Private Computation On Set Intersection With Sublinear Communication

Jonas Janneck[1], Anselme Tueno[1], Jörn Kußmaul[2], Matthew Akram[1]

SAP SE
[1] firstname.lastname@sap.com
[2] j.kussmaul@sap.com

August 31, 2022

Abstract

In this paper, we propose a new protocol for private computation on set intersection (PCI) which is an extension of private set intersection (PSI). In PSI, each party has a private set and both want to securely compute the intersection of their sets such that only the result is revealed and nothing else. In PCI, we want to additionally apply a private computation on the result. The goal is to reveal only the result of such a secure evaluation on the intersection and nothing else. We particularly focus on a client-server setting where the server's set is significantly larger than the client's set and the result of the computation should be revealed only to the client. The protocol aims at a low communication overhead which is sublinear in the server's set size. Such PSI protocols have already been realized using fully homomorphic encryption (FHE). However, they do not allow for private post-processing to enable PCI. There are also protocols enabling PCI which are in addition very fast with respect to the computational overhead. Their drawback is that they have a communication overhead which is at least linear in the larger set. We present a PSI protocol which can be used for arbitrary post-processing without creating a new protocol for every special-purpose PCI functionality. Our construction relies on the evaluation of a branching program using an FHE scheme. Using the properties of an FHE scheme, we build a non-interactive protocol with extendable functionalities. That means, we can not only securely compute the intersection but use the encrypted result to apply further computations without revealing the intersection itself. To the best of our knowledge, this results in the first PCI protocol with communication cost sublinear in the larger set. Compared to previous work, we can reduce the communication by factor 47.

1 Introduction

Imagine the scenario of ad conversion rates used to price online advertisement. In such a scenario, an advertiser is payed by a merchant if someone who has seen an ad buys a product or spends a certain amount of money at the merchant. The advertiser has a list of all the users who have seen the ad and the merchant has a list of all the users who have bought the product. Intersecting both lists yield the users for which the advertiser has to be paid. However, the parties may not want or are not allowed to reveal their whole customer data set [29]. This problem can be solved using Private Set Intersection (PSI) to reveal only the set intersection and nothing else [10, 32, 33, 41].

Depending on the use case, we can extend the setting further to find an optimal solution. For the example of ad conversion rates, the set of the advertiser is much larger than the set of the merchant, i.e. we have unbalanced sets. Moreover, we are basically not interested in the intersection itself but the final revenue that has to be paid. This can depend on the cardinality of the intersection, PSI-CAT [23, 26], or on a weighted sum of each element in the intersection, PSI-Sum [29, 38, 39]. Allowing for any of such extensions is called Private Computation on Set Intersection (PCI) [11, 34].

Furthermore, there is a very broad range of applications for PSI, e.g., contact discovery [19], genome testing [3, 46], or object-level tracking [49].

Unbalanced and Asymmetric PSI. A special case is the client-server setting, where the server has a significantly larger set than the client (unbalanced) and the intersection should be revealed only to the client (asymmetric). Moreover, the server has larger computational power and its set may undergo frequent update, while the client's computational and storage capacity is limited compared to the server. In this paper, we focus on the unbalanced and asymmetric case and refer to the server as the sender and to the client as the receiver. There is a lot of work on PSI focusing on the balanced setting, i.e. with sets of nearly the same size. These protocols yield a solution which is very efficient in terms of computational overhead [41]. However, the communication overhead depends at least linearly on the larger set such that it is not very efficient when applied to the unbalanced case. Hence, there is need for special protocols using the fact that one set is significantly smaller than the other.

Private Computation on Set Intersection. Another line of research considers the scenario of post-processing the output of the PSI computation, i.e. enable Private Computation in Set Intersection (PCI) [11, 34]. To ensure the security of the new protocol, i.e, not revealing any additional information, most generic PSI protocols cannot be used since they reveal the intersection to one of the parties and not only the post-processed result. For an example of why this is not possible, we refer to the related work in the following section. A trivial approach is to create an new protocol for every use case which is a lot of effort and opens the door to new security issues. Another approach is to create

only one PSI protocol enabling PCI and not revealing the intersection to one of the parties. One can then apply any protocol which has already proven to be secure on top of the PSI protocol. Such PSI protocols can therefore be used in a generic way and be built into larger protocols.

Our Contribution. Our goal is to combine sublinear communication with the PCI functionality, i.e. construct a protocol with only sublinear communication in the larger set size which additionally allows for efficient and non-interactive secure computations on the intersection. The idea of our solution is to represent the server's set as a branching program (BP) and evaluate the client's input on the server's set using fully homomorphic encryption (FHE). Instead of sending the encrypted output to the client, it can also be used for further post-processing extensions enabling PCI. This is possible since we do not require the client to apply any further operations (except decryption) on the received output. To improve communication and computation overhead, we propose optimizations to the basic protocol including more efficient branching program evaluations as well as improved PSI computations using different hashing techniques. Furthermore, we present a security analysis proving the security of our protocol against a semi-honest adversary in the standard model. Finally, we implement our protocol and evaluate it with different parameters. To the best of our knowledge, this results in the first PCI protocol with communication sublinear in the larger set.

Structure The remainder of the paper is structured as follows. We review related work in Section 2. Then, we present preliminaries in Section 3 before describing our basic protocol and its algorithms in Section 4. Section 5 contains optimizations to our basic protocol and the full protocol is presented in Section 6. The PCI property and further extensions are discussed in Section 7. Our protocol is analyzed theoretically in Section 8 and practically in Section 9 before Section 10 concludes our work.

2 Related Work

There are a lot of protocols solving the PSI problem. A detailed overview can be found in [41]. They differ in the setting, the tools they are built on and the functionalities they include.

Public-Key PSI. The starting point of PSI protocols, based on public-key cryptography, was built on Diffie-Hellmann key exchange [35]. Later, a protocol based on blind-RSA operations was introduced by De Cristofaro et al. [17]. The proposed public-key operations have been combined with data structures like hash tables [23], bloom filters [18, 32] or cuckoo filters [16] to reduce the computation and communication complexity. However, all these protocols have in common that they use computationally heavy public-key operations (e.g., exponentiations) and still have a communication complexity that is at least linear in the set sizes.

OT-based PSI. OT-based PSI are protocols based on oblivious transfer (OT) [40, 41]. Due to their high use of fast symmetric primitives (so-called OT-Extensions), they are the most efficient protocols with respect to computation time [41]. However, compared to public-key PSI they have a higher communication overhead. Furthermore, they are usually considered in the equally-sized input setting and do not scale well with unequal set sizes. State-of-the-art protocols have a communication complexity of $\mathcal{O}(|X|\log|X|)$ where X is the larger set [40, 41]. Like public-key PSI, most OT-based PSI protocols do not allow for generic extensions of functionalities [11].

PSI based on HE. These protocols are based on the use of a homomorphic encryption (HE) scheme. The idea is to shift the expensive computations to one party. These protocols are well suited for the unbalanced scenario in which one party has computation restrictions such as a small device, i.e. client, and the other party does not, i.e. server. Chen et al. [10] use a fully HE (FHE) scheme to reduce the communication overhead. For a server's set X and a client's set Y, they achieve communication cost linear in the size of the smaller set and only logarithmic in the larger set: $\mathcal{O}(|Y|\log|X|)$. This result has been improved by additionally extending the setting to labeled PSI [9] but keeping the same asymptotic complexity. Cong et al. [13] presented an improvement resulting in communication complexity $\tilde{\mathcal{O}}(|Y|(\log\log|X|)^2)$. All the protocols are based on a homomorphic evaluation of a polynomial representing the server's set. Specifically, they test if an input y leads to 0 when evaluating the polynomial

$$\prod_{x\in X}(y-x).$$

This line of research is the starting point of our work with respect to the communication overhead which is sublinear in the larger set size. However, we can also see that this approach cannot be used for arbitrary PCI because it needs an interaction with the client to determine if the result is 0 or not.

PCI. In circuit-based PSI, generic methods to securely compute any circuit are applied to the PSI problem. A generic method is, for example, Yao's garbled circuit [50]. Protocols based on circuit evaluation were introduced by Huang et al. [28] and later improved by Pinkas et al. [37–39]. They use suitable hashing schemes to speed up their computation by reducing the number of needed secure comparisons. In contrast to previous categories, circuit-based PSI allows for generic extensions of the functionalities, enabling PCI. Of particular interest is the work of Ciampi and Orlandi [11] and a follow-up work of Ma and Chow [34]. Ciampi and Orlandi show why state-of-the-art PSI protocols from OT, e.g., [41], cannot be combined with post-processing because they reveal the intersection to one of the parties. Moreover, they present a new protocol based on a graph structure which is evaluated using OT. They produce an "encrypted" output that can directly be used by the client for further extensions based on garbled circuits, secret-sharing [45], or HE. They achieve a small computation overhead

since they highly rely on symmetric key operations. However, a drawback of these protocols (which were built for the balanced setting) is the communication overhead which is at least linear in the larger set.

3 Preliminaries

In this section, we define homomorphic encryption, security definitions and the PSI functionality. Moreover, we shortly present how to securely evaluate a branching program.

3.1 Homomorphic Encryption

Homomorphic encryption (HE) allows computations on ciphertexts by generating an encrypted result whose decryption matches the result of a function on the plaintexts [6, 24].

HE Algorithms. An HE scheme consists of the following algorithms:

- $\mathsf{pk}, \mathsf{sk}, \mathsf{ek} \leftarrow \mathsf{KGen}(\lambda)$: This probabilistic algorithm takes a security parameter λ and outputs public, private, and evaluation keys pk, sk, and ek.
- $\mathsf{c} \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$: This algorithm takes pk and a message m and outputs a ciphertext c.
- $\mathsf{c} \leftarrow \mathsf{Eval}(\mathsf{ek}, f, \mathsf{c}_1, \ldots, \mathsf{c}_n)$: This algorithm takes ek, an n-ary function f and n ciphertexts $\mathsf{c}_1, \ldots \mathsf{c}_n$ and outputs a ciphertext c.
- $\mathsf{m}' \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{c})$: This deterministic algorithm takes sk and a ciphertext c and outputs a message m'.

We will use $[\![m]\!]$ as a shorthand notation for a ciphertext if $[\![m]\!]$ correctly decrypts to m. Further, we require IND-CPA security and the following correctness conditions. Given any set of n plaintexts $\mathsf{m}_1, \ldots, \mathsf{m}_n$, any keys pk, sk, ek, it muss hold:

- $\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, \mathsf{m}_i)) = \mathsf{Dec}(\mathsf{sk}, [\![\mathsf{m}_i]\!]) = \mathsf{m}_i \quad \forall i = 1, \ldots, n,$
- $\mathsf{Dec}(\mathsf{sk}, \mathsf{Eval}(\mathsf{ek}, f, [\![\mathsf{m}_1]\!], \ldots, [\![\mathsf{m}_n]\!])) = f(\mathsf{m}_1, \ldots, \mathsf{m}_n).$

There are two basic evaluation functions of particular interest, namely addition and multiplication of two ciphertexts or one ciphertext with one plaintext. For simplicity, we introduce special notations for these operations. By $\boxplus$, we denote $\mathsf{Eval}(\mathsf{ek}, f, [\![m_1]\!], [\![m_2]\!])$ with $f(m_1, m_2) = m_1 + m_2$. By $\boxdot$, we denote $\mathsf{Eval}(\mathsf{ek}, f, [\![m_1]\!], [\![m_2]\!])$ with $f(m_1, m_2) = m_1 \cdot m_2$. We use the same symbols to denote additions or multiplications with plaintext constant, e.g., $[\![m_1]\!] \boxplus m_2$ or $[\![m_1]\!] \boxdot m_2$.

3.2 Security

We define security in the two-party computation model [27] and start with some useful definitions.

Definition 3.1 (Probability Ensemble) *We define a* probability ensemble $X = X(a,\lambda)_{a,\lambda}$ *as an infinite sequence of random variables indexed by* $a \in \{0,1\}^*$ *and* $\lambda \in \mathbb{N}$.

Definition 3.2 (Computational Indistinguishability) *Let* $a \in \{0,1\}^*$ *and* $\lambda \in \mathbb{N}$. *Two probability ensembles* $X = X(a,\lambda)_{a,\lambda}$ *and* $Y = Y(a,\lambda)_{a,\lambda}$ *are* computationally indistinguishable*, or* $X \stackrel{c}{\equiv} Y$*, if for every non-uniform PPT algorithm* D *there exists a negligible function* μ *such that for every* a *and* λ *it holds*

$$|Pr[D(X(a,\lambda)) = 1] - Pr[D(Y(a,\lambda)) = 1]| \leq \mu(\lambda).$$

Both parties execute a protocol π on inputs x, y and the i-th party obtains output

$$\textsc{output}_i^{\pi}(x,y,\lambda).$$

The overall output is the tuple

$$\textsc{output}^{\pi} = (\textsc{output}_1^{\pi}(x,y,\lambda), \textsc{output}_2^{\pi}(x,y,\lambda)).$$

Tuple $\textsc{view}_i^{\pi}(x,y,\lambda)$ describes the view of party i during the execution of protocol π on inputs x and y with security parameter λ. This is the party's input and all received messages.

