A Framework of Private Set Intersection Protocols

Ziyuan Liang[1], Weiran Liu[2], Fan Zhang[1], Bingsheng Zhang[1], Jian Liu[1], Lei Zhang[2], and Kui Ren[1]

[1] Zhejiang University, AZFT
[2] Alibaba Group

Abstract. Private Set Intersection (PSI) is a specified protocol of secure Multi-Party Computation (MPC). PSI allows two parties to obtain the intersection of their private sets while nothing else is revealed. In contrast to the great demand for PSI in real-world applications, there is still no evaluation results of different general practical PSI framework. Most existing PSI implmentations are based on C/C++, which also makes them hard to compute in parallel. In this paper, we propose a generic Java-based PSI framework and implement all up-to-date OT-based PSI protocols within the framework until now. We evaluate these OT-based PSI protocols and the dependent cryptographic primitives and provide the best combination of primitives for constructing a best-performed OT-based PSI from the ground up. Additional optimizations are also applied to the protocols in our framework, including both generic and custom-tailored ones. We adopt filters to significantly reduce the communication of OT-based PSI protocols. The implementations in our framework support concurrence by using the natural feature of Java, which avoids to manurally allocate threads when using C/C++. We believe that our framework benefits a lot for future MPC and PSI researches and helps the promotion of PSI-based applications.

1 Introduction

1.1 Background

Secure multi-party computation (MPC) allows a group of data owners to jointly compute discrete functions on their private data, where the security of MPC requires the protocols to reveal nothing about the underlying data but the expected output. An ideal MPC protocol can be viewed as a trusted third party, who accepts private inputs, computes a function, and returns the result to the data owners. There are many general-purpose MPC frameworks [22,15,31,36,12,49,24] proposed in recent years.

Private Set Intersection (PSI) is a specific protocol of MPC which allows two parties to compute the intersection of their private sets. PSI is useful for applications where the parties need to apply a JOIN operation to private datasets, and there are many potential applications, such as mobile contact discovery, advertising conversion, and so on. However, general-purpose MPC frameworks cannot deal well with PSI. A generic PSI needs $O(n^2)$ comparisons, as each element of one party needs to be compared with each element of the other party. The performance of generic MPC becomes inefficient when the parties' sets become significantly large, and thus custom-tailored PSI protocols are in demand for practice.

Recent custom-tailored PSI protocols can be roughly divided into three categories: public key-based PSI [18,19,33,9,10], circuit-based PSI [17,45,26,21,43], and OT-based PSI [7,13,28,40,41,44,46,47,16]. Public key-based PSI has fewer communication costs but runs slowly because of the expensive public-key cryptographic operations. Circuit-based PSI runs fast but requires more communication.

OT-based PSI achieves a good balance between latency and communication costs. In contrast to that many cryptographers are trying to propose better PSI protocols with fewer costs, there is still no general-purpose PSI framework in terms of comparison and evaluation in practice. The experimental results in these works may be greatly varied because of the different execution environments, code qualities, and the choices of their dependent primitives. The choices of base primitives, such as hashes, PRFs, PRGs, and OTs, greatly influence the performance of the entire PSI protocols besides their upper-level designs. It is a great concern that how to construct a best-performed PSI protocol as well as its dependent primitives from the ground up. However, it is difficult to impartially compare the protocols unless there is a generic framework implementing and evaluating them under the same conditions. Another concern is that PSI development is not friendly to engineers, like other custom-tailored MPC protocols. It may take experienced cryptographers and engineers several months to design and implement PSI protocols. The research cost can be greatly reduced if the required dependent primitives are already packed in a framework. The researchers only need to focus on the upper-level design and invoke existing interfaces for its primitives. We are therefore motivated to propose a generic PSI framework for the sake of future PSI researches, and provide a general evaluation of all the OT-based PSI protocols.

1.2 Contribution

We propose a generic Java-based framework for all up-to-date PSI protocols, and until now, corresponding primitives and sub-protocols are implemented and packed as independent interfaces so that a protocol can be naturally constructed by combining these interfaces correctly. Afterward, we implement these PSI protocols in our framework. It takes a graduate student around two days to independently implement an existing PSI protocol based on our framework. To the best of our knowledge, our framework is the first one that is generally implemented and evaluates all these PSI protocols.

We provide an impartial evaluation of all the primitives and protocols, which consists of their latency and communication. The optimal choosing strategies for each interface are presented based on the experimental results, and we construct the PSI protocols with better performance using the optimal primitives.

Another contribution is that we apply multiple optimizations on these PSI protocols in our framework, including both generic and custom-tailored ones. We find the commons of most OT-based PSI protocols, and utilize filters to reduce the communication cost of these protocols. Parallelization is also considered in our framework. Some custom-tailored optimizations are presented to make up the shortfalls existing in some PSI protocols.

In brief, our main contributions lie as follows:

- We propose the first Java-based general-propose PSI protocols and implement up-to-date OT-based PSI in our framework until now.
- We evaluate the PSI protocols and compare their performance. We also provide the best combinations of primitives and protocols with the least costs based on the experimental results.
- Several strategies are applied to the PSI protocols to achieve better performance, including both general and custom-tailored optimizations.

1.3 Roadmap

The rest of this paper consists of the following sections. Required cryptographic and MPC primitives are presented in Section 2. We describe all up-to-date OT-based PSI protocols in our framework

in Section 3. We introduce our framework in Section 4 and provide corresponding details. Our optimizations are described in Section 5. We evaluate primitives and PSI protocols in our frameworks, and the results are presented in Section 6. Based on the results, we provide the optimal combinations of the primitives and protocols in terms of the best performance in Section 7. Related works of our paper lie in Section 8. Finally, our conclusion is given in Section 9.

2 Primitives and Building Blocks

In this section, we provide a brief introduction to cryptographic primitives common to many PSI protocols, including oblivious transfer (OT), polynomial primitives, and filters.

2.1 Security

The primitives and protocols of MPC are proved to be secure in the presence of semi-honest or malicious adversaries. In the semi-honest model, the adversary behaves curiously. It controls one of the parties and still follows the protocol specification honestly. However, it may try to learn more input information about the other honest party. A malicious adversary is not required to follow the protocol exactly and may try to get private information with all possible approaches.

