# Combining Private Set-Intersection with Secure Two-Party Computation

Michele Ciampi[1(✉)] and Claudio Orlandi[2]

[1] The University of Edinburgh, Edinburgh, UK
mciampi@ed.ac.uk

[2] Aarhus University, Aarhus, Denmark
orlandi@cs.au.dk

**Abstract.** Private Set-Intersection (PSI) is one of the most popular and practically relevant secure two-party computation (2PC) tasks. Therefore, designing special-purpose PSI protocols (which are more efficient than generic 2PC solutions) is a very active line of research. In particular, a recent line of work has proposed PSI protocols based on oblivious transfer (OT) which, thanks to recent advances in OT-extension techniques, is nowadays a very cheap cryptographic building block. Unfortunately, these protocols cannot be plugged into larger 2PC applications since in these protocols one party (by design) learns the output of the intersection. Therefore, it is not possible to perform secure *post-processing* of the output of the PSI protocol. In this paper we propose a novel and efficient OT-based PSI protocol that produces an "encrypted" output that can therefore be later used as an input to other 2PC protocols. In particular, the protocol can be used in combination with all common approaches to 2PC including garbled circuits, secret sharing and homomorphic encryption. Thus, our protocol can be combined with the right 2PC techniques to achieve more efficient protocols for computations of the form $z = f(X \cap Y)$ for arbitrary functions $f$.

## 1 Introduction

*Private Set-Intersection (PSI)* is one of the most practically relevant *secure two-party computation* (2PC) tasks. In PSI two parties hold two sets of strings $X$ and $Y$, respectively. At the end of the protocol one (or both) party should learn the intersection of the two sets $Z = X \cap Y$ and nothing else about the input of the other party. There are many real-world applications in which PSI is required. As an example, when mobile users install messaging apps, they need to discover whom among their contacts (from their address book) is also using the app, in

This research received funding from: COST Action IC1306; the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC); the European Union's Horizon 2020 research and innovation programme under grant agreements No 731583 (SODA) and No 780477 (PRIViLEDGE); "GNCS - INdAM". The work of 1st author has been done in part while visiting Aarhus University, Denmark.

D. Catalano and R. De Prisco (Eds.): SCN 2018, LNCS 11035, pp. 464–482, 2018.
https://doi.org/10.1007/978-3-319-98113-0_25

order to be able to start communicating seamlessly with them. Doing so requires users to learn the intersection of their contact list with the list of registered users of the service which is stored at the server side. This is typically done by having users send their contact list to the server that can then compute the intersection and return the result to the user. Unfortunately this solution is very problematic not only for the privacy of the user, but for the privacy of the users' contacts as well! In particular, some of the people in the contact list might not want their phone number being transferred and potentially stored by the server, but they have no control over this.[1] Note that this is not just a theoretically interesting problem and that Signal (one of the most popular end-to-end encrypted messaging app) has recently recognized this as being a real problem and offered partial solutions to it.[2] PSI has many other applications, including computing intersections of suspect lists, private matchmaking (comparing interests), *testing human genome* [3], *privacy-preserving ride-sharing* [16], *botnet detection* [32], *advertisment conversion rate* [20] and many more. From a feasibility point of view, PSI is just a special case of 2PC and therefore any generic 2PC protocol (such as [15,43]) could be used to securely evaluate PSI instances as well. However, since PSI is a natural functionality that can be applied in numerous real-world applications, many efficient protocols for this specific functionality have been proposed, with early results dating back to the 80s [30,40]. The problem was formally defined in [13] and follow up work increased the efficiency of PSI protocols to have complexity only *linear* in the inputs of the parties [8,23]. A very recent work shows how to obtain a protocol where communication complexity is linear in the size of the smaller set and logarithmic in the larger set [5]. However, these protocols still require performing expensive public-key operations (e.g., exponentiations) for every element in the input sets. As public-key operations are orders of magnitudes more expensive than symmetric key operations, these protocols are not practically efficient for large input sets. In the meanwhile, generic techniques for 2PC had improved by several orders of magnitude and the question of whether special purpose protocols or generic protocols were most efficient has been debated in [9,19]. Due to its practical relevance, PSI protocols in the *server-aided* model have been proposed as well [24]. Independent and concurrent works [11,35] (which were not publicly available at time we first posted our paper on ePrint) consider the problem of using a PSI protocol to construct more complex functionality in an efficient way. More specifically, [35] provides a way to securely compute many variants of the set intersection functionality using a clever combination of Cuckoo hashing and garbled circuit. The work of Falk et al. [11] focuses on obtaining a PSI protocol that is efficient in

[1] Some apps do not transfer the contact list in cleartext, but a hashed version instead. However, since the domain space of phone numbers is small enough to allow for brute forcing of the hashes, this does not guarantee any real privacy guarantee.

[2] Unfortunately, the Signal team has concluded that current PSI protocols are too inefficient for their application scenario and relied on trusted-hardware instead, in the style of [41]. See https://signal.org/blog/private-contact-discovery/ for more details on this.

terms of communication. In addition, the authors of [11] propose a PSI protocol where the output can be secret shared that has communication complexity of $O(m\lambda \log\log m)$, where $\lambda$ is the bit-length of the elements and $m$ is the set-size. The techniques used in our paper significantly differ from the techniques used in [11,35]. Our solution avoids the use of garbled circuits and rely on the security and the efficiency of OT and symmetric key encryption schemes.

## 1.1 OT-Based PSI

The most efficient PSI protocols today are those following the approach of PSZ [34,36]. These protocols make extensive use of a cryptographic primitive known as *oblivious transfer (OT)*. While OT provably requires expensive public-key operation, OT can be "extended" as shown by [1,21,26] i.e., the few necessary expensive public-key operations can be amortized over a very large number of OT instances, and the marginal cost of OT is only a few (faster) symmetric key operations instead. In particular, improvements in OT-extension techniques directly imply improvements to PSI protocols as shown by e.g., [27,33]. In a nutshell, the PSZ protocol introduced two important novel ideas to the state of the art of PSI. First, they give an efficient instantiation of the *private set membership protocol* (PSM) introduced in [12] based on OT. Second, they show how to efficiently implement PSI from PSM using hashing techniques. (An overview of their techniques is given below).

## 1.2 Our Contribution

The main contribution of this paper is to give an efficient instantiation of PSM that provides output in encrypted format and can therefore be combined with further 2PC protocols. Our PSM protocol can be naturally combined with the hashing approach of PSZ to give a PSI protocol with encrypted output achieving the same favourable complexity in the input sizes. This enables the combination the efficiency of modern PSI techniques with the potentials of general 2PC. Combining our protocols with the right 2PC post-processing allows more efficient evaluation of functionalities of the form $Z = f(X \cap Y)$ for any function $f$. Like in PSZ we only focus on semi-honest security. Instantiating our PSM protocol together with an actively secure OT-extension protocol such as [2,25] would result in a protocol with *privacy* but not *correctness* (i.e., the view of the protocol *without* the output can be efficiently simulated), which is a meaningful notion of security in some settings. PSI protocols with security against malicious adversaries have been proposed in e.g., [17,38,39]. It is an interesting open problem to design efficient protocols which are both secure against active (or covert) adversaries and that produce encrypted output. Also, like in PSZ, we only focus on the two-party setting. The recent result of [18] has shown that multiparty set-intersection can be computed efficiently. Extending our result to the multiparty case is an interesting future research direction. We also compare the computation complexity of our scheme for PSM with all the circuit-based PSI approaches (which can be combined with further postprocessing) proposed in [37]. More precisely, in Table 1 we compare our protocol with the protocols of [37] in terms of number of symmetric key operations, and bits exchanged between the parties. The result of this comparison is that our protocol has better performance, in terms of computational complexity, than all the circuit-based PSI approaches considered for our comparison[3]. We refer the reader to the full version for more details about this comparison.