Definition 3.3 (Semi-Honest Security) *Let* $f = (f_1, f_2)$ *be a deterministic functionality. A protocol* π securely computes f in the presence of semi-honest adversaries *if there exists PPT algorithms* $\mathcal{S}_1, \mathcal{S}_2$ *and a negligible function* μ *such that*

$$Pr[\textsc{output}^{\pi}(x,y,\lambda) \neq f(x,y)] \leq \mu(\lambda) \quad \forall x,y,\lambda$$

and

$$\{\mathcal{S}_1(1^{\lambda}, x, f_1(x,y))\}_{x,y,\lambda} \stackrel{c}{\equiv} \{\textsc{view}_1^{\pi}(x,y,\lambda)\}_{x,y,\lambda}$$

$$\{\mathcal{S}_2(1^{\lambda}, y, f_2(x,y))\}_{x,y,\lambda} \stackrel{c}{\equiv} \{\textsc{view}_2^{\pi}(x,y,\lambda)\}_{x,y,\lambda}$$

where $|x| = |y|$.

3.3 Private Set Intersection

The PSI protocol consists of a server (or sender) holding a set X and a client (or receiver) holding a set Y. We assume that both sets consist of μ–bit strings and $|X|$, $|Y|$ are publicly known with $|X| >> |Y|$. The ideal functionality $\mathcal{F}_{\mathsf{PSI}}$ takes X from the sender and Y from the receiver. It computes and outputs $X \cap Y$ to the receiver and nothing to the sender, as described in Figure 1.

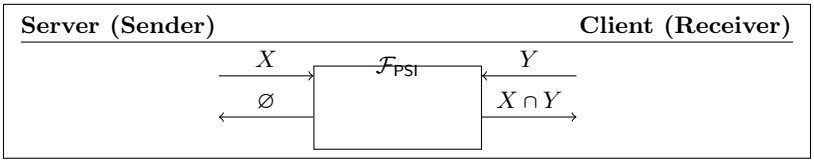


Figure 1: Illustration of the PSI Functionality

3.4 Secure Branching Program Evaluation

Our protocol relies on a branching program (BP) that is represented as a tree. We use the notation from the work of Janneck et al. [30]. We therefore start by defining our data structure and shortly present the secure evaluation.

3.4.1 Data Structure

The data structure is a binary tree consisting of inner nodes and terminal nodes. Each inner node has two child nodes and terminal nodes have no child nodes. There is a node with no parent node that is called root node. Let v be a node in the tree. We define a node data structure Node consisting of the following attributes:

- v.parent: a value representing the pointer to the parent node,
- v.left: a value representing the pointer to the left child node,
- v.right: a value representing the pointer to the right child node,
- v.lEdge: a bit representing the *edge label* to the left child node,
- v.rEdge: a bit representing the *edge label* to the right child node,
- v.cLabel: a value representing a *node label*,
- v.level: an integer representing the *node level* in the tree,
- v.cost: an integer representing the cost on the path from the root.

The pointer to the parent node v.parent is initially null and points to the respective parent node when the child node is created. This pointer remains null for the root node. The pointers to the child nodes v.left, v.right are initially null, and point to the respective nodes if they are created. The edge labels to the child nodes v.lEdge, v.rEdge are 0 on the left and 1 on the right. The node label v.cLabel is 0 or 1 for terminal nodes and undefined for inner nodes. The level v.level is 1 for the root node, 2 for the child nodes of the root and so on. The cost attribute v.cost is computed during tree evaluation.

Algorithm 1 Evaluating Nodes by Computing Decision Bits

```
1:  function EvalNodes(root, ⟦ȳ⟧)
2:      let Q be a new queue
3:      Q.enqueue(root)
4:      parse ⟦ȳ⟧ to ⟦y[1]⟧, ..., ⟦y[μ]⟧
5:      while Q.empty() = false do
6:          v ← Q.dequeue()
7:          if v.left ≠ null then
8:              ⟦v.left.cost⟧ ← ⟦y[v.level]⟧ ⊞̄ 0
9:              Q.enqueue(v.left)
10:         if v.right ≠ null then
11:             ⟦v.right.cost⟧ ← ⟦y[v.level]⟧
12:             Q.enqueue(v.right)
```

3.4.2 Algorithms

Computing Decision Bits. Let $Y = \{y_1, \ldots, y_{N_Y}\}$ be the set of the client. The client sends each input in Y bitwise encrypted to the server. For each $y \in Y$, let $\bar{y} = y[1], \ldots, y[\mu]$ be the corresponding bit string and $[\![\bar{y}]\!] = [\![y[1]]\!], \ldots, [\![y[\mu]]\!]$ the corresponding ciphertexts. The server computes the decision bits at each inner node v by comparing each $[\![y[i]]\!]$ against the edge labels of node v. This comparison is a bit equality test that returns $[\![1]\!]$ if the two bits are equal and $[\![0]\!]$ otherwise. It is therefore implemented using the boolean XNOR gate. We denote the encrypted XNOR gate by $\overline{\boxplus}$. Note that we can neglect the XNOR gate on the right branch since the branch is labeled with 1. The computation of decision bits is illustrated in Algorithm 1. The algorithm uses a queue with enqueue and dequeue functions which add and remove elements. We can reduce the complexity by computing the decision bit only once per level and use it for any node at this level. This minimizes the number of evaluations to the depth of the tree.

Aggregating Decision Bits. For each leaf node v, the server aggregates the comparison bits along the path from the root to v. This is done using homomorphic multiplication of the decision bits. The aggregated result is stored at the leaf node of the corresponding path. We implement it by using a queue and traversing the tree in BFS order as illustrated in Algorithm 2.

Finalizing. After aggregating the decision bits along the paths to the leaf nodes, each leaf node v stores either $[\![v.\mathsf{cost}]\!] = [\![0]\!]$ or $[\![v.\mathsf{cost}]\!] = [\![1]\!]$. Moreover, there is a unique leaf with $[\![v.\mathsf{cost}]\!] = [\![1]\!]$ and all other leaves have $[\![v.\mathsf{cost}]\!] = [\![0]\!]$. Then, the server aggregates the costs at the leaves by computing for each leaf v the value $[\![v.\mathsf{cost}]\!] \boxdot v.\mathsf{cLabel}$ and summing up the results of all leaves. This computation is illustrated in Algorithm 3. Note that we only need a constant multiplication in step 4 since the server's labels are stored as plaintexts.

Algorithm 2 Evaluating Path by Aggregating Decision Bits

```
1:  function EvalPaths(root)
2:      let Q be a queue
3:      let leaves be a queue
4:      Q.enqueue(root)
5:      while Q.empty() = false do
6:          v ← Q.dequeue()
7:          if v.left ≠ null then
8:              [[v.left.cost]] ← [[v.left.cost]] ⊡ [[v.cost]],
9:              if v.left.isLeaf() then
10:                 leaves.enqueue(v.left)
11:             else
12:                 Q.enqueue(v.left)
13:         if v.right ≠ null then
14:             [[v.right.cost]] ← [[v.right.cost]] ⊡ [[v.cost]],
15:             if v.right.isLeaf() then
16:                 leaves.enqueue(v.right)
17:             else
18:                 Q.enqueue(v.right)
19:     return leaves
```

Algorithm 3 Evaluating Leaves by Summing up the Costs at Leaf Nodes

```
1:  function EvalLeaves(leaves)
2:      [[b]] ← [[0]]
3:      for each v ∈ leaves do
4:          [[b]] ← [[b]] ⊞ ([[v.cost]] ⊡ v.cLabel)
5:      return [[b]]
```

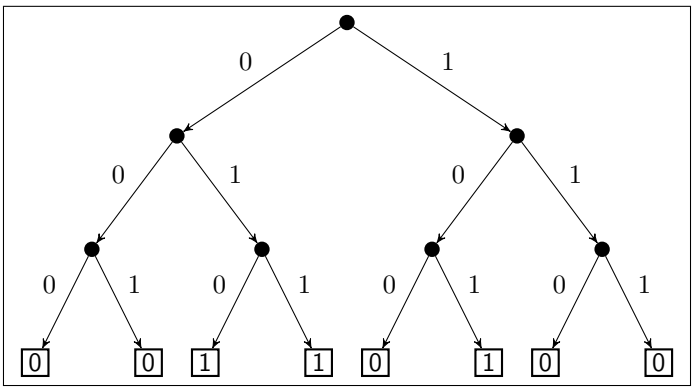


Figure 2: Binary Tree for $X = \{2, 3, 5\}$

4 Our Basic Protocol

We now describe how to use the BP evaluation to implement the PSI functionality.

Initialization. The Initialization consists of a one time key generation. The client generates an appropriate triple $(\mathsf{pk}, \mathsf{sk}, \mathsf{ek})$ of public, private and evaluation keys for a homomorphic encryption scheme. Then, the client sends $(\mathsf{pk}, \mathsf{ek})$ to the server. For each computation, the client just encrypts its input and sends it to the server. As explained before, the actual computation on the binary tree is done only by the server. The following steps describe the computation at the server starting by creating the binary tree.

Creating the BP. Let $X = \{x_1, \ldots, x_{|X|}\}$, where all inputs have bitlength μ. The server starts by creating a binary tree representing X. The basic idea consists of creating a binary tree representing all bit strings of length μ. Then, each leaf that represents an input in X is labeled with 1 (i.e. $v.\mathsf{cLabel} = 1$), otherwise the leaf is labeled with 0 (i.e. $v.\mathsf{cLabel} = 0$). Finally, we can prune all subtrees labeled with the same bit. That is, if an inner node v has two child nodes labeled with the same bit b,we remove the child nodes of v from the tree and transform v into a leaf node labeled with b, (i.e. $v.\mathsf{cLabel} = b$). This BP creation is denoted by CreateTree.

Note that at this stage the computation is done on the plaintext representation of X and is therefore very fast. As an example, assume that $\mu = 3$ and $X = \{2, 3, 5\}$, then the tree in Figure 2 represents the binary tree of X before pruning. Finally, Figure 3 illustrates the pruned tree.

Putting It Together. As illustrated in Protocol 4, the whole computation is performed by the server. The server first creates a tree representation of its input X.Then, for each $y \in Y$ the client sends the encrypted bit representation $[\![\bar{y}]\!]$

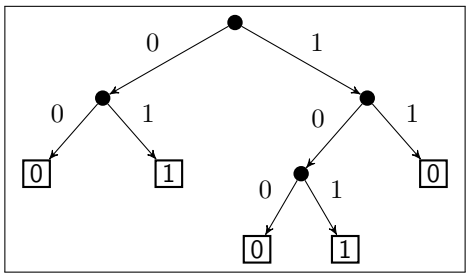


Figure 3: Pruned Binary Tree for $X = \{2, 3, 5\}$

and the server sequentially evaluates nodes (Algorithms 1), paths (Algorithm 2), and leaves (Algorithm 3) as described in Section 3.4. The server sends the resulting ciphertext, $[\![b_y]\!]$ encrypting either 1 or 0, to the client. The client decrypts and learns the result: if $b_y = 1$ then y is in the intersection, otherwise y is not in the intersection.

5 Extensions and Optimization

All of the optimization techniques presented in [30] can be used in our scenario. In this section, we present some additional optimizations useful for the PSI functionality and applicable to our basic protocol.

5.1 SIMD

Many commonly used FHE schemes are defined over a number field over a polynomial, $\mathbb{Z}_2[X]/F$. Assume this polynomial to be $F(X) \in \mathbb{Z}_2[X]$ of degree N. We further assume that $F(X)$ splits into s irreducible and distinct polynomials of degree $d = \frac{N}{s}$, i.e.

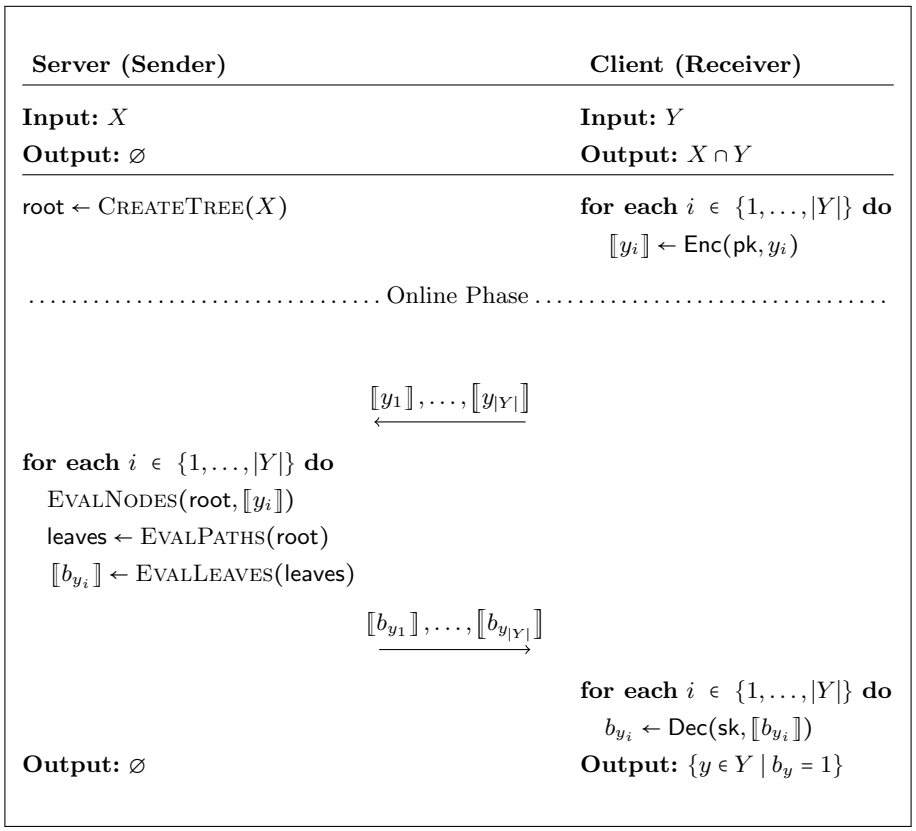
$$F(X) = \prod_{i=1}^{s} F_i(X).$$

The encryption scheme is based on plaintext space $\mathbb{Z}_2[x]/F$. Using the Chinese remainder theorem, this yields an isomorphism between the plaintext space and the factors above, the so called slots

$$\mathbb{Z}_2[X]/F \cong \mathbb{Z}_2[X]/(F_1) \otimes \cdots \otimes \mathbb{Z}_2[X]/(F_s) \cong \mathbb{Z}_{2^d} \otimes \cdots \otimes \mathbb{Z}_{2^d}.$$

The isomorphism can be used to encode multiple inputs into each of these slots and encrypt it accordingly. We can further apply slot-wise operations on the ciphertexts and obtain SIMD (single input multiple data) operations. This procedure is called ciphertext packing or just SIMD. More details can be found in the work of Smart and Vercauteren [47].

