Scaling Private Set Intersection to Billion-Element Sets

Seny Kamara
Microsoft Research

Payman Mohassel
University of Calgary

Mariana Raykova
SRI

Saeed Sadeghian
University of Calgary

ABSTRACT

We examine the feasibility of private set intersection (PSI) over massive datasets. PSI, which allows two parties to find the intersection of their sets without revealing them to each other, has numerous applications including to privacy-preserving data mining, location-based services and genomic computations. Unfortunately, the most efficient constructions only scale to sets containing a few thousand elements—even in the semi-honest model and over a LAN.

In this work, we design PSI protocols in the server-aided setting, where the parties have access to a single *untrusted* server that makes its computational resources available as a service. We show that by exploiting the server-aided model and by carefully optimizing and parallelizing our implementations, PSI is feasible for *billion*-element sets even while *communicating over the Internet*. As far as we know, ours is the first attempt to scale PSI to billion-element sets which represents an increase of five orders of magnitude over previous work.

Our protocols are secure in several adversarial models including against a semi-honest, covert and malicious server; and address a range of security and privacy concerns including fairness and the leakage of the intersection size. Our protocols also yield efficient server-aided private equality-testing (PET) with stronger security guarantees than prior work.

1 Introduction

In the problem of private set intersection (PSI), two parties want to learn the intersection of their sets without revealing to each other any information about their sets beyond the intersection. PSI is a fundamental problem in security and privacy that comes up in many different contexts. Consider, for example, the case of two or more institutions that wish to obtain a list of common customers for data-mining purposes; or a government agency that wants to learn whether anyone on its no-fly list is on a flight's passenger list. PSI has found applications in a wide range of settings such as genomic computation [4], location-based services [55], and collaborative botnet detection [54].

Secure multi-party computation. PSI is a special case of the more general problem of secure multi-party computation (MPC). In this problem, each party holds its own private input and the goal is to collectively compute a joint function of the participants' inputs without leaking additional information and while guaranteeing correctness of the output. The design and implementation of practical MPC protocols has been an active area of research over past decade with numerous efforts to improve and optimize software implementations and to develop new frameworks such as Fairplay [53, 6], VIFF [19], Sharemind [7], Tasty [40], HEKM [41], VMCrypt [52], and SCAPI [25]. While these general-purpose solutions can be used to solve the PSI problem, they usually do not provide efficient solutions. A large body of work, therefore, has focused on the design and implementation of *efficient* special-purpose PSI protocols [30, 38, 10, 17, 22, 39, 43, 42].

Limitations of MPC. While progress on efficient PSI (and MPC in general) has been impressive, existing protocols are still far from optimal for many real-world scenarios. As the trend towards "Big Data" continues, Governments and private organizations often manage massive databases that store billions of records. Therefore, for any PSI solution to be of practical interest in such settings, it needs to efficiently process sets with tens or hundreds of millions of records. Unfortunately, existing general- and special-purpose PSI solutions (especially with malicious security) are *orders of magnitude* less efficient than computing intersections on plaintext sets and hence *do not scale to massive datasets*.

Another limitation of standard approaches to PSI is that achieving fairness is not always possible. Roughly speaking, fairness ensures that either all the parties learn the output of the computation or none will. This is crucial in many real-world applications such as auctions, electronic voting, or collective financial analysis, where a dishonest participant should not be able to disrupt the protocol if it is not satisfied with the outcome of the computation. In 1986, Cleve showed that complete fairness is impossible in general, unless the majority of the players are honest [15]. A number of constructions try to achieve fairness for a specific class of functionalities [36], or consider limited (partial) notions of fairness instead [56, 31, 35].

server-aided MPC. A promising approach to address these limitations is *server-aided* or *cloud-assisted* MPC.[1] In this

[1] An alternative approach considered in the PSI literature is the use of tamper-proof hardware in the design of private set intersection [37, 29]. This approach allows for better efficiency and hence more scalable protocols. Token-based PSI makes different and incomparable trust assumptions compared to server-aided MPC, and does not seem suitable for settings that involve a cloud service.

variant of MPC, the standard setting is augmented with a small set of servers that have no inputs to the computation and that receive no output but that make their computational resources available to the parties. In this paradigm, the goal is to tradeoff the parties' work at the expense of the servers'. Server-aided MPC with two or more servers has been considered in the past [20, 21] and even deployed in practice [8], but since we focus on instantiating the server using a cloud service we are mostly interested in the *single-server* scenario.

A variety of single-server-aided protocols have been considered in the past. This includes general-purpose solutions such as [2], which combines fully-homomorphic encryption [33] with a proof system [5]; and the constructions based on Yao's garbled circuit technique [61, 62], proposed by Feige, Killian and Naor [28] in the semi-honest model and recently formalized and extended to stronger models in [44] and optimized and implemented in [47]. This also includes special-purpose protocols such as server-aided private equality-testing [55, 32].

Non-collusion. With the exception of [2], which uses heavy machinery and is only of theoretical interest at this stage, all other single-sever-aided protocols we know of are secure in a setting where the server *does not collude with the parties*. There are many settings in practice where collusion does not occur, e.g., due to physical restrictions, legal constraints and/or economic incentives. In a server-aided setting where the server is a large cloud provider (e.g., Amazon, Google or Microsoft), it is reasonable—given the consequences of legal action and bad publicity—to assume that the server will not collude with the parties.

The work of [44] attempts to formally define non-collusion in the context of MPC. For the purpose of our work, however, we use a simplified notion of non-collusion wherein two parties A and B are considered to not collude if they are not simultaneously corrupted by the adversary (e.g., either A is malicious or B is, but not both). This allows us to use the standard ideal/real-world simulation-based definitions of security for MPC and simply restrict the parties that the adversary can corrupt. In particular, we consider the adversary structures that respect the non-collusion relations described above (which we refer to as admissible subsets). So, for example, with two parties and a single server that does not collude with them we need to consider adversary structures that only contain a single malicious party. On the other hand, in a setting with multiple parties and a single server, either an arbitrary subset of the parties are corrupted or the server is. This simplified notion appears to capture the security of all existing server-aided constructions we are aware of.

1.1 Our Contributions

Motivated by the problem of PSI for massive datasets, we design and implement several new PSI protocols in the server-aided setting. Our protocols are provably secure in several adversarial models including against a semi-honest, covert and malicious server; and address a range of security and privacy concerns including fairness and intersection size-hiding. Our protocols also yield efficient server-aided private equality-testing (PET) with stronger security guarantees than prior work.

Efficiency and Comparison. All our protocols require only a linear number of block-cipher invocations (a pseudorandom permutation) in the set sizes for the parties with inputs; and the execution of a standard/plaintext set intersection algorithm for the server. This is a major improvement over all previous general- and special-purpose PSI constructions.

We then show that by making use of various optimizations, efficient data structures and by carefully parallelizing our implementations, PSI is feasible for *billion*-element sets even while communicating over the Internet. This is five orders of magnitude larger than what the best standard PSI protocols can feasibly achieve over a LAN (see the experiments in Sections 5.2 and 5.3).

Our protocols are competitive compared to non-private set intersection as well. For example, our semi-honest protocol is only 10% slower than the non-private variant (note that we use the same optimizations in both ours and the non-private protocol). This shows that achieving privacy can indeed be affordable when using the right infrastructure and optimizations (see the experiments in Section 5.4).

We also show that our constructions can easily implemented on top of existing frameworks for fast set operations. In particular, we show how to use a NoSQL database implementation, Redis (in use by various cloud-based services), in a black-box way to implement our server-aided PSIs (see experiments in Section 5.5.

Optimizations for large sets. In order to make the memory, bandwidth, and CPU usage of our implementations scalable to very large sets (up to billion-elements) and for communication over the internet, we carefully optimize every aspect of our implementation. For example, we use fast and memory-efficient data structures from the Sparsehash library [26] to implement our server-side set intersection protocol. In order to take advantage of the parallelizability of our protocols, we also use multi-threading both on the client- and the server-side, simultaneously processing, sending, receiving, and looking-up elements in multiple threads. The use of parallelization particularly improves the communication time, which dominates the total running time of our protocols. Our experiments (see Section 5.1) show that we gain up to a factor of 3 improvement in total running time in this fashion. Other important considerations include the choice of cryptographic primitives, and the truncation of ciphertexts before send and receive operations, while avoiding potential erroneous collisions.

1.2 Related Work

As far as we know, the recent work of Dong, Chen, Camenisch and Russello [24] is the only other work that proposes a (fair) server-aided PSI protocol. Their protocol, however, assumes a semi-honest server and requires public-key operations (i.e., exponentiations) which prevents their protocol from scaling to the sizes we consider in this work.

We also note that server-aided PSI protocols can be constructed from searchable symmetric encryption schemes (SSE) and, in particular, from index-based SSE schemes [59, 34, 13, 16, 14, 46, 45, 12]. We provide a detailed comparison between these notions in Appendix A and only note here that SSE schemes provide a richer functionality than needed for PSI so the design of non-SSE-based server-aided PSI protocols is well motivated.

Finally, private equality testing [27, 9, 1, 51] is a well-known and important functionality that has found numerous applications in the past, typically as a sub-protocol. Indeed, PET has recently found application in privacy-preserving proximity testing [55, 32, 58] and, in particular, the work of [55] uses a server-aided PET (in a model similar to ours) as the main cryptographic component of their construction. While previous work [55, 32, 58] suggests several sever-aided PET protocols, all these constructions assume a *semi-honest server.* By setting the set size of our intersection size-hiding protocol to 1 (note that we need to hide the intersection size to hide the output of PET), we get a an alternative instantiation of server-aided PET that is secure against a malicious server while still only using lightweight symmetric-key operations.

2 Security Definition

Our definition of security for private set intersection is in the standard ideal/real-world paradigm and follows the definitions typically used for secure MPC [11]. The main difference with standard definitions is that, in our setting, we do not allow the adversary to simultaneously corrupt the server and a (non-server) party. As discussed above, this captures a (weak) form of non-collusion between the server and the parties. In the definition that follows, the corrupted parties could be malicious, covert or semi-honest. We also consider the variants that achieve fairness and those that do/don't leak the size of intersection to the server.

Real-model execution. The real-model execution of protocol Π takes place between parties $(P_1, \ldots, P_n)$, server P_{n+1} and an adversary $\mathcal{A}$ that is allowed to corrupt an *admissible* subset of the parties. and admissible subset of parties can either be $\{P_{n+1}\}$ or any subset of $\{P_1, \cdots, P_n\}$.

At the beginning of the execution, each party $(P_1, \ldots, P_n)$ receives its input set $\mathbf{S}_i \subseteq \mathcal{U}$, random coins r_i, and an auxiliary input z_i while the server P_{n+1} receives only a set of random coins r_{n+1} and an auxiliary input z_{n+1}. The adversary $\mathcal{A}$ receives an admissible set $I \subset [n+1]$ that indicates which parties it corrupts.