### 1.3 Improving the Efficiency of Smart Contract Protocols

Most of the cryptocurrency systems are built on top of blockchain technologies where miners run distributed consensus whose security is ensured as long as the adversary controls only a minority of the miners. Some cryptocurrency systems allow to run complex programs and decentralized applications on the blockchain. In Ethereum[4] those programs are called smart contracts. Roughly speaking, the aim of a smart contract is to run a protocol and start a transaction to pay a user of the cryptocurrency systems according to the output of the protocol execution. Unfortunately, this interesting feature of the smart contracts does not come for free. Indeed, in order to execute a smart contract, it is required to pay a *gas fee* that depends on the number of instructions of the protocol to be executed. So, higher is the complexity of protocol, higher is the price to pay. In this context a cryptographic protocol that outputs intermediate values in a secret shared way is particularly useful. Suppose that two parties want to securely compute $f(X \cap Y)$ for arbitrary functions $f$, and reward another party depending on the output of this computation. Instead of writing on a smart contract the entire protocol to compute $f(X \cap Y)$, the two parties could run a sub-protocol $\Pi$ to obtain a secret share of $\chi = X \cap Y$ without using a smart contract, and then run another sub-protocol $\Pi'$ to compute $f(\chi)$, this time using a smart contract to enforce the reward policy. Following this approach it is possible to move part of the computation off-chain, thus increasing the performance and, at the same time, decreasing the costs required to execute the smart contract. Moreover, we observe that $\chi$ can be reused to compute different functions $f'$. The scenario described above is particularly interesting if one of the party can be fully malicious, but in this work we will focus on semi-honest security leaving the above as an open question.

## 2 Technical Overview

*Why PSZ and 2PC Do Not Mix.* We start with a quick overview of the PSM protocol in PSZ [34,36], to explain why their protocol inherently reveals the

[3] The complexity of the protocols proposed in [37] depends upon parameters that are also related to the used hash function. In order to make our comparison fair, we have set these parameters as showed in the first column in Table 10 of [37]. More precisely, the authors of [37] show in that table which parameters are adopted for their empirical efficiency comparison for the case where one set is much greater than the other set (which is exactly the case of PSM).

[4] http://www.ethereum.org.

**Table 1.** Computation and communication complexity comparison for the PSM case. $M$ represents the size of the set, $s$ is the security parameter and $\lambda$ is the bit-length of each element.

| | # of sym. key operations | Communication [bits] |
|---|---|---|
| Yao SCS [19] | $12\lambda M \log M + 3\lambda M$ | $2\lambda M s(1 + 3\log M)$ |
| GMW SCS [19] | $12\lambda M \log M$ | $6\lambda M(s+2)\log M$ |
| Yao PWC [37] | $4\lambda M + 6393\lambda$ | $\lambda(M3s + 3198s + 15, 6)$ |
| GMW PWC [37] | $6\lambda M + 9594\lambda$ | $\lambda(M4 + 6396 + 2sM + 6396s)$ |
| This work | $4\lambda M + 3\lambda$ | $2\lambda M s + M s$ |

intersection to one of the parties. From a high-level point of view, the protocol is conceptually similar to the PSM protocol from oblivious pseudorandom function (OPRF) of [12], except that the OPRF is replaced with a similar functionality efficiently implemented using OT. For simplicity, here we will use the OPRF abstraction. The goal of a PSM protocol is the following: the receiver $R$ has input $x$, and the sender $S$ has input a set $Y$; at the end of the protocol the receiver learns whether $x \in Y$ or not while the sender learns nothing. The protocol starts by using the OPRF subprotocol, so that $R$ learns $x^* = F_k(x)$ (where $k$ is known to $S$), whereas $S$ learns nothing about $x$. Now $S$ evaluates the PRF on her own set and sends the set $Y^* = \{y^* = F_k(y) | y \in Y\}$ to $R$, who checks if $x^* \in Y^*$ and concludes that $x \in Y$ if this is the case. In other words, we map all inputs into pseudorandom strings and then let one of the parties test for membership "in the clear". Since the party performing the test doesn't have access to the mapping (e.g., the PRF key), this party can only check for the membership of $x$ and no other points (i.e., all elements in $Y^* \setminus \{x^*\}$ are indistinguishable from random in $R$'s view). From the above description, it should be clear that the PSZ PSM inherently reveals the output to one of the parties. Turning this into a protocol which provides encrypted output is a challenging task. Here is an attempt at a "strawman" solution: we change the protocol such that $R$ still learns the pseudorandom string $x^* = F_k(x)$ corresponding to $x$, but now $S$ sends a value *for every element in the universe*. Namely, for each $i$ (in the domain of $Y$) $S$ sends an encryption of whether $i \in Y$ "masked" using $F_k(i)$ e.g., $S$ sends $c_i = F_k(i) \oplus E(i \in Y)$[5]. Now $R$ can compute $c_x \oplus x^* = E(x \in Y)$ i.e., an encrypted version of whether $x \in Y$, which can be then used as input to the next protocol. While this protocol produces the correct result, its complexity is exponential in the bit-length of $|x|$, which is clearly not acceptable. Intuitively, we know that only a polynomial number of $c_i$'s will contain encryptions of 1, and therefore we need to find a way to "compress" all the $c_i$ corresponding to $i \notin Y$ into a single one, to bring the complexity of the protocol back to $O(|Y|)$. In the following, after defining some useful notation, we give an intuitive explanation on how to do that.

[5] The exact format of the "encryption" $E(\cdot)$ would depend on the subsequent 2PC protocol and is irrelevant for this high-level description.