A straightforward optimization is to encode multiple client's inputs into one chiphertext and evaluate the protocol once. This reduces the computation as



Protocol 4: The Basic Protocol

well as the communication overhead depending on the slot size. For a slot size of s, the overhead is reduced by factor $\frac{|Y|}{\lceil |Y|/s \rceil} \approx s$.

5.2 Non-binary Trees

One generalization approach of [30] is to rely on an m-ary tree instead of a binary tree. To this end, we have to parse the elements of the sets to base m and create a tree with m children for each inner node. The client has to encrypt values from $\{0, \ldots, m-1\}$ which reduces the number of ciphertexts being sent. Moreover, the computation time on the server can be reduced as well since the server can parallelize the execution of independent sub-trees. However, the evaluation becomes more complex. One technique evaluates a polynomial on each of the branches. More specifically for the i-th node and the j-th branch:

$$P_j([\![y]\!]) = \frac{\prod_{k=0, k\neq j}^{m-1}([\![y]\!] - k)}{c_j},$$

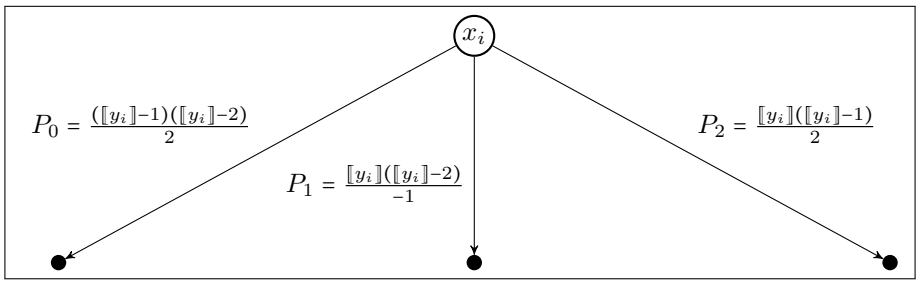


Figure 5: Example of polynomial evaluation for $m = 3$ at node x_i

for a constant

$$c_j = \prod_{k=0,k\neq j}^{m-1} (j-k).$$

In the offline phase we can rewrite the polynomials to obtain the coefficient form, i.e. an array of n coefficients starting with the highest degree x^{m-1}:

$$(c_{m-1}, \ldots, c_0).$$

An example is presented in Figure 5.

In the online phase, we only need the powers of y which might be computationally expensive. Then, we can use the coefficient forms to evaluate the polynomials by using constant multiplications and additions.

Overall effort per tree level (online phase):

- Powers from $y^2, y^3, \ldots, y^{m-1}$ (ciphertext powers),
- $m \cdot (m-2) = m^2 - 2m$ constant multiplications,
- $m \cdot (m-2) + 1 = m^2 - 2m + 1$ additions (+1 since the first polynomial has a constant term).

We can further decrease the computational costs. So far, we implicitly assumed the computation to be done over some field (we assumed c_j to have an inverse element). If we introduce some further constraints, some quite practical consequences occur. For example, the output of the polynomial is either 0 or 1 and the input is at most m. Then, it is sufficient to compute over $\mathbb{Z}_m$ which is a field if we assume m to be prime.[1] The improvement can be illustrated by considering polynomial P_0 which simplifies to

$$P_0(y) = (y^{m-1} - 1) \cdot c_0^{-1} \mod m,$$

which saves many computations.

Finally, we can introduce a trade-off between communication and computation overhead. The most expensive operations are ciphertext-ciphertext multiplications. For evaluating the polynomials for multiple branches, these are the

[1] Otherwise we could take the smallest prime larger than m.

ciphertext powers. To save some computation time, the client could precompute these powers. This can be done in the offline phase and especially on plaintexts before the encryption step.

Note that we can phrase our approach as a secure search in a trie [22]. If we push this variant to extremes, we could obtain a "tree" of depth 1 and have to evaluate polynomials of a high degree. This results in previous PSI work based on HE [9, 10, 13]. The work of Cong et al. [13] introduces very efficient techniques to compute these polynomials by using special powers precomputed by the client. They combine the special powers in different ways to obtain all powers needed to evaluate the polynomials. The same procedure could be applied to improve our extension.

5.3 Partition

To improve the computational overhead we can apply a partition on the server's set to allow for parallelization. The server can partition its set $X = \bigcup_{i=1}^{n} X_i$ with $X_i \bigcap X_j = \varnothing$ for $i \neq j$. We obtain

$$Y \bigcap X = Y \bigcap (\bigcup_i X_i) = \bigcup_i (Y \bigcap X_i).$$

Thus, we can apply an intersection procedure on each subset and aggregate the result. Note that we get a boolean result for every $y \in Y$ indicating if y is in X. Since the subsets X_i are disjoint, y can be in at most one subset and we can just add up the results to obtain the overall result.

6 Our Full Protocol

Using hashing schemes, we can improve the computation time significantly. Instead of comparing each element of Y with each element of X, we hash the values to multiple bins and only compare elements in the same bin. The application of hashing schemes for PSI protocols is well established [9, 10, 13, 37, 40, 41]. The most common schemes are simple hashing, and Cuckoo Hashing or its variants [36]. Another variant is permutation-based hashing, which was first used for PSI by Pinkas et al. [37]. It can also be combined with Cuckoo Hashing.

6.1 General Construction

We start by describing the components of our full protocol as a general construction which is then instantiated by two variants.

Cuckoo Hashing. The basic idea of optimizing the protocol via hashing is based on the work of Chen et al. [10] and described in the following. The client builds a hash table with β bins of at most one element using Cuckoo Hashing with h hash functions. The number of bins should be sufficient such that a hashing failure occurs with negligible probability. The introduction of

a constant-size stash can reduce the failure probability. The server hashes its elements into β bins using simple hashing and all h hash functions. We have to apply all the hash functions since the server does not know which hash function was used to hash a client's element into a specific bin. Note that the bin size of the server will be much larger than one.

Using this construction, we can correctly compute the intersection by computing the intersection of each bin. That is because the server used all hash functions. If an element of the client is hashed to bin B_i, the same element is also hashed to the same bin if it occurs in the server's set. However, the client should pad all the empty bins with a dummy element to prevent any leakage. If we use a stash, all elements of the stash have to be compared to all the server's elements. Depending on the construction, this can be more efficient than choosing larger Cuckoo Hashing parameters. In our case, a stash would lead to a significant overhead using the described procedure. That is why we must rely on larger Cuckoo parameters to reduce the collision probability.

Packing the Client's and Server's Elements. The idea of the packing technique is to use only one or a few evaluations to evaluate all the elements at the same time. To this end, we want to apply the SIMD technique from Section 5.1 on all the bins. The client's side is straightforward since we bound the client's bin size at one and can therefore pack each element into one slot. Hence, we can evaluate multiple inputs on one tree but we need to evaluate multiple inputs on multiple trees since we have different elements in the server's bins. We can pack the server's elements as well but with some slight modifications. Note that the main evaluation of our protocol is generic, i.e. independent of the actual underlying elements to be intersected. In fact, we can evaluate the branching program except the leaf evaluation for all the inputs and apply the leaf aggregation using packed elements. In more detail, we encode multiple leaf node labels into one Node.cLabel and evaluate Algorithm 3 using a SIMD multiplication. That allows the application of multiple inputs on multiple programs and hence the intersection on each bin with only one program evaluation.

As in the case of packed inputs, we have an additional restriction. The number of elements per bin may not exceed the number of slots of a ciphertext. This can be a problem for larger server sets. It can be avoided by choosing a larger number of bins and compute on batches in the size of ciphertext slots but that requires further evaluation procedures. In the following we present a more efficient variant. We can batch the server's elements such that each batch fits into the encoding of one ciphertext. The evaluation of the tree can then be done once and the leaf evaluation is done for every batch separately. The results of the batches are aggregated using an Xor operation. We can slightly change the data representation and let cLabel be a vector such that every entry encodes as many elements as there are available slots. The changes to Algorithm 3 are presented in a modified variant, Algorithm 4. To keep our notation correct, we must also adjust the tree generation algorithm to allow for multiple labels and hence represent multiple trees. We denote this algorithm by CreateTreeMT.

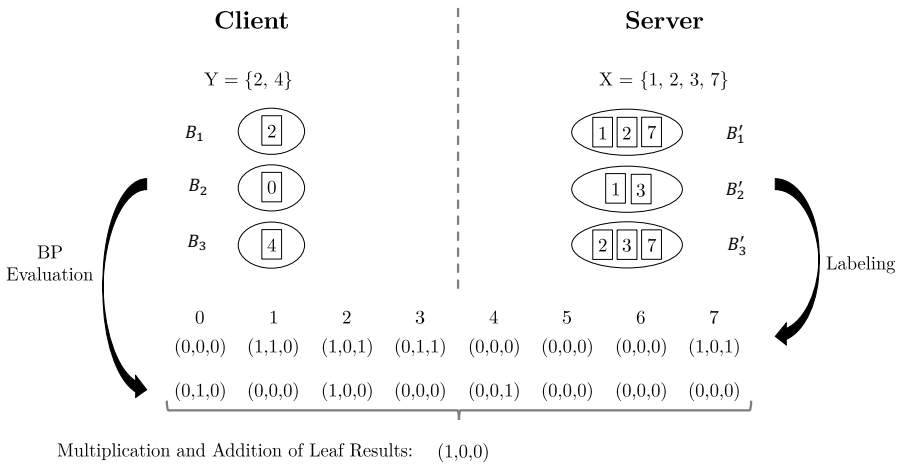


Figure 6: Graphical Representation of Example 6.1

Algorithm 4 Evaluating Leaves for Multiple Trees

```
1: function EvalLeavesMT(leaves)
2:     [[b]] ← [[0]]
3:     for each v ∈ leaves do
4:         for each label ∈ v.cLabel do
5:             [[b]] ← [[b]] ⊞ ([[v.cost]] ⊡ label)
6:     return [[b]]
```

Example 6.1 *Let us consider the following example which is graphically presented in Figure 6. The client holds set* $Y = \{2, 4\}$ *and the server holds set* $X = \{1, 2, 3, 7\}$. *They hash into three bins and the client obtains via Cuckoo Hashing with two hash functions:* $B_1 = \{2\}, B_2 = \{\delta\}, B_3 = \{4\}$ *where the second bin was empty and filled with a dummy element* δ. *For this example, we assume the dummy element to be* 0. *The server uses simple hashing and hashes every element using both hash functions. It obtains the following bins* $B'_1 = \{1, 2, 7\}, B'_2 = \{1, 3\}, B'_3 = \{2, 3, 7\}$ *Then, the server builds a single tree in which the leaf labels are arrays of size three since we have three bins. Position* i *in the array indicates if the respective element is in bin* B'_i. *We obtain* $(0, 0, 0), (1, 1, 0), (1, 0, 1), (0, 1, 1), (0, 0, 0), (0, 0, 0), (0, 0, 0), (1, 0, 1)$. *In the online phase, the client send a packed input, i.e. a bit representation of* $(2, 0, 4)$. *Evaluating the BP, we obtain the results for the inputs. Multiplying with the server's labels and summing up yields the bin-wise result.*

Since the batching builds a partition on the bin's elements (the elements in each bin are still a set), the result is still correct. For the correctness of computing the intersection on partitions, see Section 5.3. A drawback of the hashing

procedure is that we limit the usability or the impact of some optimization techniques from the previous section. We have to keep a generic structure where all the server's element can be represented. Tree optimizations like pruning can only be applied if the requirements are fulfilled for all the elements we are evaluating the BP on. For the example above, we can only prune leaves 4 and 5 whereas in the original representation of the set the graph structure can be optimized further. For a larger set and especially for a larger number of bins, the probability of reductions decreases significantly.

Size of Elements. For some real world applications it might be necessary to apply the protocol to any data representation such as phone numbers, passwords or arbitrary files. Hashing the input elements solves the problem of different data representations since we can intersect the sets of hashes. Using standard hashing schemes, we obtain outputs of 128 or 256 bit which would lead to very inefficient protocols since our complexity is based on the data representation, i.e. the length of elements. However, we can truncate the hash values to σ bit since we are only interested in matching values and the probability of a collision is still negligible for a smaller domain. For the probability of a collision to be smaller than 2^{-40}, we need the bit-length to fulfill $40 + \log|X| + \log|Y| \leq \sigma$ [9]. Note that the smaller hashing domain does not have any security implications since we compare encrypted hash values. Instead of sets X and Y, we now compare hashed sets $\tilde{X}$ and $\tilde{Y}$.