For all honest parties P_i, let $\textsc{out}_i$ denote its output and for the corrupted party P_i, let $\textsc{out}_i$ denote its view during the execution of Π. The output of the real-model execution of Π between parties $(P_1, \ldots, P_{n+1})$ in the presence of an adversary $\mathcal{A}$ is defined as:

$$\textsc{real}(k, \overline{\mathbf{S}}, \overline{z}; \overline{r}) \stackrel{def}{=} \left\{\textsc{out}_1, \ldots, \textsc{out}_{n+1}\right\},$$

where $\overline{\mathbf{S}} = (\mathbf{S}_1, \ldots, \mathbf{S}_n)$, $\overline{z} = (z_1, \ldots, z_{n+1})$ and $\overline{r} = (r_1, \ldots, r_{n+1})$.

Ideal-model execution. The ideal-model execution of protocol Π takes place between parties $(P_1, \ldots, P_n)$, server P_{n+1} and a simulator $\textsc{Sim}$ that is allowed to corrupt at most one party at a time.

As in the real-model execution, the ideal execution begins with each party $(P_1, \ldots, P_n)$ receiving its input set $\mathbf{S}_i \in \mathcal{U}$, its coins r_i and an auxiliary input z_i, while the server P_{n+1} receives only its coins r_{n+1} and an auxiliary input z_{n+1}. The simulator receives an admissible set $I \subset [n+1]$ that indicates which parties it corrupts. Each party $(P_1, \ldots, P_n)$ sends $\mathbf{S}'_i$ to the trusted party, where $\mathbf{S}'_i = \mathbf{S}_i$ if P_i is semi-honest and $\mathbf{S}'_i$ is an arbitrary set if P_i is malicious or covert. If the execution is *intersection-size hiding* (with respect to the intersection), the server receives $|\mathbf{S}'_1|$ through $|\mathbf{S}'_n|$ from the trusted party. If the execution is not size-hiding, the server receives, in addition, $|\bigcap_{j=1}^{n} \mathbf{S}'_j|$. The trusted party finally returns the intersection $\bigcap_{j=1}^{n} \mathbf{S}'_j$ to each party P_i.

If the execution is not *fair*, the parties and the server can send an abort message to the trusted party at any point throughout the execution and the trusted party will return $\perp$ to all parties. If the execution is fair, they are only allowed to send abort messages *before they receive their outputs.*

Security against *covert* [3] adversaries guarantees that a cheating adversary that diverts from the specified protocol is caught (deterred to cheat) by the honest parties with some probability, which is called *deterrence factor.* If the deterrence factor is one, then any cheating adversary is always detected. In order to handle covert adversaries as opposed to malicious, we need to slightly modify the above ideal execution. In particular, to achieve a deterrence factor of $1/t$, the trusted party will flip a bias coin that is head with probability $1/t$ and is tail otherwise. If the coin turns up head, he will reveal the honest parties inputs to the adversary and allows him to decide the honest parties' outputs. Else, the ideal execution proceeds as it did above. We refer the reader to [3] for more details.

For all honest parties P_i, let $\textsc{out}_i$ denote the output returned to P_i by the trusted party, and for all corrupted parties let $\textsc{out}_i$ be some value output by P_i. The output of an ideal-model execution between parties $(P_1, \ldots, P_{n+1})$ in the presence of a simulator $\textsc{Sim}$ is defined as

$$\textsc{ideal}(k, \overline{\mathbf{S}}, \overline{z}; \overline{r}) \stackrel{def}{=} \left\{\textsc{out}_1, \ldots, \textsc{out}_{n+1}\right\}$$

where $\overline{\mathbf{S}} = (\mathbf{S}_1, \ldots, \mathbf{S}_n)$, $\overline{z} = (z_1, \ldots, z_{n+1})$ and $\overline{r} = (r_1, \ldots, r_{n+1})$.

We now present our formal definition of security which, intuitively, guarantees that executing a protocol Π in the real model is equivalent to executing Π in an ideal model with a trusted party.

Definition 2.1 (Security). *An n-party private set intersection protocol Π is secure if for all* ppt *adversaries $\mathcal{A}$ corrupting an admissible subset of the parties, there exists a* ppt *simulator* $\textsc{Sim}$ *such that for all $\overline{\mathbf{S}} \in [2^{\mathcal{U}}]^n$, for all $\overline{z}$, and for all $i \in [n+1]$,*

$$\left\{\textsc{real}(k, \overline{\mathbf{S}}, \overline{z}; \overline{r})\right\}_{k\in\mathbb{N}} \stackrel{c}{\approx} \left\{\textsc{ideal}(k, \overline{\mathbf{S}}, \overline{z}; \overline{r})\right\}_{k\in\mathbb{N}}$$

where $\overline{r}$ is chosen uniformly at random.

3 Our Protocols

In this Section, we describe our protocols for server-aided PSI. Our first protocol is a multi-party protocol that is only secure in the presence of a semi-honest server (but any collusion of malicious parties). Our second protocol is a two-party protocol and is secure against a covert or a malicious server depending on the parameters used, and also secure when one of the parties is malicious. Our third protocol shows how one can augment the two-party protocol to achieve fairness while our fourth protocol, shows how to hide the size of the intersection[2] from the server as well. Our intersection-size hiding protocol also yields the first server-aided PET with security against a malicious server.

In all our protocols, k denotes the computational security parameter (i.e., the key length for the pseudorandom permutation (PRP)) while s denotes a statistical security parameter. For $\lambda \geq 1$, we define the set $\mathbf{S}^\lambda$ as

$$\mathbf{S}^\lambda = \big\{x\|1, \ldots, x\|\lambda : x \in \mathbf{S}\big\}$$

and $(\mathbf{S}^\lambda)^{-\lambda} = \mathbf{S}$. If $F : \mathcal{U} \to \mathcal{V}$ is a function, the $\mathbf{S}$-evaluation of F is the set $F(\mathbf{S}) = \big\{F(s) : s \in \mathbf{S}\big\}$. We also denote by F^{-1} the inverse of F where $F^{-1}(F(\mathbf{S})) = \mathbf{S}$. If $\pi : [|\mathbf{S}|] \to [|\mathbf{S}|]$ is a permutation, then the set $\pi(\mathbf{S})$ is the set that results from permuting the elements of $\mathbf{S}$ according to π (assuming a natural ordering of the elements). In other words:

$$\pi(\mathbf{S}) = \big\{x_{\pi(i)} : x_i \in \mathbf{S}\big\}.$$

We denote the union and set difference of two sets $\mathbf{S}_1$ and $\mathbf{S}_2$ as $\mathbf{S}_1 + \mathbf{S}_2$ and $\mathbf{S}_1 - \mathbf{S}_2$, respectively.

3.1 Server-aided PSI with Semi-honest Server

We first describe our server-aided protocol for a semi-honest server or any collusion of malicious parties. The protocol is described in Fig. 1 and works as follows. Let $\mathbf{S}_i$ be the set of party P_i. The parties start by jointly generating a secret k-bit key K for a pseudorandom permutation (PRP) F. Each party randomly permutes the set $F_K(\mathbf{S}_i)$ which consists of *labels* computed by evaluating the PRP over the elements of his appropriate set, and sends the permuted set to the server. The server then simply computes and returns the intersection of the labels $F_K(\mathbf{S}_1)$ through $F_K(\mathbf{S}_n)$.

Intuitively, the security of the protocol follows from the fact that the parties never receive any messages from each other, and their only possible malicious behavior is to change their own PRP labels which simply translates to changing their input set. The semi-hoenest server only receives labels which due to the pseudo-randomness of the PRP reveal no information about the set elements. We formalize this intuition in the Theorem 3.1 whose proof is omitted due to lack of space.

Theorem 3.1. *The protocol described in Fig. 1 is secure in the presence (1) a semi-honest server and honest parties or (2) a honest server and any collusion of malicious parties.*

Setup and inputs: Let $F : \{0,1\}^k \times \mathcal{U} \to \{0,1\}^{\geq k}$ be a PRP. Each party P_i has a set $\mathbf{S}_i \subseteq \mathcal{U}$ as input while the server has no input:

1. P_1 samples a random k-bit key K and sends it to P_i for $i \in [2, n]$;
2. each party P_i for $i \in [n]$ sends $\mathbf{T}_i = \pi_i(F_K(\mathbf{S}_i))$ to the server, where π_i is a random permutation;
3. the server computes $\mathrm{I} = \bigcap_{i=1}^n \mathbf{T}_i$ and returns it to all the parties;
4. each party P_i outputs $F_K^{-1}(I)$.

Figure 1: A PSI protocol with a semi-honest server

Efficiency. Each P_i invokes the PRP a total of $|\mathbf{S}_i|$ times, while the server only performs a "plaintext" set intersection and no cryptographic operations. Once can use any of the existing algorithms for set intersection. We use the folklore hash table insertion/lookup which runs in nearly linear time in parties sets.

Also note that the protocol can be executed asynchronously where each party connects at a different time to submit his message to the sever and later to obtain the output.

3.2 Server-aided PSI with Malicious Security

The previous protocol is only secure against a semi-honest server because the server can return an arbitrary result as the intersection without the parties being able to detect this. To overcome this we proceed as follows: we require each party P_i to augment its set $\mathbf{S}_i$ with λ copies of each element. In other words, they create a new set $\mathbf{S}_i^\lambda$ that consists of elements $\big\{x\|1, \ldots, x\|\lambda\big\}$ for all $x \in \mathbf{S}_i$. The parties then generate a random k-bit key for a PRP F using a coin tossing protocol and evaluate the PRP on their augmented sets. This results in sets of labels $F_K(\mathbf{S}_i^\lambda)$. Finally, they permute labels with a random permutation π_i to obtain $\mathbf{T}_i = \pi_i\big(F_K(\mathbf{S}^\lambda)\big)$ which they send to the server. The server computes the intersection I of $\mathbf{T}_1 = \pi_1(F_K(\mathbf{S}_1^\lambda))$ and $\mathbf{T}_2 = \pi_2(F_K(\mathbf{S}_2^\lambda))$ and returns the result to the parties. Each party then checks that $F_K^{-1}(\mathrm{I})$ contains all λ copies of every element and aborts if this is not the case.

Intuitively, this check allows the parties to detect if the server omitted any element in the intersection since, in order to cheat, the server has to guess which elements in I correspond to the λ copies of the element it wishes to omit. But this still does not prevent the server from cheating in two specific ways: (1) the server can return an empty intersection; or (2) it can claim to each party that all the elements from the party's input set are in the intersection.

We address these cases by guaranteeing that the set intersection is never empty and never contains all elements of an input set. To do this, the parties agree on three dummy sets $\mathbf{D}_0$, $\mathbf{D}_1$ and $\mathbf{D}_2$ of strings outside the range of possible input values $\mathcal{U}$ such that $|\mathbf{D}_0| = |\mathbf{D}_1| = |\mathbf{D}_2| = t$. The first party then adds the set $\Delta_1 = \mathbf{D}_0 + \mathbf{D}_1$ to $\mathbf{S}_1^\lambda$ and the second party adds the set $\Delta_2 = \mathbf{D}_0 + \mathbf{D}_2$ to the set $\mathbf{S}_2^\lambda$. We denote the resulting sets $\mathbf{S}_1^\lambda + \Delta_1$ and $\mathbf{S}_2^\lambda + \Delta_2$, respectively. Now, the intersection I of $(\mathbf{S}_1^\lambda + \Delta_1) \cap (\mathbf{S}_2^\lambda + \Delta_2)$ cannot be empty

[2] We note that, this is different from what is know in the literature as size-hiding PSI where the goal is the hide the size of input sets. Here, we only intend to hide the size of the intersection from the server who does not have any inputs or outputs.

since $\mathbf{D}_0$ will always be in it and it cannot consist entirely of one of the sets $\mathbf{S}_1^\lambda + \Delta_1$ or $\mathbf{S}_2^\lambda + \Delta_2$ since neither of them are contained in the intersection. We note that the three dummy sets $\mathbf{D}_0$, $\mathbf{D}_1$ and $\mathbf{D}_2$ need to be generated only once and can be reused in multiple executions of the set intersection protocol. The parties can generate the dummy values using a pseudorandom number generator together with a short shared random seed for the PRG, which they can obtain running a coin-tossing protocol. We can easily obtain dummy values inside and outside the range $\mathcal{U}$ by adding a bit to the output of the PRG, where this bit is set to zero for values inside the range and to one for values outside the range.