2.2 Oblivious Transfer

OT is one of the most fundamental protocols of MPC. The most common form of OT is 1-out-of-2 OT. In the beginning, the sender holds a pair of message (m_0, m_1), and the receiver holds a choose bit $b \in \{0, 1\}$. The final goal of OT is to allow the receiver to learn m_b, whilst the sender learns nothing and the receiver learns no information about m_{1-b}. 1-out-of-2 OT can be easily built based on public-key cryptography [4,37,8,32]. A simple case in [4] requires that the receiver generates a key pair (sk_b, pk_b) and a single public key pk_{1-b} without learning the corresponding private key. Then the sender encrypt $m_i (i \in \{0, 1\})$ with pk_i, and obviously the receiver can only decrypt m_b. To guarantee that the receiver learns nothing about sk_{1-b}, the parties can constrain pk_b and pk_{1-b} to satisfy a specific relationship to prevent the receiver to generate (pk_{1-b}, sk_{1-b}).

Public key cryptography is expensive, however, most MPC protocols use a large amount of OTs, and that is the major motivation of the researches of OT Extensions (OTE). There are two species of OTE. One is 1-out-of-2 OTE [20,1,2,23,3], which tries to generate the amount of 1-out-of-2 OT instances by using a few base OTs and cheap symmetric cryptographic primitives. The main idea of Ishai et al. [20] is based on matrix transposition. The sender and the receiver first operate a few reversed OT instances, where the receiver generates two random matrices T_0 and T_1, and the sender chooses the columns of matrices based on each bit of a secret random string s. Note that each column of $T_0 + T_1$ equals the choose bit string of the receiver, and the sender receives a matrix Q whose columns are the same as T_{s_i}. Thus each row of the matrices constructs an OT instance because the ith row of T_0 equals either the ith row of Q or this row XORed with s, based on its choice bit. The possibility of cheap OTs made a significant difference for MPC protocols with large inputs such as PSI.

Another kind of OTE is 1-out-of-n OTE [27,28,38]. Kolesnikov and Kumaresan [27] first pointed out that the protocol [20] can be explained from the perspective of encoding. Every row of Q can be regarded as $q_i = t_i \oplus C(b_i) \odot s$. They naturally expanded 1-out-of-2 OT in [20] to a 1-out-of-n case by replacing the replicated code with the Walsh-Hadamard code with sublinear communication.

However, the above OT schemes only guarantee semi-honest security. A malicious receiver gets access to the secret string s with customized incorrect coding strings, and the honest sender cannot distinguish the incorrect coding during the entire scheme. A straightforward solution is applying a consistency check on the semi-honest OTE schemes to prove that the receiver is honest. Orru et al. [38] proposed a malicious secure OTE scheme leveraging binary linear code, such as BCH code. The sender and the receiver generate some additional OT instances. After receiving Q, the sender generates some random strings and requires the receiver to apply linear operations with some OT instances randomly determined by the sender. If the receiver is honest, the sender gets the same results from applying the operations on Q's elements, otherwise, the receiver only has a negligible probability to pass the check. There are other malicious secure OTE schemes such as [2,23]. A malicious OT-based PSI scheme is required to adopt malicious OTE schemes for security considerations.

2.3 Polynomial Evaluation and Interpolation

Polynomial evaluation and interpolation play an important role in some PSI protocols [30,40]. Evaluation refers to compute the values on a specific point set $\{x_i\}$ given the coefficients of a polynomial, and interpolation refers to reconstruct the coefficients given a point set $\{(x_i, P(x_i))\}$. There are two main advantages to utilize polynomials in PSI. The point representation of polynomials has good homomorphic properties, which can be used in homomorphic encryption-based MPC protocols like Shamir's secret sharing. Another property of polynomial is that interpolating the unique polynomial needs at least $n+1$ points where n is the degree of the polynomial, which can be used to verify the cardinality of the intersection.

In most scenarios, the degree of the polynomials could be significantly high. Moenck et al. [34] proposed an efficient algorithm for quick evaluation and interpolation. The main idea is the construction of the sub-product trees. A sub-product tree is a full binary tree. The values of the leaves are one-degree polynomials $(x - x_i)$ where $\{x_i\}$ is the point set in both evaluation and interpolation cases. The value of parent nodes is the product of the corresponding child nodes. When multi-point evaluation, the evaluation polynomial traverses from the root node to the leaves. Every time the polynomial passes a node, the tree divides the polynomial by the value of the node and sends the remainder to child nodes. Note that the remainder always has a lower degree than the divisor, thus the leaves get constant results. For the i-th leaf node, all values of the nodes on the routine from the root have the same factor $(x - x_i)$, and the high-degree part thrown away is equal to 0 when $x = x_i$, thus the final constant result of each leaf is the corresponding evaluation result $P(x_i)$.

One approach of interpolation utilizes the "divide and conquer" representation of Lagrange interpolation formula Eq. 1, where $a_i = \frac{1}{\prod_{j\in\{n\}\&j\neq i}(x_i - x_j)}$. $M_R(x)$ and $M_L(x)$ refer to the values of the right and left nodes of the root, and this recursive relationship can be promoted from root to the leaves. If we get the smallest sub-equations on the leaves, we can compute from leaves to root to get the final interpolation result. The sub-equation on each leaf equals to $y_i a_i$. Note that $\frac{1}{a_i}$ is the value of $p'(x_i)$ where $p'(x)$ refers to the derivation of $p(x) = \prod_{i\in\{n\}}(x - x_i)$. Obviously $\{a_i\}$ could be computed by the fast multi-point described above, and can reuse the sub-product tree of the interpolation.

$$
\begin{aligned}
P(x) &= \sum_{i\in\{n\}} y_i \cdot L_i(x) \\
&= \sum_{i\in\{n\}} y_i \cdot a_i \prod_{j\neq i}(x - x_j) \\
&= \sum_{i=1}^{n/2} y_i \cdot a_i \prod_{j\neq i}(x - x_j) + \sum_{i=n/2+1}^{n} y_i \cdot a_i \prod_{j\neq i}(x - x_j) \\
&= P_L(x) \cdot \prod_{i=n/2+1}^{n}(x - x_i) + P_R(x) \cdot \prod_{i=1}^{n/2}(x - x_i) \\
&= P_L(x) \cdot M_R(x) + P_R(x) \cdot M_L(x)
\end{aligned} \tag{1}
$$

