Scaling Private Set Intersection to Billion-Element Sets

Seny Kamara[1], Payman Mohassel[2], Mariana Raykova[3], and Saeed Sadeghian[2]

[1] Microsoft Research
[2] University of Calgary
[3] SRI

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 [3], location-based services [58], and collaborative botnet detection [57].

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 [56,5], VIFF [19], Sharemind [6], Tasty [41], HEKM [42], VMCrypt [55], and SCAPI [26]. 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 [31,39,9,16,22,40,44,43].

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 [13]. A number of constructions try to achieve fairness for a specific class of functionalities [37], or consider limited (partial) notions of fairness instead [59,32,36].

SERVER-AIDED MPC. A promising approach to address these limitations is *server-aided* or *cloud-assisted* MPC.[4] In this 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 [7], 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 [34] with a proof system [4]; and the constructions based on Yao's garbled circuit technique [64], proposed by Feige, Killian and Naor [29] in the semi-honest model and recently formalized and extended to stronger models in [45] and optimized and implemented in [48]. This also includes special-purpose protocols such as server-aided private equality-testing [58,33].

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.

[4] An alternative approach considered in the PSI literature is the use of tamper-proof hardware in the design of private set intersection [38,30]. 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.

The work of [45] 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 (see full version for a more detailed discussion).

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.[5] 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 4.2 and 4.3).

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 4.4).

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 4.5.

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 [27] to implement our server-side set intersection protocol. In

[5] Due to space limitations we had to omit our security definitions and proofs. The full version of this work with definitions and proofs is available on request.

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 4.1) 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

The problem of PSI was introduced by Freedman, Nissim and Pinkas [31]. PSI has attracted a lot of attention and several protocols have been proposed with various levels of efficiency [51,9,39,44,17,40]. De Cristofaro and Tsudik presented the first PSI protocols with linear complexity [14,23,22]. Huang, Evans and Katz [43] proposed a protocol with $O(n \log n)$ complexity (where n is the size of the sets) based on secure two-party computation. While the Huang et al. protocol has larger complexity, experimental results [25] suggest it is competitive and even more efficient than DeCristofaro and Tsudik's protocol for large security parameters. In recent work, Dong, Chen and Wen propose a new two-party PSI protocol with linear complexity based on Bloom filters, secret sharing and oblivious transfer [25]. Though the Dong et al. protocol is linear, the underlying cryptographic operations mostly consist of symmetric-key operations. This results in the fastest two-party PSI protocol to date and is an order of magnitude faster than previous work.

As far as we know, the recent works of Dong, Chen, Camenisch and Russello [24] and of Kerschbaum [50] are the only other works that propose server-aided PSI protocols. Both protocols, however, assume a semi-honest server and require public-key operations (the latter even requires bilinear pairing operations) which prevent these protocols 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 [62,35,11,15,12,47,46,10]. In the full version of this work, [6] we provide a detailed comparison between these notions 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 [28,8,1,54] 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 [58,33,61] and, in particular, the work of [58] uses a server-aided PET (in a model similar to ours) as the main cryptographic component of their construction. While previous work [58,33,61] 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.

[6] The full version is available upon request.

2 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[7] 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 $F : \mathcal{U} \rightarrow \mathcal{V}$ 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 $\pi : [|\mathbf{S}|] \rightarrow [|\mathbf{S}|]$ 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.

2.1 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 1 whose proof is omitted due to lack of space.

Theorem 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.*

[7] 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.

Setup and inputs: Let $F : \{0,1\}^k \times \mathcal{U} \rightarrow \{0,1\}^{\geq k}$ 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 $F_K^{-1}(I)$.

Fig. 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.

2.2 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 $\{x\|1, \ldots, x\|\lambda\}$ 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 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 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 2 and 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 [53,49]. 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 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. *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 $\lambda = 2$.

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.

2.3 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

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 $\mathbf{D}_0 \not\subset F_K^{-1}(\mathrm{I})$ 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
$$\bigg(F_K^{-1}(\mathrm{I}) - \mathbf{D}_0\bigg)^{-\lambda}.$$

Fig. 2. A Server-aided PSI protocol with malicious security

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.