It turns out that adding the dummy sets provides an additional benefit. In particular, in order to cheat, by say removing or adding elements, the server not only needs to ensure λ copies remain consistent, but also has to make sure that it does not remove or add elements from the corresponding dummy sets. In other words, we now have two parameters t and λ and as stated in Theorem 3.2, the probability of undetected cheating is $1/t^{\lambda-1} + \mathsf{negl}(k)$ where k is the computational security parameter used for the PRP. Therefore, by choosing the right values of t and λ one can significantly increase security against a malicious server.

Fig. 2 presents the details of our protocol and its security is formalized in Theorems 3.2 and 3.3 below whose proof is omitted due to lack of space. This two theorem consider all possible admissible subsets of the participants that can be corrupted by the adversary.

Coin-toss. The coin tossing protocol is abstracted as a coin tossing functionality $\mathcal{F}_{\mathsf{CT}}$ which takes as input a security parameter k and returns a k-bit string chosen uniformly at random. This functionality can be achieved by simply running a simulatable coin tossing protocol [50, 48]. Such a protocol emulates the usual coin-flipping functionality in the presence of arbitrary malicious adversaries and allows a simulator who controls a single player to control the outcome of the coin flip. We note that the coin-tossing step is independent of the parties' input sets and can be performed offline (e.g., for multiple instantiations of the protocol at once). After this step, the two parties interact directly with the untrusted server until they retrieve their final result. As a result, it has negligible effect on efficiency of our constructions and is omitted from those discussions.

Our set intersection protocol in Fig. 2 provides security in the case of one malicious party, which can be any of the parties. We state formally our security guarantees in the next two theorems.

Theorem 3.2. *If F is pseudo-random, and $(1/t)^{\lambda-1}$ is negligible in the statistical security parameter s, the protocol described in Fig. 2 is secure in the presence of a malicious server and honest P_1 and P_2.*

Theorem 3.3. *The protocol described in Fig. 2 is secure in (1) the presence of malicious P_1 and an honest server and P_2; and (2) a malicious P_2 and honest server and P_1.*

Covert Security. By setting the two parameters t and λ properly, one can aim for larger probabilities of undetected cheating and hence achieve covert security (vs. malicious security) in exchange for better efficiency. For example, for deterrence factor of 1/2, one can let $t = 2$ and $\lambda = 2$.

Setup and inputs: Let $F : \{0,1\}^k \times \mathcal{U} \to \{0,1\}^{\geq k}$ be a PRP and $t, \lambda \geq 1$. P_1 and P_2 have sets $\mathbf{S}_1 \subseteq \mathcal{U}$ and $\mathbf{S}_2 \subseteq \mathcal{U}$ as input, respectively, while the server has no input:

1. P_1 chooses sets $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \subseteq \mathcal{D} \neq \mathcal{U}$ such that $|\mathbf{D}_0| = |\mathbf{D}_1| = |\mathbf{D}_2| = t$ and sends them to P_2;
2. P_2 checks that $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2$ were constructed correctly and aborts otherwise;
3. P_1 and P_2 use $\mathcal{F}_{\mathsf{CT}}$ to agree on a random k-bit key K;
4. each party P_i for $i \in \{1,2\}$ sends the set
$$\mathbf{T}_i = \pi_i\Bigg(F_K\Bigg(\mathbf{S}_i^\lambda + \Delta_i\Bigg)\Bigg)$$
to the server, where π_i is a random permutation and $\Delta_i = \mathbf{D}_0 + \mathbf{D}_i$;
5. the server returns the intersection $\mathrm{I} = \mathbf{T}_1 \cap \mathbf{T}_2$;
6. each party P_i aborts if:
   (a) either $\mathbf{D}_0 \not\subseteq F_K^{-1}(\mathrm{I})$ or $\mathbf{D}_i \cap F_K^{-1}(\mathrm{I}) \neq \emptyset$
   (b) there exists $x \in \mathbf{S}_i$ and $\alpha, \beta \in [\lambda]$ such that $x\|\alpha \in F_K^{-1}(\mathrm{I})$ and $x\|\beta \notin F_K^{-1}(\mathrm{I})$;
7. each party computes and outputs the set
$$\Bigg(F_K^{-1}(\mathrm{I}) - \mathbf{D}_0\Bigg)^{-\lambda}.$$

Figure 2: A Server-aided PSI protocol with malicious security

Efficiency. Each party P_i invokes the PRP $\lambda|\mathbf{S}_i| + 2t$ times while the server performs a "plaintext" set intersection on two sets of size $|\mathbf{S}_1| + 2t$ and $|\mathbf{S}_2| + 2t$, with no cryptographic operations.

Once again, the protocol can be run asynchronously with each party connecting at a different time to submit his message to the server and later to obtain his output.

3.3 Fair Server-aided PSI

While the protocol in Fig. 2 is secure against malicious parties, it does not achieve fairness. For example, a malicious P_1 can submit an incorrectly structured input that could cause P_2 to abort after receiving an invalid intersection while P_1 learns the real intersection. To detect this kind of misbehavior (for either party) and achieve fairness, we augment the protocol as follows.

Suppose we did not need to hide the input sets from the server but still wanted to achieve fairness. In such a case, we could modify the protocol from Fig. 2 as follows. After computing the intersection $\mathrm{I} = \mathbf{T}_1 \cap \mathbf{T}_2$, the server would commit to I (properly padded so as to hide its size) and ask that P_1 and P_2 reveal their sets $\mathbf{S}_1$ and $\mathbf{S}_2$ as well as their shared key K. The server would then check the correctness of $\mathbf{T}_1$ and $\mathbf{T}_2$ and notify the parties in case it detected any cheating (without being able to change the intersection since it is committed). This modification achieves fairness since, in the presence of a malicious P_1, P_2 will abort before the server opens the commitment. In order to hide the sets $\mathbf{S}_1$ and $\mathbf{S}_2$ from the server, it will be enough to apply an additional layer of the PRP. The first layer will account for the privacy guarantee while the second layer will enable the detection of misbehavior.

The protocol is described in detail in Fig. 3 and the next two theorems describe the adversarial settings in which it guarantees security.

Setup and inputs: Let $F : \{0,1\}^k \times \mathcal{U} \rightarrow \{0,1\}^{\geq k}$ be a PRP and $t, \lambda \geq 1$. P_1 and P_2 have sets $\mathbf{S}_1 \subseteq \mathcal{U}$ and $\mathbf{S}_2 \subseteq \mathcal{U}$ as input, respectively, while the server has no input:

1. P_1 chooses sets $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \subseteq \mathcal{D} \neq \mathcal{U} \neq \mathsf{Range}(F)$ such that $|\mathbf{D}_0| = |\mathbf{D}_1| = |\mathbf{D}_2| = t$ and sends them to P_2;
2. P_2 checks that $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2$ were constructed correctly and aborts otherwise;
3. P_1 and P_2 use $\mathcal{F}_{\mathsf{CT}}$ to agree on random k-bit keys K_1 and K_2;
4. each party P_i for $i \in \{1,2\}$ sends to the server the set:
$$\mathbf{T}_i = \pi_i\bigg(F_{K_2}\big(F_{K_1}(\mathbf{S}_i)^{\lambda} + \Delta_i\big)\bigg)$$
where π_i is a random permutation.
5. the server computes the intersection $\mathrm{I} = \mathbf{T}_1 \cap \mathbf{T}_2$ and adds enough padding elements to I until its size is equal to $|\mathbf{S}_1| + t$. We denote this new set by I'.
6. the server then sends a commitment $\mathsf{com}(\mathrm{I}')$ to P_1 and P_2
7. P_1 and P_2 reveal the sets $F_{K_1}(\mathbf{S}_1)$, $F_{K_1}(\mathbf{S}_2)$, $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2$ to the server.
8. the server verifies that each $\mathbf{T}_i$ is consistent with the appropriate opened sets. If not it aborts.
9. the server opens $\mathsf{com}(\mathrm{I}')$ and as a result the parties learn I' from which they remove the padding elements to obtain I.
10. each party P_i aborts if:
    (a) either $\mathbf{D}_0 \not\subset F_{K_2}^{-1}(\mathrm{I})$ or $\mathbf{D}_i \cap F_{K_2}^{-1}(\mathrm{I}) \neq \emptyset$
    (b) there exists $x \in \mathbf{S}_i$ and $\alpha, \beta \in [\lambda]$ such that $F_{k_1}(x) \| \alpha \in F_{K_2}^{-1}(\mathrm{I})$ and $F_{k_1}(x) \| \beta \notin F_{K_2}^{-1}(\mathrm{I})$
11. each party computes and outputs the set
$$F_{K_1}^{-1}\bigg(\big(F_{K_2}^{-1}(\mathrm{I}) - \mathbf{D}_0\big)\bigg)^{-\lambda}.$$

Figure 3: A fair server-aided PSI protocol

Theorem 3.4. *If F is pseudo-random, and $(1/t)^{\lambda-1}$ is negligible in the security parameter s, the protocol described in Fig. 3 is secure in the presence of a malicious server and honest P_1 and P_2.*

Theorem 3.5. *The protocol described in Fig. 3 is secure in (1) the presence of malicious P_1 and an honest server and P_2; and (2) a malicious P_2 and honest server and P_1, and also achieves fairness.*

Efficiency. Each party P_i invokes the PRP $2\lambda|\mathbf{S}_i| + 2t)$ times, while the server executes a "plaintext" set intersection on two sets of size $|\mathbf{S}_1| + 2t$ and $|\mathbf{S}_2| + 2t$ respectively, and also computes a commitment to this set which can also be implemented using fast symmetric-key primitives such as hashing.

3.4 Intersection Size-Hiding Server-aided PSI

Our previous protocols reveal the size of the intersection to the server which, for some applications, may be undesirable. To address this we describe a protocol that hides the size of the intersection from the server as well. The protocol is described in detail in Fig. 4 and works as follows.