A graphical presentation of the procedure can be found in Figure 7. It describes the PSI computation for hashed sets $\tilde{Y}$ and $\tilde{X}$. The client uses Cuckoo Hashing into β bins $B_1, \ldots, B_\beta$ and the server uses simple hashing into the same number of bins $B'_1, \ldots, B'_\beta$. We use $\square$ to denote an element of a bin. The term "Packed PSI" refers to an application of the packed protocol as described before.

The general construction is illustrated in Protocol 9. We assume the following common inputs: $r = \left\lceil \frac{\beta}{s} \right\rceil$ the number of batched values, $H_1, \ldots, H_h$ the hash functions for Cuckoo Hashing, and dummy element δ. As subroutines we use SimpleH() and CuckooH() to denote the simple hashing or Cuckoo hashing procedure returning a two- or one-dimensional table. Ciphertext packing or SIMD as described in Section 5.1 is represented by two subroutines. To encode multiply values into one plaintext, we use the routine Encode(). The inverse operation is denoted by Decode().

The evaluation of σ-bit values and hence the evaluation of a tree of depth σ can be further optimized. We have two variants to reduce the representation size. One is a deterministic variant (Section 6.2) using permutation-based hashing and the other one is a probabilistic variant (Section 6.3) using universal hashing.

6.2 Deterministic Variant

To improve the solution, we can use permutation-based hashing (PBH). PBH reduces the bit-length of elements that are stored in the bins [1]. This improvement suits our protocol very well since our comparison overhead mainly depends on the bit-length.

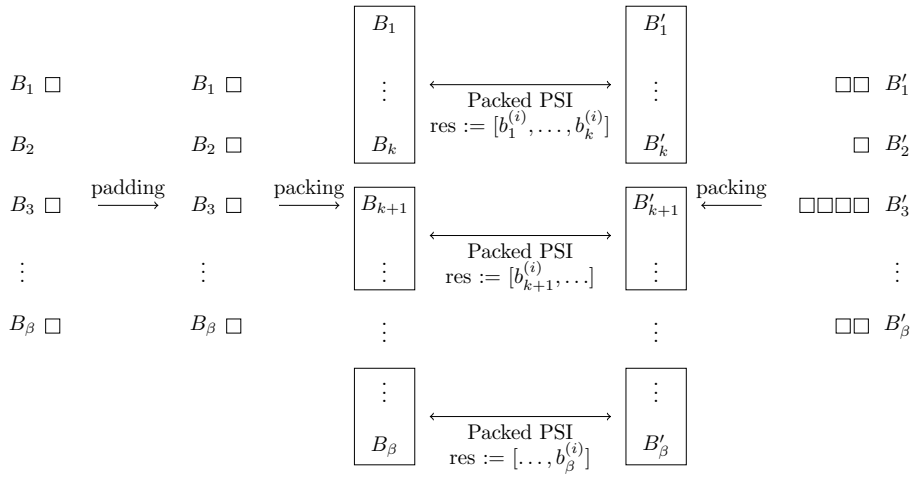


Figure 7: Schematic PSI Computation: On the left side, the client uses cuckoo hashing to hash the set $\tilde{Y}$ into β bins. On the right side, the server uses simple hashing to hash the set $\tilde{X}$ into β bins.

For example, if we consider 32-bit integers and choose $\beta = 2^{20}$ bins (see [37]), the elements in the bins have only 12 bits. We can then apply our PSI protocol for each bin and rely on comparisons of 12-bit values instead of 32 bits.

PBH ensures that two hash values which are equal and stored in the same bin represent the same value. The Cuckoo Hashing construction from the previous section also ensures the correctness of the comparison. However, combining the two techniques yields a correctness issue. The correctness of PBH is based on using the same hash function but in Cuckoo Hashing we are using multiple hash functions. Consider the following example: Let $x \neq x'$ be two elements of length μ with $x_R = x'_R$. If we apply different hash functions H_1, H_2 on x_R and x'_R, we probably obtain $H_1(x_R) \neq H_2(x'_R)$. Hence, it might also hold $x_L \oplus H_1(x_R) = x'_L \oplus H_2(x'_R)$ and they are stored in the same bin even though they are different elements.

We overcome this problem by appending a hash identifier on the element stored in the bin. Thus, we only recognize elements which are the same if they were hashed using the same hash function. For h hash functions, this increases the size of elements by $\log(h)$ bits, which again increases the depth of the BP. However, this increase is constant-size and for our practical implementations it is much less significant than the benefits of PBH which depends on the number of bins.

6.3 Probabilistic Variant

A drawback of the general construction is that the computation time depends exponentially on the domain of the server's values. In the previous section,

we restricted the domain by using permutation-based hashing but it might still be much larger than the number of server elements. Therefore, we introduce a method to limit the domain to approximately the number of elements we are evaluating on. As a result, the computation time does not depend on the representation of elements anymore but on the server's set size.

Bloom Filter. The presented method has some similarities with bloom filters. A bloom filter [4] is a space-efficient but probabilistic data structure. It can represent a set of elements and one can check if an element is in the set via a bloom filter query. A bloom filter of m bits is initialized by m 0-values. It is filled by the following procedure. Hash an element x via k hash functions $H : \{0,1\}^* \to \{1,\ldots,m\}$ to obtain k positions in the domain $\{1,\ldots,m\}$. Then, set the bits on these positions to 1. To query for an element, one has to hash the element using the same k hash functions and check if all the positions are 1. Hence, we might obtain false positive results if some of the positions were coincidentally set by different elements. Tuning the size of the filter and the number of hash functions can bound the probability of false positives. Bloom filters, or an alternative called Cuckoo Filters [20], have been used in the context of PSI before [31, 43]. However, these techniques are not able to reduce the domain into a range which is approximately the number of elements in the bins. We construct the following technique.

Our Construction. We keep the notation from the bloom filter description above. Our goal is to choose a small m since our evaluation depends on the size of the elements compared. We hash as in Bloom Filters but build a BP using the hash positions. Let us first consider the approach for one bin. Let S be the number of elements in that bin. We choose m to be approximately the same as the number of elements S inserted and insert only the position for one hash function. For a random function, the probability of a collision with only one element in the bin is obviously $\frac{1}{m}$ and the probability of no collision $1 - \frac{1}{m}$. The probability for no collision with S elements in the bin is therefore

$$\left(1 - \frac{1}{m}\right)^S.$$

Taking the complementary probability, we obtain the collision probability for the whole bin:

$$1 - \left(1 - \frac{1}{m}\right)^S. \tag{1}$$

Depending on the chosen parameters, this might be too large and would lead to a high false positive rate. However, if we apply the procedure multiple times, we can reduce the collision probability and therefore the false positive rate while the domain m on which we evaluate our comparison stays small. To apply the procedure multiple times means to evaluate the procedure on different hash values.

As for bloom filters, we choose k hash functions and apply the procedure k times. The BP is then evaluated on the hashed values and the results are multiplied. Thus, only if all k results indicate that the element is in the intersection, it will be accepted. We only obtain a false positive result if there are collision with all k hash functions. The aggregation of results increases the multiplicative depth by $\log k$.

This variant is depicted in Protocol 10. In addition to Protocol 9, we assume further common inputs. Let $\hat{H}_1, \ldots, \hat{H}_k$ be hash functions randomly chosen from a hash family. For example, the functions can be chosen by the client and then distributed together with the parameters of the encryption scheme

If the hash functions are well chosen, the false positive probability can be bounded from above. For a false positive result, the elements must collide under all k hash functions. Hence, using Equation 1, we obtain

$$\left(1 - \left(1 - \frac{1}{m}\right)^S\right)^k.$$

Note that this upper bound assumes that events of a collision with different hash functions are independent. This bound holds if we use universal hashing [8].

Definition 6.2 (Universal Hashing Family) *A family of hash functions $\mathcal{H} = \{H : U \to \{1, \ldots, m\}\}$ is called* universal *if*

$$\forall x, y \in U, x \neq y : \Pr_{H \in \mathcal{H}}[H(x) = H(y)] \leq \frac{1}{m}.$$

In fact, the events are not independent but the bound is even lower, i.e. our construction is actually more efficient in terms of false positive results. We have to ensure that a collision with one hash function does not imply a higher probability of collision with another hash function. Using universal hashing, it is the other way around, i.e. the probability of another collision decreases.

Theorem 6.3 *Let $\mathcal{H} = \{H : U \to \{1, \ldots, m\}\}$ be a universal hash family with $m > 1$. Then, for every elements $x, y \in U$ and every $k > 1$ it holds*

$$\Pr_{H_k \in \mathcal{H} \setminus \{H_1, \ldots, H_{k-1}\}}[H_k(x) = H_k(y) \mid H_1(x) = H_1(y), \ldots, H_{k-1}(x) = H_{k-1}(y)] < \frac{1}{m}.$$

The proof can be found in Appendix B.3.

Choice of Parameters. An open problem is to determine S, the numbers of elements to be hashed. For the hashing scheme described in Section 6.1, we evaluate the intersection bin-wise. Hence, we need the maximal number of elements per bin over all bins. We can compute an upper bound which is fulfilled with high probability. The upper bound is based on the number of bins which is determined by the client's set size and the number of the server's elements, which are both publicly known. Based on the maximal bin size, both client

and server can agree on the number of hash functions k. The client sends the corresponding k hash values for each element.

The server can now evaluate the procedure on its bins. To reduce the computation time, the server can possibly save some operations by using additional information of its own set. After the hashing-to-bins-step, the server knows the actual maximal number of elements over all bins. Therefore, the server needs only as many iterations k as needed to fulfill the false positive requirements. This might be smaller than the theoretical bound. Note that revealing the actual number would leak some information about the server's set. To further hide this information, the server should execute some additional operations on the final results to achieve the same amount of noise as for the theoretical bound. Note that this procedure still decreases the computation time since only the multiplications to combine the results of different hash functions are applied and the basic BP evaluation can be saved.

7 PCI and Extended Functionalities

In this section, we present possible extensions to our PSI protocol starting with examples for PCI. Furthermore, we consider Updatable PSI, Labeled PSI, and the possibility to outsource the computation to a third party.

7.1 PCI

Our goal was to construct a protocol that allows Private Computation on Set Intersection (PCI). Our construction computes the intersection between the client's set Y and the server's set X. More specifically, the servers computes an encrypted bit $[\![b_y]\!]$ which tells if $y \in X \cap Y$. The output makes it possible to do any further computations on the intersection without first decrypting the result. This property has two advantages. First, we do not reveal the intersection itself to any of the parties which allows a simple extension without modifying the PSI protocol and obtaining a secure protocol at the same time (assuming the applied extension is secure). Second, we do not even need another interaction between the parties since the server has a meaningful output. Previous low-communication PSI protocols lack these two properties [9, 10, 13]. Some examples for PCI are PSI-CAT [23, 26], Threshold-PSI [23], and PSI-Sum [38, 39]. We start by describing solutions for these common PSI extensions and then generalize the construction to allow for arbitrary extensions.

PSI-CAT. In PSI-CAT the goal is to reveal only the cardinality of the intersection. We can homomorphically compute the size of the intersection S:

$$[\![S]\!] = \sum_{i=1}^{|Y|} [\![b_{y_i}]\!] .$$

Then, $[\![S]\!]$ can be sent to the client.

Threshold-PSI. In Threshold-PSI the goal is to reveal the intersection or the intersection size only if it exceeds a specific threshold t. In a first step, we can compute the size of the intersection as above:

$$[\![S]\!] = \sum_{i=1}^{|Y|} [\![b_{y_i}]\!].$$

We can then apply a homomorphic comparison to a plaintext threshold and obtain

$$[\![b_{CAT}]\!] = [\![S \geq t]\!].$$

For secure integer comparison, we refer to Tueno and Janneck [48].

For the variant where the size should be revealed, the server can send

$$[\![b_{CAT}]\!] \boxdot [\![S]\!]$$

to the client. For the variant revealing the intersection, the server sends

$$[\![b_{CAT}]\!] \boxdot [\![b_{y_i}]\!] \quad \forall i = 1, \ldots, \beta.$$

Note that the computations need additional multiplications. However, the number of operations is known to both parties and the parameters for the scheme can be chosen accordingly or the intersection result can be bootstrapped.

PSI-Sum. In PSI-Sum, the goal is to sum up payloads associated with each input for all the inputs of the intersection. Let p_i be the payload corresponding to element y_i. Then, we obtain the result

$$[\![b_{sum}]\!] = \sum_{i=1}^{|Y|} [\![b_{y_i}]\!] \cdot p_i.$$

Note that this construction does not even need a homomorphic multiplication.