Theorem 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.*

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 $\mathbf{D}_0 \not\subset F^{-1}_{K_2}(\mathrm{I})$ or $\mathbf{D}_i \cap F^{-1}_{K_2}(\mathrm{I}) \neq \emptyset$
    (b) there exists $x \in \mathbf{S}_i$ and $\alpha, \beta \in [\lambda]$ such that $F_{k_1}(x)\|\alpha \in F^{-1}_{K_2}(\mathrm{I})$ and $F_{k_1}(x)\|\beta \notin F^{-1}_{K_2}(\mathrm{I})$
11. each party computes and outputs the set
$$F^{-1}_{K_1}\bigg(\big(F^{-1}_{K_2}(\mathrm{I}) - \mathbf{D}_0\big)\bigg)^{-\lambda}.$$

Fig. 3. A fair server-aided PSI protocol

Theorem 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.

2.4 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 6 and 7 whose proof is omitted due to lack of space.

Theorem 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 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 $\lambda|S_1| + 2t$.

3 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 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.

3.1 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.

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:
$$\mathbf{T}_1 = \pi_1\bigg(F_{K_1}\Big(\mathbf{S}_1^{\lambda} + \Delta_1\Big)\bigg)$$
6. The server returns to P_1:
$$\mathbf{T}_1' = \pi_3\bigg(F_{K_2}(\mathbf{T}_1)\bigg),$$
where π_3 is a random permutation
7. P_2 sends
$$\mathbf{T}_2' = \pi_2\bigg(F_{K_2}\Big(F_{K_1}\Big(\mathbf{S}_2^{\lambda} + \Delta_2\Big)\Big)\bigg)$$
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 $\mathbf{D}_0 \not\subset \mathbf{I}^{-1}$ or $\mathbf{D}_i \cap \mathbf{I}^{-1} \neq \emptyset$
    (b) there exists $x \in \mathbf{S}_i$ and $\alpha, \beta \in [\lambda]$ such that $x\|\alpha \in \mathbf{I}^{-1}$ and $x\|\beta \notin \mathbf{I}^{-1}$ 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
$$\bigg(\mathbf{I}^{-1} - \mathbf{D}_0\bigg)^{-\lambda}.$$

Fig. 4. An intersection size-hiding server-aided PSI

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 [52]. Let $\mathbf{S} \subset \mathcal{U}$ be a set and $\mathtt{A}$ be an array of size $|\mathbf{S}|$ that stores each element of $\mathbf{S}$. To randomly permute $\mathbf{S}$, for all items $\mathtt{A}[i]$, we generate an index $j \leq [|\mathbf{S}|]$ uniformly at random and swap $\mathtt{A}[i]$ with $\mathtt{A}[j]$. We sampled the random j by applying AES to $\mathtt{A}[i]$ 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.

3.2 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 [27]. 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 [60] 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 [63] 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.

3.3 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.

3.4 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.

4 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.

4.1 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 4.1. 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.

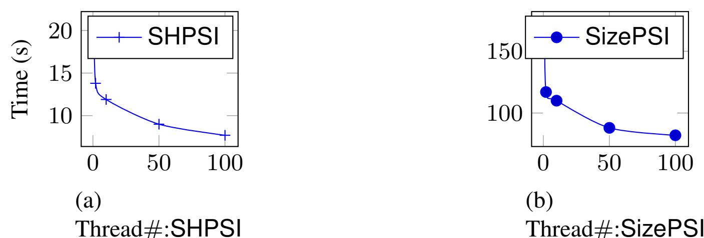

Fig. 5. Effect of multi-threading on the runtime of our protocols. Set size is 10 Million.

4.2 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.

4.3 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 pro-

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.

tocol [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).

4.4 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.

4.5 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	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.

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.

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. 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.
4. B. Barak and O. Goldreich. Universal arguments and their applications. In *CCC*, 2002.
5. A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *CCS*, 2008.
6. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, 2008.
7. 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.
8. 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.
9. Jan Camenisch and Gregory Zaverucha. Private intersection of certified sets. *FC*, pages 108–127, 2009.
10. 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.
11. Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS*, pages 442–455, 2005.
12. 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.