The high-level idea to hiding the size of the intersection from the server is simply to not have it compute the intersection at all. Instead, P_1 will compute the intersection while the server will only play an auxiliary role and help P_1. The parties P_1 and P_2 generate a shared secret key K_1 for a PRP. Similarly, P_2 and the server generate a shared secret key K_2, also for a PRP. P_1 uses K_1 (which it shares with P_2) to send $F_{K_1}(\mathbf{S}_1)$ to the server who uses K_2 (which it shares with P_2) to return a random permutation of $F_{K_2}(F_{K_1}(\mathbf{S}_1))$ to P_1. P_2 then randomly permutes $F_{K_2}(F_{K_1}(\mathbf{S}_2))$ and sends it to P_1. P_1 then computes the intersection of the two sets and sends the result to P_2. Since P_2 knows both K_2 and K_1, he can remove both layers of encryption and learn the intersection (as usual, he aborts if the intersection is not well-formatted). Finally, P_2 needs to let P_1 learn the intersection as well. Sending the intersection directly to him is not secure since a malicious P_2 may lie about the output. Instead, P_2 will notify the server who will reveal to P_1 the random permutation he used to permute $F_{K_2}(F_{K_1}(\mathbf{S}_1))$. This allows P_1 to learn the location of each element in the intersection in his set and recover the intersection itself using that information (P_1 also aborts if the intersection is not well-formatted).

We formalize security of this protocol in Theorems 3.6 and 3.7 whose proof is omitted due to lack of space.

Theorem 3.6. *If F is pseudo-random, and $(1/t)^{\lambda-1}$ is negligible in the security parameter s, the protocol described in Fig. 4 is secure and intersection-size hiding in the presence of a malicious server and honest P_1 and P_2.*

Theorem 3.7. *The protocol described in Fig. 4 is secure in (1) the presence of malicious P_1 and an honest server and P_2; and (2) a malicious P_2 and honest server and P_1.*

Efficiency. P_1 invokes the PRP, $\lambda|S_1|+2t$ times. He also performs the "plaintext" set intersection on two sets of size $|\mathbf{S}_1| + 2t$ and $|\mathbf{S}_1| + 2t$ respectively. P_2 invokes the PRP, $2(\lambda|S_1| + 2t)$ while the server invokes the PRP $\lambda|S_1| + 2t$.

4 Our Implementation

In this section we describe the details of our implementation, including our choice of primitives and our optimization and parallelization techniques.

We implemented three of our protocols: the one described in Figure 1, which is secure against a semi-honest server; the

Setup and inputs: Let $F : \{0,1\}^k \times \mathcal{U} \to \{0,1\}^{\geq k}$ be a PRP and $t, \lambda \geq 1$. P_1 and P_2 have sets $\mathbf{S}_1 \subseteq \mathcal{U}$ and $\mathbf{S}_2 \subseteq \mathcal{U}$ as input, respectively, while the server has no input:

1. P_1 chooses sets $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \subseteq \mathcal{D} \neq \mathcal{U}$ such that $|\mathbf{D}_0| = |\mathbf{D}_1| = |\mathbf{D}_2| = t$ and sends them to P_2;
2. P_2 checks that $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2$ were constructed correctly and aborts otherwise;
3. P_1 and P_2 use $\mathcal{F}_{\mathsf{CT}}$ to agree on a random k-bit key K;
4. The party P_2 and the server use the functionality $\mathcal{F}_{\mathsf{CT}}$ to generate a k-bit key K_2
5. P_1 sends to the server:
$$\mathbf{T}_1 = \pi_1\Big(F_{K_1}\left(\mathbf{S}_1^{\lambda} + \Delta_1\right)\Big)$$
6. The server returns to P_1:
$$\mathbf{T}_1' = \pi_3\Big(F_{K_2}(\mathbf{T}_1)\Big),$$
where π_3 is a random permutation
7. P_2 sends
$$\mathbf{T}_2' = \pi_2\Big(F_{K_2}\left(F_{K_1}\left(\mathbf{S}_2^{\lambda} + \Delta_2\right)\right)\Big)$$
to P_1 where π_2 is a random permutation
8. P_1 computes $\mathrm{I} = \mathbf{T}_1' \cap \mathbf{T}_2'$ and returns the result to P_2
9. Let $\mathrm{I}^{-1} = F_{K_1}^{-1}\big(F_{K_2}^{-1}(\mathrm{I})\big)$
10. P_2 checks that I has the right form and aborts if
    (a) either $\mathbf{D}_0 \not\subset \mathrm{I}^{-1}$ or $\mathbf{D}_i \cap \mathrm{I}^{-1} \neq \emptyset$
    (b) there exists $x \in \mathbf{S}_i$ and $\alpha, \beta \in [\lambda]$ such that $x \| \alpha \in \mathrm{I}^{-1}$ and $x \| \beta \notin \mathrm{I}^{-1}$ for some $\beta \in [\lambda]$.
11. If P_2 does not abort, it notifies the server who sends π_3 to P_1. P_1 uses π_3 to map the values in $\mathbf{T}_1'$ to the values in $\mathbf{T}_1$ and respectively $\mathbf{S}_1$. Since $I \subset \mathbf{T}_1'$, P_1 learns the values in the set I^{-1}.
12. P_1 checks that I has the right form as in Step 10 and aborts if the check fails.
13. Each party computes and outputs the set
$$\Big(\mathrm{I}^{-1} - \mathbf{D}_0\Big)^{-\lambda}.$$

Figure 4: An intersection size-hiding server-aided PSI

one of Figure 2, which is secure against a malicious server; and the one of Figure 4, which hides the intersection size from the server. In the following, we refer to these protocols by SHPSI , MPSI , and SizePSI , respectively. Our implementation is in C++ and uses the Crypto++ library v.5.62 [18]. The code can be compiled on Windows and Linux and will be released publicly once when the paper is made public. Throughout, we will sometimes refer to parties that are not the server as *clients.*

To make our implementation scale to massive-size sets, we had to optimize each step of the protocols, use efficient data structures, and make extensive use of the parallelization via multi-threading.

4.1 Client Processing

The main operations during the client processing steps are the application of a PRP to generate labels and the application of a random permutation to shuffle labels around. We now describe how each of these operations is implemented.

PRP instantiation. We considered two possibilities for implementing the PRP: (1) using the Crypto++ implementation of SHA-1 (as a random oracle); (2) using the Crypto++ implementation of AES which uses the AES Instruction Set (Intel AES-NI). We ran micro benchmarks with over a million invocations and concluded that the Crypto++ AES implementation was faster than the SHA-1 implementation. As a result, we chose the Crypto++ AES implementation to instantiate the PRP. For set elements larger than the AES block size, we used AES in the CBC mode.

Random permutation instantiation. We instantiated the random permutations using a variant of the Fisher-Yates shuffle [49]. Let $\mathbf{S} \subset \mathcal{U}$ be a set and `A` be an array of size $|\mathbf{S}|$ that stores each element of $\mathbf{S}$. To randomly permute $\mathbf{S}$, for all items `A`$[i]$, we generate an index $j \leq [|\mathbf{S}|]$ uniformly at random and swap `A`$[i]$ with `A`$[j]$. We sampled the random j by applying AES to `A`$[i]$ and using the first $\log(|\mathbf{S}|)$ bits of the output.

Communication and Truncation. For our protocols—especially when running over the Internet— communication is the main bottleneck. Our experiments showed that the send and receive functions (on Windows Winsock) have a high overhead and so invoking them many times heavily slows down communication. To improve performance we therefore store the sets $\mathbf{T}_i$ in a continuous data structure in memory. This allows us to make a single invocation of the send function. Naturally, our memory usage becomes lower-bounded by the size of the sets $\mathbf{T}_i$.

Since we need to send all labels, the only solution to reduce communication complexity is to truncate the labels. Note that the output of a PRP is random so any substring of its output is also a random. This property allows us to truncate the labels without affecting security. The problem with truncation, however, is that it introduces false positives in the intersection computation due to possible collisions between the labels of different set elements. In particular, when working with a set $\mathbf{S}$, and truncating the AES output to ℓ bits, the probability of collision is less than $|\mathbf{S}|/2^{\ell/2}$ (this follows from the birthday problem). So when working with sets of tens or hundreds of millions of elements, we need to choose $80 \leq \ell \leq 100$ to reduce the probability of a collision to 2^{-20}. Another issue with truncation is that the clients cannot recover the set elements from the labels by inverting the PRP anymore. To address this, we simply store tables at the clients that map labels to their set elements.

4.2 Server Intersection

For the intersection operation that is performed by the server—or the client in the case of SizePSI —we considered and implemented two different approaches. The first is is based on a custom implementation whereas the second is based on the open-source Redis NoSQL database.

Our custom implementation. The trivial pair-wise comparison approach to compute set intersection has a quadratic complexity and does not scale to large sets. We therefore implemented the folklore set intersection algorithm based on hash tables, wherein the server hashes the elements of the first set into a hash table, and then tries to lookup the elements of the second set in the same table. Any element with a successful lookup is added to the intersection. The server then outputs a boolean vector indicating which elements of the second set are in the intersection and which are not.

To implement this algorithm, we used the `dense_hash_set` and `dense_hash_map` implementation from the Sparsehash library [26]. In contrast to their *sparse* implementation which focuses on optimizing memory usage, the dense implementation focuses on speed. The choice of data structure was critical in our ability to scale to billion-element datasets, in terms of both memory usage, and computational efficiency.

The Redis-based implementation. As an alternative to our custom implementation of the server, we also used the Redis NoSQL database. Redis is generally considered to be one of the most efficient NoSQL databases and is capable of operating on very large datasets (250 million in practice). Redis is open source and implemented in ANSI C (for high performance). It is also employed by several cloud-based companies such as Instagram, Flickr and Twitter. This highlights an important benefit of our PSI protocols (with the exception of the size-hiding protocol), which is that the server-side computations consists only of set intersection operations. As such any database can be used at the server.

Looking ahead, we note that our experiments were run on a Windows Server and that the Redis project does not directly support Windows. Fortunately, the Microsoft Open Tech group develops and maintains an experimental Windows port of Redis [57] which we used for our experiments. Unfortunately, the port is not production quality yet and we therefore were not able to use it for very large sets, i.e., for sets of size larger than 10 million (this is the reason for the "X" in one row of table 4).

We integrated the Windows port of the Redis C client library, hiredis [60] in our implementation with minor modifications. Instead of sending the labels to the server, we send them as sets of insertion queries to the Redis server. This is followed by a set intersection query which returns the result. We note that our custom server uses the same interface. To improve the mass insertion of sets, we employ the Redis pipelining feature. Pipelining adds the commands to a buffer according to the Redis protocol and sends them as they are ready. At the end, we have to wait for a reply for each of the commands. The extra delay caused by this last step, as well as the overhead of the Redis protocol, makes Redis less efficient than our custom implementation.

4.3 Output Checks

Recall that in the case of MPSI , the clients have to perform various checks on the output set I they receive from the server. In particular, they need to verify that each element in I has λ copies, that $\mathbf{D}_0$ is in I and that $\mathbf{D}_i$ is not. We use two additional data structures to facilitate these verification steps. The data structures are created by each client separately. The first structure is a dictionary `mv`, implemented with `dense_hash_set`, that maps the indices of the elements in (the truncated version of) $\mathbf{T}_i$ to the index of the element in $\mathbf{S}_i$ that it is associated with (all λ copies of the same element are mapped to the same index). The truncated labels of the elements in $\mathbf{D}_0$ and $\mathbf{D}_1$ are mapped to the values -2 and -3, respectively. The truncated labels of the elements in $\mathbf{D}_0$ are then inserted into a `dense_hash_set` data structure.

During verification, the clients can now easily use the `mv` structure and the `dense_hash_map` map to keep track of the number of copies of each element in $\mathbf{S}_i$ and to quickly check that $\mathbf{D}_0$ is present and that $\mathbf{D}_i$ is not.