We can even go a step further and consider the case of private payloads, i.e. payloads p_i are only known to the client. In this scenario, the client can send encryptions of p_i to the server and the server can then homomorphically compute

$$[\![b_{sum}]\!] = \sum_{i=1}^{|Y|} [\![b_{y_i}]\!] \boxdot [\![p_i]\!].$$

PCI-capable PMT. Many PSI protocols, including ours, can be seen as the combination of Cuckoo hashing and a secure private membership test (PMT) protocol. PMT protocols enable a client to test for one element y whether it is included in a set X given by the server. Performing a PMT for every element in the Cuckoo hash table yields a secure PSI protocol. However, this construction does not allow PCI since the client receives the PMT result. Ciampi and Orlandi describe a variation of the PMT functionality that allows arbitrary PCI. Here, instead of a boolean b indicating whether $y \in X$, the client learns γ^1 if $y \in X$

and γ^0 otherwise. Both γ^0 and γ^1 are arbitrary strings input by the server. To support PCI, γ^0 and γ^1 could encode input wire labels for garbled circuits, secret-shares of the PMT result b for arithmetic circuits [15] or HE ciphertexts $\mathsf{Enc}(b)$ where the server holds the secret key. Our protocol can also realize this PMT variation if the server sends back

$$\gamma^1 \cdot [\![b_{y_i}]\!] \boxplus \gamma^0 \cdot [\![1 - b_{y_i}]\!] .$$

Generalization The previous examples describe customized functionalities on the calculated intersection which are of practical interest. With the PCI-capable PMT approach, any functionality f can be computed on the intersection [11]. Formally, given a function f that should be applied to the intersection, we can securely compute the ideal (asymmetric) PCI_f functionality defined as

$$\mathcal{F}^{PCI_f}(X,Y) = (\mathcal{F}_1^{PCI_f}(X,Y), \mathcal{F}_2^{PCI_f}(X,Y)) = (\varnothing, f(X \cap Y)).$$

in the semi-honest security setting. However, using Ciampi and Orlandi's PCI-capable PMT approach, either the client has to perform the PCI computation[2] or at least one more round of communication is needed. In our case, we can simplify the PCI computation without additional rounds of communication. Remark, in our protocol, the client sends the encryption of all elements $[\![y_i]\!]$ to the server. Since the server also obtains $[\![b_{y_i}]\!] = [\![1]\!]$ if $y_i \in (X \cap Y)$ and $[\![b_{y_i}]\!] = [\![0]\!]$ otherwise, the server can (efficiently) construct FHE circuits $\hat{f}$ to compute any functionality f on the intersection. This construction is the same as the FHE-based PCI-capable PMT approach but on the server's side and with encrypted client elements. Such generic computations are possible because the server can address each element of the intersection on its own using a homomorphic multiplication of the boolean output and the element. For example, adding intersection elements to an arbitrary encrypted value $[\![x]\!]$ can be implemented by computing

$$[\![x]\!] \boxplus ([\![b_{y_i}]\!] \boxdot [\![y_i]\!]) . \tag{2}$$

If the element is not in the intersection, 0 is added. Dealing with elements that are not in the intersection (i.e. $[\![b_{y_i}]\!] = [\![0]\!]$) seems to be an unavoidable overhead since the size of the intersection needs to be hidden from the server (or even from the client itself).

Now for PCI_f, in the online phase, instead of sending back $[\![b_{y_i}]\!]$, the server evaluates and sends $\hat{f}\left(([\![b_{y_1}]\!], \ldots, [\![b_{y_\beta}]\!]), ([\![y_1]\!], \ldots, [\![y_\beta]\!])\right)$ to the client. An overview is depicted in Protocol 11. Note, in our improved protocol we operate on packed ciphertexts. However, generic computations as described above are still possible with packed ciphertexts. Addressing all the elements separately, e.g., to compute the sum, can be achieved by rotations [25]. Thus, we aggregate multiple packed ciphertexts into one tuple and evaluate the homomorphic circuit on these tuples.

[2] In the secret-sharing scenario, the server and client jointly perform the PCI computation.

7.2 Updatable PCI

For many real-world applications of PSI, like contact discovery [19], it is desirable to support updates of the input sets (and the resulting intersection) without executing the whole PSI protocol again. A proposed requirement for updatable PSI is that the complexity of an update should only scale with the size of the update (i.e. the number of additions and deletions). Kiss et al. [32] and Badrinarayanan et al. [2] introduce different updatable PSI models and protocols that (partially) meet this complexity requirement. Our protocol can support efficient updates of the server set where updates are assumed to happen more frequently. In the full protocol execution, we store the encrypted client elements $[\![y_i]\!]$ for all bins $i \in \{1, \ldots, \beta\}$. Afterwards, for adding or deleting a server element x, the server computes all h bin indices $\mathcal{J} \coloneqq \{H_1(x), \ldots, H_h(x)\} \subseteq \{1, \ldots, \beta\}$ where x could be placed. The server homomorphically compares

$$[\![x = y_j]\!] \coloneqq x[1] \boxplus [\![y_j[1]]\!] \boxdot \cdots \boxdot x[\mu] \boxplus [\![y_j[\mu]]\!]$$

for each bin $j \in \mathcal{J}$. Further, the server computes $x \boxdot \sum_{j \in \mathcal{J}} [\![x = y_j]\!]$ (in a larger plaintext space) and sends it back to the client. If the client obtains $[\![0]\!]$, the only revealed information is that an item (which is not part of the intersection) has been updated. If the result is $[\![x]\!]$, it depends on whether x has already been in the calculated intersection. If yes, the client learns that x has been removed from the server set and thus the intersection. Otherwise, the client adds x to the intersection. As such, server additions and deletions are performed in the exact same way.

However, such updates allow no generic PCI on an updated intersection. Our protocol can support server updates with a computation complexity of $\mathcal{O}(|Y|)$ which can be used for generic PCI without additional client interactions. We extend the previous update approach as follows. In the full protocol execution, the server additionally stores all bin results $[\![b_{y_i}]\!]$. Now, the encrypted booleans for PCI are updated as $[\![b_{y_j}]\!] = [\![b_{y_j}]\!] \boxplus [\![x = y_j]\!]$ for $j \in \mathcal{J}$. For all other bin results $[\![b_{y_i}]\!]$, the server adds $[\![0]\!]$ (with the same noise level as $[\![x = y_j]\!]$) to hide which bins have been updated. Remark that both update approaches can be easily adjusted to work with permutation-based hashing and packed ciphertexts.

7.3 Labeled PSI

Labeled PSI has been considered in recent works on PSI based on homomorphic encryption [9, 13]. In labeled PSI, the sender holds a label L_i for each of its elements $x \in X$. In addition to the intersection, the receiver obtains label L_i for each item in the intersection, i.e. $y_i \in X \cap Y$. In previous techniques, the sender builds interpolation polynomials which are then evaluated on the encrypted receiver's input. The receiver obtains the label if the element is in the intersection and a random element otherwise. Such a construction introduces additional computation and communication overhead.

In contrast, our protocol can be adapted with nearly no additional computation or communication overhead. In the final step of our BP evaluation

algorithm (Algorithm 3), we just multiply the cost of the paths with the label of the leave nodes. In the standard PSI case, these labels are 1 or 0 indicating if the element is in the set. For labeled PSI, we can replace the 1's with the corresponding label. Thus, the client obtains the label if the element is in the intersection and 0 otherwise. The computational overhead stays the same. The same holds for the communication overhead except for the case in which we need a larger larger ciphertext to represent the labels than we would needed in the standard PSI case.

7.4 Third-Party Computation

Our protocol is built for a client-server scenario in which the server holds the larger set and has a lot of computation power. In case the latter is not given, the server might outsource the computational effort to a third party. With some slight modifications, we are able to extend our protocol such that the server can outsource the main computation to a third party in a secure way. Like the client, the server encrypts its program and the third party can evaluate the protocol on encrypted inputs Y and an encrypted tree built from X. To this end, the server applies the hashing procedure, builds the tree and encrypts the leaf node labels. Then, the server sends the encrypted labels to the third-party which can then use the standard evaluation process with only one difference. The multiplications with leaf nodes are no longer constant multiplications but ciphertext-ciphertext multiplications. In this scenario, we cannot apply direct tree optimization procedures since the leaf nodes would not be unambiguous anymore. The client can decrypt the result using the help of a proxy [24]. An extension to also obfuscate possible optimization is described in Appendix C.

8 Analysis

We first present a security analysis of our protocol followed by some complexity considerations.

8.1 Security

General Remark. Common security definitions, e.g. Definition 3.3, assume the inputs x, y to have the same bitlength, i.e. $|x| = |y|$. For our unbalanced PSI setting this seems not to be the case at first glance since the sets have different sizes. This is a general problem which is mainly of formal nature but is not explicitly discussed in the literature. However, we can still adjust our model to fulfill the formal requirements. Note that both set sizes as well as the length of elements are common knowledge. Hence, the client with the smaller set could pad its input to the same bitlength as the larger set. Both the ideal functionality and the protocols do not use the additional "input". In this way we can bypass the formal problem without changing any functional aspect. In the following, we ignore the padding step to not make the proofs unnecessarily complicated.

Semi-Honest Security. To give an intuition, let us first consider the security against a semi-honest adversary in an informal way. We have dealt with correctness in the previous sections. Regarding privacy, we must think about what a corrupted party can learn. On the one hand, the server operates on IND-CPA ciphertexts, i.e. does not have any (unencrypted) information belonging to the client. Thus, a corrupted server should not be able to learn anything. On the other hand, the only result the client receives from the server is the bit indicating if its element is in the server's set. This information would be learned by the client anyway because the required client output is the intersection.

Theorem 8.1 (Semi-Honest Security of Full Protocol) *If the underlying encryption scheme is IND-CPA secure, the full protocol with the general construction (Protocol 9)* Π_{PSI} *securely computes functionality* $\mathcal{F}_{\mathsf{PSI}}$ *in the presence of a semi-honest adversary.*

Theorem 8.2 (Semi-Honest Security of Deterministic Variant) *If the underlying encryption scheme is IND-CPA secure, the full protocol with the deterministic variant* Π_{PSI} *(Section 6.2) securely computes functionality* $\mathcal{F}_{\mathsf{PSI}}$ *in the presence of a semi-honest adversary.*

Theorem 8.3 (Semi-Honest Security of Probabilistic Variant) *If the underlying encryption scheme is IND-CPA secure and the chosen hashing family is universal, the full protocol with the probabilistic variant* Π_{PSI} *(Protocol 10) securely computes functionality* $\mathcal{F}_{\mathsf{PSI}}$ *in the presence of a semi-honest adversary.*

Theorem 8.4 (Semi-Honest Security of PCI) *If the underlying encryption scheme is IND-CPA secure, the PCI construction (Protocol 11) for a function* f*,* $\Pi_{\mathsf{PCI}_\mathsf{f}}$*, securely computes functionality* $\mathcal{F}_{\mathsf{PCI}_\mathsf{f}}$ *in the presence of a semi-honest adversary.*

Proofs can be found in Appendix B.1.

Receiver Privacy in Case of a Malicious Sender. For a malicious server, we can additionally guarantee privacy for the client. To obtain a full simulation, we would need (among other things) to ensure correctness of the output. Our protocol does not provide such a property since the sender can do arbitrary computations on the receiver's input, e.g., outputting the whole input set as an intersection result or an empty set. However, we can at least ensure privacy of the receiver in the setting of a malicious sender. In our scenario, this is very simple since we can use a definition of privacy for the case the sender does not obtain an output [27]. This yields the following privacy property

$$\{\mathsf{View}^{\mathcal{A}}_{\pi,\mathcal{A}(\mathsf{z}),1}(X,Y,\lambda)\}_{X,Y,Y',\mathsf{z},\lambda} \stackrel{c}{\equiv} \{\mathsf{View}^{\mathcal{A}}_{\pi,\mathcal{A}(\mathsf{z}),1}(X,Y',\lambda)\}_{X,Y,Y',\mathsf{z},\lambda},$$

for a PPT adversary $\mathcal{A}$ with auxiliary input z.

Theorem 8.5 (Receiver Privacy) *If the underlying encryption scheme is IND-CPA secure, the full protocol with the general construction* π *(Protocol 9) fulfills receiver privacy in the presence of a malicious sender.*

A proof can be found in Appendix B.2. The same technique can be used to prove receiver privacy in the malicious setting for the other protocol variants, similar to the semi-honest setting.

Security in the Third-Party Setting. The easiest way to securely implement the third-party setting (Section 7.4) is to encrypt both the client and server input with the client's public key. Then, client and server send their inputs to the third party. To obtain the same security properties, we additionally have to assume non-colluding parties, especially no collusion between client and third-party. Otherwise all of the server's input could be easily extracted.

8.2 PSI Extension

Our goal was to design a protocol with low communication costs, as well as the ability to enable PCI. With our protocol, we can combine the PSI computation with any computation, i.e. for any functionality f, we can compute $f(X \cap Y)$ without revealing $X \cap Y$. The complexity of the extensions mainly depends on the complexity of the function that should be applied, i.e. the number of operations of the corresponding FHE circuit. Unfortunately, we apply the operations on $\mathcal{O}(|Y|)$ elements because the server does not know which of the elements are in the intersection. As described in Section 7.1, we argue this is a necessary overhead to hide the size of the intersection. Still, in our improved protocol this does not make a big difference since we are operating on packed ciphertexts anyway.[3] Compared to a plain circuit of f, we might need additional operations in some cases. In Section 7, we show how to retrieve the encrypted input elements by multiplying with the boolean result. If the PCI functionality operates on original elements, we have to apply these operations. Note that our applications of the standard PSI extensions (PSI-CAT, PSI-Sum, Threshold-PSI) only operate on the encrypted result bits and therefore do not need to include the encryptions of the client elements.

8.3 Complexity

We consider both communication and computation complexity starting with communication, for which our goal is to achieve a sublinear overhead.

8.3.1 Communication

Using HE, we can achieve communication sublinear in the larger set since we can shift most of the computation to one party and only need to communicate the initial inputs. Thus, instantiating our protocol with an FHE scheme, we would get a communication complexity independent of the server's set, i.e. $\mathcal{O}(|Y|)$. In practice, a leveled FHE scheme is often the more efficient choice because it is a lot faster. However, in terms of communication we have a new problem.

