of Computer Security* 13, 4 (2005).
[67] Kenton Varda. 2008. Protocol Buffers: Google's Data Interchange Format. https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html.
[68] Ryan Wails, Aaron Johnson, Daniel Starin, Arkady Yerukhimovich, and S. Dov Gordon. 2019. Stormy: Statistics in Tor by Measuring Securely. *ACM Conference on Computer and Communications Security (CCS)*.
[69] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-scale secure multiparty computation. In *ACM Conference on Computer and Communications Security (CCS)*.
[70] Douglas Wikström. 2005. A Sender Verifiable Mix-Net and a New Proof of a Shuffle. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*.

## A COMMITMENTS AND COINS

For reference, in the section we repeat the one-to-many commitment functionality given in [12] (Figure 12), present an accountable coin-flipping functionality (Figure 13), and give a protocol that UC-realizes the coin flipping functionality and the corresponding proof (Figure 14).

$\mathcal{F}_{\text{MCOM}}^{1:M}$

(1) **Commit Phase:** Upon receiving a message (COMMIT, $m$) from $\text{CP}_j$ with $m \in \{0,1\}$, record the tuple ($\text{CP}_j$, $m$) and send (RECEIPT, $\text{CP}_j$) to all parties.
(2) **Prove Phase:** Upon receiving a message (REVEAL) from $\text{CP}_j$, if a tuple ($\text{CP}_j$, $m$) was previously recorded, then send the message (REVEAL, $m$) to all parties.

Fig. 12. $\mathcal{F}_{\text{MCOM}}^{1:M}$, the ideal functionality for one-to-many commitments.

$\mathcal{F}_{\text{COIN}}$

(1) **(Round 1)** Wait to receive GENERATECOINS from each CP
(2) **(Round 2)** If received GENERATECOINS from each CP, generate $\vec{b} \in \{0,1\}^n$ at random and broadcast $\vec{b}$ to all CPs. Otherwise, broadcast (BLAME, $L$) where $L$ where $L$ is the set of the CPs from whom messages are missing from Round 1.

Fig. 13. $\mathcal{F}_{\text{COIN}}$, the ideal coin flipping functionality.

$\pi_{\text{COIN}}$

(1) **(Round 1) Commitments.** Each $\text{CP}_j$ selects $b_j \in \{0,1\}$ at random and commits to $b_j$ to every other CP using $\mathcal{F}_{\text{MCOM}}^{1:M}$.
(2) If a CP has received a commitment receipt from all other CPs, each addressed to the set of all CPs, proceed to Round 2. Otherwise, broadcast (BLAME, $L$) where $L$ is the set of CPs for which a correct receipt was not received and then exit.
(3) **(Round 2) Opening.** Each $\text{CP}_j$ opens $b_j$ to all CPs using $\mathcal{F}_{\text{MCOM}}^{1:M}$. If a CP receives a valid opening from each CP, output $b = \bigoplus_j b_j$. Otherwise, output (BLAME, $L$) where $L$ is the set of CPs who have failed to open their commitments to a single bit.

Fig. 14. $\pi_{\text{COIN}}$, an accountable coin-flipping protocol

THEOREM A.1. *The protocol $\pi_{COIN}$ UC-realizes the $\mathcal{F}_{COIN}$ functionality in the static-corruption, synchronous, ($\mathcal{F}_{BC}$, $\mathcal{F}_{MCOM}^{1:M}$)-hybrid model if there is at least one honest CP.*

PROOF. We define a simulator $S$ as follows: $S$ runs a copy of the adversary $A$, and simulates the honest parties and ideal functionalities honestly. If a malicious party commits during the commit phase and opens it to an element of $\{0,1\}$ in the opening phase, S sends GENERATECOIN to $\mathcal{F}_{\text{COIN}}$

on behalf of that malicious party. We observe that every dishonest party $P$ either completes a valid opening in the second phase of the protocol, in which case $\mathcal{F}_{\text{COIN}}$ has received GenerateCoin from $P$ (through $S$), or $P$ has failed to produce a valid opening, in which it is blamed by all honest (simulated) CPs. Finally, in the case that all parties open commitments and $\mathcal{F}_{\text{COIN}}$ outputs a uniformly random bit, this bit is identically distributed to $\oplus_j b_j$ in the simulation, since $b_j$ is random for honest $j$, and dishonest CPs must commit to their bits without any information about the honest parties' selections. □

We will need a vector of $n$ coins, so we write $\mathcal{F}_{\text{N-COIN}}$ as $n$ independent copies of $\mathcal{F}_{\text{COIN}}$, which can be joined into a single protocol using the JUC theorem from [13]. Similarly, we will require commitments to strings of bits, so we write $\mathcal{F}_{\text{S-MCOM}}^{1:\text{M}}$ to indicate the protocol where $\mathcal{F}_{\text{MCOM}}^{1:\text{M}}$ is invoked multiple times to produce a commitment to a bit string rather than a single bit.