The evaluation and interpolation algorithms are concise and beautiful, but there are other detailed problems in implementation. There are amounts of polynomial multiplication and division operations in the algorithms, which greatly affects the complexity. Approaches such as FFT and NTT are recommended for their $O(NlogN)$ complexity. Another practical problem is that the evaluation and interpolation point sets are not always equal to 2^n to fill balanced sub-product trees. The padding strategies are different for evaluation and interpolation. For evaluation, the points can be separated into several smaller sub-product trees, because the evaluation is independent for each point. Irrelevant points can also be padded to the sub-product trees. Note that the padded points cannot be 0. The strategy of polynomial division involves obtaining the inverse of the divisor, and the zero points that lead to divisors cannot be inverted. Moreover, the leaves of a single tree are required to be no more than the degree of the evaluation polynomial to guarantee the degree of the root value to be no more than that of the polynomial, or the first division fails. As for interpolation, there is only a single tree, and 1s can be padded to empty leaves.

2.4 Filters

Filters are a kind of approximate set membership data structure, which helps to check the existence of elements in a set without providing all set members. The canonical one is the Bloom filter [5], which maps an element to several bit entries with multiple hash functions. Bloom filter is a table of buckets with a fixed bucket size of 1. There are k different hash functions. When x is inserted, it flips all bits on positions $\{h_i(x)\}$ to 1. To check whether x is contained in the filter or not, it checks the positions $\{h_i(x)\}$. If all $\{h_i(x)\}$ equal to 1, x is considered to be contained, and vice versa. Note that the Bloom filter guarantees "absolutely no" but only "possibly yes". Even a value is not inserted, its hashing positions might be occupied by other inserted values, and maybe further regarded as inserted. The false-positive rate is controllable with appropriate parameter settings. For the same reason, the Bloom filter does not support deleting an inserted value.

Cuckoo filter [14] is a data structure similar to Bloom filter [5], which allows for fast membership testing. The cuckoo filter uses cuckoo hashing to avoid the shortcomings of the Bloom filter. Cuckoo hashing[39] is a different hashing strategy compared with common hash tables. Cuckoo hashing uses an m-entry array with k different hash functions $\{h_i\} : \{0,1\}^* \to [m]$. When a value x is inserted, it randomly picks i, and inserts x into the entry $h_i(x)$. If the entry has been occupied, insert the value evicted with the same strategy, until the insertion does not lead to eviction, or the times of eviction reaches a threshold. In principle, it can also check whether other entries $h_j(x)_{(j\neq i)}$ are empty or

not before performing an eviction, and inserts the value into other empty entries. If the times of eviction reaches the upper bound, the latest evicted value is placed into small-size storage called stash [25]. If the stash is full, it leads to a failure and leads to rehashing.

Different from the Bloom filter, the inserted entry size of the cuckoo filter is fixed to the length of a tag, and the empty entries are padded with a one-byte placeholder. There are two hash functions in the cuckoo filter. We use one to calculate the tag $H_t(x)$, and use the other to calculate the first entry $H_1(x)$. The second possible entry is calculated with $H_t(x) \oplus H_1(x)$. The insertion and eviction strategies are the same as those of cuckoo hashing. The reason why the second possible entry is not calculated by another hash function is that, when an eviction occurs, the cuckoo filter might not get access to the original x. In this way, the other possible entry can be easily computed by the current entry and the tag. Compared to the Bloom filter, the cuckoo filter supports deletion and performs better for lookup because fewer hash functions are computed. There are other filters such as Morton filter [6] and Vacuum filter [48], which further considered data's local dependencies to reduce memory loading from disks. Morton Filters use fixed-size block stores to further compress the empty blocks in the cuckoo filter, and provides a fingerprint counter array to record the positions of fingerprints in the storage array, which achieves a better compression rate than the cuckoo filter. Vacuum Filter uses a "divide-into-chucks" strategy to reduce the wasted space in the cuckoo filter. Note that the entries are required to equal the exponents of 2. The load factor of the cuckoo filter is only around 50% when it switches from the original entry number to the next large exponent. Vacuum filters divide the entry space into several chucks with fixed sizes, and the two candidate entries of one element lie in the same chunk, and thus vacuum filters steady the compression rate of the cuckoo filters. Vacuum filters additionally adopt different sizes of chucks to further improve the memory access by the local dependencies of the candidate entry pairs. However, the parameter settings of the vacuum filter mentioned in [48] cause high insertion fault rate. We fix this problem by adjusting the parameters later in Section 5.

3 Protocols

We briefly introduce the OT-based PSI implemented in our framework, which is divided into two categories based on their security assumptions.

3.1 Semi-honest Secure PSI

Hazay-Lindell (HL) 08 Hazay and Lindell [16] proposed one of the earliest OT-based PSI protocols. It uses OT in PSI protocols in a direct way. The precondition of a PSI protocol is to construct an Oblivious Pseudo-Random Function (OPRF). It inputs x from P_1 and nothing from P_2, and outputs k to P_2 and $PRF_k(x)$ to P_1, during which P_1 learns nothing about k, and P_2 learns nothing about x. P_1 obtains PRF results of all its inputs. P_2 locally computes PRF results of its inputs with $\{k\}$, and sends them to P_1. P_1 compares the PRF results and obtains the corresponding intersection. OPRF in HL08 is constructed with public-key cryptography. The main idea is that each bit of an element matches a pair of subkeys. The OT receiver gets one of each pair by regarding its element bits as choose bits. Each element corresponds to a specific key multiplied by every bit's subkey, and thus the OPRF is constructed.