[3] It does affect the asymptotic behavior, and is especially prevalent in cases where the client's set is large, which does not fit into our setting.

Leveled FHE has an additional depth parameter D as an input and only allows to evaluate circuits of depth at most D. The circuit the server has to evaluate depends on its set and for a larger set, the depth of circuits increases as well. For both variants, the depth of the BP to be evaluated increases logarithmically with the size of the server's set, i.e. for a BP depth d, it holds $d \in \mathcal{O}(\log |X|)$. We are able to evaluate a BP of depth d with a multiplicative depth which is logarithmic in the BP depth. For detailed information, we refer to the work of Janneck et al. [30]. Combining these results, we obtain an overall multiplicative depth D with $D \in \mathcal{O}(\log \log |X|)$. Commonly used leveled FHE schemes like BGV [7] or BFV [5, 21] introduce quasipolynomial communication overhead, in detail $\tilde{\mathcal{O}}(D^2)$ for depth parameter D. Hence, we obtain communication cost $\tilde{\mathcal{O}}(|Y|(\log \log |X|)^2)$. The communication cost still depends on the server's set but in a magnitude which is really small for real-world applications.

8.3.2 Computation

The computational complexity is the weak point of our protocol since it asymptotically depends exponentially on the element size or the server's set size. The reduction mechanisms can reduce the complexity to evaluate the BP but as mentioned in Section 6 their impact is significantly reduced using the hashing construction. Other optimizations such as PBH and the probabilistic variant can reduce the computation time but do not decrease the overhead in terms of the asymptotic complexity. However, for practical uses it has a significant impact and can make the protocol applicable for real world applications. In the following we present the number of operations to give an overview of the variants of our protocol.

Deterministic Variant. Let s be the number of slots for plaintext encoding and β the number of bins required for the Cuckoo Hashing (depends on the client's set size $|Y|$) and σ the bitlength of input elements. We apply the complete procedure $\left\lceil \frac{\beta}{s} \right\rceil$ times.

Each procedure consists of a SIMD BP evaluation of depth $\sigma - \log \beta$ (because of PBH). Such a BP evaluation needs 2^d multiplications for d the depth of the BP. If the depth is of relevance, we can compute the BP with logarithmic depth. In this case, we need even more multiplications, approximately $\log d \cdot 2^{d \log d}$ [30].

Since $\sigma \in \mathcal{O}(\log |X|)$, we obtain a depth $D \in \mathcal{O}(\log \log |X|)$. This matches the results of the previous subsection.

Probabilistic Variant. In addition to parameters s and β from the deterministic variant, we also need to consider the number of hash functions k. We now need $\left\lceil \frac{\beta}{s} \right\rceil \cdot k$ SIMD BP evaluations. The depth of such a BP does not depend on σ anymore but on a chosen parameter m which is the size of the image of the hash functions. We have chosen m near S, the number of elements in the server's bins. The parameter S depends on the size of the server's set, the number of bins, and the actual values. We refrain from a detailed analysis and

only present the average to give a rough overview. The average number of elements is $|X|/\beta$ and thus m and the BP depth are of the same magnitude. For a concrete analysis, we have to compute an upper bound which will be higher than the average case.

For the probabilistic variant, we have to take the additional multiplications based on the second hashing procedure into account. For k hash functions, we need k multiplications for each of the $\lceil \frac{\beta}{s} \rceil$ rounds which increases the multiplicative depth by $\log k$, which is a constant factor.

9 Evaluation

In this section, we describe implementation details and present the results of our evaluation.

9.1 Implementation

We implemented both variants of our protocol as a proof of concept. The implementation is based on BGV [7] from HElib [44]. Since the evaluation of a branching program is a basic building block of our protocol, we base its implementation on Janneck et al. [30]. For Cuckoo Hashing, we used three hash functions and Tabulation Hashing [8] which has been proven to provide a low hashing failure rate [42]. The number of bins is at least $\beta = 1.5 \cdot |Y|$ which yields a very low failure probability of less than 2^{-40} [41]. To instantiate the hash functions of our probabilistic variant, we relied on the linear congruential hash family which is known to be universal [8]. We conducted experiments with different parameters to elaborate advantages and disadvantages of our methods. All experiments were performed on an AWS instance equipped with 24 virtual cores of an Intel Xeon scalable processor with up to 4GHz and 192 GB of RAM running Ubuntu 20.04.

9.2 Experimental Results

We start by examining the communication results and then explore the computation results.

9.2.1 Communication

In the theoretical analysis in Section 8.3 we have seen that we achieve a communication overhead sublinear in the server's set. This is a substantial improvement compared to previous PCI work [11, 34]. In Table 1 and Figure 8, we present the communication in MB between client and server for different sizes for our protocol and previous work executing the PSI functionality. The data of the previous work is estimated using the analysis of Ma and Chow [34], our data is measured during the protocol execution. We use the unbalanced setting, with size $|Y| = 500$ for the client's set and different sizes of the server's set. The protocol of Ma and Chow has better communication than Ciampi and Orlandi

for server set sizes 2^{20} and larger. Our protocol has a larger communication overhead for small sizes of the server's set but has a significant improvement on larger sets. For $|X| = 2^{24}$ we can reduce the communication by factor 47.

Table 1: Communication (in MB) for Different Server's Set Sizes and Different Numbers of Hash Functions k for Our Probabilistic Variant

Server's Set Size $\lvert X\rvert$	2^{12}	2^{16}	2^{20}	2^{24}
Ciampi and Orlandi [11]	41	299	3605	53204
Ma and Chow [34]	54	339	3465	41480
Ours ($k = 8$)	141	223	332	441
Ours ($k = 16$)	277	441	659	877

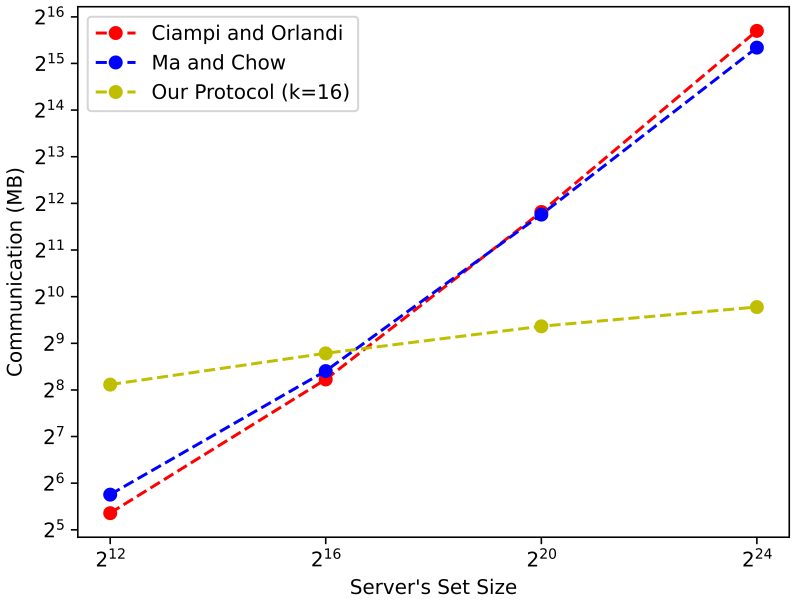


Figure 8: Communication (in MB) for the PSI Functionality

9.2.2 Computation

We first compare how the element bitlength and the size of the server's set influences the computational overhead of the server and then analyze the running time of different post-processing routines.

Element Bitlength. To analyze the influence of the element size, we fix the server set at size $|X| = 2^{10}$. Table 2 presents the running times for both variants. We can clearly see that the deterministic variant strongly depends on the element size whereas the element size does influence the running time of the probabilistic significantly.[4]

Table 2: Running Time (in s) for $|X| = 2^{10}$ and Different Element Bitlengths

Element Bitlength σ	12	16	20	22
Deterministic Variant	0.941	14.811	274.856	1170.140
Probabilistic Variant	1.672	1.009	0.935	1.009

Server's Set Size. We now increase the size of the server's set and analyze both variants. The element size is fixed to $\sigma = 16$ bits and for the probabilistic variant we use $k = 4$ hash functions. The results are depicted in Table 3. As expected, the deterministic variant has the same running time for all sizes of the server's set whereas the running time of probabilistic variant increases.

Table 3: Running Time (in s) for $\sigma = 16, k = 4$ and Different Server's Set Sizes

Server's Set Size $\|X\|$	2^{10}	2^{12}	2^{14}	2^{16}
Deterministic Variant	14.811	14.759	14.906	14.774
Probabilistic Variant	4.666	7.396	28.015	48.064

PCI. For PCI, we evaluate different protocols on the set intersection and measure the required time for the post-processing. The computation is applied on the resulting representation which is a boolean result for each client's element. Thus, the computational complexity of the post-processing depends on the client's set size. Results are depicted in Table 4. For real-world use cases, we can first compute the set intersection and different post-processing could be applied if it is needed. Therefore, we only have to compute the comparably expensive basic PSI functionality once and evaluate the post-processing very efficiently.

Table 4: Running Time (in ms) for Secure Post-Processing on an Intersection

Number of Bins β	64	128	256	512	1024	2048
Cardinality	20.52	28.67	42.08	71.23	128.30	257.88
Weighted Sum	265.39	579.91	806.85	1719.63	2811.75	5425.84

Comparison. In the previous work, the client obtains an encrypted tree and traverses it using multiple OTs. The evaluation of the tree is therefore done by the client. Moreover, the additional PCI functionality has to be applied by the

[4]The comparably high value for the probabilistic variant and $\sigma = 12$ seems to be an outlier.

client as well or the parties have to establish a new communication round as described in Section 7.1. In our protocols, all computations are still solely done by the server and we do not need any further interaction with the client. Hence, we can not only outsource the effort of the PSI computation to the server but the entire PCI functionality. This opens up opportunities to apply the protocol in the asymmetric case where the client has only small computation power and the server has much larger computing resources with plenty of options for parallelization. Furthermore, previous work depends on the bitlength of input elements. This holds only for our deterministic variant whereas our probabilistic variant can process inputs of arbitrary length.

10 Conclusion

We proposed a new protocol for private set intersection between a server and a client. Both parties have private sets where it is assumed that the server's set is significantly larger. The result of the computation is revealed only to the client. Our protocol delegates the complete computation to a server using fully homomorphic encryption (FHE). The server homomorphically evaluates the client's inputs on a branching program representing its set. With this construction we achieve sublinear communication. Moreover, we enable PCI, i.e. arbitrary post-processing on the intersection without any further interaction. However, the computation costs do not scale efficiently with the size of the server's set such that we do not achieve a very efficient computational overhead. Therefore, our protocol is applicable to scenarios with large computing resources of the server where we need low communication costs and the ability to apply efficient post-processing, i.e. without any further client interaction.

References

[1] Y. Arbitman, M. Naor, and G. Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 787–796. IEEE, 2010.

[2] S. Badrinarayanan, P. Miao, and T. Xie. Updatable private set intersection. *Cryptology ePrint Archive*, 2021.

[3] P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik. Countering gattaca: efficient and secure testing of fully-sequenced human genomes. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 691–702, 2011.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual Cryptology Conference*, pages 868–886. Springer, 2012.

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *ECCC*, 18:111, 2011.

[7] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[8] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[9] H. Chen, Z. Huang, K. Laine, and P. Rindal. Labeled psi from fully homomorphic encryption with malicious security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1223–1237, 2018.

[10] H. Chen, K. Laine, and P. Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1255, 2017.

[11] M. Ciampi and C. Orlandi. Combining private set-intersection with secure two-party computation. In *International Conference on Security and Cryptography for Networks*, pages 464–482. Springer, 2018.

[12] A. Cohen. What about bob? the inadequacy of cpa security for proxy reencryption. In D. Lin and K. Sako, editors, *Public-Key Cryptography – PKC 2019*, pages 287–316, Cham, 2019. Springer International Publishing.

[13] K. Cong, R. C. Moreno, M. B. da Gama, W. Dai, I. Iliashenko, K. Laine, and M. Rosenberg. Labeled psi from homomorphic encryption with reduced computation and communication. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1135–1150, 2021.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[15] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 643–662, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[16] A. C. Davi Resende and D. de Freitas Aranha. Faster unbalanced private set intersection in the semi-honest setting. *Journal of Cryptographic Engineering*, 11(1):21–38, 2021.

[17] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *International Conference on Financial Cryptography and Data Security*, pages 143–159. Springer, 2010.

[18] S. K. Debnath and R. Dutta. Secure and efficient private set intersection cardinality using bloom filter. In *International Conference on Information Security*, pages 209–226. Springer, 2015.

[19] D. Demmler, P. Rindal, M. Rosulek, and N. Trieu. Pir-psi: Scaling private contact discovery. *Proceedings on Privacy Enhancing Technologies*, 4:159–178, 2018.

[20] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[21] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.

[22] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, sep 1960.

[23] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *International conference on the theory and applications of cryptographic techniques*, pages 1–19. Springer, 2004.

[24] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, New York, NY, USA, 2009. ACM.

[25] S. Halevi and V. Shoup. Algorithms in helib. In *Annual Cryptology Conference*, pages 554–571. Springer, 2014.

[26] P. Hallgren, C. Orlandi, and A. Sabelfeld. Privatepool: Privacy-preserving ridesharing. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 276–291. IEEE, 2017.

