Enabling Two-Party Secure Computation on Set Intersection *

Ferhat Karakoç[†1] and Alptekin Küpçü[‡2]

[1]Ericsson Research, İstanbul, Türkiye
[2]Koç University, İstanbul, Türkiye

Abstract

In this paper, we propose the first linear two-party secure-computation private set intersection (PSI) protocol, in the semi-honest adversary model, computing the following functionality. One of the parties (P_X) inputs a set of items $X = \{x_j \mid 1 \leq j \leq n_X\}$, whereas the other party (P_Y) inputs a set of items $Y = \{y_i \mid 1 \leq i \leq n_Y\}$ and a set of corresponding data pairs $D_Y = \{(d_i^0, d_i^1) \mid 1 \leq i \leq n_Y\}$ having the same cardinality with Y. While P_Y outputs nothing, P_X outputs a set of data $D_X = \{d_i^{b_i} \mid b_i = 1 \text{ if } y_i \in X, b_i = 0 \text{ otherwise}\}$. This functionality is generally required when the PSI protocol is used as a part of a larger secure two-party computation such as threshold PSI or any function of the intersection in general. In literature, there are linear circuit and secure-computation PSI proposals, such as Pinkas et al. PSI protocol (Eurocrypt 2019), our PSI protocol (CANS 2020) and Chandran et al. PSI protocol (PETS 2022), for similar functionalities but having a cuckoo table mapping in the functionality, which complicates the application of different secure computation techniques on top of the output of the PSI protocol. We also show that the idea in the construction of our secure-computation PSI protocol having the functionality mentioned above can be utilized to convert the existing circuit PSI and secure-computation PSI protocols into the protocols realizing the functionality not having the cuckoo table mapping. We provide this conversion method as a separate protocol, which is one of the main contributions of this work. While creating the protocol, as a side contribution, we provide a one-time batch oblivious programmable pseudo-random function based on garbled Bloom filters.

Keywords— Private set intersection, two-party computation, Bloom filters, oblivious transfer, cuckoo hashing, circuit-PSI, OPPRF

1 Introduction

Private set intersection (PSI) protocols are one of the commonly used two party secure communication primitives, where two parties, P_X and P_Y, have their own respective

*This is the full version of the paper (1) presented at CANS 2020 and (2) in IACR ePrint archive.

[†]ferhat.karakoc@ericsson.com
[‡]akupcu@ku.edu.tr

private sets, X and Y, and at least one of the parties learn the intersection $X \cap Y$ but nothing more. Since it fits very well into a real world problem and finds many application areas such as health genome testing, online advertising, and discovery of contact lists, considerable amount of custom PSI protocols have been proposed in the literature. Another recent PSI use case is the Cyber-Physical-Social Systems (3), which is an extension of Cyber-Physical Systems (4). The studies (5; 6) utilizes PSI to have a privacy-preserving profile matching scheme in this use case. In some use cases, instead of having the set intersection itself, a function of it is required. However, revealing the intersection to at least one of the parties makes the PSI protocol not usable as a building block in a larger secure computation protocol, because in that larger protocol, intermediate information would leak due to the nature of the employed PSI protocol.

In this work, we focus on designing a PSI protocol in the semi-honest security model, which allows P_Y obliviously to send data to P_X, where neither P_X nor P_Y know the choice bits, which depend on the intersection. This type of PSI protocols can be called *PSI with bi-oblivious data transfer*. This functionality allows the realization of *secure-computation PSI protocols* which are usable as a building block in larger secure computation protocols. *Secure-computation PSI protocols* include the functionality of *circuit PSI protocols* which is defined as outputting secret shares of the intersection to the parties (7). More precisely, in the *PSI with bi-oblivious data transfer*, P_Y inputs a set of data pairs $D_Y = \{(d_i^0, d_i^1) \mid 1 \leq i \leq n_Y\}$ in addition to a set of items $Y = \{y_i \mid 1 \leq i \leq n_Y\}$ as usual, and P_X inputs a set of items $X = \{x_i \mid 1 \leq i \leq n_X\}$. While P_Y outputs nothing, P_X outputs a set of data $D_X = \{d_i^{b_i} \mid 1 \leq i \leq n_Y, b_i \in \{0,1\}\}$ where $b_i = 1$ if $y_i \in X$, $b_i = 0$ otherwise. With different choices for (d_i^0, d_i^1), different functionalities can be realized. For example, when each $(d_i^0, d_i^1) = (0, 1)$, we obtain regular PSI. When (d_i^0, d_i^1) is a pair of two strings, we obtain PSI with data transfer (8; 9). To see how our protocol can be utilized for cardinality computation, consider d_i^0 and d_i^1 respectively as additively-homomorphic encryption of '0' and '1' ($E_k(0)$ and $E_k(1)$ for key k picked by P_Y), respectively, and that our protocol is followed by additively-homomorphic evaluation of the obtained values by P_X, and then P_Y decrypts the result. This corresponds to PSI cardinality. Alternatively, d_i^0 and d_i^1 output values can be secret shares of the membership result for each item of Y, or labels for the corresponding input wires for garbled-circuit-based secure computation protocols. For example, the value d_i^0 can be a wire label corresponding to zero for wire i and d_i^1 can be wire label corresponding to one for wire i. This way, after our protocol concludes, the parties have the wire labels corresponding to the intersection, without knowing the intersection. The computation can then continue, for example, computing a threshold over the intersection cardinality, or any other secure two-party computation protocol whose input should be the intersection. More applications and details are given in Section 6.

Related Work: To the best of our knowledge, protocols that output a function of the membership results were proposed by Ciampi and Orlandi (10), Pinkas et al. (11), Falk et al. (12), ourselves (1), and Chandran et al. (7) in addition to the circuit based solutions of (13; 14). Among these protocols, only the protocols proposed in (10; 1) allows application of any secure computation protocol on the output of the PSI protocol by providing different kinds of outputs, while the others allow only learning the secret shares of the items in the intersection. This type of protocols that output secret shares is called circuit PSI protocols. We name the PSI protocols allowing execution of any the of secure computation protocols *secure-computation set intersection protocols* or *secure-computation PSI protocols*.

In (10), a custom private set membership protocol (PSM) (where one of the parties has only one item instead of a set) based on oblivious navigation of a graph was introduced, and this PSM protocol was converted to a PSI protocol with $O(n \log n / \log \log n)$ communication and computation complexities using the hashing tech-

niques proposed in (15; 16; 14), where n is the number of items in the sets. (12) has a communication complexity of $O(n \log\log n)$ when the output can be secret shared. In (11), Pinkas et al. proposed a PSI protocol with $O(n)$ communication and $\omega(n(\log\log n)^2)$ computation using the oblivious programmable pseudo-random function (OPPRF) in (17). That protocol uses OPPRF to check the private set membership relation in the hashed bins, where the result is not output in clear text, and then deploys a comparison circuit for the output of the membership result that can be given to a function as the input. Our work in (1) followed a similar approach with the circuit-PSI protocol of (11) but utilized garbled Bloom filters to have the first protocol with linear communication and computation complexity. (7) also followed the idea of (11) but in a different way from (1) to have another linear-cost secure-computation PSI protocol. Note that, to the best of our knowledge, the existing linear-cost circuit-PSI (11; 7) and secure-computation PSI protocols (1) have a mapping such that $f : \{1, 2, ..., n\} \rightarrow \{1, 2, ..., \beta\}$ which maps the elements of one of the party to a cuckoo table, where n is the number of items in the set of the party and β is the size of the cuckoo table. The reason why these protocols have this mapping in their functionality is the usage of cuckoo table in their constructions. Having such a mapping reduces the feasibility of usage of these PSI protocols in larger secure computation protocols because the output of the PSI protocol becomes a function of the cuckoo table not the set of the party.

Also in literature, there have been special purpose PSI protocols such as (18; 19; 20; 21; 22; 23; 24; 25; 26; 27), which output a specific function of the intersection such as cardinality of the set, intersection-sum, or a threshold function.

Our Contribution: In this work, we convert our linear secure-computation PSI protocol introduced in (1) to a secure-computation PSI protocol having natural functionality, which does not include any other parameters except the sets to be intersected and the sizes of the sets, by removing the cuckoo table mapping from the functionality, so that the new protocol can be easily utilized to build larger secure computation protocols using the PSI protocol as a primitive. We then generalize the conversion idea to make it applicable to other linear-cost circuit PSI protocols to remove the cuckoo table mapping from their functionalities as well. While improving our protocol in (1), we continue to follow the idea of Pinkas et al. (11) in that we first run a PSM protocol for each bin in the cuckoo hash table and then execute a comparison protocol, but we diverge from their idea in the following ways. The first one is that we construct a Bloom-filter (BF) based PSM protocol by modifying Dong et al. PSI solution (28) to reduce the computation complexity. The second point is that, instead of using a comparison circuit, we execute Ciampi-Orlandi PSM protocol as a secure equality testing protocol such as the one used in (29), which makes the equality testing free by using the base oblivious transfer already executed in the BF-based PSM protocol. Following these two methods along the idea of Pinkas et al., we are able construct the first custom PSI protocol having linear computation and communication complexities in the number of items for the functionality we consider (outputting not the result set, but a function of the membership results), to the best of our knowledge.

Overview of our conversion protocol: To remove the cuckoo table mapping from the protocol functionality, we incorporate additional steps that use additively homomorphic encryption. P_Y obliviously learns additively homomorphic encryption of '0' or '1' under the key of P_X for each bin of the cuckoo table depending on the bin-wise membership, and then P_Y obliviously sends the d_i^0 or d_i^1 for only the bins where the items of P_Y were located. We provide a new protocol consisting of these new steps, which can also be utilized for other circuit-PSI protocols to remove the weird cuckoo table mapping from their functionality. The overhead of our conversion protocol (to convert cuckoo table mapping into natural PSI functionality) does not increase the asymptotic complexity of the underlying circuit PSI protocol, i.e., keeps the complexity linear.

Scenarios that our conversion protocol fits best: Since the complexity of our conversion protocol depends on the number of items in Y, its concrete overhead is light when the size of Y is less than the size of X. For example, when $n_X = 2^{16}$ and $n_Y = 2^6$, our overhead over the best concrete complexity circuit PSI in a wide-are network setting is only 6%. As the size of X increases, the relative overhead decreases even further. This case is mainly seen in applications where one of the parties P_Y is a mobile device while the other party P_X is a server. Private contact discovery problem could be given as an example use case (30). More concretely, a user of a messaging application would like to discover his/her contacts who use the messaging application, without revealing the information of other contacts to the messaging application server. In this case, one party (i.e., the user) has a set containing a few hundred items, while the other party (i.e., the messaging server) could have a set containing a million items (30).