4.4 Parallelizability and Multi-threading

One of the main advantages of our protocols is that they are highly parallel. To exploit this we used the POSIX thread library for the portable implementation of threads and their synchronization. At the beginning of the protocol, each client creates a certain number TCP connections with the server and starts a thread for each connection. In Step 1, the clients start preparing the values and send them in parallel to the server. In Steps 2,3, and 4, the server inserts the elements in the hash table. Since Sparsehash is not a thread-safe library, these steps cannot be performed in parallel. Finally, in Step 5, the server performs a parallel lookup of the second client's set and returns the intersection as a boolean vector. We report on the effect of multi-threading on the running time of our protocols in the next section.

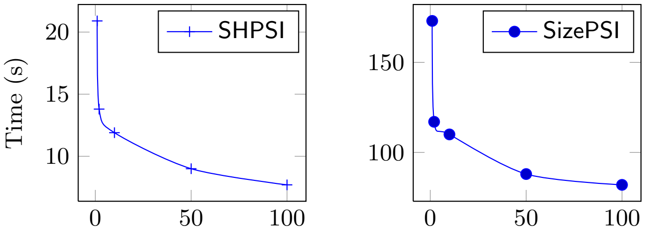
5 Experimental Evaluation

Next, we evaluate the performance and scalability of our implementations. In particular, we investigate the effect of multi-threading on the efficiency of our protocols, we evaluate the scalability of SHPSI by executing it on billion-element sets, and we compare the efficiency of our protocols with state-of-the-art two-party PSI protocols as well as with non-private solutions.

We generate the input sets on the fly and as part of the execution. Each element is a 16 byte value. We note that, for our implementation, the size of the intersection does not effect computation or communication. This is because the server does not return the intersection but a bit vector that indicates whether each element of the partie's set is in the intersection or not.

5.1 Effect of Multi-Threading

To demonstrate the effect of parallelization, we ran an experiment where we increased the number of threads for a given set size (10 Million) for both the SHPSI and the SizePSI protocols. Results are presented in two separate graphs in Figure 5.1. The use of parallelization particularly improves the communication time which dominates the total running time of our protocols. We get up to a factor of 3 improvement in total running time by increasing the number of threads.



(a) Thread#:SHPSI

(b) Thread#:SizePSI

Figure 5: Effect of multi-threading on the runtime of our protocols. Set size is 10 Million.

5.2 Scalability of our SHPSI Protocol

We examine the scalability of our protocol in the WAN setup. We run SHPSI for sets ranging from $100K$ to 1 billion elements. The total running times and the size of communication (for each client) are provided in Table 1. Note that even for sets with 1 billion elements, our protocol runs on the order of minutes.

We used 3 Windows Azure services connected over the Internet. The server was an 8-core Windows server 2012 VM with 14GB of memory located in the West US region. For each client, we used a 8-core Windows server 2012 VM with 7GB of memory. The clients were both located in the East US region to guarantee that they were not on the same network as the server. We chose to run our clients in the Cloud (as opposed to locally) to provide a somewhat uniform platform that can be used by others to reproduce our experiments. For the billion-element sets, we increased the client's RAM to 14 GB and the server's to 24GB.

Set Size	Threads #	Comm.	Total
100K	20	1	532 (ms)
1M	20	10	1652 (ms)
10M	100	114	7 (s)
100M	100	1239	53 (s)
1B	100	12397	580 (s)

Table 1: Scalability of SHPSI . Communication is in MegaBytes.

5.3 Comparison with Standard PSI

We compare SHPSI which provides security against a semi-honest server and our SizePSI protocol with malicious security against the state-of-the-art two-party PSI protocol [23] (we used an implementation provided to us by the authors). We stress that the protocol of [23] is secure against semi-honest adversaries in the standard MPC setting. The point of this comparison is simply to demonstrate that server-aided protocols can allow for significant efficiency improvements over standard two-party protocols. The provided implementation of [23] is intended for LAN setting and can be compiled under Linux, so we used the same setup for our comparison. In this setting, our experimental testbed consisted of 3 machines, each of which was a 3GHz Xeon server with 16GB of memory running Linux as their OS. The timings are provided in Table 2. They include the total running time for each protocol, starting from when the clients start running until they output the result of the intersection (i.e., the communication times are included). We only went up to sets of $100K$ elements in order to keep the running time of the protocol of [23] manageable.

Set size	[23] (ms)	SHPSI (ms)	SizePSI (ms)
1000	600	2	13
10000	6725	12	112
50000	116155	59	488
100000	559100	117	996

Table 2: Comparison of SHPSI , SizePSI and [23]. Times include communication (10 Threads).

5.4 Comparison with Plaintext Set Intersection

In this experiment, we compare SHPSI , and SizePSI (with $\lambda = 3$ and $t = 1000000$, yielding $s \approx 40$) with a plaintext set intersection for a wide range of set sizes. In particular, we implemented and tested a non-private server-aided set intersection execution, where the clients send their *plaintext* sets and receive the intersection from the server. We employed all the optimizations and parallelization applied to our own protocols (such as multi-threading, choice of data structures etc.) to the plaintext protocol as well. This experiment was just so we could compare the overhead incurred by our protocols over plaintext intersection. The times are in Table 3. Note that our SHPSI protocol is at most 10% slower than the plaintext intersection for most set sizes while SizePSI is a factor of 4-10 slower. This is in contrast to the setting of standard MPC where going from semi-honest to malicious security increases computation and communication by orders of magnitude.

Set Size	SHPSI C.	SizePSI C.	Plain T.	SHPSI T.	SizePSI T.
100K	1MB	7.4MB	530	532	2000
1M	10MB	74.3MB	1600	1652	10232
10M	114MB	619MB	7102	7717	82323
20M	228MB	1.2GB	10780	11662	185123

Table 3: Comparison of our SHPSI and SizePSI to plaintext set intersection. T. is short for total time. C. is short for communication and times are in millisecond.

5.5 Porting to NoSQL Databases

In our final experiment we replace our custom server with a Redis server with which the clients interact using insertion and set intersection queries. Table 4 show details of some of our timings. The experiment shows a nice feature of our SHPSI and MPSI protocols i.e. that they can be easily plugged into existing NoSQL database implementation without the need to make any changes to them.

Set Size	Plain T.	SHPSI T.	MPSI T.
1000	380.3	381.0	857.4
10000	934.0	939.7	2020.0
100000	2170.4	2239.8	7368.3
1000000	5798.9	6496.3	61544.9
10000000	47041.5	54020.5	X

Table 4: Comparison of our SHPSI and MPSI to plaintext set intersection when server is implemented by Redis. T. is short for total time in milliseconds.

6 References

[1] Bill Aiello, Yuval Ishai, and Omer Reingold. Priced oblivious transfer: How to sell digital goods. In *EUROCRYPT 2001*, pages 119–135. 2001.

[2] G. Asharov, A. Jain, A. Lopez-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *EUROCRYPT*, 2012.

[3] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC*, pages 137–156, 2007.

[4] Pierre Baldi, Roberta Baronio, Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Countering gattaca: efficient and secure testing of fully-sequenced human genomes. In *CCS*, pages 691–702, 2011.

[5] B. Barak and O. Goldreich. Universal arguments and their applications. In *CCC*, 2002.

[6] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *CCS*, 2008.

[7] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, 2008.

[8] P. Bogetoft, D. Christensen, I. Damgard, M. Geisler, T. Jakobsen, M. Krøigaard, J. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In *FC*, 2009.

[9] Fabrice Boudot, Berry Schoenmakers, and Jacques Traore. A fair and efficient solution to the socialist millionaires' problem. *Discrete Applied Math.*, 111(1):23–36, 2001.

[10] Jan Camenisch and Gregory Zaverucha. Private intersection of certified sets. *FC*, pages 108–127, 2009.

[11] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[12] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO '13*. Springer, 2013.

[13] Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security (ACNS '05)*, volume 3531 of *Lecture Notes in Computer Science*, pages 442–455. Springer, 2005.

[14] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In *Advances in Cryptology - ASIACRYPT '10*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[15] R. Cleve. Limits on the security of coin flips when half the processors are faulty. In *STOC*, pages 364–369, 1986.

[16] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM Conference on Computer and Communications Security (CCS '06)*, pages 79–88. ACM, 2006.

[17] Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Moti Yung. Efficient robust private set intersection. In *ACNS*, pages 125–142. Springer, 2009.

[18] Wei Dai. Crypto++ library. http://www.cryptopp.com/, 2013. [Online; accessed 08-May-2013].

[19] I. Damgard, M. Geisler, M. Krøigaard, and J.-B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *PKC*, 2009.

[20] I. Damgard and Y. Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In *CRYPTO*, 2005.

[21] I. Damgard, Y. Ishai, M. Krøigaard, J.-B. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *CRYPTO*, 2008.

[22] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. *ASIACRYPT*, pages 213–231, 2010.

[23] Emiliano De Cristofaro and Gene Tsudik. Experimenting with fast private set intersection. In *Trust and Trustworthy Computing*, Lecture Notes in Computer Science.

[24] Changyu Dong, Liqun Chen, Jan Camenisch, and Giovanni Russello. Fair private set intersection with a semi-trusted arbiter. Cryptology ePrint Archive, Report 2012/252, 2012. http://eprint.iacr.org/.

[25] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. Scapi: The secure computation application programming interface. Technical report, Cryptology ePrint Archive, Report 2012/629, 2012. http://crypto. biu. ac. il/scapi, 2012.

[26] Donovan Hide et al. Sparsehash library. https://code.google.com/p/sparsehash/, 2013. [Online; accessed 08-May-2013].

[27] Ronald Fagin, Moni Naor, and Peter Winkler. Comparing information without leaking it. *Communications of the ACM*, 39(5):77–85, 1996.

[28] U. Feige, J. Killian, and M. Naor. A minimal model for secure computation (extended abstract). In *STOC*,

1994.

[29] Marc Fischlin, Benny Pinkas, Ahmad-Reza Sadeghi, Thomas Schneider, and Ivan Visconti. Secure set intersection with untrusted hardware tokens. In *Topics in Cryptology–CT-RSA 2011*, pages 1–16. Springer, 2011.

[30] Michael Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *EUROCRYPT 2004*, pages 1–19. Springer, 2004.

[31] J. Garay, P. MacKenzie, M. Prabhakaran, and K. Yang. Resource fairness and composability of cryptographic protocols. *Theory of Cryptography*, pages 404–428, 2006.

[32] Ran Gelles, Rafail Ostrovsky, and Kina Winoto. Multiparty proximity testing with dishonest majority from equality testing. In *Automata, Languages, and Programming*. 2012.

[33] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[34] E-J. Goh. Secure indexes. Technical Report 2003/216, IACR ePrint Cryptography Archive, 2003. See http://eprint.iacr.org/2003/216.

[35] S. Gordon and J. Katz. Partial fairness in secure two-party computation. *EUROCRYPT*, pages 157–176, 2010.

[36] S.D. Gordon, C. Hazay, J. Katz, and Y. Lindell. Complete fairness in secure two-party computation. *J. of the ACM*, 58(6):24, 2011.

[37] Carmit Hazay and Yehuda Lindell. Constructions of truly practical secure protocols using standardsmartcards. In *CCS*, pages 491–500, 2008.