## 2.1 Our Protocol

We introduce some useful (and informal) notation in order to make easier to understand the ideas behind our construction. We let $Y = \{y_1, \ldots, y_M\}$ be the input set of the sender $\mathsf{S}$, and we assume w.l.o.g., that $|Y| = M = 2^m$.[6] All strings have the same length e.g., $|x| = |y_i| = \lambda$.[7] We will use a special symbol $\perp$ such that $x \neq \perp \, \forall x$. We use a function $\mathsf{Prefix}(x, i)$ that outputs the $i$ most significant bits of $x$ ($\mathsf{Prefix}(x, i) \neq \mathsf{Prefix}(x, j)$ when $i \neq j$ independently of the value of $x$) and for simplicity we define $\mathsf{Prefix}(Y, i)$ to be the set constructed by taking the $i$ most significant bits of every element in $Y$. The protocol uses a symmetric key encryption scheme $\mathsf{Sym} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with the additional property that given a key $k \leftarrow \mathsf{Gen}(1^s)$ it is possible to efficiently verify if a given ciphertext is in the range of $k$ (see Sect. 3 for a formal definition). Finally, the output of the protocol will be one of two strings $\gamma_0, \gamma_1$ chosen by $S$, respectively denoting $x \notin Y$ and $x \in Y$. The exact format of the two strings depends on the protocol used for post-processing. For instance if the post-processing protocol is based on: (1) *garbled circuits*, then $\gamma_0, \gamma_1$ will be the labels corresponding to some input wire; (2) *homomorphic encryption*, then $\gamma_b = \mathsf{Enc}(pk, b)$ for some homomorphic encryption scheme $\mathsf{Enc}$; (3) *secret-sharing*, then $\gamma_b = s_2 \oplus b$ where $s_2$ is a uniformly random share chosen by $S$, so that if $R$ defines its own share as $s_1 = \gamma_b$ then it holds that $s_1 \oplus s_2 = b$.[8] In order to "compress" the elements of $Y$ we start by considering an undirected graph with a level structure of $\lambda + 1$ levels. The vertices in the last level of this graph will correspond to the elements of $Y$. More precisely, we associate the secret key $k_{b_\lambda b_{\lambda-1} \ldots b_1}$ of a symmetric key encryption scheme $\mathsf{Sym}$ to each element $y = b_\lambda b_{\lambda-1} \ldots b_1 \in Y$. The main idea is to allow the receiver to obliviously navigate this graph in order to get the key $k_{b_\lambda b_{\lambda-1} \ldots b_1}$ if $x = y$, for some $y = b_\lambda b_{\lambda-1} \ldots b_1 \in Y$, or a special key $k^\star$ otherwise. Moreover we allow the receiver to navigate the graph efficiently, that is, every level of the graph is visited only once. Once a key $k$ is obtained by the receiver, the sender sends $O(|Y|)$ ciphertexts in a such a way that the key obtained by the receiver can decrypt only one ciphertext. Moreover the plaintext of this ciphertext will correspond to $\gamma_0$ or $\gamma_1$ depending on whether $x \in Y$ or not.

**First Step: Construct the Graph *G*.** Each graph level $i \in \{0, \ldots, \lambda\}$ has size at most $|\mathsf{Prefix}(Y, i)| + 1$. More precisely, for every $t = b_\lambda b_{\lambda-1} \ldots b_{\lambda-i} \in \mathsf{Prefix}(Y, i)$ there is a node in the level $i$ of $G$ that contains a key $k_{b_\lambda b_{\lambda-1} \ldots b_{\lambda-i}}$. In addition, in the level $i$ there is a special node, called *sink node* that contains

[6] Sets can always be padded with dummy elements, but the complexity of the protocol can match $M$ that in practice can be $M \approx 2^{m-1}$.

[7] We can assume $\lambda$ to be smaller than the (statistical) security parameter $s$ and we will denote the bit decomposition of $x$ by $x = x_\lambda \ldots x_1$. Otherwise before running the protocol the parties can hash their input down and run the protocol with inputs $h(x)$ and $h(Y) = \{h(y_1), \ldots, h(y_M)\}$. Clearly if $x = y_i$ then $h(x) = h(y_i)$, and for correctness we need that $Pr[h(x) \in h(Y) \wedge x \notin Y] < 2^{-s}$.

[8] Here we use $\oplus$-secret sharing without loss of generality. Any 2-out-2 secret sharing would work here.

a key $k_i^\star$ (which we refer to as *sink key*). The aim of $k_i^\star$ is to represent all the values that do not belong to $\mathsf{Prefix}(i, Y)$. Let us now describe how the graph $G$ is constructed. First, for $i = 1, \ldots, \lambda$ the key (for a symmetric key encryption scheme) $k_i^\star$ is generated using the generation algorithm $\mathsf{Gen}(\cdot)$. As discussed earlier the aim of these keys is to represent the elements that do not belong to $Y$. More precisely, the sink key $k_i^\star$, with $i \in \{1, \ldots, \lambda\}$ represents all the values that do not belong to $\mathsf{Prefix}(Y, i)$ and the key $k_\lambda^\star$ (the last sink key) will be used to encrypt the output $\gamma_0$ corresponding to non-membership in the last step of our protocol. Note that if $\mathsf{Prefix}(x, i) \notin \mathsf{Prefix}(Y, i)$ then $\mathsf{Prefix}(x, j) \notin \mathsf{Prefix}(Y, j)$ for all $j > i$. Therefore, once entered in a sink node, the *sink path* is never abandoned and thus the final sink key $k_\lambda^\star$, will be retrieved (which allows recovery of $\gamma_0$). Let us now give a more formal idea of how $G$ is constructed.

– The root of $G$ is empty, and in the second level there are two vertices $k_0$ and $k_1$ where[9], for $b = 0, 1$

$$k_b = \begin{cases} k \leftarrow \mathsf{Gen}(1^s), & \text{if } b \in \mathsf{Prefix}(Y, 1) \\ k_1^\star, & \text{otherwise} \end{cases}$$

– For each vertex $k_t$ in the level $i \in \{1, \ldots, \lambda\}$ and for $b = 0, 1$ create the node $k_{t||b}$ as follows (if it does not exists) and connect $k_t$ to it.

$$k_{t||b} = \begin{cases} k \leftarrow \mathsf{Gen}(1^s), & \text{if } t||b \in \mathsf{Prefix}(Y, i+1) \\ k_{i+1}^\star, & \text{if } t||b \notin \mathsf{Prefix}(Y, i+1) \\ k_{i+1}^\star, & \text{if } k_t = k_i^\star \end{cases}$$

We observe that a new node $k_{t||b}$ is generated only when $t||b \in \mathsf{Prefix}(Y, i)$. In the other cases the sink node $k_{i+1}^\star$ is used.

In Fig. 1 we show an example of what the graph $G$ looks like for the set $Y = \{010, 011, 110\}$. In this example it is possible to see how, in the 2nd level, all the elements that do not belong to $\mathsf{Prefix}(Y, 2)$ are represented by the sink node $k_2^\star$. Using this technique we have that in the last level of $G$ one node ($k_3^\star$ in this example) is sufficient to represent all the elements that do not belong to $Y$. Therefore, we have that the last level of $G$ contains at most $|Y| + 1$ elements. We also observe that every level of $G$ cannot contain more than $|Y| + 1$ nodes.

**Second Step: Oblivious Navigation of *G*.** Let $x = x_\lambda x_{\lambda-1} \ldots x_1$ be the receiver's ($\mathsf{R}$'s) private input and $Y$ be the sender's ($\mathsf{S}$'s) private input. After $\mathsf{S}$ constructs the graph $G$ we need a way to allow $\mathsf{R}$ to obtain $k_{x_\lambda x_{\lambda-1} \ldots x_1}$ if $x \in Y$ and the sink key $k_\lambda^\star$ otherwise. All the computation has to be done in such a way that no other information about the set $Y$ is leaked to the receiver, and as well that no information about $x$ is leaked to the sender. In order to do so we use $\lambda$ executions of 1-out-of-2 OT. The main idea is to allow the receiver to select

[9] In abuse of notation we refer to a vertex using the key represented by the vertex itself.