13. R. Cleve. Limits on the security of coin flips when half the processors are faulty. In *STOC*, pages 364–369, 1986.
14. Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography*, pages 143–159, 2010.
15. R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *ACM CCS*, pages 79–88, 2006.
16. Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Moti Yung. Efficient robust private set intersection. In *ACNS*. Springer, 2009.
17. Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Moti Yung. Secure efficient multiparty computing of multivariate polynomials and applications. In *ACNS*, pages 130–146. 2011.
18. Wei Dai. Crypto++ library. http://www.cryptopp.com/, 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.
25. Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In *ACM CCS*, pages 789–800, 2013.
26. Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. Scapi: The secure computation application programming interface, 2012.
27. Donovan Hide et al. Sparsehash library. https://code.google.com/p/sparsehash/, 2013. [Online; accessed 08-May-2013].
28. Ronald Fagin, Moni Naor, and Peter Winkler. Comparing information without leaking it. *Communications of the ACM*, 39(5):77–85, 1996.
29. U. Feige, J. Killian, and M. Naor. A minimal model for secure computation (extended abstract). In *STOC*, 1994.
30. Marc Fischlin, Benny Pinkas, Ahmad-Reza Sadeghi, Thomas Schneider, and Ivan Visconti. Secure set intersection with untrusted hardware tokens. In *CT-RSA*, pages 1–16. 2011.
31. Michael Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *EUROCRYPT 2004*, pages 1–19. Springer, 2004.
32. J. Garay, P. MacKenzie, M. Prabhakaran, and K. Yang. Resource fairness and composability of cryptographic protocols. *Theory of Cryptography*, pages 404–428, 2006.
33. Ran Gelles, Rafail Ostrovsky, and Kina Winoto. Multiparty proximity testing with dishonest majority from equality testing. In *Automata, Languages, and Programming*. 2012.
34. C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.
35. E-J. Goh. Secure indexes. Technical Report 2003/216, IACR ePrint Cryptography Archive, 2003. See http://eprint.iacr.org/2003/216.
36. S. Gordon and J. Katz. Partial fairness in secure two-party computation. *EUROCRYPT*, pages 157–176, 2010.
37. 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.
38. Carmit Hazay and Yehuda Lindell. Constructions of truly practical secure protocols using standardsmartcards. In *CCS*, pages 491–500, 2008.

39. Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *TCC*, 2008.
40. Carmit Hazay and Kobbi Nissim. Efficient set operations in the presence of malicious adversaries. *Public Key Cryptography–PKC 2010*, pages 312–331, 2010.
41. W. Henecka, S. Kogl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS*, 2010.
42. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.
43. Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.
44. Stanislaw Jarecki and Xiaomin Liu. Fast secure computation of set intersection. *SCN*, pages 418–435, 2010.
45. S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party comptuation. Technical Report 2011/272, IACR ePrint Cryptography Archive, 2011.
46. S. Kamara and C. Papamanthou. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security (FC '13)*, 2013.
47. S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *ACM Conference on Computer and Communications Security (CCS '12)*. ACM Press, 2012.
48. Seny Kamara, Payman Mohassel, and Ben Riva. Salus: a system for server-aided secure function evaluation. In *CCS*, pages 797–808, 2012.
49. J. Katz, R. Ostrovsky, and A. Smith. Round efficiency of multi-party computation with a dishonest majority. In *EUROCRYPT*, 2003.
50. F. Kerschbaum. Outsourcing private set intersection using homomorphic encryption. In *Asia CCS '12*, 2012.
51. Lea Kissner and Dawn Song. Privacy-preserving set operations. In *CRYPTO*, pages 241–257. 2005.
52. Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
53. Y. Lindell. Parallel coin-tossing and constant-round secure two-party computation. In *CRYPTO*, 2001.
54. Helger Lipmaa. Verifiable homomorphic oblivious transfer and private equality test. In *ASIACRYPT*. 2003.
55. L. Malka. Vmcrypt: modular software architecture for scalable secure computation. In *CCS*, 2011.
56. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *USENIX Security*, 2004.
57. Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. Botgrep: finding p2p bots with structured graph analysis. In *USENIX Security*, 2010.
58. Arvind Narayanan, Narendran Thiagarajan, Mugdha Lakhani, Michael Hamburg, and Dan Boneh. Location privacy via private proximity testing. In *NDSS*, 2011.
59. B. Pinkas. Fair secure two-party computation. *Eurocrypt*, pages 647–647, 2003.
60. Henry Rawas. Redis windows port. https://github.com/MSOpenTech/redis, 2013. [Online; accessed 08-May-2013].
61. Gokay Saldamli, Richard Chow, Hongxia Jin, and Bart Knijnenburg. Private proximity testing with an untrusted server. In *SIGSAC*, pages 113–118, 2013.
62. D. Song, D. Wagner, and A. Perrig. Practical techniques for searching on encrypted data. In *IEEE S&P*, pages 44–55, 2000.
63. Tang Yaguang. hiredis win32. https://github.com/texnician/hiredis-win32, 2013. [Online; accessed 08-May-2013].
64. A. Yao. Protocols for secure computations. In *FOCS*, 1982.