[38] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *TCC*, 2008.

[39] Carmit Hazay and Kobbi Nissim. Efficient set operations in the presence of malicious adversaries. *Public Key Cryptography–PKC 2010*, pages 312–331, 2010.

[40] W. Henecka, S. Kogl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS*, 2010.

[41] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[42] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

[43] Stanislaw Jarecki and Xiaomin Liu. Fast secure computation of set intersection. *SCN*, pages 418–435, 2010.

[44] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party comptuation. Technical Report 2011/272, IACR ePrint Cryptography Archive, 2011.

[45] S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security (FC '13)*, 2013.

[46] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press, 2012.

[47] Seny Kamara, Payman Mohassel, and Ben Riva. Salus: a system for server-aided secure function evaluation. In *CCS*, pages 797–808, 2012.

[48] J. Katz, R. Ostrovsky, and A. Smith. Round efficiency of multi-party computation with a dishonest majority. In *EUROCRYPT*, 2003.

[49] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[50] Y. Lindell. Parallel coin-tossing and constant-round secure two-party computation. In *CRYPTO*, 2001.

[51] Helger Lipmaa. Verifiable homomorphic oblivious transfer and private equality test. In *ASIACRYPT*. 2003.

[52] L. Malka. Vmcrypt: modular software architecture for scalable secure computation. In *CCS*, 2011.

[53] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, 2004.

[54] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. Botgrep: finding p2p bots with structured graph analysis. In *USENIX Security*, 2010.

[55] Arvind Narayanan, Narendran Thiagarajan, Mugdha Lakhani, Michael Hamburg, and Dan Boneh. Location privacy via private proximity testing. In *NDSS*, 2011.

[56] B. Pinkas. Fair secure two-party computation. *Eurocrypt*, pages 647–647, 2003.

[57] Henry Rawas. Redis windows port. https://github.com/MSOpenTech/redis, 2013. [Online; accessed 08-May-2013].

[58] Gokay Saldamli, Richard Chow, Hongxia Jin, and Bart Knijnenburg. Private proximity testing with an untrusted server. In *SIGSAC*, pages 113–118, 2013.

[59] D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE Symposium on Research in Security and Privacy*, pages 44–55. IEEE Computer Society, 2000.

[60] Tang Yaguang. hiredis win32. https://github.com/texnician/hiredis-win32, 2013. [Online; accessed 08-May-2013].

[61] A. Yao. Protocols for secure computations. In *FOCS*, 1982.

[62] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, pages 162 –167, oct. 1986.

APPENDIX

A Server-aided PSI vs. SSE

In this Section, we provide a comparison between the notions of server-aided PSI and SSE and, in particular, to index-based SSE constructions [59, 34, 13, 16, 14, 46, 45, 12]. An index-based SSE scheme takes as input a secret key K and a dataset $\{(w_i, v_i)\}_{i=1}^n$, where the w_i's are keywords from a universe $\mathbf{W}$ and the v_i's are arbitrary strings. It outputs an encrypted index γ that can be queried using tokens generated from the key K and a keyword w. So, given a token τ for keyword w, γ can be queried to recover the data item v_i associated with w.

Server-aided PSI from SSE. Informally, a server-aided PSI protocol can be constructed from any (index-based) SSE scheme as follows. The parties P_1 and P_2 first generate two shared keys: K_1 for the SSE scheme; and K_2 for a pseudo-random permutation π. P_1 sends to the server an encrypted index for the dataset $\{(x_i, \pi_{K_2}(x_1))\}_{i=1}^n$, where $\mathbf{S}_1 = \{x_1, \ldots, x_n\} \subseteq \mathbf{W}$, and P_2 sends a set of tokens $(\tau_1, \ldots, \tau_{|\mathbf{S}_2|})$, where τ_i is the token for the ith element in P_2's set $\mathbf{S}_2 \subseteq \mathbf{W}$.

To compute the intersection, the server queries γ with each token $\tau_1, \ldots, \tau_{|\mathbf{S}_2|}$ and returns the recovered items to P_1 and P_2 who invert the PRP to find the intersection. [3]

SSE from server-aided PSI. Similarly, SSE schemes can be constructed from a certain kind of server-aided PSI protocol. More precisely, from protocols that: (1) include a setup phase to generate a shared key; and (2) require a single round of communication with the server, i.e., P_1 and P_2 send a single message to the server and the server returns a single message. Note that all our protocols have this structure and therefore yield SSE schemes.

Given a dataset $\{(w_i, f_i)\}_{i=1}^n$, where the f_i's are files, the client works as follows. It first encrypts the files using a symmetric encryption scheme which results in ciphertexts $(c_1, \ldots, c_n)$. It then simulates the setup phase of the protocol (playing both P_1 and P_2) n times in order to generate n keys $(K_1, \ldots, k_n)$. Once the keys are generated it executes n simulations of P_1, using the key K_i and the input set $\{w : w \in f_i\}$ for the ith execution (here $w \in f_i$ means that there exists a pair (w, f_i) in the dataset). In each execution, P_1 generates a message for the server $\mathbf{msg}_i$. The client then sends the encrypted files $(c_1, \ldots, c_n)$ and $\gamma = (\mathbf{msg}_{1,1}, \ldots, \mathbf{msg}_{1,n})$ to the server.

To search for a disjunction of keywords $w_1 \vee \cdots \vee w_d$, the client executes n simulations of P_2 (using the same coins as before), using the key K_i and the input set $\{w_1, \ldots, w_\ell\}$ for the ith execution. In each execution, P_2 will generate a message $\mathbf{msg}_{2,i}$. The client then sends the messages $\tau = (\mathbf{msg}_{2,1}, \ldots, \mathbf{msg}_{2,n})$ to the server.

Given γ and τ as above, the SSE server does the following: for all $i \in [n]$, it simulates the PSI server with messages $\mathbf{msg}_{1,i}$ and $\mathbf{msg}_{2,i}$ as input and returns a result $\mathbf{msg}_{3,i}$ to the client. From the set of results $(\mathbf{msg}_{3,1}, \ldots, \mathbf{msg}_{3,n})$, the client can extract the intersections between the keywords in f_i and the disjunction and if that intersection is non-empty then the ith file matched the query.

Note that the resulting schemes have several limitations. First, if the underlying server-aided PSI protocol is not size-hiding (with respect to the server), the SSE scheme leaks the number of terms (in the disjunction) that are contained in each file. Another limitation is that the search time for the server and the token size are linear in the number of files. Finally, we point out that if the underlying server-aided PSI protocol does not leak the intersection to the server (as required by our definition) then only the client can learn whether a file matched its query. This is undesirable for SSE as it requires an extra round of interaction for the client to then download the relevant files.

B Proof of Theorem 3.1

Theorem 3.1. *The protocol described in Fig. 1 is secure in the presence (1) a semi-honest server and honest parties or (2) an honest server and any collusion of malicious parties.*

Proof sketch. We construct a simulator $\textsc{Sim}_{n+1}$ who receives $|\mathbf{S}_1|$, $|\mathbf{S}_2|$ and $|\mathbf{S}_1 \cap \mathbf{S}_2|$ from the functionality and simulates a semi-honest server P_{n+1} by emulating the execution of the protocol between P_{n+1} and honest P_1 through P_n. $\textsc{Sim}_{n+1}$ first generates n arbitrary sets $\widetilde{\mathbf{S}}_1$ through $\widetilde{\mathbf{S}}_n$ such that $|\widetilde{\mathbf{S}}_i| = |\mathbf{S}_i|$ and that $|\bigcap_{i=1}^n \widetilde{\mathbf{S}}_i| = |\bigcap_{i=1}^n \mathbf{S}_i|$. It then creates n sets $\mathbf{T}_1, \ldots, \mathbf{T}_n$ such that

$$\mathbf{T}_i = \pi_i\Big(F_K\big(\widetilde{\mathbf{S}}_i\big)\Big),$$

where π_i is a random permutation and K is a random k-bit key. $\textsc{Sim}_{n+1}$ then sends $\mathbf{T}_1$ through $\mathbf{T}_n$ to the server who returns a set $\mathrm{I} = \bigcap_{i=1}^n \mathbf{T}_i$. $\textsc{Sim}_{n+1}$ then outputs whatever the server P_{n+1} outputs. It follows by construction and by the pseudo-randomness of F that the server's view is indistinguishable during its view in the real-model execution and, therefore, so is its output.

We now construct a simulator $\textsc{Sim}_1$ that simulates an adversary $\mathcal{A}$ who corrupts any collusion of malicious parties $C \subset [n]$ by emulating the execution of the protocol between those parties and the honest parties in $[n] \setminus C$ and the honest P_{n+1}. $\textsc{Sim}_1$ receives the key K from the $\mathcal{F}_{\mathsf{CT}}$ functionality. After receiving the set of labels $\mathbf{T}_i$ for $i \in C$, it recovers $\mathbf{S}_i$ by computing $F_K^{-1}(\mathbf{T}_i)$ and sends it to the trusted party. After receiving the intersection $\bigcap_{i=1}^n \mathbf{S}_i$, it computes $\mathrm{I} = F_K(\bigcap_{i=1}^n \mathbf{S}_i)$ and sends it to $\mathcal{A}$. Finally, $\textsc{Sim}_1$ outputs whatever $\mathcal{A}$ outputs. It follows by construction and by the pseudo-randomness of F that the joint distribution of views of P_i for $i \in C$ is indistinguishable from their joint view during a real-model execution.

■

C Proofs of Theorems 3.2 and 3.3

We begin by establishing a Lemma that will be useful for the proof of Theorem 3.2. Informally, the Lemma bounds the probability that a (polynomial-time) adversary can identify the labels of all λ copies of some set $\mathbf{Z} \subset \mathbf{X}$ given a set $\pi\big(F_K(\mathbf{X}^\lambda + \mathbf{Y})\big)$ of randomly permuted labels.

Lemma C.1. *Let $\mathbf{X} \subset \mathcal{U}$ and $\mathbf{Y} \subset \mathbf{D} \neq \mathcal{U}$. If $F : \{0,1\}^k \times \mathcal{U} \to \{0,1\}^{\geq k}$ is pseudo-random, then for all* PPT *adversaries $\mathcal{A}$,*

$$\Pr\left[\mathcal{A}\big(\pi\big(F_K(\mathbf{X}^\lambda + \mathbf{Y})\big)\big) = F_K(\mathbf{Z}^\lambda)\right] \leq \binom{|\mathbf{X}|}{|\mathbf{Z}|} \cdot \binom{|F_K(\mathbf{X}^\lambda + \mathbf{Y})|}{\lambda \cdot |\mathbf{Z}|}^{-1} + \mathsf{negl}(k),$$

where the probability is over the choice of π and K and the coins of $\mathcal{A}$, and $\mathbf{Z}$ is some subset of $\mathbf{X}$.

[3] This can be made more efficient using similar optimizations as those described in Section 4.

Proof sketch. Note that the pseudo-randomness of F guarantees that each label $\ell \in \pi(F_K(\mathbf{X}^\lambda + \mathbf{Y}))$ reveals no partial information about the element it encodes. In addition, the random permutation π guarantees that the position of a label ℓ reveals no partial information about the element it encodes.