[27] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

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

[29] M. Ion, B. Kreuter, A. E. Nergiz, S. Patel, S. Saxena, K. Seth, M. Raykova, D. Shanahan, and M. Yung. On deploying secure computing: Private intersection-sum-with-cardinality. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 370–389. IEEE, 2020.

[30] J. Janneck, A. Boudi, A. Tueno, and M. Akram. Secure branching program evaluation. Cryptology ePrint Archive, Paper 2022/1075, 2022. `https://eprint.iacr.org/2022/1075`.

[31] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert. Mobile private contact discovery at scale. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1447–1464, 2019.

[32] Á. Kiss, J. Liu, T. Schneider, N. Asokan, and B. Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proc. Priv. Enhancing Technol.*, 2017(4):177–197, 2017.

[33] V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu. 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, October 24-28, 2016*, pages 818–829, 2016.

[34] J. P. Ma and S. S. Chow. Secure-computation-friendly private set intersection from oblivious compact graph evaluation. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 1086–1097, 2022.

[35] C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*, pages 134–134. IEEE, 1986.

[36] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[37] B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 515–530, 2015.

[38] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai. Efficient circuit-based psi with linear communication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 122–153. Springer, 2019.

[39] B. Pinkas, T. Schneider, C. Weinert, and U. Wieder. Efficient circuit-based psi via cuckoo hashing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 125–157. Springer, 2018.

[40] B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on {OT} extension. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 797–812, 2014.

[41] B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on OT extension. *ACM Trans. Priv. Secur.*, 21(2):7:1–7:35, 2018.

[42] M. Pătraşcu and M. Thorup. The power of simple tabulation hashing. *Journal of the ACM (JACM)*, 59(3):1–50, 2012.

[43] A. C. D. Resende and D. F. Aranha. Faster unbalanced private set intersection. In *International Conference on Financial Cryptography and Data Security*, pages 203–221. Springer, 2018.

[44] V. S. Shai Halevi. Helib - an implementation of homomorphic encryption, December 2021. https://github.com/homenc/HElib.

[45] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, nov 1979.

[46] L. Shen, X. Chen, D. Wang, B. Fang, and Y. Dong. Efficient and private set intersection of human genomes. In *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 761–764. IEEE, 2018.

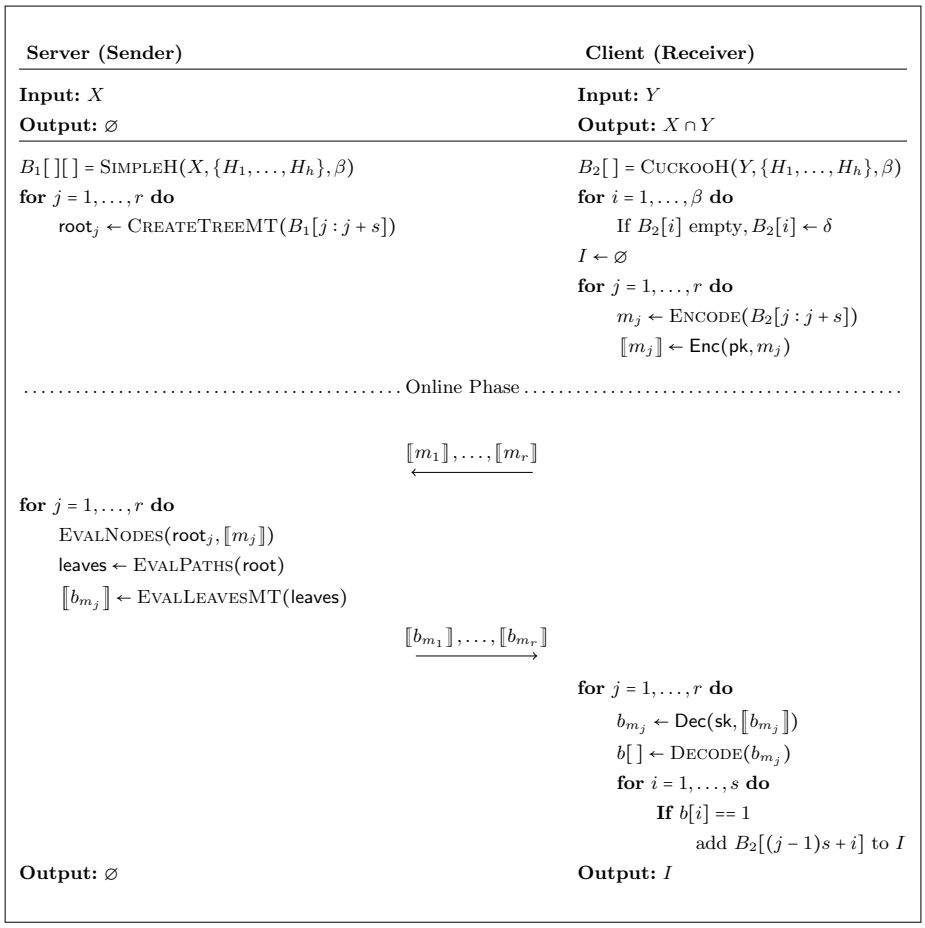
[47] N. P. Smart and F. Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[48] A. Tueno and J. Janneck. A method for securely comparing integers using binary trees. *Cryptology ePrint Archive*, 2021.

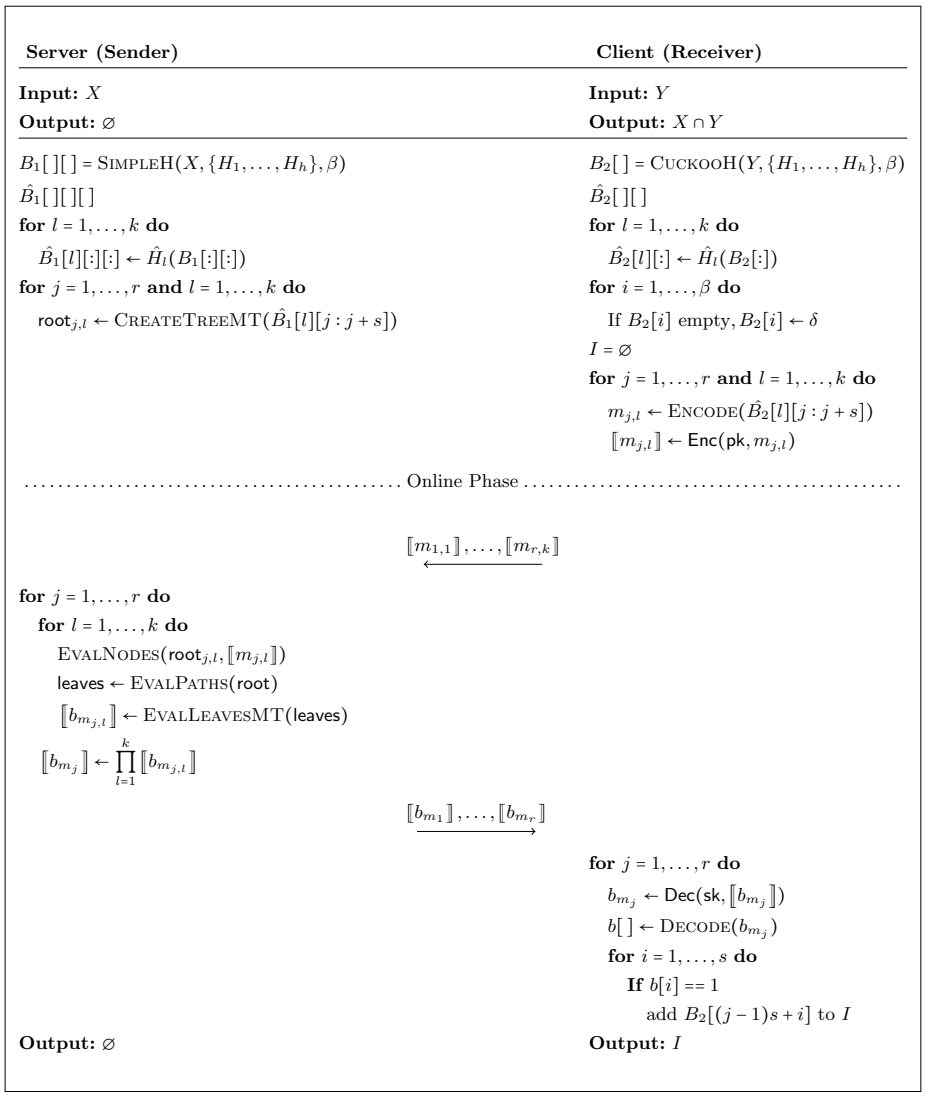
[49] A. Tueno, F. Kerschbaum, D. Bernau, and S. Foresti. Selective access for supply chain management in the cloud. In *2017 IEEE Conference on Communications and Network Security, CNS 2017, Las Vegas, NV, USA, October 9-11, 2017*, pages 476–482, 2017.

[50] A. C. Yao. Protocols for secure computations. In *SFCS '82*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

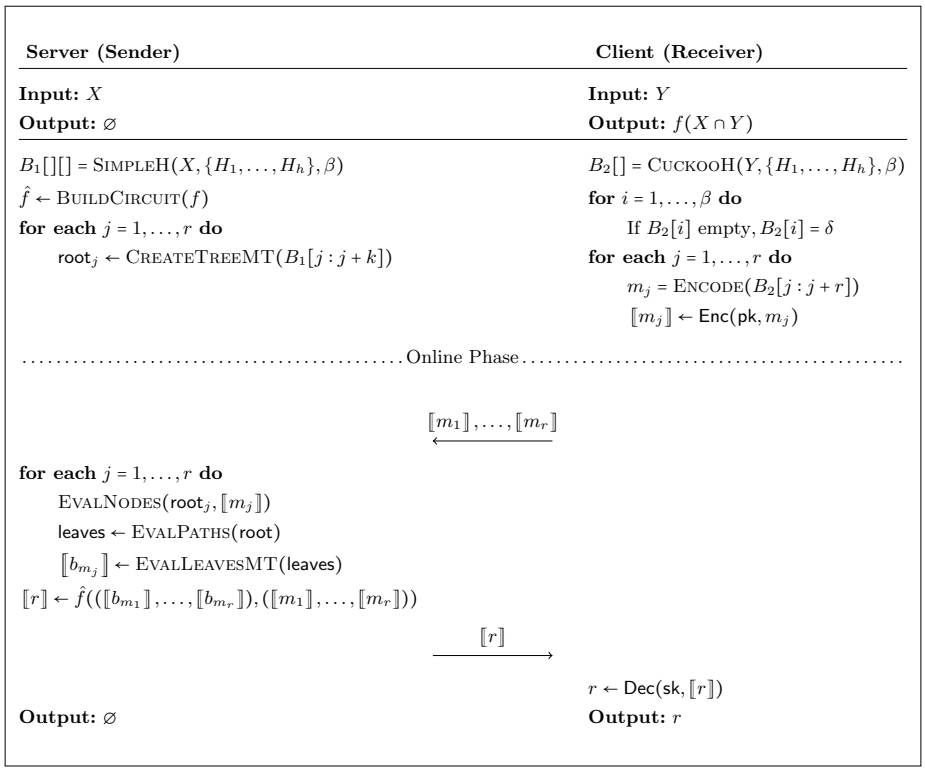
A Protocol Descriptions



Protocol 9: Full Protocol with General Hashing Construction



Protocol 10: Full Protocol with Probabilistic Variant



Protocol 11: Protocol for f-PSI with Basic Hashing Construction

B Postponed Proofs

B.1 Semi-Honest-Security Proofs

Proof of Theorem 8.1. Correctness has been shown in Section 4 and Section 6. For the privacy property we construct two simulators $\mathcal{S}_1, \mathcal{S}_2$ simulating the client's and the server's view.

We start with a simulation of the server, $\mathcal{S}_1$. The server gets no output from the ideal functionality, hence the simulator looks like $\mathcal{S}_1(1^\lambda, X, \varnothing)$. The server receives only one message, thus its view is easy to simulate.
We define the following simulator $\mathcal{S}_1$:

1. Choose β random elements $r_1, \ldots, r_\beta$ from the ciphertext space of the encryption scheme.
2. Send $[\![r_1]\!], \ldots, [\![r_\beta]\!]$ to the server.

Hence, the simulator outputs

$$\{\mathcal{S}_1(1^\lambda, X, \varnothing)\}_{X,Y,\lambda} = \{([\![r_1]\!], \ldots, [\![r_\beta]\!])\}_{X,Y,\lambda}.$$

By δ we denote the dummy element of the client. It follows

$$\{([\![r_1]\!], \ldots, [\![r_\beta]\!])\}_{X,Y,\lambda} \stackrel{c}{\equiv} \{([\![\tilde{y}_1]\!], \ldots, [\![\tilde{y}_\beta]\!]) \mid \tilde{y_1}, \ldots \tilde{y}_\beta \in Y \cup \{\delta\}\}_{X,Y,\lambda}, \quad (3)$$

where we use, as before, $\tilde{y_1}, \ldots, \tilde{y_\beta}$ to denote elements in the client's bins. This holds, since we assume the encryption scheme to be IND-CPA secure. This is equivalent to real-or-random security which means, that β random encryptions cannot be distinguished from encryptions of real client elements. Finally we have

$$\{([\![\tilde{y}_1]\!], \ldots, [\![\tilde{y}_\beta]\!]) \mid \tilde{y_1}, \ldots \tilde{y}_\beta \in Y \cup \{\delta\}\}_{X,Y,\lambda} = \{\mathsf{View}_1^{\Pi_{\mathsf{PSI}}}(X, Y, \lambda)\}_{X,Y,\lambda}$$

and constructed a simulator $\mathcal{S}_1$ simulating the server's view.