Summary of our contributions including our preliminary publication (1):

- We provide a generic conversion protocol that converts circuit PSI and secure-computation PSI protocols having the cuckoo table mapping functionality into secure-computation PSI protocols having the natural PSI functionality.
- We provide the first linear-cost secure-computation PSI protocol with natural functionality by applying the conversion protocol on our PSI protocol introduced in (1).
- We show that the concrete overhead of the conversion protocol is light especially when one of the parties has much fewer number of items than the other party, by comparing our results with the performance of the protocol of (7), which has the best concrete performance results so far. For example, when $n_X = 2^{16}$ and $n_Y = 2^6$, the overhead of our solution becomes only 6%).
- As already presented in (1), we introduce a batch OPPRF protocol that allows us to construct secure-computation PSI protocols with linear communication and computation complexity.

The following items summarize the new contributions compared to the original CANS 2020 paper (1).

- A new conversion protocol (in Section 5) that can be utilized to convert secure computation PSI protocols including cuckoo table mapping into secure computation PSI protocols having the natural functionality.
- An improved version of our protocol (in Section 5.3) presented in the CANS 2020 paper, having the natural PSI functionality.
- An improved version of Chandran et al. circuit PSI protocol (7), having the natural PSI functionality (in Section 5.4).
- Utilization of the secure-computation PSI protocol with natural functionality to have different types of outputs (in Section 6).

2 Preliminaries and Similar Protocols

Notation: P_X and P_Y are the parties who run the protocol, X and Y are the corresponding item sets of the parties, D_Y is the set of message pairs inputted by P_Y, and D_X is the set of corresponding received messages by P_X depending on the intersection $X \cap Y$.

The remaining notation we use throughout the paper is as follows:

ℓ	:	The length of the items in the sets
κ	:	Security parameter
η	:	Statistical correctness parameter
n_X	:	The number of items in X
n_Y	:	The number of items in Y
n	:	$max(n_X, n_Y)$
m	:	Bloom filter size
n_h	:	Number of hash functions used in Bloom filter
H_i	:	Set of k hash functions used in the construction of Bloom filters for i-th bin in the cuckoo table where $H_i = \{h_{i,1}, ..., h_{i,k}\}$
β	:	The number of bins in cuckoo table

2.1 Sub-Protocols

Oblivious Transfer: A 1-out-of-2 oblivious transfer (OT) (31) is a secure two-party protocol that realizes Functionality 1.

OT is one of the commonly used primitives in secure protocols and considerable amount of studies on it have been seen in literature, such as (32). To reduce the number of asymmetric key operation executions, OT extension (OTE) method was proposed in (33) and practically realized with some studies such as (34). To execute 1-out-of-2 OT for m pairs of length ℓ (OT_ℓ^m) it is enough to run OT_κ^κ, called base OTs, where κ is the security parameter, which keeps the number of heavy public key operations as a constant independent from the number of pairs m and item lengths ℓ.

In recent works, it was shown that the number of rounds can be 2 instead of 3 for an OT extension protocol by executing some of the computations in the offline phase of the protocol (35; 36). In our solution, we don't consider the preprocessing operations and so we don't use these constructions in our protocols.

Cuckoo hashing (37) is a hashing primitive that allows to map items of a set to bins, where there is at most one item in each bin. This primitive employs two hash functions h_0 and h_1 and maps n items to a table T of $(1 + \epsilon)n$ bins. An item x_i is inserted into bin $T[h_b(x_i)]$. If this bin already accommodates a previous item x_j, then x_j is relocated to bin $T[h_{1-b}(x_j)]$. If in that bin there is another item, then this procedure is repeated until there is no need or a replacement threshold is reached. If a threshold is employed, then a stash is used to store the items that are not located into the bins.

Bloom Filter Based PSI: A Bloom filter (BF) (38) is a representation of a set $X = x_1, ..., x_n$ of n elements using an m-bit string BF. BF is constructed with the help of a set of n_h independent and uniform hash functions ($H = h_1, ..., h_{n_h}$) where $h_i : \{0,1\}^\ell \rightarrow \{1, 2, ..., m\}$ as follows: BF is first set to 0^m. Then, for each item in X, $BF[h_i(x_j)]$ is set to 1 where $1 \leq i \leq n_h$ and $1 \leq j \leq n$. To check whether an item x is in the set X, one checks $BF[h_i(x)]$ is equal to 1 or not for each i ($1 \leq i \leq n_h$). If for all i ($1 \leq i \leq n_h$) the corresponding bit in BF is equal to 1, then it means that the item is probably in the set. Otherwise (for some i the corresponding bit is 0), the item is not in the set.

A Bloom filter based PSI was proposed by Dong et al. (28). In that solution, a variant of BF called Garbled Bloom Filter (GBF) was used. A GBF of a set X,

---

Functionality 1 Oblivious Transfer

Inputs. The sender inputs a pair (x^0, x^1), the receiver inputs a choice bit $b \in \{0, 1\}$.

Outputs. The functionality returns the message x^b to the receiver and returns nothing to the sender.

GBF, is similar to BF except that while for each hash function h_i in H we have $BF[h_i(x)] = 1$, $GBF[h_i(x)]$ is a secret share of x: that is,

$$\bigoplus_{i=1}^{n_h} GBF[h_i(x)] = x$$

and other cells are random values instead of simple zeros. In the first step of the protocol, P_1 and P_2 construct a GBF (GBF_X) using the GBF building algorithm provided in (28) and a BF (BF_Y), respectively. Then, P_1 and P_2 run m-pair oblivious transfer of ℓ-bit strings (OT_ℓ^m) where P_1's input is $(0^\ell, GBF_X[i])$ and P_2's input is $BF_Y[i]$ for the i-th OT, and the output of P_2 is $GBF_Y[i]$. In this way, P_2 learns $GBF_X[i]$ if $BF_Y[i] = 1$. P_2 checks, for each item $y_j \in Y$, whether it is in X or not, by comparing

$$\bigoplus_{i=1}^{n_h} GBF_Y[h_i(y_j)] \stackrel{?}{=} y_j.$$

Oblivious Pseudo-Random Function Based PSM: An oblivious pseudo-random function (OPRF), introduced in (39), is a two-party protocol where party P_1 holds a key K, party P_2 holds a string x, and at the end of the protocol P_1 learns nothing, while P_2 learns $F_K(x)$ where F is a pseudo-random function family that gets a κ-bit key K and an ℓ-bit input string x and outputs an ℓ-bit random-looking result. An oblivious programmable pseudo-random function (OPPRF) (17) is similar to an OPRF except that in OPPRF, the protocol outputs predefined values for some of the programmed inputs. In that protocol P_2 should not be able to distinguish which inputs are programmed. Note that OPPRF is very similar to PSI with data transfer (8; 9) by just setting the data of the latter to random values. Indeed, the GBF-based construction of OPPRF in (17) is essentially the GBF-based construction in (9). In this paper, we extend this GBF-based construction to batch OPPRF.

The basic idea in OPRF based PSM protocols is as follows. P_1 holds a key K to compute a pseudo-random function F_K, P_2 learns $F_K(y)$ for his item y obliviously, and P_1 sends $F_K(x_i)$ for her items $x_i \in X$ to P_2. P_2 checks if $F_K(y)$ is in the set $\{F_K(x_i)\}$. An example PSI protocol can be found in (14). In the OPRF solution, P_2 learns whether or not his item is in the set of P_1. This solution cannot be used in our setting where nobody learns the result in cleartext and the parties only learn a function result of the intersection. Pinkas et al. (11) converted the OPRF solution to the setting we consider using an oblivious programmable pseudo-random function. In that solution, P_1 sends the same (random) output r for the items in her set. Otherwise, she sends some random output to P_2. Then P_1 and P_2 run a circuit to check the equality of r and the outputs P_1 sent to P_2. At the end of this equality check circuit, one party obtains a function based on the result of the equality, i.e, of the membership.

Usage of Ciampi-Orlandi PSM Protocol to Test Equality of Two Strings: The private set membership (PSM) protocol proposed by Ciampi and Orlandi (10) works in the setting that P_1 and P_2's inputs are a set of items X and an item y, respectively, and at the end of the protocol, P_2 learns a function of the membership relation and P_1 learns nothing. The protocol is based on oblivious graph tracing and uses oblivious transfer. In our construction, we use that protocol for the case that P_1's input is just one item instead of a set, as considered in (29). In this case, the PSM protocol becomes a secure equality testing outputting a function (we call functional equality testing - FEQT) protocol that realizes Functionality 2. This simplification also greatly increases efficiency, helping us achieve linear costs. Protocol 1 presents the steps of Ciampi-Orlandi PSM protocol for the case of testing two strings as used in (29).

Functionality 2 Functional Secure Equality Testing

Inputs. P_1 inputs x and a pair of strings (d_0, d_1), P_2 inputs y.

Outputs. The functionality checks the equality of x and y and returns d_0 or d_1 according to the truth value of $x \stackrel{?}{=} y$ to P_2.

Protocol 1 Ciampi-Orlandi PSM Protocol to test equality of two strings

Parameters. $E_k(.)$ is a symmetric encryption under the key k with a polynomial-time verification algorithm outputting whether a given ciphertext is in the range of $E_k(.)$ with false positive probability being $2^{-\eta}$.
Inputs. P_1 inputs x and a string pair (d_0, d_1), P_2 inputs y.

Outputs. P_2 outputs d_0 or d_1 according to the truth value of $x \stackrel{?}{=} y$. P_1 outputs nothing.
The protocol steps:

1. P_1 prepares the message pairs (s_0^i, s_1^i) for $x[i]$ $(1 < i < \ell)$ as follows: ($x[i]$ denotes the i-th bit of x and $x[1]$ is the right-most bit)
   - chooses random symmetric keys k_ℓ and k_ℓ^* and sets $s_{x[\ell]}^\ell = k_\ell$ and $s_{1-x[\ell]}^\ell = k_\ell^*$
   - For $i = (\ell - 1)$ to 1
     - chooses random symmetric keys k_i and k_i^* and sets $s_{x[i]}^i = \{E_{k_{i+1}}(k_i), E_{k_{i+1}^*}(k_i^*)\}$ and $s_{1-x[i]}^i = \{E_{k_{i+1}}(k_i^*), E_{k_{i+1}^*}(k_i^*)\}$.
     - permutes the ciphertexts in $s_{x[i]}^i$ and $s_{1-x[i]}^i$ randomly.
2. P_1 sends $E_{k_1}(d_1)$ and $E_{k_1^*}(d_0)$ to P_2 in random order.
3. P_2 learns corresponding $s_{y[i]}^i$'s by running OT from P_1 for $1 < i < \ell$.
4. P_2 recovers only one of the keys k_1 or k_1^* by decrypting the ciphertexts in the following way:
   - decrypts the ciphertexts in $s_{y[\ell-1]}^{\ell-1}$ using $s_{y[\ell]}^\ell$ as the key where the plaintext in the encryption domain is the key that will be used to decrypt the ciphertexts in $s_{y[\ell-2]}^{\ell-2}$.
   - decrypts the ciphertexts in $s_{y[i]}^i$ using the plaintext recovered from $s_{y[i+1]}^{i+1}$ as the key to recover the key used in the next received message $s_{y[i-1]}^{i-1}$.
5. P_2 decrypts the ciphertexts $E_{k_1}(d_1)$ and $E_{k_1^*}(d_0)$ using the key recovered in Step 4 where only one of the plaintexts will be in the domain and this plantext will be equal to d_1 or d_0. P_2 outputs the result.