Dong-Chen-Wen (DCW) 13 Dong et al. [13] utilized a garbled Bloom filter and a Bloom filter to construct efficient semi-honest PSI protocols. Garbled Bloom Filter is a new data structure first

proposed in [13], which replaces the bits in hash entries with random strings, and guarantees that the xor result of the k entries equals a specific map value of the inserted element. The receiver generates a Bloom Filter and the sender generates a garbled Bloom filter based on their own sets. The receiver inputs bits of the Bloom filter as choose bits in OT, in which the sender delivers the string of garbled Bloom filter or random string when the choice bit equals 1 or 0. Note that replacing the garbled Bloom filter strings of non-intersection entries has no effect on the entries of the intersection, and thus the receiver checks the intersection garbled Bloom filter to obtain the final results. They also proposed a malicious PSI protocol, however, Mikkel [29] proved that the malicious protocol in [13] is insecure against a malicious receiver. Rindal and Rosulek improved this idea to malicious security in [46]

Pinkas-Schneider-Zohner (PSZ) 14 Pinkas et al. [44] proposed two PSI protocols. One is the optimization of DCW13. Note that DCW13 requires 1-out-of-2 message OT to deliver the intersection garbled Bloom filter, and PSZ14 pointed out that it can be replaced with random OT for delivering a random garbled Bloom filter, which can reduce a round of communication in OT, and save the amount of communication cost. Another PSI protocol in PSZ14 is based on hashing, which utilizes cuckoo hashing to split elements into entries. On the one hand, an element is only possibly inserted into two specific entries, which simplifies the comparison of masked results. On the other hand, if one party inserts elements into cuckoo hashing, and the other inserts into simple hashing of the hash functions, and thus the original PSI is divided into smaller scales.

After cuckoo hashing insertion, the elements are split into bytes, and each byte corresponds to a 1-out-of-256 OT instance of [27]. The OPRF result of the element is constructed by XORing random OT keys of each byte, and then two parties obtain the intersection by comparing each hash bin's OPRF results.

Kolesnikov-Kumaresan-Rosulek-Trieu (KKRT) 16 Kolesnikov et al. [28] proposed an efficient semi-honest PSI scheme. KKRT16 benefits a lot from its efficient OPRF, which is naturally extended from 1-out-of-n OT [27]. In the sender's OTE matrix Q, every row equals to $q_i = t_{0i} \oplus C(r_i) \odot s$, where r_i is the choose string. When C denotes a pseudorandom code, $q_i \oplus C(r_j) \odot s$ can be regarded as $PRF_{s,q_i}(r_j)$. Because $q_i \oplus C(r_j) \odot s = t_{0i} \oplus (C(r_i) \oplus C(r_j)) \odot s$, it equals to t_{0i} only when $r_j = r_i$. The sender learns s and q_i, and can evaluate PRF_{s,q_i} on any r_j, while learning nothing about r_i. The receiver obtains $t_{0i} = PRF_{s,q_i}(r_i)$, while learns nothing about s and q_i. Thus every row of Q represents a PRF instance of key (s, q_i). All instances shares the same s but have different q_i. The construction is named as Batched Related Key OPRF (BaRK-OPRF).

Note that BaRK-OPRF is a single-point OPRF instance, which indicates that the OT receiver can only evaluate once at a specific point. Thus KKRT16 also adopts the strategy of cuckoo hashing and simple hashing. Two parties insert their elements into cuckoo hashing and simple hashing respectively, then apply OPRF to each hash bin. If there are identical PRF results in the same entry, the corresponding elements belong to the intersection. Unfortunately, KKRT16 becomes insecure dealing with malicious adversaries like [27]. A corrupted receiver can steal secrets from the sender with illegal coding.

Pinkas-Rosulek-Trieu-Yanai (PRTY) 19 Pinkas et al. [40], proposed a PSI protocol, named SpOT-Light. The OPRF protocol in PRTY19 is a multi-point one, different from the single-point

one in KKRT16. Therefore, the sender and receiver do not need cuckoo hashing and corresponding dummy block paddings, which may save communication costs.

The multi-point OPRF is constructed from 1-out-of-2 OT in [20]. Under semi-honest assumptions, the first messages of ith rows of the OTE matrix, $m_{i,0}$ where $i \in [N]$, construct an OPRF as $F(i) = m_{i,0}$. When $i \in X$, the receiver sets the corresponding choose bit as 0, otherwise 1. Thus he only gets access to $F(X)$, while the sender gets $F(i)$ for all $i \in [N]$. Then the sender computes $F(Y)$, and the receiver obtains the intersection by comparing $F(X)$ and $F(Y)$. However in real-world PSI, N might be exponentially large, making this approach extremely inefficient. To reduce the communication cost, PRTY19 points out that because $n_1, n_2 << N$, the receiver can interpolate a polynomial to focus the useful point set $(X, F(X))$, and sends the polynomial to the sender, instead of sending the N-row matrix. The OPRF results of the intersection stay identical, but communication costs become less.

The biggest challenge of this approach is evaluating and interpolating an extremely high-degree polynomial, which slows down the performance. Thus the authors presented a fast version of PRTY19, different from the above low-communication one. They proposed 2-choice hashing which is similar to cuckoo hashing, but multiple elements can be inserted into the same entry. 2-choice hashing has two hash functions identical to cuckoo hashing, and an element is always inserted into the entry with fewer elements between the two candidates. The receiver interpolates low-degree polynomials for all the entries and the sender computes $F(y)$ with 2 candidate polynomials for each y. This approach improves performance but increases the communication cost. The authors recommended that users can choose the low-communication or the low-computation version of PRTY19 based on their application scenarios.

Chase-Miao (CM) 20 Chase and Miao [7] proposed a PSI protocol for networks with moderate bandwidth. They constructed a multi-point OPRF to obtain the intersection. At a high level, the sender constructs two matrices A and B of special form from its input elements. The bits in the matrices corresponding to the elements are set to be identical, where the others are set to be different. Then the columns of the matrices are inputted to OT as the sender's inputs. That means, if a receiver's element is in the intersection, then its input to the subsequent hash function will be the same as one of the sender's inputs to the hash function, and vice versa.

3.2 Malicious Secure PSI

Rindal-Rosulek (RR) 16 Rindal and Rosulek [46] further improved the idea of DCW13 [13] to malicious security in an efficient approach. The main idea is a cut-and-choose challenge. Two parties generate more OT instances than those the Bloom filter actually needs. The sender randomly chooses some OT instances and requires the receiver to open the selected OTs, which authenticates whether the receiver operates honestly or not. The sender aborts when receiving an incorrect authentication result. As the sender chooses the set randomly, the entries in the Bloom filter and the unopened OT instances do not match naturally. Thus the receiver generates a permutation that permutes OT instances into the order of Bloom filter according to the choose bit string. Then the sender and receiver operate the semi-honest protocol like [13].