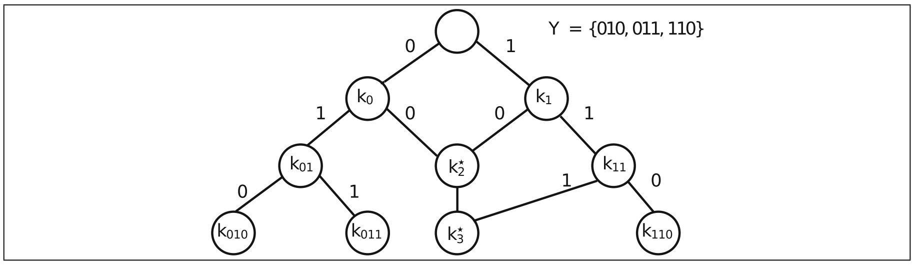

**Fig. 1.** Example of how the graph $G$ appears when the sender holds the set $Y$.

which branch to explore in $G$ depending on the bits of $x$. More precisely, in the first execution of OT, $\mathsf{R}$ will receive the key $k_{x_\lambda}$ iff there exists an element in $Y$ with the most significant bit equal to $x_\lambda$, the sink key $k_1^\star$ otherwise. In the second execution of OT, $\mathsf{R}$ uses $x_{\lambda-1}$ as input and $\mathsf{S}$ uses $(c_0,\, c_1)$ where $c_0$ is computed as follows:

– For each key in the second level of $G$ that has the form $k_{t||0}$, the key $k_{t||0}$ is encrypted using the key $k_t$.
– For every node $v$ in the first level that is connected to a sink node $k_2^\star$ in the second level, compute an encryption of $k_2^\star$ using the key contained in $v$.
– Pad the input with random ciphertexts up to the upper bound for the size of this layer (more details about this step are provided later).
– Randomly permute these ciphertexts.

The procedure to compute the input $c_1$ is essentially the same (the only difference is that in this case we consider every key with form $k_{t||1}$ and encrypt it using $k_t$). Roughly speaking, in this step every key contained in a vertex $u$ of the second level is encrypted using the keys contained in the vertex $v$ of the previous level that is connected to $u$. For example, following the graph provided in Fig. 1, $c_0$ would be equal to $\{\mathsf{Enc}(k_0, k_2^\star), \mathsf{Enc}(k_1, k_2^\star)\}$ and $c_1$ to $\{\mathsf{Enc}(k_0, k_{01}), \mathsf{Enc}(k_1, k_{11})\}$. Thus, after the second execution of OT $\mathsf{R}$ receives $c_{x_{\lambda-1}}$ that contains the ciphertexts described above where only one of these can be decrypted using the key $k$ obtained in the first execution of OT. The obtained plaintext corresponds to the key $k_{x_\lambda x_{\lambda-1}}$ if $\mathsf{Prefix}(x, 2) \in \mathsf{Prefix}(Y, 2)$, to the sink key $k_2^\star$ otherwise. The same process is iterated for all the levels of $G$. More generally, if $\mathsf{Prefix}(x, j) \in \mathsf{Prefix}(Y, j)$ then after the $j-$th execution of OT $\mathsf{R}$ can compute the key $k_{x_\lambda x_{\lambda-1} \ldots x_{\lambda-j}}$ using the key obtained in the previous phase. Conversely if $\mathsf{Prefix}(x, j) \notin \mathsf{Prefix}(Y, j)$ then the sink key $k_j^\star$ is obtained by $\mathsf{R}$. We observe that after every execution of OT $\mathsf{R}$ does not know which ciphertext can be decrypted using the key obtained in the previous phase, therefore he will try to decrypt all the ciphertext until the decryption procedure is successful. To avoid adding yet more indexes to the (already heavy) notation of our protocol we deal with this using a private-key encryption scheme with efficiently verifiable range. We note that this is not necessary and that when implementing the protocol one can instead use the *point-and-permute technique* [4]. This, and other optimisations and extensions of our protocol, are described in Sect. 5.

**Third Step: Obtain the Correct Share.** In this step $\mathsf{S}$ encrypts the output string $\gamma_0$ using the key $k^\star_\lambda$ and uses all the other keys in the last level of $G$ to encrypt the output string $\gamma_1$.[10] At this point the receiver can only decrypt either the ciphertext that contains $\gamma_0$ if $x \notin Y$ or one (and only one) of the ciphertexts that contain $\gamma_1$ if $x \in Y$. In the protocol that we have described so far $\mathsf{R}$ does not know which ciphertext can be decrypted using the key that he has obtained. Also in this case we can use a point-and-permute technique to allow $\mathsf{R}$ to identify the only ciphertext that can be decrypted using his key.

*On the Need for Padding.* As describe earlier, we might need to add some padding to the OT sender's inputs. To see why we need this we make the following observation. We recall that in the $i$-th OT execution the sender computes an encryption of the keys in the level $i$ of the artificial graph $G$ using the keys of the previous level $(i-1)$.[11] As a result of this computation the sender obtains the pair $(c^i_0, c^i_1)$, that will be used as input of the $i$-th OT execution, where $c^i_0$ (as well as $c^i_1$) contains a number of encryptions that depends upon the number of vertices on level $(i-1)$ of $G$. We observe that this leaks information about the structure of $G$ to the receiver, and therefore leaks information about the elements that belong to $Y$. Considering the example in Fig. 1, if we allow the receiver to learn that the 2nd level only contains 3 nodes, then the receiver would learn that all the elements of $Y$ have the two most significant bits equal to either $t$ or $t'$ for some $t, t' \in \{0,1\}^2$ (in Fig. 1 for example we have $t = 01$ and $t' = 11$; note however that the receiver would not learn the actual values of $t$ and $t'$).

We note that the technique described in this section can be seen as a special (and simpler) example of securely evaluating a branching program. Secure evaluation of branching programs has previously been considered in [22,31]: unfortunately these protocols cannot be instantiated using OT-extension and therefore will not lead to practically efficient protocols (the security of these protocols is based on *strong* OT which, in a nutshell, requires the extra property that when executing several OTs in parallel, the receiver should not be able to correlate the answers with the queries beyond correlations which follow from the output).

Finally, we note that the work of Chor et al. [6] uses a data structure similar to the one described here to achieve private information retrieval (PIR) based on keywords. The main difference between keyword based PIR and PSM is that in PSM the receiver should not learn any other information about the data stored by the sender, so their techniques cannot be directly applied to our setting.

## 3 Definitions and Tools

We denote the security parameter by $s$ and use "$||$" as concatenation operator (i.e., if $a$ and $b$ are two strings then by $a||b$ we denote the concatenation of

[10] The key $k^\star_\lambda$ could not exists; e.g. if $Y$ contains all the strings of $\lambda$ bits.

[11] The only exception is the first OT execution where just two keys are used as input.

$a$ and $b$). For a finite set $Q$, $x \leftarrow Q$ denotes a sampling of $x$ from $Q$ with uniform distribution. We use the abbreviation PPT that stands for probabilistic polynomial time. We use $\mathsf{poly}(\cdot)$ to indicate a generic polynomial function. We assume the reader to be familiar with standard notions such as *computational indistinguishability* and the *real world/ideal world* security definition for secure two-party computation (see the full version [7] for the actual definitions).

## 3.1 *Special* Private-Key Encryption

In our construction we use a private-key encryption scheme with two additional properties. The first is that given the key $k$, it is possible to efficiently verify if a given ciphertext is in the range of $k$. With the second property we require that an encryption under one key will fall in the range of an encryption under another key with negligible probability As discussed in [28], it is easy to obtain a private-key encryption scheme with the properties that we require. According to [28, Definition 2] we give the following definition.

**Definition 1.** *Let* $\mathsf{Sym} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a private-key encryption scheme and denote the range of a key in the scheme by* $\mathsf{Range}_s(k) = \{\mathsf{Enc}(k, x)\}_{x \in \{0,1\}^s}$.