2.2 Security Definitions

Since there are two parties who run the protocol, it is enough to prove that the protocol is secure when one of the parties is corrupted. There are two possible cases: either P_1

or P_2 is corrupted.

We follow the simulation-based security proof paradigm. Since we only consider honest-but-curious adversaries, the existence of a simulation in the "ideal world" whose protocol transcript is computationally indistinguishable from the adversary's view in the protocol execution in the "real world" (together with the parties' outputs in both worlds) proves that the protocol is secure. The basic idea in this proof paradigm is that if it is possible for the simulator to create a protocol transcript indistinguishable from the real execution transcript, then the transcript doesn't reveal any piece of information about the private input of the honest party. This security proof paradigm was formalized in (40) as follows. Protocol π implements the functionality $\mathcal{F} = (\mathcal{F}_1, \mathcal{F}_2)$ where the output of P_1 and P_2 are $\mathcal{F}_1(x, y)$ and $\mathcal{F}_2(x, y)$, respectively, and x and y are the inputs of the parties. The view of P_i for $i \in \{1, 2\}$ (denoted as $view_i^{\pi}(x, y)$) in the execution of the protocol π is the input of P_i, the internal random number coin tosses, the messages received from the other party in the execution of the protocol, and the outputs. The existence of probabilistic polynomial-time (PPT) algorithms S_i (the simulators) that takes the input of P_i and the output of P_i such that

$$\{S_i(w_i, \mathcal{F}_i(x, y))\}_{x,y} \approx \{view_i^{\pi}(x, y)\}_{x,y}$$

for $i \in \{1, 2\}$ where $w_1 = x$ and $w_2 = y$ proves that the protocol π realizes the functionality $\mathcal{F}$ securely.

As for the underlying primitives, namely OT and FEQT, whose functionalities were presented as Functionalities 1 and 2, respectively, there exists simulators who can simulate the view for both parties. These simulators take the input and output of the corresponding party as input, and produce indistinguishable views as output. In our proofs, we make use of these simulators for the underlying primitives.

Lastly, in our proofs, we provide simulators that create simulated views (including the outputs) which are indistinguishable from the real views. In all our proofs, this is either obvious (directly comes from the security of the underlying primitive, or comes from the fact that the simulated values are picked from the same distribution as the original ones), or were proven by others (in which case we also cite those papers).

3 Bloom Filter Based OPPRF Construction

We present a one-time OPPRF construction based on PSI protocols proposed in (28) and (9). For our usage, we put secret shares of random values chosen by the sender as the data to be transferred by the PSI protocol (9).

The OPPRF functionality we use in our PSM protocol is given in Functionality 3 and our construction that implements the functionality is presented in Protocol 2. The probability of false negative is zero because when $y \in X$, P_2 learns all shares required to recover the related programmed value. There may be false positives only with probability that is negligible in n_h and η, where n_h is the number of hash functions used in GBF construction and η is the minimum bit length of each cell in GBF, as shown in (28). Note that we use only one programmed value (t). Because of that, the functionality is secure only if the receiver makes only one query. For the purposes of PSM, we notice that one query is enough.

Functionality 3 (One-Time) Oblivious Programmable Pseudo Random Function

---

Inputs. P_1 inputs predefined items $X = \{x_1, ..., x_{n_X}\}$ and a programmed value t, P_2 inputs y.

Outputs. The functionality returns t to P_2 if $y \in X$; otherwise returns a random value to P_2, and returns nothing to P_1.

Protocol 2 Our One-Time OPPRF Protocol

Parameters. A set of hash functions $H = \{h_1, ..., h_{n_h}\}$

Inputs. P_1 inputs a set of items $X = \{x_1, ..., x_{n_X}\}$ and a programmed value t, P_2 inputs an item y.
Outputs. P_1 outputs nothing. P_2 outputs t if $y \in X$, otherwise outputs a random value.
The protocol steps:

1. P_1 constructs a garbled Bloom filter GBF_X having $\max(\eta, \ell)$-bit strings in each cell such that

$$\bigoplus_{i=1}^{n_h} GBF_X[h_i(x_j)] = t$$

   for $1 \leq j \leq n_X$.

2. P_2 constructs a (standard) Bloom filter BF_y for the item y.

3. P_1 and P_2 run m oblivious transfers where P_1's input is $(0, GBF_X[i])$ and P_2's input is $BF_y[i]$ for the i-th oblivious transfer, and the output of P_2 is 0 if $BF_y[i] = 0$ or $GBF_X[i]$ if $BF_y[i] = 1$. Call the output of P_2 as $GBF_y[i]$.

4. P_1 outputs nothing and P_2 outputs $\bigoplus_{i=1}^{n_h} GBF_y[h_i(y)]$.

Asymptotic Complexity. Since the number of hash functions used in the construction of Bloom filters is a constant related to the statistical correctness parameter that is independent of the number of items, Protocol 2 requires $O(n)$ hash function computations for the construction of the garbled Bloom filter in Step 1. Also, the size of the Bloom filters is $m = O(n_X)$, which makes the total asymptotic complexity of running oblivious transfers in Step 3 $O(n_X)$. Step 2 requires $O(n_X)$ non-cryptographic computation and space. Considering the complexity of Step 4 as $O(1)$, we conclude that the OPPRF protocol has a communication, computation, and space complexity of $O(n_X)$.

Theorem 1. *Protocol 2 securely realizes Functionality 3 when P_1 is corrupted by a semi-honest adversary $\mathcal{A}$, assuming that the OT protocol is semi-honest secure.*

Proof. The input set X and the programmed value t are given to the simulator S. The simulator computes a garbled Bloom filter GBF_X using its random tape such that

$$\bigoplus_{i}^{n_h} GBF_X[h_i(x_j)] = t$$

for $1 \leq j \leq n_X$. S runs the simulator of OT as the sender m times, where for the i-th run, the input of the simulator is $((0, GBF_X[i]), \perp)$. Here, $(0, GBF_X[i])$ is the input of the sender in the OT protocol and there is no output of the sender. Thus, the simulated view and output of the parties, and the view of the adversary in the real execution of the protocol and the output of the parties are indistinguishable. □

Theorem 2. *Protocol 2 securely realizes Functionality 3 when P_2 is corrupted by a semi-honest adversary $\mathcal{A}$, assuming that the OT protocol is semi-honest secure.*

Proof. The input item y and the output $\bigoplus_{i=1}^{n_h} GBF_y[h_i(y)]$ are given to the simulator S. The simulator constructs the Bloom filter using y regularly, and creates GBF'_y by running the following steps:

1. Set random values to $GBF'_y[h_i(y)]$ for $1 \leq i < n_h$.
2. Set
$$GBF'_y[h_{n_h}(y)] = \bigoplus_{i=1}^{n_h} GBF_y[h_i(y)] \oplus \bigoplus_{i=1}^{n_h-1} GBF'_y[h_i(y)]$$
3. Set $GBF'_y[i] = 0$ if $BF_y[i] = 0$.

Finally, S runs the OT simulator as the receiver m times, where in the i-th, run the receiver's input is $BF_y[i]$ and the receiver's output is $GBF'_y[i]$. The proof concludes when we show that GBF'_y is indistinguishable from GBF_y. The cells in both GBF'_y and GBF_y are equal to '0' for the indices i where $BF_y[i] = 0$. Now we need to show that for the remaining n_h cells these GBFs are indistinguishable. Any combination of $(n_h - 1)$ cells are random due to the property of secret sharing and the xor of n_h cells equals to $\bigoplus_{i=1}^{n_h} GBF_y[h_i(y)]$ in both GBFs, which concludes the proof. □

4 Our Private Set Membership Protocol

In this section, we propose a new PSM protocol that realizes Functionality 4. As discussed in the introduction, our protocol does not output the membership result, but instead outputs some function of it, so that it can be directly integrated into a larger secure computation protocol. After this section, we show how to extend our protocol to set intersection as well.

In the construction of the protocol, we use the following idea of (11): If $y \in X$, then both parties learn the same random value. Otherwise, they learn different random values. Then, the parties run a comparison protocol that outputs a function of the equality instead of the equality itself (Functionality 2). Our solution diverges from the solution of (11) in two ways. To realize the first part, (11) makes use of an OPPRF construction based on polynomials. We propose a new OPPRF construction based on Bloom filters. The selection of Bloom filters enables us to reduce the computation complexity of the protocol to a linear complexity. The other difference is that we utilize Ciampi-Orlandi PSM protocol (10) for secure equality testing as done in (29) for Functionality 2, instead of running a comparison circuit.

Functionality 4 Private Set Membership

Inputs. P_1 inputs $X = \{x_1, ..., x_{n_X}\}$ and a pair of strings (d_0, d_1), P_2 inputs y.

Outputs. The functionality checks the membership of y in X and returns d_1 to P_2 if $y \in X$. Otherwise, returns d_0 to P_2.

The overall view of our PSM protocol is as follow. To achieve private set membership, the parties first run the one-time OPPRF protocol based on garbled Bloom filters, where P_1 outputs r (a random value chosen by P_1), whereas P_2 learns some random value that may be r or something different. The value P_2 learns is always random and cannot distinguish which of the two random values it received; but, this random value is equal to r if and only if $y \in X$. Following this part, the parties run a secure functional equality testing protocol, where at the end of the protocol P_2 learns the function result of the equality relation, which is also the function result of the membership relation. We make use of the PSM protocol of Ciampi-Orlandi (10) for secure functional equality testing by reducing the number of items of the sender set to one. We present our semi-honest secure PSM solution in Protocol 3.

Protocol 3 Our Private Set Membership Protocol

Parameters. A set of hash functions $H = \{h_1, ..., h_{n_h}\}$.