Rindal-Rosulek (RR) 17 Rindal and Rosulek further proposed RR17 [47], which is much more efficient than RR16. The idea of RR17 comes from the PSZ14 protocol [44], and mainly adopts

OT extensions and phasing hash from [42]. RR17 is extended to malicious security using a dual execution strategy. The parties first hash their elements into multiple bins. For each bin, the sender and receiver apply μ OT-based OPRF instances. The sender gets $\{\mathbf{F}_i\}$, where $i \in [\mu]$ and the receiver gets $\mathbf{F}_i(x_i)$. Then they apply reversed OPPF, which performs identically as the first round except for the direction. The sender gets $\mathbf{F}_j^{inv}(y_j)$ and the receiver gets $\{\mathbf{F}_j^{inv}\}$, where $j \in [\mu]$. The final OPRF is constructed as follows. The sender independently computes $\mathbf{F}_i^Y(y_j) = \mathbf{F}_i(y_j) + \mathbf{F}_j^{inv}(y_j)$ for any (i, j), and the receiver independently computes $\mathbf{F}_j^X(x_i) = \mathbf{F}_i(x_i) + \mathbf{F}_j^{inv}(x_i)$. $\mathbf{F}_i^Y(y_j)$ equals to $\mathbf{F}_j^X(x_i)$ only when $x_i = y_j$. The authors also provided an Encode-Commit version of their protocol to offer communication/computation trade-offs.

Pinkas-Rosulek-Trieu-Yanai (PRTY) 20 Pinkas et al. [41] constructed a multi-point OPRF protocol with 1-out-of-N OTE [38]. The authors proposed a new data structure named probe-and-XOR of strings (PaXoS). PaXoS can be regarded as a generic garbled Bloom filter [13]. The key-value pairs are encoded as a string vector $D = \mathtt{Enc}(\{(x, y)\})$, and every key corresponds to a Boolean vector $v(x)$. The value y is decoded as $y = \mathtt{Dec}(D, x) = v(x) \cdot D$, where y is usually implemented as the hash value $H(x)$. PaXoS has a good linear property. There are two approaches to encode D. One is generating random Boolean v, which has poor efficiency. The encoding requires solving a random linear system of equations on an extremely large scale, and decoding requires to XOR half of the strings in D on average. The other is the garbled Bloom filter, which has good efficiency but requires more space. The authors proposed a new PaXoS named garbled cuckoo table, which combines the idea of garbled Bloom filter with a small-scale random matrix, and $v(x) = l(x)||r(x)$. $l(x)$ indicates the long vector with only two non-zero entries corresponding to the two hash function results, and $r(x)$ indicates the random part. The garbled cuckoo table requires less storage but obtains a good efficiency meanwhile.

The authors pointed out that if D is inputted as the choose bit matrix in [38], it can be easily inferred that $q_i = t_i \oplus C(d_i) \odot s$ as discussed in Section 2. Then, we have

$$\begin{aligned} F(x) = \mathtt{Dec}(T, x) &= \mathtt{Dec}(Q \oplus C(D) \odot s, x) \\ &= \mathtt{Dec}(Q, x) \oplus \mathtt{Dec}(C(D), x) \odot s \\ &= \mathtt{Dec}(Q, x) \oplus C(\mathtt{Dec}(D, x)) \odot s \\ &= \mathtt{Dec}(Q, x) \oplus C(H(x)) \odot s \end{aligned} \tag{2}$$

where x is an element and C is a linear coding scheme. The left part $\mathtt{Dec}(T, x)$ can be executed by the receiver independently, and the right part $\mathtt{Dec}(Q, x) \oplus C(H(x)) \odot s$ can be executed by the sender. Thus a multi-point OPRF is constructed to be used in PRTY20 PSI protocol. Another advantage of PRTY20 is that it can be extended from semi-honest cases to malicious cases naturally by enlarging the scale of the OTE matrix instead of changing the phases of protocols.

4 Our Framework

4.1 Overview

We implement a generic mixed-protocol PSI framework, which includes state-of-the-art OT-based PSI protocols. It is designed in a user-friendly modular and extensive way, intended for future PSI research and practice. We implement our framework in Java, which includes around 30,000 valid code lines as well as around 13,000 comments.

Figure. 1 shows an overview of our framework. It consists of three main modules from the bottom layer to the top: Common Tools, OTE, and PSI. Common Tools includes multiple sub-modules which implement the basic tools and cryptographic primitives invoked by other high-level modules. OTE module is constructed based on the common tools, which include up-to-date OT and OTE protocols. Thus the OT-based PSI protocols can be constructed in the PSI module based on OTE and other primitives.

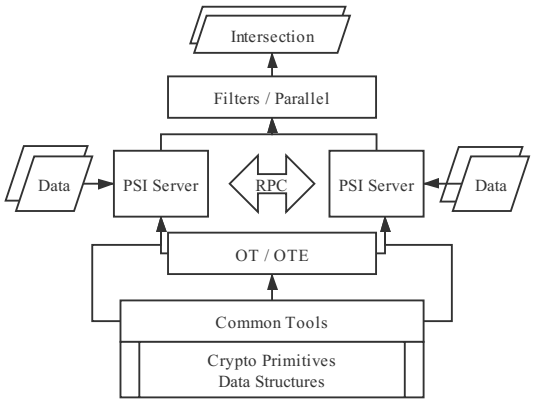


Fig. 1: Overview of our Framework.

We propose generic abstract interfaces for all possible MPC protocols, and the participants in the protocols are extracted as abstract classes. We divide MPC protocols into 2PC and MPC. In 2PC protocols, each party defines the protocol specifications (ProtocolSpec), the parties' specifications (PartySpec), and the Remote Procedure Call (RPC) instance used for communication.