1. *We say that* $\mathsf{Sym}$ *has an* efficiently verifiable range *if there exists a ppt algorithm $M$ such that* $M(1^s, k, c) = 1$ *if and only if* $c \in \mathsf{Range}_s(k)$. *By convention, for every* $c \notin \mathsf{Range}_s(k)$, *we have that* $\mathsf{Dec}(k, c) = \bot$.
2. *We say that* $\mathsf{Sym}$ *has an* elusive range *if for every probabilistic polynomial-time machine* $\mathcal{A}$, *there exists a negligible function* $\nu(\cdot)$ *such that* $\mathrm{Prob}_{\mathrm{k} \leftarrow \mathsf{Gen}(1^{\mathrm{s}})}[\mathcal{A}(1^{\mathrm{s}}) \in \mathsf{Range}_{\mathrm{s}}(\mathrm{k})] < \nu(\mathrm{s})$.

Most of the well known techniques used to construct a private-key encryption scheme (e.g. using a PRF) can be used to obtain a *special* private-key encryption scheme as well. The major difference is that a special encryption scheme has (in general) ciphertexts longer than a standard encryption scheme.

## 4 Our Protocol $\Pi^{\in}$

In this section we provide the formal description of our protocol $\Pi^{\in} = (\mathsf{S}, \mathsf{R})$ for the *set-membership* functionality $\mathcal{F}^{\in} = (\mathcal{F}^{\in}_{\mathsf{S}}, \mathcal{F}^{\in}_{\mathsf{R}})$ where

$$\mathcal{F}^{\in}_{\mathsf{S}} \colon \big(\{\{0,1\}^{\lambda}\}^M \times (\gamma^0, \gamma^1)\big) \times \{0,1\}^{\lambda} \longrightarrow \bot \qquad \text{and}$$
$$\mathcal{F}^{\in}_{\mathsf{R}} \colon \big(\{\{0,1\}^{\lambda}\}^M \times (\gamma^0, \gamma^1)\big) \times \{0,1\}^{\lambda} \longrightarrow \{\gamma^0, \gamma^1\}$$
$$\big(Y, (\gamma^0, \gamma^1), x\big) \longmapsto \begin{cases} \gamma^1 & \text{if } x \in Y \\ \gamma^0 & \text{otherwise} \end{cases}$$

where $\gamma^0$ and $\gamma^1$ are arbitrary strings and are part of the sender's input. Therefore our scheme protects both $Y$ and $\gamma^{1-b}$, when $\gamma^b$ is received by $\mathsf{R}$.

For the formal description of $\Pi^{\in}$, we collapse the first and the second step showed in the information description of Sect. 2 into a single one. That is, instead of constructing the graph $G$, the sender only computes the keys at level $i$ in order to feed the $i$-th OT execution with the correct inputs. The way in which the keys are computed is the same as the vertices for $G$ are computed, we just do not need to physically construct $G$ to allow $\mathsf{S}$ to efficiently compute the keys. In our construction we make use of the following tools.

1. A protocol $\Pi_{\mathcal{OT}} = (\mathsf{S}_{\mathcal{OT}}, \mathsf{R}_{\mathcal{OT}})$ that securely computes the following functionality

$$\begin{aligned} \mathcal{F}_{\mathcal{OT}} \colon (\{0,1\}^\star \times \{0,1\}^\star) \times \{0,1\} &\longrightarrow \{\perp\} \times \{0,1\}^\star \\ ((c_0, c_1), b) &\longmapsto (\perp, c_b). \end{aligned}$$

2. A symmetric key encryption scheme $\mathsf{Sym} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with efficiently verifiable and elusive range.
3. In our construction we make use of the following function: $\delta \colon i \longmapsto \mathsf{min}\{2^i, |Y|\}$.

This function computes the maximum number of vertices that can appear in the level $i$ of the graph $G$. As discussed before, the structure of $G$ leaks information about $Y$. In order to avoid this information leakage about $Y$, it is sufficient to add some padding to the OT sender's input so that the input size become $|Y|$. Indeed, as observed above, every level contains at most $|Y|$ vertices. Actually, it is easy to see that $\mathsf{min}\{|Y|, 2^i\}$ represents a better upper bound on the number of vertices that the $i$-th level can contain. Therefore, in order to compute the size of the padding for the sender's input we use the function $\delta$.

## 4.1 Formal Description

*Common input:* security parameter $s$ and $\lambda$.
$\mathsf{S}$*'s input:* a set $Y$ of size $M$, $\gamma^0 \in \{0,1\}^s$ and $\gamma^1 \in \{0,1\}^s$.
$\mathsf{R}$*'s input:* an element $x \in \{0,1\}^\lambda$.

**First stage**
1. For i $= 1, \dots, \lambda$, $\mathsf{S}$ computes the sink key $k_i^\star \leftarrow \mathsf{Gen}(1^s)$.
2. $\mathsf{S}$ computes $k_0 \leftarrow \mathsf{Gen}(1^s), k_1 \leftarrow \mathsf{Gen}(1^s)$. For $b = 0, 1$, if $b \notin \mathsf{Prefix}(Y, 1)$ then set $k_b = k_1^\star$[12]. Set $(c_0^1, c_1^1) = (k_0, k_1)$.
3. $\mathsf{S}$ and $\mathsf{R}$ execute $\Pi_{\mathcal{OT}}$, where $\mathsf{S}$ acts as the sender $\mathsf{S}_{\mathcal{OT}}$ using $(c_0^1, c_1^1)$ as input and $\mathsf{R}$ acts as the receiver $\mathsf{R}_{\mathcal{OT}}$ using $x_\lambda$ as input. When the execution of $\Pi_{\mathcal{OT}}$ ends $\mathsf{R}$ obtains $\kappa_1 := c_{x_\lambda}^1$.

**Second stage.** For $i = 2, \dots, \lambda$:
1. $\mathsf{S}$ executes the following steps.
   1.1 Define the empty list $c_0^i$ and for all $t \in \mathsf{Prefix}(Y, i-1)$ execute the following steps.

[12] We observe that if $Y$ is not empty (like in our case) then there exists at most one bit $b$ s.t. $b \in \mathsf{Prefix}(Y, 1)$.

- If $t||0 \in \mathsf{Prefix}(Y, i)$ then compute $k_{t||0} \leftarrow \mathsf{Gen}(1^s)$ and add $\mathsf{Enc}(k_t, k_{t||0})$ to the list $c_0^i$. Otherwise, if $t||0 \notin \mathsf{Prefix}(Y, i)$ then compute and add $\mathsf{Enc}(k_t, k_i^\star)$ to the list $c_0^i$.

1.2 If $|c_0^i| < \delta(i-1)$ then execute the following steps.
  - Compute and add $\mathsf{Enc}(k_{i-1}^\star, k_i^\star)$ to the list $c_0^i$.
  - For $j = 1, \dots, \delta(i-1) - |c_0^i|$ compute and add $\mathsf{Enc}(\mathsf{Gen}(1^s), 0)$ to $c_0^i$.[13]

1.3 Permute the elements inside $c_0^i$.

1.4 Define the empty[14] list $c_1^i$ and for all $t \in \mathsf{Prefix}(Y, i-1)$ execute the following step.
  - If $t||1 \in \mathsf{Prefix}(Y, i)$ then compute $k_{t||1} \leftarrow \mathsf{Gen}(1^s)$ and add $\mathsf{Enc}(k_t, k_{t||1})$ to the list $c_1^i$. Otherwise, if $t||1 \notin \mathsf{Prefix}(Y, i)$ compute and add $\mathsf{Enc}(k_t, k_i^\star)$ to the list $c_1^i$.