Inputs. P_1 inputs a set of items $X = \{x_1, ..., x_{n_X}\}$ and a pair of strings (d_0, d_1), P_2 inputs an item y.
Outputs. P_2 outputs d_1 if $y \in X$. Otherwise, P_2 outputs d_0. P_1 outputs nothing.

The protocol steps:

1. P_1 picks an η-bit random value r and sets $T = \{t_1 = r, ..., t_{n_X} = r\}$.
2. P_1 and P_2 run Protocol 2 for one-time OPPRF with the respective inputs (X,T) and y. Denote the output of P_2 as r'.
3. P_1 and P_2 run Protocol 1 for functional equality testing with the respective inputs $(r, (d_0, d_1))$ and r'. The output of the PSM protocol is the output of Protocol 1.

Asymptotic Complexity of our PSM protocol. Protocol 2 requires $O(n)$ hash function computations as stated in the previous section. FEQT employs $O(\eta)$ operations for η oblivious transfers in the Ciampi-Orlandi PSM protocol. Thus, the asymptotic computation complexity of our PSM protocol becomes $O(n_X)$. The communication complexity comes from the oblivious transfers. Considering the oblivious transfer extension communication complexity as linear in the number of OTs, the communication complexity of Protocol 3 is also $O(n_X)$.

Theorem 3. *Protocol 3 securely realizes Functionality 4 when P_1 is corrupted by a semi-honest adversary $\mathcal{A}$, assuming that the OPPRF and FEQT protocols are semi-honest secure.*

Proof. The simulator S is given the input set X. S picks a random value r using its random tape and sets $T = \{t_1 = r, ..., t_{n_X} = r\}$. The simulator S runs the simulator of OPPRF protocol with the input $((X, T), \perp)$. Then, S runs the simulator of FEQT protocol with the input $(r, \perp)$. This completes the whole simulation, and indistinguishability is a direct result of the underlying simulators. □

Theorem 4. *Protocol 3 securely realizes Functionality 4 when P_2 is corrupted by a semi-honest adversary $\mathcal{A}$, assuming that the OPPRF and FEQT protocols are semi-honest secure.*

Proof. The simulator S is given the input item y and the output d_b for $b = y \stackrel{?}{\in} X$. The simulator picks a η-bit random value r''. S runs the simulator of OPPRF with the input (y, r'') and the simulator of FEQT with the input (r'', d_b). S does not know the uniform random value r' used in the real execution, but it follows the same distribution as r'', and therefore they are perfectly indistinguishable. The computational indistinguishability comes from the FEQT and OPPRF simulations, which are based on OT simulations. □

5 A Protocol to Improve Secure-Computation PSI Functionality

To the best of our knowledge, existing custom secure-computation PSI protocols (10; 1) including the circuit PSI ones (11; 12; 7) need a mapping function in their function-

alities which prevents to have a natural functionality. The reason of seeing such a functionality is the use of cuckoo tables in these protocols. Functionality 5 presents such a functionality that includes the mapping denoted by f.

Functionality 5 PSI functionality including cuckoo table mapping

Inputs. P_X inputs $X = \{x_1, ..., x_{n_X}\}$ and $M_X = \{(m_j^0, m_j^1) \mid 1 \leq j \leq \beta = O(n)\}$. P_Y inputs $Y = \{y_1, ..., y_{n_Y}\}$.
Outputs. P_X outputs nothing. P_Y outputs a mapping f from Y to the cuckoo table, such that $f : \{1, 2, ..., n\} \rightarrow \{1, 2, ..., \beta\}$, and P_Y also outputs $M_Y = \{m_j^{b_j} \mid 1 \leq j \leq \beta, b_j \in \{0, 1\}\}$ where $b_j = 1$ for $j = f(i)$ if $y_i \in X$, $b_j = 0$ otherwise.

Existence of the mapping in the functionality prevents the secure-computation PSI protocols from having the natural functionality (see Functionality 6) where one of the parties should learn the membership results for its items (or the items of the other party) in the set of the other party (or in its own set) in a somehow encrypted form. Construction of some kind of protocols such as threshold PSI protocols can be possible using primitives having Functionality 5, but there are some drawbacks of using PSI protocols realizing Functionality 5 when used within the construction of larger secure two-party protocols. These drawbacks can be collected in three points: 1) performance, 2) usability, and 3) security. Below, we provide some examples to explain these points.

Functionality 6 PSI with natural functionality

Inputs. P_X inputs $X = \{x_1, ..., x_{n_X}\}$. P_Y inputs $Y = \{y_1, ..., y_{n_Y}\}$ and $D_Y = \{(d_i^0, d_i^1) \mid 1 \leq i \leq n_Y\}$.
Outputs. P_X outputs $D_X = \{d_i^{b_i} \mid 1 \leq i \leq n_Y, b_i \in \{0, 1\}\}$ where $b_i = 1$ if $y_i \in X$, $b_i = 0$ otherwise. P_Y outputs nothing.

5.1 Comparison of Functionality 5 and 6 in terms of Performance, Usability, and Security.

Performance: One drawback of Functionality 5 is that, due to the cuckoo table mapping, the parties have more outputs than the number of items in the sets. While this may not result in a potential increase in the asymptotic complexity, it leads to an increase in the concrete computation and communication cost for the remaining secure computation protocol on the set intersection. When we consider the cardinality computation on the set intersection, in the case that one party has only hundreds of items while the other has millions of items, the cardinality circuit on top of the encrypted PSI result needs to get millions of inputs if a PSI protocol having Functionality 5 is used. On the other hand, only a circuit with hundreds of inputs will be enough for the remaining secure computation if a PSI protocol with Functionality 6 is employed instead. This example clearly shows the performance drawback of Functionality 5.

Usability: It could be argued that general construction of a larger secure computation protocol after Functionality 5 may become more complex when compared to the usage a PSI protocol having Functionality 6. In the case of Functionality 5, the circuit that is to be executed on the PSI result needs to get the cuckoo table mapping function and it may need to invert that mapping function, which makes the construction of the secure two-party computation circuit more complex. The design of such circuits taking into account the cuckoo table mapping is not a straightforward task, as opposed to simply continuing computation after Functionality 6.

Security: The last but maybe most important point is about security, which could be the most serious drawback of Functionality 5. The parties need to be very careful about the construction of circuits on the PSI output, so that it does not leak any information about their sets. For example, when an identity circuit is employed on the intersection in order to have a classical PSI protocol at the end, the usage of the identity circuit on the output of the PSI having Functionality 5 will reveal information about Y because P_X will learn the location of the items in the cuckoo table of P_Y. Another example could be the following. Assume that the output of the secure-computation PSI protocol is wire labels for inputs of a circuit. Since the mapping is known by the party who constructs the cuckoo table, the circuit should be generated by that party and shared with the other party. If the party who generates the circuit is not careful about the generation of the circuit, it can generate the circuit by ignoring the inputs corresponding to the randomly filled bins in the cuckoo table, and so this may lead to information leakage to the other party. For example, the receiving party can analyse the circuit to detect if there is any input wire that does not affect the result in the plain (not garbled) circuit. If there is such an input wire, it could be understood that the corresponding cuckoo table bin does not include an item from the set, which may reveal some kind of information about the set. For example, P_Y may want to make some optimization on the circuit considering the mapping function in the cuckoo table, but this optimized circuit, which will be sent to P_X, may leak some information about the cuckoo table computed from Y. This will result in information leakage about the set of P_Y. A very simple example for such optimization could be that P_Y designs a circuit that ignores the inputs from the bins which do not include any item from Y, to reduce the computation of the circuit, but this optimization will reveal some information about the cuckoo table to P_X.

5.2 Conversion Protocol to Natural Functionality

To overcome of these drawbacks of secure-computation PSI protocols having Functionality 5, we introduce a conversion protocol that converts these PSI protocols into a secure-computation PSI protocol that realizes Functionality 6 that does not include the cuckoo table mapping.

The main idea in our conversion protocol presented in Protocol 4 is allowing P_Y who knows the cuckoo table mapping to eliminate the results corresponding to the random bins in the cuckoo table in the main PSI protocol, without leaking any information to P_X. To be able to realize this functionality without leaking any information, P_X construct the data set M_X by setting the message pairs as the additively homomorphic encryption of '0' and '1' values. Learning one message from the message pairs will not leak any information to P_Y about the membership because the encryption results will look random to P_Y. Since the response from P_Y will include only encryption of the data in the data set of P_Y, P_X will not be able to identify the location of the places of the items of Y in the cuckoo table, which prevent information leakage about the items of P_Y to P_X.

In addition to having a simple functionality that makes the construction of larger secure computation protocol easier, the conversion protocol also makes the functionality of circuit PSI and secure-computation PSI protocols more flexible. While the circuit-PSI supports the functionality where each party learns only secret shares of the intersection, Functionality 6 not only support secret share type of outputs but also supports other type of outputs, for example, the receiving party can learn the (partially) homomorphic encryption result of the intersection under encryption key of the sender party, or the receiving party can learn the corresponding one of wire labels chosen by the sender party. Section 6 provides detailed explanation how to use the functionality for different types of outputs.

Asymptotic Complexity. The steps that are related to the protocol conversion are

Protocol 4 The conversion protocol

Inputs. P_X inputs $X = \{x_1, ..., x_{n_X}\}$. P_Y inputs $Y = \{y_1, ..., y_{n_Y}\}$ and $D_Y = \{(d_i^0, d_i^1) \mid 1 \leq i \leq n_Y\}$.
Outputs. P_X outputs $D_X = \{d_{b_j}^j \mid 1 \leq j \leq n_Y, b_j \in \{0,1\}\}$ where $b_j = 1$ if $y_j \in X$, $b_j = 0$ otherwise. P_Y outputs nothing.
The protocol steps:

1. P_X constructs a set $M_X = \{(E_x(0), E_x(1)) \mid 1 \leq j \leq \beta\}$ where E is a additively homomorphic encryption algorithm, $E_x(.)$ is encryption under the key of P_X, and β is the number of bins in the cuckoo table, which can be computed from n_X. Note that this step can be executed in advance as an offline operation, since it is independent of Y (or even X).

2. P_X and P_Y run a protocol realizing Functionality 5. P_X inputs X and $M_X = \{(E_x(0), E_x(1)) \mid 1 \leq j \leq \beta\}$. P_Y inputs Y. After this execution P_X learns nothing and P_Y learns the mapping f such that $f : \{1, 2, ..., n\} \rightarrow \{1, 2, ..., \beta\}$, and $M_Y = \{E_x(b_j) \mid 1 \leq j \leq \beta, b_j \in \{0,1\}\}$ where $b_j = 1$ for $j = f(i)$ if $y_i \in X$, $b_j = 0$ otherwise.