4.2 Common Tools

Pseudo-Random Generator (PRG) and Pseudo-Random Function (PRF) are two basic cryptographic primitives. A PRG $\mathcal{G}$ holds a fixed random seed and an input data block. The output of $\mathcal{G}$ is a random element in the output space. A PRF is a deterministic algorithm with two inputs, the input block, and a key. The output is a deterministic element in the output space but seems random. We implement PRG and PRF with hashes and block ciphers in our framework. We provide three constructions to build a PRG, respectively based on CTR-mode block cipher, stream cipher, and hashes. Furthermore, we provide two constructions of PRF under different assumptions. In the random oracle model (ROM), we construct PRF by hashing the concatenation of the input and key, as the collision-resistant hash function is regarded as a random oracle. $F_{ROM}(k, x) = H_l(k||x)$. In the standard model, our PRF is constructed with PRG and block ciphers such as AES. We refer to the PRF instantiation in [7]. We use the CBC-mode block cipher to fix the input length of PRF, and then use a PRG to extend the output to an arbitrary length. $F_{SM}(k, x) = PRG_l(Enc_{CBC}(k, x))$. Thus we obtain a PRF with arbitrary input and output length. We define our main interfaces in PRG and PRF classes as below. The dependent hashes and ciphers come from JDK or Bouncy Castle (BC) libraries, and a detailed evaluation is presented in Section 6.

```
Prg.extend(byte[] key, int randomBitLength);
Prf.setKey(byte[] key);
Prf.getBigInteger(byte[] message, int bitLength);
```

Another optimization is that we implement factories that automatically choose the basic primitives and building blocks with the best performance based on the expected input and output length, including hashes, ciphers, PRFs, and PRGs. For example, an optimal PRF can be initialized by invoking the factory interface, based on the most frequent bit length of the intended output.

```
PrfFactory.getInstance(int mostFreqBitLength);
```

Then some building blocks are constructed based on the basic primitives, including binhashes, filters, and other data structures. The implementation of binhashes includes [42,39,40]. We extract binhashes as an abstract class, and different species of binhashes extend different insertion strategies. The optimal parameters are hardcoded in its implementation, and thus it automatically completes initialization based on the expected insertion number.

We also implement an abstract filter class as well as some species such as BigInteger set filter, Bloom filter [5], sparse Bloom filter [46], and cuckoo filter [14]. The BigInteger set filter is a simple list containing all inserted elements in the format of BigInteger and can be regarded as a filter with a compression rate of 1. We provide a filter factory that automatically sets the parameters based on the filter species and expected insertion number. On the one hand, several OT-based PSI protocols invoke specific filters [13,46]. On the other hand, we prove that filters help reduce the communication costs for most OT-based PSI protocols. The raw idea is to replace the final step of sending and comparing the OPRF results. We require the sender to construct and send a filter instead, and then the receiver receives the filter and checks if each of his OPRF results is contained in the filter. Thus the final round communication is compressed. We also provide a formal security analysis in Appendix A, to demonstrate that under certain conditions the strategy guarantees an identical security level to the original protocols.

PaXoS is another data structure we implement in our framework, which acts as a filter but aims at encoding and decoding instead of compression. The details have been mentioned in Section 3. Our framework includes three different PaXoS species, including random matrix PaXoS, garbled bloom filter [13], and garbled cuckoo table [41].

4.3 RPC

Another optimization of our framework is that our remote procedure call (RPC) interfaces are separated independently from other cryptographic operations. An RPC class records the participants and provides interfaces such as connecting, disconnecting, sending, receiving, which dealing with data packages in MPC protocols. A data packet consists of a data packet spec containing the metadata for configuration, and a BigInteger list containing the data expressed in the format of BigInteger.

We implement three RPC species in our framework. Network RPC is implemented by TLS handshake protocols. Memory RPC is a cross-thread RPC designed for local tests, whose data sender and receiver stay in the same machine, and they read and write on the same piece of the memory region. A memory RPC instance preserves a specific memory buffer. The sender puts the data packet

in the buffer as a key-value pair, where the metadata and data act as the key and value respectively. The receiver receives the data packet by reading the corresponding value in the hashmap according to the metadata. As memory RPC needs no initialization, the connect / disconnect interfaces are set to be empty. Another RPC is file RPC. As its name suggests, the sender packs the data packets into files with names concatenated by their configure information, and the receiver reads the data packets by reading the files. Note that the sender is required to write an additional status file, to prevent the conflict that the receiver reads before the sender finishes writing. The status file contains a short string and is written after the data file is finished. When the receiver queries to receive the data packet, it first checks the existence of the corresponding status file. If exists, it deletes the status file and reads the data packet correctly. If not, it sleeps to wait for the sender to finish writing. Similar to memory RPC, file RPC has no initialization and empty connect/disconnect interfaces. File RPC is convenient to be extended into remote mode when the sender and receiver get access to the same cloud database.

4.4 OT

The OT implementations in our framework can be divided into three abstract classes, base OT, 1-out-of-2 OTE, and 1-out-of-n OTE. Each protocol in our framework preserves a spec class, which defines the communication steps and security model. The steps help divide a protocol into several pieces based on its communication rounds. Thus the original protocol can be regarded as the two (maybe more) parties sending and receiving rounds of data packets as well as locally generating and handling them. This model helps us to extract the similarities of various protocols into abstract classes. The security model is also labeled in the class. A complex protocol is constructed by multiple simpler building blocks, which are further constructed by basic primitives. Note that the building blocks and primitives have different security models, and the security model of a high-level complex protocol is determined by the most insecure building block it adopts. For example, if a malicious-secure PSI protocol instance invokes a semi-honest secure OTE, the PSI instance will be regarded to be semi-honest secure.