It follows that, given $\pi(F_K(\mathbf{X}^\lambda + \mathbf{Y}))$, a polynomial-time adversary $\mathcal{A}$ will output a set $F_K(\{x\}^\lambda)$ for some $x \in \mathbf{X}$, with probability at most

$$|\mathbf{X}| \cdot \binom{|F_K(\mathbf{X}^\lambda + \mathbf{Y})|}{\lambda}^{-1} + \mathsf{negl}(k),$$

since the best it can do is guess the labels. More generally, it will output a set $F_K(\mathbf{Z}^\lambda)$ for some $\mathbf{Z} \subset \mathbf{X}$, with probability at most

$$\binom{|\mathbf{X}|}{|\mathbf{Z}|} \cdot \binom{|F_K(\mathbf{X}^\lambda + \mathbf{Y})|}{\lambda \cdot |\mathbf{Z}|}^{-1} + \mathsf{negl}(k).$$

■

We are now ready to proceed to the proof of Theorem 3.2.

Theorem 3.2. *If F is pseudo-random, and $(1/t)^{\lambda-1}$ is negligible in the statistical security parameter s, the protocol described in Fig. 2 is secure in the presence of a malicious server and honest P_1 and P_2.*

Proof sketch. We construct a simulator $\textsc{Sim}_3$ who receives $|\mathbf{S}_1|$, $|\mathbf{S}_2|$ and $|\mathbf{S}_1 \cap \mathbf{S}_2|$ from the functionality and simulates a malicious server by emulating the execution of the protocol between the server and honest P_1 and P_2. $\textsc{Sim}_3$ first generates two arbitrary sets $\widetilde{\mathbf{S}}_1$ and $\widetilde{\mathbf{S}}_2$ such that $|\widetilde{\mathbf{S}}_1| = |\mathbf{S}_1|$, $|\widetilde{\mathbf{S}}_2| = |\mathbf{S}_2|$ and that $|\widetilde{\mathbf{S}}_1 \cap \widetilde{\mathbf{S}}_2| = |\mathbf{S}_1 \cap \mathbf{S}_2|$. It then generates three sets $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \in \mathcal{D}$ each of size t and constructs sets $\mathbf{T}_i$ for $i \in \{1, 2\}$ such that

$$\mathbf{T}_i = \pi_i\Big(F_K\Big(\widetilde{\mathbf{S}}_i^\lambda + \Delta_i\Big)\Big),$$

where π_1 and π_2 are random permutations and K is a random k-bit key. It then sends $\mathbf{T}_1$ and $\mathbf{T}_2$ to server who returns a set I. $\textsc{Sim}_3$ performs the following checks and sends an abort message if any of them succeed:

1. either $\mathbf{D}_0 \not\subset F_K^{-1}(\mathrm{I})$ or $\mathbf{D}_i \cap F_K^{-1}(\mathrm{I}) \neq \emptyset$
2. there exists $x \in \mathbf{S}_i$ and $\alpha, \beta \in [\lambda]$ such that $x\|\alpha \in F_K^{-1}(\mathrm{I})$ and $x\|\beta \notin F_K^{-1}(\mathrm{I})$

It follows by construction and by the pseudo-randomness of F that, conditioned on the server returning $\mathrm{I} = \mathbf{T}_1 \cap \mathbf{T}_2$, the view of the server in this simulated execution is indistinguishable from its view in the real-model execution with honest P_1 and P_2. Note that if the server returns an $\mathrm{I} \neq \mathbf{T}_1 \cap \mathbf{T}_2$, the only difference in the executions will be if all the checks fail. To see why, observe that if the checks fail then P_1 and P_2 will output some $\mathrm{I} \neq \mathbf{S}_1 \cap \mathbf{S}_2$ in the real execution, whereas they will output $\mathrm{I} = \mathbf{S}_1 \cap \mathbf{S}_2$ in the ideal execution since their output is computed by the trusted party. We therefore need to show that the probability that (1) the server returns an incorrect intersection I and (2) that $\textsc{Sim}_3$ does not abort, is negligible.

Claim. $\Pr[\mathrm{I} \neq \mathbf{T}_1 \cap \mathbf{T}_2 \bigwedge \textsc{Sim}_3 \text{ does not abort}] \leq 1/t^{\lambda-1} + \mathsf{negl}(k)$.

Clearly, if $\mathrm{I} \neq \mathbf{T}_1 \cap \mathbf{T}_2$ then I either contains elements not in $\mathbf{T}_1 \cap \mathbf{T}_2$ or is missing elements from $\mathbf{T}_1 \cap \mathbf{T}_2$. We will bound the probability that the server either adds elements or removes elements without causing an abort.

We now bound the probability that the server adds elements from $\mathbf{S}_1$ that are outside the intersection (i.e., from the set $\mathbf{S}_1 - (\mathbf{S}_1 \cap \mathbf{S}_2) = \mathbf{S}_1 - \mathbf{S}_2$) in such a way that $\textsc{Sim}_3$ does not abort, i.e., in such a way that the three checks above are satisfied. Note that in order to do this, given $\mathbf{T}_1$ and $\mathbf{T}_2$, the best the server can do is to try to identify the labels $F_K(\mathbf{R}^\lambda)$ out of the set of labels $\mathbf{T}_1 - \mathbf{T}_2$ for some $\mathbf{R} \subset \mathbf{S}_1 - \mathbf{S}_2$. But note that

$$\mathbf{T}_1 - \mathbf{T}_2 = \pi\Big(F_K\Big(\big(\mathbf{S}_1 - \mathbf{S}_2\big)^\lambda + \mathbf{D}_1\Big)\Big),$$

for some random permutation π, therefore, by setting $\mathbf{X} = \mathbf{S}_1 - \mathbf{S}_2$, $\mathbf{Y} = \mathbf{D}_1$ and $\mathbf{Z} = \mathbf{R}$ in Lemma C.1 and noting that

$$|\mathbf{T}_1 - \mathbf{T}_2| = \lambda \cdot |\mathbf{S}_1 - \mathbf{S}_2| + t,$$

it follows that the server will output $F_K(\mathbf{R}^\lambda)$ with probability at most

$$\epsilon = \binom{|\mathbf{S}_1 - \mathbf{S}_2|}{|\mathbf{R}|} \cdot \binom{\lambda \cdot |\mathbf{S}_1 - \mathbf{S}_2| + t}{\lambda \cdot |\mathbf{R}|}^{-1} + \mathsf{negl}(k).$$

Setting $n = |\mathbf{S}_1|$, $m = |\mathbf{S}_1 \cap \mathbf{S}_2|$ and $r = |\mathbf{R}|$, we have

$$\begin{aligned} \epsilon &= \binom{n-m}{r} \cdot \binom{\lambda \cdot (n-m) + t}{\lambda \cdot}^{-1} + \mathsf{negl}(k) \\ &= \frac{\frac{(n-m)\cdots(n-m-r+1)}{r\cdots 1}}{\frac{((n-m)\lambda+t)\cdots((n-m-r)\lambda+t+1)}{(r\lambda)\cdots 1}} + \mathsf{negl}(k) \\ &\leq \frac{1}{\frac{(n-m)\lambda+t}{r\lambda}\cdots\frac{(n-m)\lambda+t+r+1}{r+1}} + \mathsf{negl}(k) \\ &\leq \frac{1}{t^{r(\lambda-1)}} + \mathsf{negl}(k) \end{aligned}$$

In other words, the probability that the server adds elements from $\mathbf{S}_1 - \mathbf{S}_2$ without $\textsc{Sim}_3$ sending an abort message is at most negligibly close to $1/t^{r(\lambda-1)}$. The server's best strategy is therefore to set $r = 1$ and only add a single element to the intersection. Note, however, that this achieves a still small probability of $1/t^{\lambda-1}$, which is negligible in λ.

A similar analysis holds for the case where the server adds elements from $\mathbf{S}_2 - \mathbf{S}_1$.

Next, we consider the case where the server removes items from the intersection $\mathbf{S}_1 \cap \mathbf{S}_2$ in such a way that $\textsc{Sim}_3$ does not abort (again, in such a way that the three checks above are satisfied). To do this, given $\mathbf{T}_1$ and $\mathbf{T}_2$, the best the server can do is to try to identify the labels $F_K(\mathbf{R}^\lambda)$ from the set of labels

$$\mathbf{T}_1 \cap \mathbf{T}_2 = \pi\Big(F_K\big((\mathbf{S}_1 \cap \mathbf{S}_2)^\lambda + \mathbf{D}_0\big)\Big)$$

for some set $\mathbf{R} \subset \mathbf{S}_1 \cap \mathbf{S}_2 - \mathbf{D}_0 \subset \mathbf{S}_1 \cap \mathbf{S}_2$. Setting $\mathbf{X} = \mathbf{S}_1 \cap \mathbf{S}_2$, $\mathbf{Y} = \mathbf{D}_0$ and $\mathbf{Z} = \mathbf{R}$ and applying Lemma C.1, we have that the server will succeed with probability at most

$$\epsilon = \binom{|\mathbf{S}_1 \cap \mathbf{S}_2|}{|\mathbf{R}|} \cdot \binom{\big|F_K\,\big((\mathbf{S}_1 \cap \mathbf{S}_2)^\lambda + \mathbf{D}_0\big)\big|}{\lambda \cdot |\mathbf{R}|}^{-1} + \mathsf{negl}(k).$$

Setting $m = |\mathbf{S}_1 \cap \mathbf{S}_2|$ and $r = |\mathbf{R}|$ we have

$$\begin{aligned}
\epsilon &= \binom{m}{r} \cdot \binom{\lambda \cdot m + t}{\lambda \cdot r}^{-1} \\
&= \frac{\frac{m\cdots(m-r+1)}{r\cdots 1}}{\frac{(m\lambda+t)\cdots((m-r)\lambda+t+1)}{(r\lambda)\cdots 1}} + \mathsf{negl}(k) \\
&\leq \frac{1}{\frac{m\lambda+t}{r\lambda}\cdots\frac{(m-r)\lambda+r+t+1}{r+1}} + \mathsf{negl}(k) \\
&\leq \frac{1}{t^{r(\lambda-1)}} + \mathsf{negl}(k)
\end{aligned}$$

The above probability is maximized when $r = 1$, i.e., when the server removes a single element from the intersection. Thus, the maximum probability with which the server can remove elements from the intersection without $\textsc{Sim}_3$ aborting is $1/t^{\lambda-1}$, which is negligible in λ.

■

Theorem 3.3. *The protocol described in Fig. 2 is secure in* (1) *the presence of malicious P_1 and an honest server and P_2; and* (2) *a malicious P_2 and honest server and P_1.*

Proof sketch. We first show that the protocol is secure in the presence of a malicious P_1 and honest server and P_2. We construct a simulator $\textsc{Sim}_1$ that that simulates a corrupted P_1 by emulating the execution of the protocol between P_1 and honest P_2 and server.