3. P_Y computes the set $\{E_x(b_{f(i)}d_i^1 + (1 - b_{f(i)})d_i^0) \mid 1 \leq i \leq n_Y\}$ and sends it to P_X.

4. P_X decrypts the items in the received set to learn the corresponding values and outputs $D_X = \{d_i^{b_i} \mid 1 \leq i \leq n_Y, b_i \in \{0,1\}\}$ where $b_i = 1$ if $y_i \in X$, $b_i = 0$ otherwise.

steps 1, 3 and 4. Note that in step 1, P_X computes β encryption of '0' and '1' values which are independent of the input X. Thus these encryption operations can be done offline before the execution of the protocol. As a result, the overhead complexity of the protocol comes from Step 3 and Step 4. P_Y needs to compute and send n_Y encryption results and P_X needs to decrypt these encryption results in steps 3 and 4, respectively. Thus, the overhead of the conversion protocol on top of the protocol realizing Functionality 5 is $O(n_Y)$ both in computation and communication.

Theorem 5. *Protocol 4 securely realizes Functionality 6 when P_X is corrupted by a semi-honest adversary, assuming that the protocol in Step 2 securely realizes Functionality 5 against semi-honest adversaries and that the encryption scheme used is CPA-secure.*

Proof. The simulator S is given the input $X = \{x_1, ..., x_{n_X}\}$ and the output $D_X = \{d_i^{b_i} \mid 1 \leq i \leq n_Y, b_i \in \{0,1\}\}$. S computes step 1 as done by honest P_X to have additively homomorphic encryptions M_X of zeros and ones. S then invokes the simulator of the protocol realizing Functionality 5, by giving $((X, M_X), \perp)$ as input. The last thing to complete to proof is to show that S is able to compute a set that is indistinguishable with the set of ciphertexts in step 3. S computes the set $\{E_X(d_i^{b_i}) \mid 1 \leq i \leq n_Y\}$ by directly encrypting the values in the given output D_X. Since E is a secure randomized encryption scheme, the set $\{E_X(d_i^{b_i}) \mid 1 \leq i \leq n_Y\}$ and the set of ciphertexts in step 3 are computationally indistinguishable (otherwise a straightforward reduction can be performed to the CPA security of E). □

Theorem 6. *Protocol 4 securely realizes Functionality 6 when P_Y is corrupted by a semi-honest adversary, assuming that the protocol in Step 2 is semi-honest secure.*

Proof. As seen from the protocol, the view of P_Y consists of the input Y of P_Y and the transcript of the secure protocol executed in step 2. There is no additional step in the conversion protocol which affects the view of P_Y. Thus directly running the simulator of the underlying protocol is enough. □

5.3 Application to our PSI protocol presented at CANS 2020

In this section, we present how the conversion protocol can be applied to our CANS 2020 PSI protocol to show a concrete example and to improve our PSI protocol. Since the PSI protocol utilizes a one-time OPPRF construction introduced in our CANS 2020 paper, we first give that construction and then show the implementation of the conversion protocol on the PSI protocol.

5.3.1 Batch One-Time OPPRF

We propose a new batch one-time OPPRF construction in Protocol 5 that implements Functionality 7, to be used in our PSI protocol. For the construction of a batch OPPRF from Protocol 2, instead of using different garbled Bloom filters for each programmed value set, we construct only one garbled Bloom filter, and store the shares of programmed values in the same garbled Bloom filter. Note that for each set X_i, a different set of hash functions (hash function set H_i for the programmed value set X_i) is used, since there might be some items which belong to more than one set. In our PSI protocol we use only one programmed value t_i for each X_i.

Functionality 7 Batch One-Time Oblivious Programmable Pseudo Random Function

---

Inputs. P_1 inputs a predefined set of item sets $X = \{X_1, ..., X_\beta\}$ and corresponding programmed value set $T = \{t_1, ..., t_\beta\}$. P_2 inputs a set of items $Y = \{y_1, ..., y_\beta\}$.
Outputs. The functionality checks the membership relations $y_i \in X_i$ and returns t_i if $y_i \in X_i$ to P_2; otherwise returns a random r_i to P_2, for each i where $1 \leq i \leq \beta$.

Asymptotic Complexity. Since the size of the garbled Bloom filter is linear in the number of items to be stored in it and OT extension is also linear in the number of OT executions, the computation and communication complexities of our batch one-time OPPRF protocol becomes linear in the total number of programmed values in the programmed value sets.

Theorem 7. *Protocol 5 securely realizes Functionality 7 when P_1 is corrupted by a semi-honest adversary $\mathcal{A}$, assuming that the OT protocol is semi-honest secure.*

Proof. The simulator S is given the input set of sets X and the programmed values set T. The simulator computes a garbled Bloom filter using its random tape such that $\bigoplus_{j=1}^{k} GBF_X[h_{i,j(x_{i,l})}] = t_i$. S runs the simulator of the OT protocol as the sender with the input $(GBF_X, \perp)$. This concludes the simulation. Indistinguishability directly comes from the garbled Bloom filter construction following the protocol, and the OT simulator. □

Theorem 8. *Protocol 5 securely realizes Functionality 7 when P_2 is corrupted by a semi-honest adversary $\mathcal{A}$, assuming that the OT protocol is semi-honest secure.*

Proof. The input set Y and the output R' are given to the simulator S. The simulator constructs a Bloom filter for Y and a garbled Bloom filter GBF'_Y following the steps:

Protocol 5 Bloom Filter Based Batch One-Time OPPRF Protocol

Parameters. A set of hash function sets $H = \{H_1, ..., H_\beta\}$ where $H_i = \{h_{i,0}, ...h_{i,n_h}\}$
Inputs. P_1 inputs a set of item sets $X = \{X_1, ..., X_\beta\}$ and corresponding programmed value sets $T = \{t_1, ..., t_\beta\}$. P_2 inputs a set of items $Y = \{y_1, ...y_\beta\}$.
Outputs. P_2 outputs a set of random values $R' = \{r'_1, ..., r'_\beta\}$, where $r'_i = t_i$ if $y_i \in X_i$; otherwise r'_i is a random value; for $1 \leq i \leq \beta$.

The protocol steps:

1. P_1 constructs a garbled Bloom filter GBF_X having $\max(\eta, \ell)$-bit strings in each cell such that

$$\bigoplus_{j=1}^{n_h} GBF_X[h_{i,j}(x_{i,l})] = t_i$$

   for $1 \leq i \leq \beta$ and $1 \leq j \leq n_h$.

2. P_2 constructs a Bloom filter BF_Y for the items in Y.

3. P_1 and P_2 run m oblivious transfers where P_1's input is $(0, GBF_X[i])$ and P_2's input is $BF_Y[i]$ for the i-th oblivious transfer, and the output of P_2 is 0 if $BF_y[i] = 0$ or $GBF_X[i]$ if $BF_Y[i] = 1$. Call the OT output P_2 obtains as $GBF_Y[i]$.

4. P_2 outputs $R' = \{r'_1, ..., r'_\beta\}$ where

$$r'_i = \bigoplus_{j=1}^{n_h} GBF_Y[h_{i,j}(y_i)]$$

1. Constructs a BF BF_Y for Y.
2. Constructs a GBF GBF'_Y such that $\bigoplus_{j=1}^{k} GBF'_Y[h_{i,j}(y_i)] = r'_i$ for $1 \leq i \leq \beta$.
3. Sets $GBF'_Y[i] = 0$ if $BF_Y[i] = 0$.

Then, S runs the simulator of the OT protocol as the receiver with the input (BF_Y, GBF'_Y). Note that the garbled bloom filters GBF'_Y and GBF_Y are indistinguishable as discussed in the proof of Theorem 2. □

5.3.2 Improving CANS 2020 PSI construction

Our PSM protocol can be used to build an efficient PSI protocol using the hashing techniques introduced in (15; 16). In this technique, one party constructs a cuckoo table as mentioned in Section 2.1 using two hash functions and the other party maps her items into bins in a hash table using the two hash functions that are applied on each item. Then, a private set membership protocol is applied on each bin where the party who constructs the cuckoo table inputs the (single) item in the i-th bin, and the other party inputs the set of items in the i-th bin of its hash table, for the i-th execution of the PSM protocol. If one were to directly employ our PSM construction to obtain a PSI protocol using this hashing technique, the computation and communication complexities of the full PSI protocol would be $O(n \log n / \log \log n)$, since the number of items in each hash table bin is $O(\log n / \log \log n)$ and the number

of bins is $O(n)$. Note that with this usage, for each bin, P_2 and P_1 run $O(n)$ parallel OPPRF protocols and then apply $O(n)$ parallel FEQT protocols. Instead of following this straightforward way, we show that it is possible to make the communication and computation complexities linear while extending our PSM solution to a PSI solution using our batch one-time OPPRF protocol.

Our full PSI protocol that realizes Functionality 6 is introduced in Protocol 6. Note that in Step 5 of the protocol, the bins of the hash table are given as the item sets to the batch one-time OPPRF protocol. While there are many items in the bins of the hash table, most of them are random values and the total number of non-random items in the hash table will be the product of the number of items (n) and the number of cuckoo hash functions (chosen as 3 in our protocol). Thus, the size of the garbled Bloom filter constructed in the batch one-time OPPRF protocol will be $O(n)$, which allows our PSI protocol to have linear complexity.

Protocol 6 Our improved PSI protocol

Parameters. A set of hash function sets $H = \{H_1, ..., h_\beta\}$ where $H_i = \{h_{i,1}, ..., h_{i,n_h}\}$ for $1 \leq i \leq \beta$.
Inputs. P_X inputs $X = \{x_1, ..., x_{n_X}\}$. P_Y inputs $Y = \{y_1, ..., y_{n_Y}\}$ and $D_Y = \{(d_i^0, d_i^1) \mid 1 \leq i \leq n_Y\}$.
Outputs. P_X outputs $D_X = \{d_{b_j}^j \mid 1 \leq j \leq n_Y, b_j \in \{0,1\}\}$ where $b_j = 1$ if $y_j \in X$, $b_j = 0$ otherwise. P_Y outputs nothing.
The protocol steps:

1. P_X constructs a set $M_X = \{(E_x(0), E_x(1)) \mid 1 \leq j \leq \beta\}$ where $E_x(.)$ is encryption under the public key of P_X and E is a partially homomorphic encryption algorithm.

2. P_X constructs a hash table for the set X.

3. P_Y constructs a cuckoo table for the set Y. f denotes the mapping such that $f(i) = j$ if y_i is mapped to the j-th bin of the cuckoo table.

4. P_X picks a set of β η-bit random values $R = \{r_1, ..., r_\beta\}$.

5. P_X and P_Y run Protocol 5 with their respective inputs: (hash table, R) and cuckoo table. Let the output of P_Y be $R' = \{r'_1, ..., r'_\beta\}$.