1.5 If $|c_1^i| < \delta(i-1)$ then execute the following steps.
  - Compute and add $\mathsf{Enc}(k_{i-1}^\star, k_i^\star)$ to the list $c_1^i$.
  - For $j = 1, \dots, \delta(i-1) - |c_1^i|$ compute and add $\mathsf{Enc}(\mathsf{Gen}(1^s), 0)$ to $c_1^i$.

1.6 Permute the elements inside $c_1^i$.

2. S and R execute $\Pi_{\mathcal{OT}}$, where S acts as the sender $\mathsf{S}_{\mathcal{OT}}$ using $(c_0^i, c_1^i)$ as input and R acts as the receiver $\mathsf{R}_{\mathcal{OT}}$ using $x_{\lambda-i+1}$ as input. When the execution of $\Pi_{\mathcal{OT}}$ ends, R obtains $c_{x_{\lambda-i+1}}^i$.

**Third stage**

1. S executes the following steps.
   - 1.1 Define the empty list $l$.
   - 1.2 For every $t \in \mathsf{Prefix}(Y, \lambda)$ compute and add $\mathsf{Enc}(k_t, \gamma^1)$ to $l$.
   - 1.3 If $|l| < 2^\lambda$ then compute and add $\mathsf{Enc}(k_\lambda^\star, \gamma^0)$ to $l$.
   - 1.4 Permute the elements inside $l$ and send $l$ to R.
2. R, upon receiving $l$ acts as follows.
   - 2.1 For $i = 2, \dots, \lambda$ execute the following steps.
     - For every element $t$ in the list $c_{x_{\lambda-i+1}}^i$ compute $\kappa \leftarrow \mathsf{Dec}(\kappa_{i-1}, t)$. If $\kappa \neq \perp$ then set $\kappa_i = \kappa$.
   - 2.2 For all $e \in l$ compute $\mathsf{out} \leftarrow \mathsf{Dec}(\kappa_\lambda, e)$ and output $\mathsf{out}$ if and only if $\mathsf{out} \neq \perp$.

**Theorem 1.** *Suppose $\Pi_{\mathcal{OT}}$ securely computes the functionality $\mathcal{F}_{\mathcal{OT}}$ and* Sym *is a special private-key encryption scheme, then $\Pi^{\in}$ securely computes $\mathcal{F}^{\in}$.*

We refer the reader to the full version of this work [7] for the formal proof.

[13] In this step, as well as in the step 1.5 of this stage, the function $\delta$ is used to compute the right amount of fake encryption to be added to the list that will we used as input of $\mathsf{R}_{\mathcal{OT}}$.

[14] The following three steps are equal to the previous three steps (1.1, 1.2 and 1.3), the only difference is that $t||1$ is considered instead of $t||0$.

*Round Complexity: Parallelizability of Our Scheme.* In the description of our protocol in Sect. 4.1 we have the sender and the receiver engaging $\lambda$ sequential OT executions. We now show that this is not necessary since the OT executions can be easily parallelized, given that each execution is independent from the other. That is, the output of a former OT execution is not used in a latter execution. For simplicity, we assume that $\Pi_{\mathcal{OT}}$ consists of just two rounds, where the first round goes from the receiver to the sender, and the second goes in the opposite direction. We modify the description of the protocol of Sect. 4.1 as follows. The Step 3 of the *first stage* and step 2 of the *second stage* are moved to the beginning of the *third stage*. When $\mathsf{S}$ sends the last round of $\Pi_{\mathcal{OT}}$, he also performs the step 1 of the *third stage*. Therefore the list $l$ is sent together with the last rounds of the $\lambda$ $\Pi_{\mathcal{OT}}$ executions. Roughly speaking, in this new protocol $\mathsf{S}$ first computes all the inputs $(k_0, k_1, c_0^1, c_1^1, \ldots, c_0^\lambda, c_1^\lambda)$ for the OTs. Then, upon receiving the $\lambda$ first rounds of $\Pi_{\mathcal{OT}}$ computed by $\mathsf{R}$ using as input the bits of $x$, $\mathsf{S}$ sends $\lambda$ second round of $\Pi_{\mathcal{OT}}$ together with the list $l$. We observe that in this case the $\mathsf{S}$'s inputs to the $\lambda$ executions of $\Pi_{\mathcal{OT}}$ can be pre-computed *before* any interaction with $\mathsf{R}$ begins.

## 5 Optimisations and Extensions

*Point and Permute.* In our protocol the receiver must decrypt every ciphertext at every layer to identify the correct one. This is suboptimal both because of the number of decryptions and because encryptions that have efficiently verifiable range necessarily have longer ciphertexts. This overhead can be removed using the standard *point-and-permute technique* [4] which was introduced in the context of garbled circuits. Using this technique we can add to each key in each layer a *pointer* to the ciphertext in the next layer which can be decrypted using this key. This has no impact on security.

*One-Time Pad.* It is possible to reduce the communication complexity of our protocol by using *one-time pad encryption* in the last $\log s$ layers of the graph, in the setting where the output values $\gamma^0, \gamma^1$ are such that $|\gamma^b| < s$. For instance, if the output values are bits (in case we combine our PSM with a GMW-style protocol), then the keys (and therefore the ciphertexts) used in the last layer of the graph only need to be 1 bit long. Unfortunately, since the keys in the second to last layer are used to mask up to two keys in the last layer, the keys in the second to last layer must be of length 2 and so on, which is why this optimisation only gives benefits in the last $\log s$ layer of the graph.

*PSM with Secret Shared Input.* Our PSM protocol produces an output which can be post-processed using other 2PC protocols. It is natural to ask whether it is possible to design efficient PSM protocols that also work on encrypted or secret-shared inputs. We note here that our protocol can also be used in the setting in which the input string $x$ is *bit-wise secret-shared* between the sender and the receiver i.e., the receiver knows a share $r$ and the sender knows a share

$s$ s.t., $r \oplus s = x$. The protocol does not change for the receiver, who now inputs the bits of $r = r_\lambda, \ldots, r_1$ to the $\lambda$ one-out-of-two OTs (instead of the bits of $x$ as in the original protocol). The sender, at each layer $i$, will follow the protocol as described above if $s_i = 0$ and instead swap the inputs to the OT if $s_i = 1$. It can be easily verified that the protocol still produces the correct result and does not leak any extra information.

*Keyword Search.* Our PSM protocol outputs an encryption of a bit indicating whether $x \in Y$ or not. The protocol can be easily modified to output a value dependent on $x$ itself and therefore implement "encrypted keyword search". That is, instead of having only two output strings $\gamma^1, \gamma^0$ representing membership and non-membership respectively, we can have $|Y|+1$ different output strings (one for each element $y \in Y$ and one for non-membership). This can be used for instance in the context where $Y$ is a database containing id's $y$ and corresponding values $v(y)$, and the output of the protocol should be an encryption of the value $v(x)$ if $x \in Y$ or a standard value $v(\perp)$ if $x \notin Y$. The modification is straightforward: instead of using all the keys in the last layer of the graph to encrypt the same value $\gamma^1$, use each key $k_y$ to encrypt the corresponding value $v(y)$ and the sink key (which is used to encrypt $\gamma^0$ in our protocol) to encrypt the value $v(\perp)$.