We implement both random OT and message OT of each OT/OTE protocol, and we discover that almost all OT/OTE protocols implement random OTs to determine the random key arrays. The real messages are encrypted and decrypted using these keys. Thus we implement the conversion from random OT/OTE to message OT/OTE in the outside abstract OT/OTE classes and require each protocol to implement random OT/OTE respectively based on their design. Several efficient 1-out-of-2 base OT protocols [37,8,32] are implemented in our framework, and the interfaces are packed as follows.

```
% AbstractBaseOtSender
void sendMessagePair(Cipher cipher, int messageBitLength, BigInteger[]
    messagePair);
BigInteger[] sendKeyPair();
% AbstractBaseOtReceiver
BigInteger receiveMessage(Cipher cipher, int messagBitLength, boolean choice)
    ;
BigInteger receiveKey(boolean key);
```

The OTE protocols in our framework are classified as 1-out-of-2 OTE [1,3,20,23], and 1-out-of-n OTE [27,28,38]. The corresponding interfaces are abstracted as follows.

```
% AbstractToOteSender
void sendMessagePairArray(Cipher cipher, int messageBitLength, BigInteger[][][]
    messagePairArray);
BigInteger[][][] sendKeyPairArray(int arrayLength);
% AbstractToOteReceiver
BigInteger[] receiveMessageArray(Cipher cipher, int messagBitLength, boolean
    [] choiceArray);
BigInteger[] receiveKeyArray(boolean[] choiceArray);
```

```
% AbstractNoOteSender
void sendMessageTupleArray(Cipher cipher, Hash hash, int messageBitLength,
    int tupleSize, BigInteger[][][] messageTupleArray);
BigInteger[][][] sendKeyTupleArray(Hash hash, int arrayLength, int tupleSize);
% AbstractNoOteReceiver
BigInteger[] receiveMessageArray(Cipher cipher, Hash hash, int
    messagBitLength, int tupleSize, int[] choiceTupleArray);
BigInteger[] receiveKeyArray(Hash hash, int[] choiceArray);
```

Cipher refers to the encryption scheme to encrypt messages with random OT keys. Hash is the hash function used in 1-out-of-n OTE.

Note that the choices in 1-out-of-n OTE are encoded by coders as mentioned in Section 2, and thus we additionally implement coders as primitives in our common tools, including the reputation coder in [20], Walsh-Hadamard coders in [27], linear BCH coders in [38,41], and pseudo-random coder in [28]. Matrix transposition is the major performance bottleneck of OTE. We implement and test several different types of bit matrix to achieve the best performance. We implement bit matrices in the format of binary, BigInteger, and byte array, and find out that the byte array format using the interaction transposition strategy in [1] performs best. An interesting point is that the efficient interaction transposition does not work well in parallel because of the data dependency, and achieves similar performance compared with naive transposition.

4.5 PSI

With all these primitives completed, we are fully prepared to implement any up-to-date OT-based PSI protocols, including [7,13,28,40,41,44,46,47,16]. The PSI participants can directly invoke OT/OTE instances without defining the communication outside OT/OTE. We also implement the naive PSI and DH-based PSI [18] as benchmarks for performance comparison.

Another of our biggest advantages is that new PSI protocols can be easily developed and extended by adding the required primitives, which provides a user-friendly platform for comparing and developing PSI protocols. For example, if we further equip our framework with a circuit submodule, then up-to-date circuit-based PSI protocols can be implemented in our framework, which is planned to be our future work.

5 Optimizations

We apply multiple optimization strategies in our framework to reduce the latency or communications of the protocols, including both general strategies and custom-tailored modifications.

Table 1: Communication Costs of Filters.

Filter Name	Entry Num	Entry Size	Filter Size
BigInteger Set	n	l	nl
Bloom Filter	$n\lambda/ln_2$	1	$n\lambda/ln_2$
Cuckoo Filter	$2^{\lceil log(0.26n+1)\rceil}$	48	$48n + 8 * 2^{\lceil log(0.26n+1)\rceil}$

5.1 Saving Communication Using Filters

The final step of most OT-based PSI protocols is that the sender sends his OPRF results to the receiver, who will later compare the OPRF results to obtain the intersection. Note that the OPRF result intersection of the parties can be regarded as multiple public set inclusion operations, and a set inclusion can be naturally implemented by checking the element in the filters.

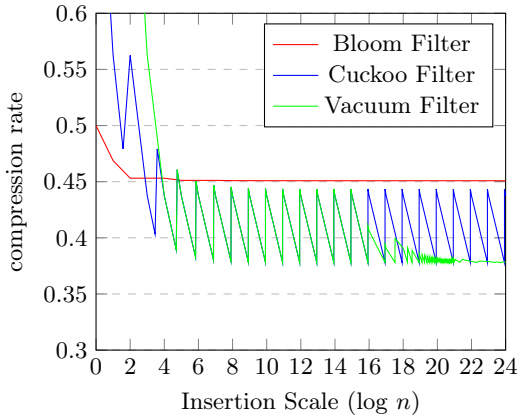
In our optimized protocols, the sender sends a filter in which his OPRF results are inserted, and the receiver successively checks whether each of his OPRF results is contained in the filter or not to determine the output intersection. The advantage is that the communication cost can be reduced according to the compression rate of the filter.

We compute the storage occupation of the Bloom filter and cuckoo filter and compare them with the plaintext set filter as a baseline. The theoretical occupation of the filters is presented in Table 1. Vacuum Filter is not listed because of its complex structure, which makes it hard to evaluate its compression rate theoretically. The empty entries in the cuckoo filters are padded with single-byte placeholders rather than 48-bit fingerprints. Then we fix the false positive rate of the filters to 2^{-40} and test the compression rates of the filters in practice. The results are presented in Figure. 2, which demonstrates that filters can efficiently reduce communication when there are a large number of insertions, and the cuckoo filter always performs better than the Bloom filter when n is larger than 2^5. The compression rate of the set filter depends on the element length l, and l depends on the protocol security, which is always not smaller than $\lambda + log(n_1 n_2)$. Thus we recommend using filters. Another concern is that the fixed entry number leads to the strange jaggies of cuckoo filter, which is required to be padded to 2^i, where $i \in \mathbb{N}^+$. Thus the peaks and valleys appear at the threshold points, which only depend on the load factor of the cuckoo filter. The vacuum filter performs identically to the cuckoo filter when there are less than 2^{16} elements expected for insertion. As the insertion scale grows even larger, the compression rate of the vacuum filter becomes smooth and steady and keeps at the valleys of the cuckoo filter. Thus we recommend using vacuum filters when the expected insertion is larger than 2^{16}.