6. P_X and P_Y run β parallel executions of Protocol 1 for functional equality testing, where for the j-th run, the inputs of P_X and P_Y are $(r_j, (E_x(0), E_x(1)))$ and r'_j.

7. P_Y computes the set $\{E_x(b_{f(i)}d_i^1 + (1 - b_{f(i)})d_i^0) \mid 1 \leq i \leq n_Y\}$ and sends it to P_X.

8. P_X decrypts the items in the received set to learn the corresponding values and outputs $D_X = \{d_i^{b_i} \mid 1 \leq i \leq n_Y, b_i \in \{0,1\}\}$ where $b_i = 1$ if $y_i \in X$, $b_i = 0$ otherwise.

Note that when we use two hash functions for cuckoo hashing, then there will be some items in Y which cannot be placed into the table and have to be moved to a stash. For each of these items in the stash, a PSM protocol also has to be executed. When we consider the number of these items as $\omega(1)$, then the complexity of our PSI protocol

becomes bigger than $O(n)$. To make the complexity linear, Pinkas et al. proposed to use dual execution or a stash-less cuckoo hashing (11). In dual execution, after the first run of the PSI protocol, P_2 learns the membership result for its items except the ones in the stash. Then the parties run the PSI protocol swapping their roles, that is, P_1 constructs a cuckoo table for X and P_2 constructs a hash table for the items in the stash. Since there may be some items of P_1 which have not been placed in the cuckoo table and moved to a stash, P_1 and P_2 should run the PSM protocol for their items in the stashes. However, this usage does not realize the Functionality 6 that we consider, since in the second run, P_2 learns the function of the membership result between its items in the stash and the set X, and in the final PSM protocols run for the items in the stashes, P_2 again learns the function of the membership result between its items in the stash and P_1's items in the stash. That is, P_2 learns two different results for its items in the stash that makes the protocol diverge from Functionality 6. Because of these two reasons, we make use of the second method of Pinkas et al., which is the usage of stash-less cuckoo hashing with three hash functions.

The steps that helps to have such a functionality are Steps 1, 7 and 8 in Protocol 6. Note that these steps can also be injected to other existing PSI protocols to make them address Functionality 6.

Asymptotic Complexity. For simplicity we take $n = n_X = n_Y$. While it seems that there are $O(n \log n / \log \log n)$ items in the hash table of P_X, which makes the length of the Bloom filters $O(n \log n / \log \log n)$, the actual number of items is $O(3n) = O(n)$ since the other items are random values padded to the bins to make the number of items in the bins $O(\log n / \log \log n)$. Thus, the complexity of Step 5 of Protocol 6 becomes $O(n)$. Since the number of bins is $O(n)$ and for each bin only one equality testing is executed in Step 6, the complexity of Step 6 will be $O(n)$. As stated in Section 5, the overhead of the application of conversion protocol (steps 1, 7 and 8 in Protocol 6) is $O(n_Y)$. Thus, the communication and computation complexities of our PSI protocol becomes $O(n)$.

Security. The security arguments of the conversion protocol is presented with Theorems 5 and 6 with the assumption that the PSI protocol is semi-honest secure. Also, the security of the PSI protocol is shown in (1). Thus these two security arguments completes the security proof of our improved protocol presented in Protocol 6.

5.4 Application on Chandran et al. PSI protocol

Before presenting how our conversion protocol can be applied on the Chandran et al. PSI protocol (7), we remind their PSI protocol with the following mains steps.

1. *Hashing (steps 1 and 2 of the PSI protocol in (7)):* P_Y and P_X (P_0 and P_1 in (7)) respectively compute the cuckoo table and the hash table of their sets Y and X. This corresponds to the steps 1 and 2 of our PSI protocol in (1).
2. *Relaxed Batch OPPRF (steps 3-5):* P_Y and P_X compute the relaxed batch OPPRF respectively using the cuckoo table and hash table as their inputs. In the end of the computation, for each bin of the tables P_Y learns a set (W_i) of three items while P_X learns a target value t_j. This corresponds to the steps 3 and 4 of our PSI protocol. Note that in our PSI protocol, P_Y learns an item instead of a set of items for each cuckoo table bin.
3. *Comparing OPPRF Outputs (step 6):* P_Y and P_X run the PSM protocol for each corresponding bin of the cuckoo table and hash table. In the end of the protocol, they learn secret shares of the membership result for the each bin. This step is similar to the step 5 in our protocol. Note that, since in our protocol P_Y learns an item instead of a set, we run an equality testing protocol instead of a PSM protocol.

4. *Circuit (step 7):* The parties run a circuit by providing the secret shares and the cuckoo table as input.

As seen from the brief and high level description of the PSI protocol of (7), the protocol outputs secret shares for β bins and the circuit needs to get the cuckoo table as input in addition to the secret shares. To convert that protocol to a protocol realizing Functionality 6, the following modifications can be applied on the PSI protocol.

- P_X executes step 1 of Protocol 4 to have the set of encryption results of '0's and '1's (i.e., $M_X = \{(E_x(0), E_x(1)) \mid 1 \leq j \leq \beta\}$). This will correspond to step 1 of our improved PSI protocol (Protocol 6).
- Hashing and Relaxed Batch OPPRF main steps are executed as done in the original protocol. This step is similar to the steps 2-5 in our improved PSI protocol.
- Instead of using their PSM protocols, a PSM protocol realizing Functionality 8 is executed by P_X and P_Y assuming the role of P_2 and P_1, respectively. This functionality is very similar to Functionality 4 except the difference that the party who holds an item inputs the string pair instead of the party who holds the set. We introduce a protocol (Protocol 7) for this functionality, which is very similar to Protocol 3. This step corresponds to step 6 of our protocol except the difference that we run β parallel equality testing protocol while β parallel PSM protocol is executed here.
- P_Y and P_X runs steps 3 and 4 of the conversion protocol presented in Protocol 4, as done in our improved PSI protocol.

Functionality 8 Secure Computation Private Set Membership Functionality 2

Inputs. P_1 inputs $X = \{x_1, ..., x_{n_X}\}$, P_2 inputs y and a pair of strings (d_0, d_1).

Outputs. The functionality checks the membership of y in X and returns d_1 to P_1 if $y \in X$. Otherwise, returns d_0 to P_1.

Protocol 7 Secure Computation Private Set Membership Protocol 2

Parameters. A set of hash functions $H = \{h_1, ..., h_{n_h}\}$.

Inputs. P_1 inputs a set of items $X = \{x_1, ..., x_{n_X}\}$, P_2 inputs an item y and a pair of strings (d_0, d_1).
Outputs. P_1 outputs d_1 if $y \in X$. Otherwise, P_1 outputs d_0. P_2 outputs nothing.

The protocol steps:

1. P_1 picks an η-bit random value r and sets $T = \{t_1 = r, ..., t_{n_X} = r\}$.
2. P_1 and P_2 run Protocol 2 for one-time OPPRF with the respective inputs (X,T) and y. Denote the output of P_2 as r'.
3. P_1 and P_2 run Protocol 1, by respectively assuming the roles of P_2 and P_1, for functional equality testing with the respective inputs (r) and $(r', (d_0, d_1))$. The output of the PSM protocol is the output of Protocol 1.

With these changes in the PSI protocol of (7), we have a PSI protocol that realizes the natural functionality expected from a PSI protocol that allow secure computation on the PSI result.

6 Applications of Secure-Computation PSI Protocol with Natural Functionality

In this section, we present some applications of our protocol to exemplify how it can be integrated into a larger two-party protocol.

6.1 Having a Classical PSI Protocol

If one would like to compute the intersection itself, this is easily doable using our protocol. P_Y prepares the set D_Y by setting $(d_i^0, d_i^1) = (0, 1)$ for $1 \leq i \leq n_Y$ and inputs its set Y and D_Y to our PSI protocol. P_X inputs its set X. After running our PSI protocol, P_Y learns nothing and P_X learns the set $D_X = \{d_i^{b_i} \mid 1 \leq i \leq n_Y, d_i^{b_i} \in \{0, 1\}\}$ where $d_i^{b_i} = 1$ if $y_i \in X$, $d_i^{b_i} = 0$ otherwise. P_X sends D_X to P_Y, and so P_Y learns the intersection.

6.2 Having a PSI Cardinality Protocol

To just learn the cardinality of the intersection and no additional information about the intersection, P_Y chooses a key pair for an additively homomorphic encryption scheme and prepares the data set D_Y by setting $(d_i^0, d_i^1) = (E_{pk}(0), E_{pk}(1))$ for $1 \leq j \leq n_Y$. P_Y inputs Y and D_Y and P_X inputs X to our protocol and P_X learns the data set D_X where $d_i = E_{pk}(1)$ if $y_i \in X$ and the other d_i values are encryption of '0' under the public key of P_Y. P_X homomorphically sums up the d_i values using the additive homomorphism property of the encryption scheme and obtains the result $\sum E_{pk}(d_i) = E_{pk}(|X \cap Y|)$. P_X sends $E_{pk}(|X \cap Y|)$ to P_Y and P_Y learns the cardinality by decrypting the received ciphertext.

6.3 Having a Threshold PSI Protocol

Assume that the parties want to learn the intersection if and only if the cardinality of the intersection is equal to or bigger than a threshold t. Our PSI protocol can be utilized as follows. P_Y generates a key K for a probabilistic encryption scheme and divides the decryption key into n_Y secret shares such that t of the shares are enough to construct the key (using a t out of n_Y threshold secret sharing scheme). P_Y also sets $(d_i^0, d_i^1) = ((E_K(0), r_i), (E_K(1), k_i))$ where $E_K(.)$ is the encryption operation under the key K chosen by P_Y, r_i is a random value having the same length with the key, and k_i is the i-th secret share of the decryption key K. After running our PSI protocol, P_X obtains the data set D_X such that $d^i = (E_K(1), k_i)$ for the i-th items of Y that are in the intersection. If the number of items are equal to or bigger than the threshold t, then P_X will be able to construct the decryption key and able to decrypt the left parts of the d_i values that are the encryption of '0' or '1', which allows P_X to learn the intersection in a similar way as explained in Section 6.1. Otherwise, if the number of items in the intersection is fewer than t, P_X will have fewer than t key shares and will not be able to construct the decryption key, and therefore will not be able to decrypt the ciphertexts of '0' and '1' correctly, which means that P_X will not be able to learn the intersection. When a threshold secret sharing scheme such as (41) is used, P_X needs to compute all possible (potentially exponentially-many, depending on t and n_Y) combinations of secret shares to construct the decryption key. To avoid considering all possible combinations of secret shares for construction of the decryption

key and to not leak more information, usage of other tools such as the Reed-Solomon decoding algorithm, as done in (42), can be investigated as future work.