Now we consider a simulator $\mathcal{S}_2$ for the client's view, which again receives one message. In contrast to the server, the client gets an output which has to be simulated correctly.
We construct the following simulator $\mathcal{S}_2(1^\lambda, Y, X \cap Y)$:

1. Hash all elements from Y according the Cuckoo Hashing scheme from protocol Π_{PSI} into β bins.
2. Fill the empty bins with dummy elements δ such that there are β elements $\tilde{y}_1, \ldots, \tilde{y}_\beta$ from $Y \cup \{\delta\}$ (again the same procedure as in the protocol).[5]
3. For all $i = 1, \ldots, \beta$ if $\tilde{y}_i \in X \cap Y$ encrypt a 1, i.e. $m_i = \mathsf{Enc}(1)$. Otherwise encrypt a 0, i.e. $m_i = \mathsf{Enc}(0)$.
4. Extract the number of multiplications M and additions A that are applied by the server to every ciphertext. This number depends on the size of the server's set and the algorithm which are both publicly known.
5. Multiply each ciphertext M times with 1 and add A times 0, resulting in $m'_1, \ldots, m'_\beta$.
6. Send $m'_1, \ldots, m'_\beta$ to the client.

Hence, the simulator outputs

$$\{\mathcal{S}_2(1^\lambda, Y, X \cap Y)\}_{X,Y,\lambda} = \{(m'_1, \ldots, m'_\beta)\}_{X,Y,\lambda}.$$

The view executing the real protocol is

$$\{\mathsf{View}_2^{\Pi_{\mathsf{PSI}}}(X, Y, \lambda)\}_{X,Y,\lambda} = \{([\![b_1]\!], \ldots, [\![b_\beta]\!])\}_{X,Y,\lambda},$$

where b_i indicates if the i-th element is in the intersection. Compared to the server, the client can decrypt received messages. Step 3 of the simulator ensures

$$\mathsf{Dec}(m_i) = \mathsf{Dec}([\![b_i]\!]) \quad \forall i = 1, \ldots, \beta.$$

[5]This step is not necessary but eases the understanding.

Step 5 does not change the value of the underlying plaintext, thus we obtain

$$\mathsf{Dec}(\llbracket b_i \rrbracket) = \mathsf{Dec}(m_i) = \mathsf{Dec}(m'_i) \quad \forall i = 1, \ldots, \beta$$

and at least the plaintexts of the received messages are indistinguishable.

However, the ciphertexts have to be computational indistinguishable as well. If we apply operations on an FHE ciphertext, we increase the noise and can therefore distinguish a fresh ciphertext from a ciphertext which is the result of some evaluations. Step 5 of the simulator ensures that the number of operations matches with the number of operations the server applies in the real world. Hence, the noise level of $\llbracket c_i \rrbracket$ cannot be distinguished from the noise level of m'_i. Finally, it follows

$$\{(m'_1, \ldots, m'_\beta)\}_{X,Y,\lambda} \stackrel{c}{\equiv} \{(\llbracket b_1 \rrbracket, \ldots, \llbracket b_\beta \rrbracket)\}_{X,Y,\lambda}$$

which concludes the proof. ■

Proof of Theorem 8.2. The only difference to the general construction is that we store only parts of the message in the Cuckoo Table. Nevertheless, we can construct the same simulator for the server as in the proof of Theorem 8.1 and obtain the exact same equivalence as in Equation (3).

The same holds for the simulator of the client. It must simply store for each Cuckoo Table element (element's right part) the corresponding original element. Hence, the same proof as for Theorem 8.1 can be applied. ■

Proof of Theorem 8.3. To prove privacy, we can construct the same simulator for the server as in the proof of Theorem 8.1 and obtain the exact same equivalence as in Equation (3) except that we need more ciphertexts to simulate all the different hash functions. The same holds for the simulator of the client. Hence, the same proof as for Theorem 8.1 can be applied. The remaining task is to ensure correctness to achieve semi-honest security. By Theorem 6.3, we obtain a decreasing false positive rate since we use a universal hashing family. Thus, we can choose a sufficiently large number of hash functions to obtain any statistical security requirement. ■

Proof of Theorem 8.4. Correctness is ensured since we assume the PCI to correctly implement functionality f. For privacy, we consider simulators $\mathcal{S}_1$ and $\mathcal{S}_2$. For $\mathcal{S}_1$, the simulator for the server's view, we take the same simulator as for the proof of Theorem 8.1. It fulfills the same security guarantees since the PCI does not influence the server's view.

Constructing a simulator for the client's view is an even simpler task than for the PSI protocol. For $\mathcal{S}_2(1^\lambda, Y, f(X,Y))$, we define:

1. Encrypt the final result, $m = \mathsf{Enc}(f(X,Y))$.
2. Apply the same kind of operations on m as an honest server would do, resulting in a noisy m'.

3. Send m' to the client.

The sets created by the simulator and the one of the real view are indistinguishable with the same arguments as in the proof of Theorem 8.1. ■

B.2 Malicious Security Proofs

Proof of Theorem 8.5. Since the protocol consists of only one interaction, the sender's view is just the encrypted bins. Let $\tilde{y}_1, \ldots, \tilde{y}_\beta \in Y \cup \{\delta\}$ be the elements of the receiver's bins when executing the protocol on input Y. Then, we have

$$\{\mathsf{View}^{\mathcal{A}}_{\pi,\mathcal{A}(z),1}(X,Y,\lambda)\}_{X,Y,Y',z\in\{0,1\}^*,\lambda} = \{([\![\tilde{y}_1]\!],\ldots,[\![\tilde{y}_\beta]\!])\}.$$

Analogously, let $\tilde{y}'_1, \ldots, \tilde{y}'_\beta \in Y \cup \{\delta\}$ be the elements of the receiver's bins when executing the protocol on input Y'. Hence, it holds

$$\{\mathsf{View}^{\mathcal{A}}_{\pi,\mathcal{A}(z),1}(X,Y',\lambda)\}_{X,Y,Y',z\in\{0,1\}^*,\lambda} = \{([\![\tilde{y}'_1]\!],\ldots,[\![\tilde{y}'_\beta]\!])\}.$$

From the IND-CPA security of the underlying scheme, it follows

$$\{([\![\tilde{y}_1]\!],\ldots,[\![\tilde{y}_\beta]\!])\} \stackrel{c}{\equiv} \{([\![\tilde{y}'_1]\!],\ldots,[\![\tilde{y}'_\beta]\!])\}.$$

■

B.3 Proof of Universal Hashing Implications

To proof Theorem 6.3, we introduce some notation in addition to Definition 6.2.

Definition B.1 *For a family of hash functions $\mathcal{H} = \{H : U \to \{1,\ldots,m\}\}$ we define two random variables for elements $x, y \in U$:*

$$\delta_H(x,y) := \begin{cases} 1 & , H(x) = H(y) \text{ and } x \neq y \\ 0 & , \text{else} \end{cases}$$

$$\delta_{\mathcal{H}}(x,y) := \sum_{H\in\mathcal{H}} \delta_H(x,y)$$

Note that $\delta_{\mathcal{H}}$ is in fact a "random" variable without any randomness since $\mathcal{H}$ is the whole family. This notation is another variant of defining universal hashing [14] which is equivalent to our definition.

Lemma B.2 *A hash family $\mathcal{H} = \{H : U \to \{1,\ldots,m\}\}$ is universal iff $\forall x, y \in U$ it holds $\delta_{\mathcal{H}}(x,y) \leq \frac{|\mathcal{H}|}{m}$.*

Proof. Let $\mathcal{H} = \{H : U \to \{1,\ldots,m\}\}$ be a universal hash family, then it holds

$$\Pr_{H\in\mathcal{H}}[H(x) = H(y)] \leq \frac{1}{m}.$$

By definition of δ_H, we obtain

$$\Pr_{H \in \mathcal{H}}[\delta_H(x,y) = 1] \leq \frac{1}{m}$$

and

$$\mathbb{E}[\delta_H(x,y)] = \Pr_{H \in \mathcal{H}}[\delta_H(x,y) = 0] \cdot 0 + \Pr_{H \in \mathcal{H}}[\delta_H(x,y) = 1] \cdot 1 \leq \frac{1}{m}.$$

By definition of $\delta_{\mathcal{H}}$ it follows

$$\mathbb{E}[\delta_{\mathcal{H}}(x,y)] = \sum_{H \in \mathcal{H}} \mathbb{E}[\delta_H(x,y)] \leq \frac{|\mathcal{H}|}{m}.$$

$\delta_{\mathcal{H}}$ is deterministic since we take the sum over all hash functions, it holds

$$\delta_{\mathcal{H}}(x,y) \leq \frac{|\mathcal{H}|}{m}.$$

For the other direction, assume that $\delta_{\mathcal{H}}(x,y) \leq \frac{|\mathcal{H}|}{m}$. That means, there are at most $\frac{|\mathcal{H}|}{m}$ hash functions under which x and y collide. If we choose one hash function uniformly at random, we obtain

$$\Pr_{H \in \mathcal{H}}[H(x) = H(y)] \leq \frac{|\mathcal{H}|/m}{|\mathcal{H}|} = \frac{1}{m}$$

which concludes the proof. ■

We can now prove that the probability of additional collisions decreases.

Theorem B.3 *Let $\mathcal{H} = \{H : U \to \{1, \ldots, m\}\}$ be a universal hash family with $m > 1$. Then, for every elements $x, y \in U$ and every $k > 1$ it holds*

$$\Pr_{H_k \in \mathcal{H} \setminus \{H_1, \ldots, H_{k-1}\}}[\delta_{H_k}(x,y) = 1 \mid \delta_{H_1}(x,y) = \ldots = \delta_{H_{k-1}}(x,y) = 1] < \frac{1}{m}.$$

Proof. Let $\mathcal{H} = \{H : U \to \{1, \ldots, m\}\}$ be a universal hash family with $m > 1$. If we randomly choose hash function H_k from $\mathcal{H} \setminus \{H_1, \ldots, H_{k-1}\}$ and $H_1, \ldots, H_{k-1}$ led to a collision, then there are at most $\frac{|\mathcal{H}|}{m} - (k-1)$ left which lead to a collision. Hence, we obtain

$$\begin{aligned}
&\Pr_{H_k \in \mathcal{H} \setminus \{H_1, \ldots, H_{k-1}\}}[H_k(x) = H_k(y) \mid H_1(x) = H_1(y), \ldots, H_{k-1}(x) = H_{k-1}(y)] \\
&= \Pr_{H_k \in \mathcal{H} \setminus \{H_1, \ldots, H_{k-1}\}}[\delta_{H_k}(x,y) = 1 \mid \delta_{H_1}(x,y) = \ldots = \delta_{H_{k-1}}(x,y) = 1] \\
&\leq \frac{|\mathcal{H}|/m - (k-1)}{|\mathcal{H}| - (k-1)} \\
&= \frac{1}{m} \cdot \underbrace{\frac{|\mathcal{H}| - m(k-1)}{|\mathcal{H}| - (k-1)}}_{<1} \\
&< \frac{1}{m},
\end{aligned}$$

concluding the proof. ■

Algorithm 5 Evaluating Nodes (Encrypted Case)

```
1:  function EvalNodes(root, [[ȳ]])
2:      let Q be a new queue
3:      Q.enqueue(root)
4:      parse [[ȳ]] to [[y[1]]], ..., [[y[μ]]]
5:      while Q.empty() = false do
6:          v ← Q.dequeue()
7:          if v.left ≠ null then
8:              [[v.left.cost]] ← [[y[v.level]]] ⊞ [[v.rEdge]]      ▷ Encrypted label
9:              Q.enqueue(v.left)
10:         if v.right ≠ null then
11:             [[v.right.cost]] ← [[y[v.level]]] ⊞ [[v.lEdge]]     ▷ Encrypted label
12:             Q.enqueue(v.right)
```

C Extended Third-Party Computation

If we want to apply tree optimizations in the third-party case and want to guarantee the server's privacy, we must create the tree in a generic way such that the resulting structure does not leak anything about the server's set. Hence, we discard the requirement that branches are always labeled with 0 on the left and 1 on the right. This obfuscates possible optimizations like pruning.[6] The server sends these encrypted values and the client sends the encrypted elements to the third party which applies the same evaluation as before except for an encrypted node evaluation algorithm. In contrast to the plaintext case, the evaluation has to be done homomorphically. We need the original Xnor as mentioned in the description of Algorithm 1. Thus, for a boolean input x, we want to compute

$$x \overline{\oplus} v.\mathsf{lEdge} \quad \text{and} \quad x \overline{\oplus} v.\mathsf{rEdge}.$$

Since $v.\mathsf{lEdge} = 1 - v.\mathsf{rEdge}$, we can simplify to

$$x \oplus v.\mathsf{rEdge} \quad \text{and} \quad x \oplus v.\mathsf{lEdge}.$$

The modification can be found in Algorithm 5 where lines 8 and 11 changed compared to the basic protocol.

We must also modify the evaluation of leaves (Algorithm 3). Instead of multiplying by a plaintext in step 4, we now must apply a homomorphic multiplication.

Overall, we need an additional homomorphic Xor for each node, i.e. depth of the BP additional homomorphic additions, and for each leaf a homomorphic multiplication instead of a constant multiplication. Hence, the multiplicative depth is increased by one.

[6] This obfuscation might not be enough for a full security proof such that we might need to omit tree optimizations.