Upon receiving the sets $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2$ from P_1, $\textsc{Sim}_1$ checks that they are correctly formed and aborts if they are not. If they are correctly formed, it proceeds to simulating the coin tossing step with P_1 and sets the output to a randomly chosen key K. After receiving $\mathbf{T}_1$, it computes $F_K^{-1}(\mathbf{T}_1)$ and checks that (1) the result includes $\mathbf{D}_0$; (2) that the result does not contain any elements of $\mathbf{D}_2$; and (3) that the set $F_K^{-1}(\mathbf{T}_1) - \mathbf{D}_1$ contains at least λ copies of each element. If any of the checks fail it aborts, otherwise it proceeds to recover the set:

$$\mathbf{S}_1 = \left(F_K^{-1}(\mathbf{T}_1) - \mathbf{D}_0 - \mathbf{D}_1\right)^{-\lambda}.$$

$\textsc{Sim}_1$ then sends $\mathbf{S}_1$ to the trusted party and receives the set $\mathbf{S}_1 \cap \mathbf{S}_2$ from which it computes

$$\mathrm{I} = \pi\left(F_K\left(\left(\mathbf{S}_1 \cap \mathbf{S}_2\right)^{\lambda} + \mathbf{D}_0\right)\right).$$

where π is a random permutation. Finally, $\textsc{Sim}_1$ sends I to P_1 and outputs whatever P_1 outputs.

It follows by construction and by the pseudo-randomness of F that, conditioned on P_1 generating the sets $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2$ correctly and on P_1 constructing its set $\mathbf{T}_1$ correctly (i.e., so that it includes λ copies of each element and the set $\mathbf{D}_0$), the view of P_1 during the simulation is indistinguishable from its view during the real-model execution and, therefore, so is its output. If, on the other hand, either the dummy sets or $\mathbf{T}_1$ violate any of the checks performed by $\textsc{Sim}_1$, both executions abort.

Security against a malicious P_2 and honest P_1 and server is analogous to the case of a malcious P_1.

■

D Proof of Theorems 3.4 and 3.5

Theorem 3.4. *If F is pseudo-random, and $(1/t)^{\lambda-1}$ is negligible in the security parameter s, the protocol described in Fig. 3 is secure in the presence of a malicious server and honest P_1 and P_2.*

Since the proof of security for the case where the server is malicious and P_1 and P_2 are honest is analogous to the proof of Theorem 3.2, we omit the proof for the above theorem and focus only on the cases where either P_1 or P_2 are malicious.

Theorem 3.5. *The protocol described in Fig. 3 is secure in* (1) *the presence of malicious P_1 and an honest server and P_2; and* (2) *a malicious P_2 and honest server and P_1, and also achieves fairness.*

Proof sketch.

Towards showing the first case, we construct a simulator $\textsc{Sim}_1$ that simulates a corrupted P_1 by emulating the execution of the protocol between P_1 and honest P_2 and server. Upon receiving the sets $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2$ from P_1, $\textsc{Sim}_1$ checks that they are correctly formed and aborts if they are not. If they are correctly formed, it proceeds to simulating the coin tossing steps with P_1 and sets the outputs to randomly chosen keys K_1 and K_2. After receiving $\mathbf{T}_1$, it computes and sends

$$\mathbf{S}_1 = F_{K_1}^{-1}\left(\left(F_{K_2}^{-1}(\mathbf{T}_1) - \mathbf{D}_0 - \mathbf{D}_1\right)^{-\lambda}\right)$$

to the trusted party. Upon receiving the intersection $\mathbf{S}_1 \cap \mathbf{S}_2$ from the trusted party, $\textsc{Sim}_1$ a commitment $\mathsf{com}(\mathrm{I}')$ to P_1, where

$$\mathrm{I}' = F_{K_2}\left(F_{K_1}(\mathbf{S}_1 \cap \mathbf{S}_2)^{\lambda} + \mathbf{D}_0\right) + \mathbf{W}$$

where $\mathbf{W}$ is a set of padding elements of size $|\mathbf{S}_1| + t - |\mathrm{I}'|$. After receiving $F_{K_1}(\mathbf{S}_1')$, for some $\mathbf{S}_1'$, and $\mathbf{D}_0', \mathbf{D}_1', \mathbf{D}_2'$ from P_1, $\textsc{Sim}_1$ checks that $\mathbf{T}_1$ was constructed correctly. If not, it aborts otherwise it opens the commitment for P_1. $\textsc{Sim}_1$ then outputs whatever P_1 outputs.

It clearly follows by construction and by the pseudo-randomness of F that, conditioned on P_1 constructing the dummy sets and $\mathbf{T}_1$ correctly, it's view during the simulation is indistinguishable from its view during the real-model execution and, therefore, that so is its output. If, on the other hand, either the dummy sets or $\mathbf{T}_1$ are not constructed correctly, both executions will abort before P_1 receives its output. In particular, if $\mathbf{T}_1$ violates the checks then the real-model execution will be aborted by the server before it opens its commitment, and the ideal-model execution will be aborted by $\textsc{Sim}_1$ before opening its commitment.

Security against a malicious P_2 and honest P_1 and server is analogous to the case of a malicious P_1.

■

E Proof of Theorems 3.6 and 3.7

Theorem 3.6. *If F is pseudo-random, and $(1/t)^{\lambda-1}$ is negligible in the security parameter s, the protocol described* *in Fig. 4 is secure and intersection-size hiding in the presence of a malicious server and honest P_1 and P_2.*

Proof sketch. We construct a simulator $\textsc{Sim}_3$ who receives $|\mathbf{S}_1| = n$ and $|\mathbf{S}_2| = m$ from the functionality and simulates the execution of the protocol with the server. $\textsc{Sim}_3$ generates a random set $S_1 \subset \mathcal{U}$ of size n as well as sets $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \subseteq \mathcal{D} \neq \mathcal{U}$ and a PRP key K_1. It also runs the coin tossing protocol with the server to choose a PRP key K_2. Then, $\textsc{Sim}_3$ computes a set

$$\mathbf{T}_1 = \pi_1\Big(F_{K_1}\Big(\mathbf{S}_1^\lambda + \Delta_1\Big)\Big)$$

where $\Delta_1 = \mathbf{D}_0 \cup \mathbf{D}_1$ and π_1 is a random permutation. It sends $\mathbf{T}_1$ to server, and $\textsc{Sim}_3$ receives back from the server the set $\mathbf{T}_1'$ that he evaluated. $\textsc{Sim}_3$ checks using K_1 that $\mathbf{T}_1'$ is of the form $\{F_{K_2}(s_{\pi_3(j)}) : s_j \in \mathbf{T}_1\}$ for some permutation π_3. If it is not, $\textsc{Sim}_3$ sends an abort message for both P_1 and P_2 to the ideal functionality. Then $\textsc{Sim}_3$ receives a permutation σ from the server and if $\sigma \neq \pi_3$, the simulator sends to the ideal functionality an abort message for P_1. The views in the real and the ideal executions are indistinguishable unless in the real execution the server manages to compute F_{K_2} incorrectly on a subset of values from $\mathbf{S}_1$, which means that he returns a set of incorrect PRP values in $\mathbf{T}_1$ that does not include any elements from $\mathbf{D}_0 \cup \mathbf{D}_1$, and includes all λ copies of at least one element in the intersection (the probability that the server adds values to the set of P_1 is negligible since it does not know K_1). However, the probability for this negligible as we saw in the proof for Theorem 3.2.

■

Theorem 3.7. *The protocol described in Fig. 4 is secure in* (1) *the presence of malicious P_1 and an honest server and P_2; and* (2) *a malicious P_2 and honest server and P_1.*

Proof sketch.
Malicious P_1. We construct a simulator $\textsc{Sim}_{P_1}$ who receives $n = |\mathbf{S}_1|$, $m = |\mathbf{S}_2|$ and simulates the protocol execution with P_1 as follows. It receives sets $\mathbf{D}_0$, $\mathbf{D}_1$, $\mathbf{D}_2$, checks whether they are disjoint subsets of $\mathcal{D} \neq \mathcal{U}$ and if not, sends an abort to P_1. The, it runs the coin tossing protocol with P_1 to choose a PRP key K_1. $\textsc{Sim}_{P_2}$ receives a set $\mathbf{T}_1$ from P_1 and sends back

$$\mathbf{T}_1' = \pi_3\Big(F_{K_2}(\mathbf{T}_1)\Big),$$

where π_3 is a random permutation. The simulator uses K_1 to extract the input set $\mathbf{S}_1$ underlying $\mathbf{T}_1$. $\textsc{Sim}_{P_2}$ sends $\mathbf{S}_1/(\mathbf{D}_0 \cup \mathbf{D}_1)$ to the ideal functionality $\mathcal{F}_{\mathsf{SPSI}}$ and receives back the intersection set J. $\textsc{Sim}_{P_1}$ constructs a random set $\mathbf{S}_2$ of size m that has intersection J with $\mathbf{S}_1$, and sends to P_1 the set

$$\mathbf{T}_2' = \pi_2\Big(F_{K_2}\Big(F_{K_1}\Big(\mathbf{S}_2^\lambda + \Delta_2\Big)\Big)\Big),$$

where $\Delta_2 = \mathbf{D}_0 \cup \mathbf{D}_2$. After P_1 returns the intersection I, $\textsc{Sim}_{P_2}$ checks that $\mathrm{I}^{-1} = F_{K_1}^{-1}(F_{K_2}^{-1}(\mathrm{I}))$ includes all elements from $\mathbf{D}_0$ and none of $\mathbf{D}_2$ and also λ copies for each of the d elements in J. If this check fails, $\textsc{Sim}_{P_1}$ sends an abort message to P_1. Otherwise, $\textsc{Sim}_{P_1}$ sends to P_1 the permutation π_3. The view in the real and the simulated execution are indistinguishable unless in the real execution P_1 manages to return an incorrect set intersection that passes the check for being of the correct form. As we saw in the proof of Theorem 3.2 this can happen only with negligible probability.

Malicious P_2. We construct a simulator $\textsc{Sim}_{P_2}$ who receives $n = |\mathbf{S}_1|$, $m = |\mathbf{S}_2|$ andinteracts with P_2 as follows. It generates sets $\mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_2 \subseteq \mathcal{D} \neq \mathcal{U}$ and sends them to P_2. It runs the coin tossing protocol with P_2 to establish keys K_1, K_2. Then, it receives the set $\mathbf{T}_2'$ that P_2 prepared and extracts the set of input values $\mathbf{S}_2$ using the keys K_1 and K_2 (any value x such that $F_{K_2}(F_{K_1}(x\|\alpha)) \in \mathbf{T}_2'$ for some $\alpha \leq \lambda$). $\textsc{Sim}_{P_2}$ sends $\mathbf{S}_2/(\mathbf{D}_0 \cup \mathbf{D}_2)$ to the ideal functionality $\mathcal{F}_{\mathsf{SPSI}}$ and receives back an intersection set J. $\textsc{Sim}_{P_2}$ computes

$$\mathbf{T}_1' = \Big(F_{K_2}\Big(F_{K_1}\Big(J^\lambda + \Delta_1\Big)\Big)\Big),$$

where $\Delta_1 = \mathbf{D}_0 \cup \mathbf{D}_1$ and returns to P_2 the set $\mathbf{T}_1' \cap \mathbf{T}_2'$. If $\textsc{Sim}_{P_2}$ receives from P_2 an abort message, it sends the ideal functionality abort for P_1. The views in the real and the ideal executions are indistinguishable since in both cases P_1 receives output only if P_2 does not send an abort message. If P_1 receives an output set, in both cases he receives the same intersection set that P_2 gets since the server is honest and sends the correct permutation π_3 to P_1.

■