On the other hand, one needs to be very careful with such constructions. When a verifiable secret sharing scheme such as (43) is used, then P_X will be able to learn which secret shares are valid ones, and this allows P_X to learn which items in the cuckoo table is in the hash table of P_X, thereby leaking more information than what is desired.

6.4 General Secure Computation over Set Intersection

In general, any secure two-party computation can be computed where the input contains the set intersection, on top of our protocol. Below, we analyze different types of secure two-party computation techniques, and explain how our protocol can be employed to provide the intersection as input to the corresponding continuation protocol securely. Our protocol's outputs will be the continuation protocol's inputs.

6.4.1 Secret share type of outputs.

P_Y sets the set D_Y as follows. For each index i where $1 < i < n_Y$, P_Y chooses a random string (s_i) with enough length and sets $d_i^0 = s_i$ and $d_i^1 = s_i \oplus 1$. According to the functionality, if $y_i \in X$ then P_X learns d_i^1 where s_i and d_i^1 will be the secret shares of the membership. Similar argument will be valid also for the case that $y_i \notin X$.

6.4.2 Homomorphic encryption result type of outputs.

To be able to have such kind of outputs, P_Y generates a key pair to be used in the (partially, somewhat, fully) homomorphic encryption, and constructs the set D_Y by setting $d_i^0 = E(0)$ and $d_i^1 = E(1)$ where E is a (partially, somewhat, fully) homomorphic encryption under the public key of P_Y. In the end of the protocol, P_X will be able to learn $E(1)$ is $y_i \in X$ or $E(0)$ otherwise.

6.4.3 Wire label type of outputs.

P_Y constructs a circuit that gets our protocol's output as its input and computes the function on the intersection. For this aim, P_Y chooses wire label pairs (w_j^0, w_j^1) for each input wire that represent the membership result of y_j on X, and creates the set D_Y as $d_j^0 = w_j^0$ and $d_j^1 = w_j^1$. Then P_X learns the corresponding wire labels depending on the intersection and obliviously evaluates the circuit. With this approach, only the output of the function on the intersection will be learned without leaking any other information about the intersection.

6.5 Handling the Associated Data

In same cases, in addition to the set, the parties may have some data associated with each item in the set. One example in a real world scenario could be advertisement use cases, where one party (P_X) holds a set of identifiers and the other party (P_Y) has the set of identifiers and the amount of payment made by people having those identifiers. The parties require a protocol that reveals only the sum of the associated data (payments) of the items (identifiers) in the intersection. (7; 11) propose such protocols by modifying their circuit PSI protocol. In our protocol, we do not need to modify the protocol; instead, it is enough to fill the set D_Y in an appropriate way (see below) to support this advertisement type of use case with associated (payment) data.

One way could be utilization of secret shares. At the end of the protocol, the parties learn secret shares of the associated data of the items in the intersection. For the other items, the parties learn secret shares of '0'. Then the parties execute a

summation protocol that takes secret shares of the items to be summed up. For this purpose, the sender party P_Y includes the secret shares of the associated data in the set D_Y as follows. For each index i where $1 \leq i \leq n_Y$, P_Y chooses a random string (r_i) with enough length and respectively includes r_i and $r_i \oplus a_i$ in d_i^0 and d_i^1, respectively, where a_i is the associated data. According to the functionality, if $y_i \in X$ then P_X learns the secret share of the associated data, otherwise P_X learns the secret share of '0'.

If some computation using homomorphic encryption is desired, then instead of including r_i and $r_i \oplus a_i$ in d_i^0 and d_i^1, the values d_i^0 and d_i^1 are respectively set to $E_y(0)$ and $E_y(a_i)$, where E_y is an additively homomorphic encryption scheme using the public key of P_Y. After learning the corresponding data values $E_y(0)$ or $E_y(a_i)$, P_X uses them to homomorphically compute the encryption result of the sum of 0 or a_i values, and then returns the resulting ciphertext to P_Y. After decryption, P_Y learns the sum of the associated data for the intersection.

7 Performance Evaluation

7.1 Concrete Complexity

Parameter Choices. We take the number of hash functions used in the construction of Bloom filters as $n_h = \eta$ and follow the choice of (28) to set the size of the Bloom filter as taking $m = 1.44kn$. Note that taking $n_h = \eta$ doesn't reduce the security level to statistical correctness parameter because the result of BF-based OPPRF protocol are random numbers which then be inputs of the equality testing protocol. Following the parameters in (11), we choose the number of bins as $1.27n$ and the number of cuckoo hashes as 3, which makes the probability of having at least one item in the stash 2^{-40}, consistent with our preferred statistical correctness parameter η.

Concrete Complexity of our PSM protocol. For the Bloom filters, P_1 and P_2 compute $n \times n_h$ and n_h hash functions, respectively. For the OT-extension in the OPPRF part, they run m oblivious transfer whose total computation complexity is approximately equal to $3m$ symmetric key operations thanks to the oblivious transfer extension (34). Finally, the parties execute Ciampi-Orlandi PSM protocol where the number of items in the set of P_1 is one, which makes the computation complexity 6η symmetric key operations at P_1 and 5η symmetric key operations at P_2 (the reader can refer to (29) for the complexity calculation for the FEQT protocol) [1]. Thus the computation complexity of the protocol at the party where majority of workload is done is $n \times n_h + 3m + 6\eta$. Since we choose $m = 1.44 \times n_h \times n$ and $n_h = \eta$ then the complexity becomes $5.32n\eta + 6\eta$. For the parameter $\eta = 40$ the complexity will be $212.8n + 240$ symmetric key operations. The communication complexity comes from the oblivious transfers. In the OPPRF step, the message lengths in the oblivious transfer is η bits, while for the FEQT part, it is $2(\kappa + \eta)$ bits. Considering that the total number of bits transferred in the OT extension equals to 2 times the items' length times number of pairs, the communication complexity of the protocol becomes $2\times m\times\eta+2\times\eta\times2\times(\kappa+\eta) = 2\times(1.44\times n\times\eta)\times\eta+2\eta\times2\times(\kappa+\eta) = 2.88n\eta^2+4\kappa\eta+4\eta^2$.

Concrete Complexity of our Conversion Protocol Overhead The computational overhead introduced by our conversion protocol comes from the steps 1, 3 and 4 in Protocol 4. In step 1, P_X needs to compute $2 \times \beta$ encyption operations, but since these operations are independent from the input of P_X, this step can be executed offline before the execution of the protocol. In step 3, for each of the items in the set P_Y needs to execute one encryption, one inverse, $2 \times |d| + 1$ multiplication operations

[1] The item lengths in the GBF and so the lengths of the items to be tested for equality are $\max(\eta, \ell)$ bits as stated in Protocol 2. In concrete complexity analysis, we take it as η for simplicity considering that in practice generally $\eta > \ell$.

where $|d|$ denotes the bit length of the items in D_Y. These operation numbers comes from the following equation:

$$E_x(b_{f(i)}d_i^1 + (1 - b_{f(i)})d_i^0) = (E_x(b_{f(i)}))^{d_i^1} \times (E(1)/E_x(b_{f(i)}))^{d_i^0}$$

Finally, in step 4, P_X performs n_Y decryption operations. Regarding the communication overhead, we only need to consider the data transfer from P_Y to P_X in step 3 where n_Y encryption results are sent in that communication.

7.2 Experimental Verification

Setup. We implemented Ciampi-Orlandi PSM protocol, our PSM and PSI protocols, and our conversion protocol using C programming language and GMP library. Runtime estimates are done for LAN and WAN under the assumption that the bandwidth in LAN (respectively in WAN) is 1 Gbps (100 Mbps) and RTT is 1 ms (100 ms). In our experiment setup, P_1 and P_2 run on the same machine as different processes and communicate with each other over a TCP channel. We run the protocols for different size of sets and item lengths on a single CPU core of a computer that has 2.1 Ghz 16-core Intel Xeon CPU with 64 GB RAM. In the experiments, we chose RSA 2048 as asymmetric encryption algorithm in base OT, the statistical correctness parameter η as 40 bits, AES as the encryption algorithm, SHA-256 with different initialization vectors as the hash functions. We take the f function such that it outputs 128-bit wire labels. We take the number of hash functions in the construction of Bloom filters in our protocols as $n_h = 40$. The results are the averages over 10 executions of the protocols.

PSM. Table 1 shows the total amount of data transmitted between P_1 and P_2 during the execution of the protocols and the run-times in LAN and WAN setting. As can be seen from the table, our BF-based semi-honest PSM protocol has linear complexity both on computation and communication, and we provide comparable performance. Our asymptotic advantage becomes visible with larger ℓ values.

Table 1: Performance results of Ciampi-Orlandi and our PSM protocols.

Protocol		Ciampi-Orlandi PSM			Our PSM		
Set size n		$n = 2^{12}$	$n = 2^{14}$	$n = 2^{16}$	$n = 2^{12}$	$n = 2^{14}$	$n = 2^{16}$
Comm. [MB]	$\ell = 32$	5.4	21.3	84.5	6.0	23.6	93.8
	$\ell = 48$	8.1	31.8	126.5	6.5	25.4	101.0
	$\ell = 64$	10.7	42.3	168.5	7.4	29.0	115.4
LAN [s]	$\ell = 32$	2.05	4.20	11.72	2.58	6.51	22.03
	$\ell = 48$	2.44	5.76	18.12	2.66	6.56	22.34
	$\ell = 64$	2.80	7.36	24.40	2.66	6.60	22.54
WAN [s]	$\ell = 32$	10.21	36.39	139.44	11.65	42.12	163.75
	$\ell = 48$	14.69	53.82	209.32	12.42	44.90	174.97
	$\ell = 64$	18.97	71.30	279.05	13.78	50.37	196.90

Overhead of Our PSI Conversion Protocol. We also implemented our PSI conversion protocol to validate our performance analysis and compare the overhead with the performance of our PSI protocol and Chandran et al. PSI protocol. We used Paillier encryption scheme (44) for the additevely homomorphic encryption need and run the implementation in the same computation environment mentioned above. We chose the length of the prime numbers as 2048-bit. We excluded the cost of step 1 since it can be done offline. The remaining (online) cost of our conversion protocol is independent of the number of items in P_X. While the length of the items in the sets X and Y does not affect the cost of the protocol, the length of the items in the data set D_Y has an impact on the cost. In the experiment, we selected the length of these items as 128-bits, considering them as wire labels for a garbled circuit. Table 2 presents our

conversion protocol overhead with different set sizes of Y, where we choose the set size n_Y as 2^4, 2^6, and 2^8 considering the use case of private contact discovery mentioned in Section 1. Note that since the performance of our conversion protocol is independent of n_X, we fixed it to 2^{16} in the experiments. Table 3 presents the overhead cost of our conversion protocol on top of the PSI protocols introduced in (1) and (7). The percentage values in the table are computed using the performance results presented in Table 2 and the PSI protocols' performance results given in (1) and (7). According to the results, it can be concluded that our conversion protocol converts the PSI protocol to having the natural functionality with neglectable overhead, especially when the number of items in Y is much fewer than the number of items in X.