*PSI from PSM.* We can follow the same approach of PSZ [34,36] to turn our PSM protocol into a protocol for PSI. Given a receiver with input $X$ and a sender with input $Y$ the trivial way to construct PSI from PSM is to run $|X|$ copies of PSM, where in each execution the receiver inputs a different $x$ from $X$ and where the sender always inputs her entire set $Y$. As described above, the complexity of our protocol (as the complexity of the PSM protocol of PSZ) is proportional in the size of $|Y|$, so this naïve approach leads to quadratic complexity $O(|X| \cdot |Y|)$. PSZ deals with this using *hashing* i.e., by letting the sender and the receiver locally preprocess their inputs $X, Y$ before engaging in the PSM protocols. The different hashing techniques are explained and analysed in [37, Sect. 3]. We present the intuitive idea and refer to their paper for details: in PSZ the receiver uses *Cuckoo hashing* to map $X$ into a vector $X'$ of size $\ell = O(|X|)$ such that all elements of $X$ are present in $X'$ and such that every $x'_i \in X'$ is either an element of $X$ or a special $\perp$ symbol. The sender instead maps her set $Y$ into $\ell = |X'|$ small buckets $Y'_1, \ldots, Y'_\ell$ such that every element $y \in Y$ is mapped into the "right bucket" i.e., the hashing has the property that if $y = x'_i$ for some $i$ then $y$ will end up in bucket $Y'_i$ (and potentially in a few other buckets). Now PSZ uses the underlying PSM protocol to check whether $x'_i$ is a member of $Y'_i$ (for all $i$'s), thus producing the desired result. The overall protocol complexity is now $O(\sum_{i=1}^{l} |X'| \cdot |Y'_i|)$ which (by careful choice of the hashing parameters) can be made sub-quadratic. In particular, if one is willing to accept a small (but not negligible) failure probability, the overall complexity becomes only linear in the input size. Since this technique is agnostic of the underlying PSM protocol, we can apply the same technique to our PSM protocol to achieve a PSI protocol that produces encrypted output.

## 6 Applications

The major advantage provided by $\Pi^{\in}$ is that the output of the receiver can be an arbitrary value chosen by the sender as a function of $x$ for each value $x \in Y \cup \{\perp\}$. This is in contrast with most of the approaches for set membership, where the value obtained by the receiver is a fixed value (e.g. 0) when $x \in Y$, or some random value otherwise. We now provide two examples of how our protocol can be used to implement more complex secure set operations. The examples show some guiding principles that can be used to design other applications based on our protocol. Without loss of generality in the following applications only the receiver will learn the output of the computation. Moreover we assume that the size of $X$ and $Y$ is equal to the same value $M$.[15] Also for simplicity we will describe our application using the naïve PSI from PSM construction with quadratic complexity, but using the PSZ approach, as described in Sect. 5, it is possible to achieve linear complexity using hashing techniques. Finally, in both our applications we exploit the fact that additions can be performed locally (and for free) using secret-sharing based 2PC. In applications in which round complexity is critical, the protocols can be redesigned using garbled circuits computing the same functionality, since the garbled circuit can be sent from the sender to the other messages of the protocol. However in this case additions have to be performed inside the garbled circuit.

*Computing Statistics of the Private Intersection.* Here we want to construct a protocol where sender and receiver have as input two sets, $X$ and $Y$ respectively, and want to compute some statistics on the intersections of their sets. For instance the receiver has a list of id's $X$ and that the sender has a list of id's $Y$ and some corresponding values $v(Y)$ (thus we use the variant of our protocol for *keyword search* described in Sect. 5). At the end of the protocol the receiver should learn the average of $v(X \cap Y)$ (and not $|X \cap Y|$). The main idea is the following: the sender and the receiver run $M$ executions of our protocol where the receiver inputs a different $x_i$ from $X$ in each execution. The sender always inputs the same set $Y$, and chooses the $|Y| + 1$ outputs $\gamma_i^y$ for all $y \in Y \cup \{\perp\}$ for all $i = 1, \ldots, M$ in the following way: $\gamma_i^y$ is going to contain two parts, namely an arithmetic secret sharing of the bit indicating whether $x_i \in Y$ and an arithmetic secret sharing of the value $v(y)$. The arithmetic secret sharing will be performed using a modulo $N$ large enough such that $N > M$ and $N > M \cdot V$ where $V$ is some upper bound on $v(y)$ so to be sure that no modular reduction will happen when performing the addition of the resulting shares. Concretely the sender sets $\gamma_i^y = (-u_i^2 + 1 \mod N, -v_i^2 + v(y) \mod N)$ for all $y \in Y$ and $\gamma_i^{\perp} = (-u_i^2 \mod N, -v_i^2 \mod N)$. After the protocol the receiver defines her shares $u_i^1, v_i^1$ to be the shares contained in her output of the PSM protocol, and then both parties add their shares locally to obtain secret sharing of the size of the intersection and of the sum of the values i.e., $U^1 = \sum_i u_i^1$, $V^1 = \sum_i v_i^1$,

[15] We assume this only to simplify the protocol description, indeed our protocol can be easily instantiated when the two sets have different size.

$U^2 = \sum_i u_i^2$, and $V^2 = \sum_i v_i^2$. Now the parties check if $(U^1, U^2)$ is a sharing of 0 and, if not, they compute and reveal the result of the computation $\frac{V^1+V^2}{U^1+U^2}$. Both these operations can be performed using efficient two-party protocols for comparison and division such as the one in [10,42].

*Threshold PSI.* In this example we design a protocol $\Pi^t = (P_1^t, P_2^t)$ that securely computes the functionality $\mathcal{F}^t = (\mathcal{F}^t_{P_1^t}, \mathcal{F}^t_{P_2^t})$ where

$$\mathcal{F}^t_{P_1^t}\colon \{\{0,1\}^\lambda\}^M \times \{\{0,1\}^\lambda\}^M \longrightarrow \perp \text{ and}$$
$$\mathcal{F}^t_{P_2^t}\colon \{\{0,1\}^\lambda\}^M \times \{\{0,1\}^\lambda\}^M \longrightarrow \{\{0,1\}^\lambda\}^\star$$
$$(S_1, S_2) \longmapsto \begin{cases} S_1 \cap S_2 & \text{if } |S_1 \cap S_2| \geq t \\ \perp & \text{otherwise} \end{cases}$$

That is, the sender and the receiver have on input two sets, $S_1$ and $S_2$ respectively, and the receiver should only learn the intersection between these two sets if the size of the intersection is greater or equal than a fixed (public) threshold value $t$. In the case that the size of the intersection is smaller that $t$, then no information about $S_1$ is leaked to $P_2^t$ and no information about $S_2$ is leaked to $P_1^t$. (This notion was recently considered in [16] in the context of privacy-preserving ride-sharing). As in the previous example, the sender and the receiver run $M$ executions of our protocol where the receiver inputs a different $x_i$ from $S_2$ in each execution. The sender always inputs the same set $S_1$, and chooses the two outputs $\gamma_i^0, \gamma_i^1$ in the following way: $\gamma_i^b$ is going to contain two parts, namely an arithmetic secret sharing of 1 if $x_i \in Y$ or 0 otherwise, as well as encryption of the same bit using a key $k$. The arithmetic secret sharing will be performed using a modulus larger than $M$, so that the arithmetic secret sharings can be added to compute a secret-sharing of the value $|S_1 \cap S_2|$ with the guarantee that no overflow will occur. Then, the sender and the receiver engage in a secure-two party computation of a function that outputs the key $k$ to the receiver if and only if $|S_1 \cap S_2| > t$. Therefore, if the intersection is larger than the threshold now the receiver can decrypt the ciphertext part of the $\gamma$ values and learn which elements belong to the intersection. The required 2PC is a simple comparison with a known value (the threshold is public) which can be efficiently performed using protocols such as [14,29].