Table 2: Performance of our conversion protocol (independent of n_X).

n_Y	Comm. [MB]	LAN [s]	WAN [s]
2^4	0.008	0.09	0.10
2^6	0.031	0.34	0.39
2^8	0.125	1.34	1.53

Table 3: Overhead of our conversion protocol on top of PSI protocols in (1) and (7), $n_X = 2^{16}$.

PSI Protocol	n_Y	Comm.	LAN	WAN
PSI in (1)	2^4	0.001%	0.106%	0.010%
	2^6	0.005%	0.399%	0.040%
	2^8	0.021%	1.574%	0.157%
PSI in (7)	2^4	0.008%	4.918%	1.495%
	2^6	0.032%	18.579%	5.830%
	2^8	0.129%	73.224%	22.870%

8 Conclusion

As a main contribution, we proposed a conversion protocol having some steps that enables linear-complexity circuit PSI and secure-computation PSI protocols to realize the expected natural functionality of secure computation PSI protocols. To illustrate how the conversion protocol can be applied to the existing protocols, we applied the conversion on our PSI protocol introduced at CANS 2020 and on Chandran et al. protocol introduced at PETS 2022. While converting these linear protocols into PSI secure computation protocols having the natural functionality, our conversion protocol also keeps the complexity in linear. In addition saving the asymptotic complexity, the overhead of our protocol becomes negligible especially for the cases that the number of items in one party is very small when compared with the number of items in the other party. This assumption is especially valid for the non-silo use cases, e.g., one party can be a mobile phone having less number of items and the other party can be a server having huge amount of items. Finally, in the conversion protocol, we used public key cryptography (partially homomorphic encryption). Improvement of this construction with the use of symmetric key primitives to have better performance is left as a future work.

As side contributions, we propose a private set intersection (PSI) protocol achieving linear communication and computation complexities while outputting a function of the membership results to be used in larger secure two-party protocols to compute other functionalities over the intersection set. To construct such a protocol, we first used one-time oblivious programmable pseduo-random function (OPPRF) based on existing Bloom filter based PSI solutions and then proposed a private set membership (PSM) protocol. To reduce the complexity while converting the PSM solution to a PSI protocol using hashing techniques, we constructed another primitive that is

called a batch one-time OPPRF. Finally, using these new constructions, we introduced our PSI protocol with linear communication and computation complexities. We also implemented our protocols to validate our performance analysis and show concrete efficiency of our protocols. We leave security against malicious adversaries, and multi-party PSI with bi-oblivious data transfer as future work.

Acknowledgements

We acknowledge support from TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project number 119E088 and under the 1515 Frontier RD Laboratories Support Program with project number 5169902. We thank Sherman Chow and Karl Norrman for their valuable comments.

References

[1] F. Karakoç and A. Küpçü, "Linear complexity private set intersection for secure two-party protocols," in *Cryptology and Network Security - 19th International Conference, CANS 2020, Vienna, Austria, December 14-16, 2020, Proceedings* (S. Krenn, H. Shulman, and S. Vaudenay, eds.), vol. 12579 of *Lecture Notes in Computer Science*, pp. 409–429, Springer, 2020.

[2] F. Karakoç and A. Küpçü, "Linear complexity private set intersection for secure two-party protocols," *IACR Cryptol. ePrint Arch.*, p. 864, 2020.

[3] J. J. Zhang, F. Wang, X. Wang, G. Xiong, F. Zhu, Y. Lv, J. Hou, S. Han, Y. Yuan, Q. Lu, and Y. Lee, "Cyber-physical-social systems: The state of the art and perspectives," *IEEE Trans. Comput. Soc. Syst.*, vol. 5, no. 3, pp. 829–840, 2018.

[4] J. Zeng, L. T. Yang, M. Lin, H. Ning, and J. Ma, "A survey: Cyber-physical-social systems and their system-level design methodology," *Future Gener. Comput. Syst.*, vol. 105, pp. 1028–1042, 2020.

[5] Y. Qian, X. Xia, and J. Shen, "A profile matching scheme based on private set intersection for cyber-physical-social systems," in *IEEE Conference on Dependable and Secure Computing, DSC 2021, Aizuwakamatsu, Japan, January 30 - February 2, 2021*, pp. 1–5, IEEE, 2021.

[6] M. Etemad, F. Beato, A. Küpçü, and B. Preneel, "Are you really my friend? efficient and secure friend-matching in mobile social networks," in *2017 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2017, Paris, France, April 26-28, 2017*, pp. 122–131, IEEE, 2017.

[7] N. Chandran, D. Gupta, and A. Shah, "Circuit-psi with linear complexity via relaxed batch OPPRF," *Proc. Priv. Enhancing Technol.*, vol. 2022, no. 1, pp. 353–372, 2022.

[8] E. D. Cristofaro and G. Tsudik, "Practical private set intersection protocols with linear complexity," in *FC* (R. Sion, ed.), 2010.

[9] Y. Zhao and S. S. M. Chow, "Are you the one to share? secret transfer with access structure," *Proc. Priv. Enhancing Technol.*, vol. 2017, no. 1, pp. 149–169, 2017.

[10] M. Ciampi and C. Orlandi, "Combining private set-intersection with secure two-party computation," in *SCN*, pp. 464–482, 2018.

[11] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai, "Efficient circuit-based PSI with linear communication," in *EUROCRYPT*, pp. 122–153, 2019.

[12] B. H. Falk, D. Noble, and R. Ostrovsky, "Private set intersection with linear communication from general assumptions," in *ACM WPES*, 2019.

[13] Y. Huang, D. Evans, and J. Katz, "Private set intersection: Are garbled circuits better than custom protocols?," in *NDSS*, 2012.

[14] B. Pinkas, T. Schneider, and M. Zohner, "Scalable private set intersection based on OT extension," *ACM Trans. Priv. Secur.*, vol. 21, no. 2, pp. 7:1–7:35, 2018.

[15] B. Pinkas, T. Schneider, and M. Zohner, "Faster private set intersection based on OT extension," in *USENIX Security*, 2014.

[16] B. Pinkas, T. Schneider, G. Segev, and M. Zohner, "Phasing: Private set intersection using permutation-based hashing," in *USENIX Security*, 2015.

[17] V. Kolesnikov, N. Matania, B. Pinkas, M. Rosulek, and N. Trieu, "Practical multi-party private set intersection from symmetric-key techniques," in *ACM CCS*, 2017.

[18] A. Shamir, "On the power of commutativity in cryptography," in *ICALP*, 1980.

[19] C. A. Meadows, "A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party," in *IEEE S&P*, 1986.

[20] E. D. Cristofaro, P. Gasti, and G. Tsudik, "Fast and private computation of cardinality of set intersection and union," in *CANS*, 2012.

[21] V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu, "Efficient batched oblivious PRF with applications to private set intersection," in *ACM CCS*, 2016.

[22] S. K. Debnath and R. Dutta, "Secure and efficient private set intersection cardinality using bloom filter," in *ISC*, 2015.

[23] A. Davidson and C. Cid, "An efficient toolkit for computing private set operations," in *ACISP*, 2017.

[24] Y. Zhao and S. S. M. Chow, "Can you find the one for me?," *ACM WPES*, 2018.

[25] M. Ion, B. Kreuter, A. E. Nergiz, S. Patel, M. Raykova, S. Saxena, K. Seth, D. Shanahan, and M. Yung, "On deploying secure computing commercially: Private intersection-sum protocols and their business applications," in *IEEE EuroS&P*, 2020.

[26] M. Ion, B. Kreuter, E. Nergiz, S. Patel, S. Saxena, K. Seth, D. Shanahan, and M. Yung, "Private intersection-sum protocol with applications to attributing aggregate ad conversions," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 738, 2017.

[27] S. Lv, J. Ye, S. Yin, X. Cheng, C. Feng, X. Liu, R. Li, Z. Li, Z. Liu, and L. Zhou, "Unbalanced private set intersection cardinality protocol with low communication cost," *Future Gener. Comput. Syst.*, vol. 102, pp. 1054–1061, 2020.

[28] C. Dong, L. Chen, and Z. Wen, "When private set intersection meets big data: an efficient and scalable protocol," in *ACM CCS*, 2013.

[29] F. Karakoç, M. Nateghizad, and Z. Erkin, "SET-OT: A secure equality testing protocol based on oblivious transfer," in *ARES*, 2019.

[30] A. C. D. Resende and D. F. Aranha, "Faster unbalanced private set intersection," in *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers* (S. Meiklejohn and K. Sako, eds.), vol. 10957 of *Lecture Notes in Computer Science*, pp. 203–221, Springer, 2018.

[31] M. O. Rabin, "How to exchange secrets by oblivious transfer," tech. rep., Harvard Aiken Computation Laboratory Technical Report TR-81, 1981.

[32] S. Kim, S. Kim, and G. Lee, "Secure verifiable non-interactive oblivious transfer protocol using RSA and bit commitment on distributed environment," *Future Gener. Comput. Syst.*, vol. 25, no. 3, pp. 352–357, 2009.

[33] D. Beaver, "Correlated pseudorandomness and the complexity of private computations," in *ACM STOC*, 1996.

[34] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently," in *CRYPTO*, 2003.

[35] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient two-round OT extension and silent non-interactive secure computation," in *ACM CCS*, 2019.

[36] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, "Efficient pseudorandom correlation generators: Silent OT extension and more," in *CRYPTO*, 2019.

[37] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[38] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[39] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold, "Keyword search and oblivious pseudorandom functions," in *TCC*, 2005.

[40] Y. Lindell, "How to simulate it - A tutorial on the simulation proof technique," in *Tutorials on the Foundations of Cryptography.* (Y. Lindell, ed.), pp. 277–346, Springer International Publishing, 2017.

[41] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[42] E. Zhang, J. Chang, and Y. Li, "Efficient threshold private set intersection," *IEEE Access*, vol. 9, pp. 6560–6570, 2021.

[43] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings* (J. Feigenbaum, ed.), vol. 576 of *Lecture Notes in Computer Science*, pp. 129–140, Springer, 1991.

[44] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT*, pp. 223–238, 1999.