## References

1. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer and extensions for faster secure computation. In: 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, 4–8 November 2013, pp. 535–548 (2013)
2. Asharov, G., Lindell, Y., Schneider, T., Zohner, M.: More efficient oblivious transfer extensions with security for malicious adversaries. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 673–701. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_26

3. Baldi, P., Baronio, R., Cristofaro, E.D., Gasti, P., Tsudik, G.: Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, 17–21 October 2011, pp. 691–702 (2011)
4. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols (extended abstract). In: Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, Baltimore, Maryland, USA, 13–17 May 1990, pp. 503–513 (1990)
5. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, 30 October–03 November 2017, pp. 1243–1255. ACM (2017)
6. Chor, B., Gilboa, N., Naor, M.: Private information retrieval by keywords. IACR Cryptology ePrint Archive 1998, 3 (1998). Appeared in the Theory of Cryptography Library
7. Ciampi, M., Orlandi, C.: Combining private set-intersection with secure two-party computation. Cryptology ePrint Archive, Report 2018/105 (2018). https://eprint.iacr.org/2018/105
8. De Cristofaro, E., Tsudik, G.: Practical private set intersection protocols with linear complexity. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 143–159. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14577-3_13
9. De Cristofaro, E., Tsudik, G.: Experimenting with fast private set intersection. In: Katzenbeisser, S., Weippl, E., Camp, L.J., Volkamer, M., Reiter, M., Zhang, X. (eds.) Trust 2012. LNCS, vol. 7344, pp. 55–73. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30921-2_4
10. Dahl, M., Ning, C., Toft, T.: On secure two-party integer division. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 164–178. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32946-3_13
11. Falk, B.H., Noble, D., Ostrovsky, R.: Private set intersection with linear communication from general assumptions. IACR Cryptology ePrint Archive 2018, 238 (2018)
12. Freedman, M.J., Ishai, Y., Pinkas, B., Reingold, O.: Keyword Search and Oblivious Pseudorandom Functions. In: Kilian, J. (ed.) TCC 2005. LNCS, vol. 3378, pp. 303–324. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30576-7_17
13. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 1–19. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_1
14. Garay, J., Schoenmakers, B., Villegas, J.: Practical and secure solutions for integer comparison. In: Okamoto, T., Wang, X. (eds.) PKC 2007. LNCS, vol. 4450, pp. 330–342. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71677-8_22
15. Goldreich, O., Micali, S., Wigderson, A.: How to play ANY mental game or a completeness theorem for protocols with honest majority. In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing, New York, USA, pp. 218–229 (1987)
16. Hallgren, P., Orlandi, C., Sabelfeld, A.: Privatepool: privacy-preserving ridesharing. In: IEEE 30th Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, 21–25 August 2017, pp. 276–291 (2017)

17. Hazay, C., Lindell, Y.: Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In: Canetti, R. (ed.) TCC 2008. LNCS, vol. 4948, pp. 155–175. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78524-8_10
18. Hazay, C., Venkitasubramaniam, M.: Scalable multi-party private set-intersection. In: Fehr, S. (ed.) PKC 2017. LNCS, vol. 10174, pp. 175–203. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54365-8_8
19. Huang, Y., Evans, D., Katz, J.: Private set intersection: are garbled circuits better than custom protocols? In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, 5–8 February 2012 (2012)
20. Ion, M., et al.: Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738 (2017)
21. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_9
22. Ishai, Y., Paskin, A.: Evaluating branching programs on encrypted data. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 575–594. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70936-7_31
23. Jarecki, S., Liu, X.: Fast secure computation of set intersection. In: Garay, J.A., De Prisco, R. (eds.) SCN 2010. LNCS, vol. 6280, pp. 418–435. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15317-4_26
24. Kamara, S., Mohassel, P., Raykova, M., Sadeghian, S.: Scaling private set intersection to billion-element sets. In: Christin, N., Safavi-Naini, R. (eds.) FC 2014. LNCS, vol. 8437, pp. 195–215. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45472-5_13
25. Keller, M., Orsini, E., Scholl, P.: Actively secure OT extension with optimal overhead. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9215, pp. 724–741. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47989-6_35
26. Kolesnikov, V., Kumaresan, R.: Improved OT extension for transferring short secrets. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 54–70. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_4
27. Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious PRF with applications to private set intersection. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016, pp. 818–829 (2016)
28. Lindell, Y., Pinkas, B.: A proof of security of yao's protocol for two-party computation. J. Cryptol. **22**(2), 161–188 (2009)
29. Lipmaa, H., Toft, T.: Secure equality and greater-than tests with sublinear online complexity. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013. LNCS, vol. 7966, pp. 645–656. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39212-2_56
30. Meadows, C.A.: A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In: Proceedings of the 1986 IEEE Symposium on Security and Privacy, Oakland, California, USA, 7–9 April 1986, pp. 134–137 (1986)
31. Mohassel, P., Niksefat, S.: Oblivious decision programs from oblivious transfer: efficient reductions. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 269–284. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32946-3_20

32. Nagaraja, S., Mittal, P., Hong, C., Caesar, M., Borisov, N.: BotGrep: finding P2P bots with structured graph analysis. In: Proceedings of the 19th USENIX Security Symposium, Washington, DC, USA, 11–13 August 2010, pp. 95–110 (2010)
33. Orrù, M., Orsini, E., Scholl, P.: Actively secure 1-out-of-$N$ OT extension with application to private set intersection. In: Handschuh, H. (ed.) CT-RSA 2017. LNCS, vol. 10159, pp. 381–396. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52153-4_22
34. Pinkas, B., Schneider, T., Segev, G., Zohner, M.: Phasing: private set intersection using permutation-based hashing. In: 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, 12–14 August 2015, pp. 515–530 (2015)
35. Pinkas, B., Schneider, T., Weinert, C., Wieder, U.: Efficient circuit-based PSI via Cuckoo hashing. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10822, pp. 125–157. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78372-7_5
36. Pinkas, B., Schneider, T., Zohner, M.: Faster private set intersection based on OT extension. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, 20–22 2014, pp. 797–812 (2014)
37. Pinkas, B., Schneider, T., Zohner, M.: Scalable private set intersection based on OT extension. Cryptology ePrint Archive, Report 2016/930 (2016)
38. Rindal, P., Rosulek, M.: Improved private set intersection against malicious adversaries. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10210, pp. 235–259. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56620-7_9
39. Rindal, P., Rosulek, M.: Malicious-secure private set intersection via dual execution. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, 30 October–03 November 2017, pp. 1229–1242. ACM (2017)
40. Shamir, A.: On the power of commutativity in cryptography. In: de Bakker, J., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 582–595. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10003-2_100
41. Tamrakar, S., Liu, J., Paverd, A., Ekberg, J., Pinkas, B., Asokan, N.: The circle game: scalable private membership test using trusted hardware. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, 2–6 April 2017, pp. 31–44 (2017)
42. Toft, T., et al.: Primitives and applications for multi-party computation. Ph.D. thesis, University of Aarhus, Denmark (2007)
43. Yao, A.C.: Protocols for secure computations (extended abstract). In: 23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3–5 November 1982, pp. 160–164 (1982)