The original parameter settings in Vacuum filters bring an non-negligible probability of insertion faults. The 4-layer chucks lead the inserted items to gather in smaller chucks, and many items expected to be in larger chucks will hold two entries in the same smaller chucks by accident when the size of two different chucks are similar. Thus the items appearing in smaller chucks can be much

more than expected. We adjust the parameters of the chuck sizes to fix these problems by adding the expected load factors of smaller chucks from (0.75, 0.5, 0.25) to (0.9, 0.7, 0.4), which significantly reduces the insertion faults, and keeping the compression rate of the original design.

Fig. 2: The compression rates of filters under different expected insertion. The filters have identical false positive rate as 2^{-40}.



Another concern is its security effects on the original protocol. We provide a detailed security analysis in the random-oracle model in the Appendix based on simulation security proofs. We conclude that the filters do not break the security when its false-positive rate is small enough. In the original protocol, the length of OPRF results is required to be long enough based on the set size for security. Similarly, when using filters, the parties are required to choose an optimal filter based on the number of OPRF instances and expected insertions. The false-positive rate of a filter depends on its structure and parameters, and our framework automatically chooses the best parameters.

In addition, we have considered whether this filter optimization can be applied to other PSI protocols, including DH-based PSI. A classic DH-based PSI works as follows. The client determines a subkey, who later encrypts his set elements and sends the intermediates to the server. The server determines the other subkey, who encrypts both the server and client's intermediates and sends back the server's intermediates as well as the client's final encrypted results concatenated with the client's intermediates. The client encrypts the server's intermediates and compares two sets of final results to obtain the intersection. DH-based PSI has some similarities compared with OT-based PSI, as they both map the private sets to a pseudo-random message space to obtain the intersection. Unfortunately, our filter optimization does not work for DH-based PSI. The reason is that DH-based PSI maps the elements twice, while OT-based PSI maps only once. Thus the client cannot match the elements with corresponding final results when there are more than one mappings.

5.2 Parallelization

Parallelization is an efficient approach for reducing latency, which has been adopted by many existing SMPC protocols. We also apply parallelization in our framework.

Most original implementations of PSI protocols use C++, and the developers are required to manually set threads for parallelization, which is inconvenient and may not obtain the best performance in different scenarios. Our framework is implemented in Java, which provides the parallel stream class, and can automatically execute in parallel according to the computational power. A parallel parameter is set to determine whether a protocol is implemented in parallel or not. Note that parallelization does not always get better performance, as it needs additional initialization for distributing threads. Three conditions are considered to determine whether a piece of code needs to be parallel. The codes contain cryptographic computations including hashing, symmetric or asymmetric cryptography, and thus are required to be executed for $O(n)$ or more times. Another requirement is that there is no data dependency among different loops. We manually pick up these codes and apply parallelization.

5.3 Big-endian Format and Common Conversions

We fix the byte arrays in our framework in a big-endian format. Our main consideration is that BigIntegers in Java are represented in big-endian, and most of the cryptographic operations are applied to BigIntegers. It is faster to convert BigIntegers to big-endian byte arrays rather than small-endian. Moreover, most byte arrays in our framework are required to represent in fixed lengths by padding enough "0"s. We prefer the "0"s to appear at the beginning of the arrays, and thus the paddings are less probable for misreading. These conversion interfaces are packed in the CommonConversion module in our framework. We additionally pack other conversion and padding interfaces among common data types besides byte arrays, including BigIntegers, bools, and boolean arrays, etc.

5.4 Custom-tailored Optimizations and Corrections

There are some custom-tailored optimizations and corrections for some of these OT-based PSI protocols, which we found during implementation.

In Step 5 of the malicious-secure PSI protocol in [46], a random permutation π is generated to map each OT instance to the Bloom filter according to its choice bits. However, the original protocol description is not precise enough. The protocol only guarantees there are sufficient OT instances to choose bits equaling 1, and the instances choosing 0 cannot satisfy the demand of the Bloom filter. A more precise description of π is to first map the "0" OT instances to "0" positions in the Bloom filter, and then pad "1" OT instances to remaining positions.

PSI protocols in [40] use Lagrange Polynomials for interpolation. We replace Lagrange Polynomial with Newton Interpolation and find that the latter performs better. Moreover, there is a minor problem in the polynomial evaluation algorithm of [40]. The authors recommended that the subset size n of evaluated points and the degree d of the evaluated polynomial satisfy $n, d \in (2^{l-1}, 2^l]$, where l indicates an integer. However, the evaluation fails when $n = d = 2^l$, because the polynomial of root node may have a degree bigger than d and thus we modify it to $n, d \in [2^{l-1}, 2^l)$. Pinkas et al. [41] proposed two methods to construct a garbled cuckoo table, based on 2-core and DFS respectively. We expected the DFS approach to perform better before evaluation, but our experimental results show that they may have similar computational performance. We think it is because we adopted Java instead of C/C++.

Many protocols in our framework use elliptic curves, including some base OT [37,32,8] and PSI protocols [18,16,9,10]. We adopt secp256k1, which has a co-factor equaling 1, in terms of reducing costs of point multiplications.

6 Performance Evaluation

6.1 Experiment Design

Our goal is to find the combinations from the ground up for the best performance. A greedy strategy is rationally based on the architecture of our framework. The modules in our framework can be roughly divided into three levels based on their dependencies. Cryptographic primitives such as hashes, ciphers, PRF, and PRG are generated independently. The primitives are combined to construct base OTs and OTEs, and the OT-based PSI protocols consist of all the above modules. Thus we test the modules in our framework from the ground up and look for the optimal combinations achieving the best performances.

For primitives including hashes, ciphers, PRFs, and PRGs, we have tested their running times when required to the output value of specific lengths from 2^0 to 2^{20}, and thus we get an optimal strategy for choosing these primitives when outputs of different bit lengths are required. Then we test the running time and communication of building blocks including OT and OTE protocols under different network conditions, and finally provide a complete evaluation of different OT-based PSI protocols.

6.2 Benchmark Environment

Our benchmarks are implemented on a physical machine with 8-core Intel(R) Core(TM) i9-9900K 3.60GHz CPU and 32GB RAM. We simulate four network connection cases, including a LAN and three WAN networks. In the LAN setting, the bandwidth is set to be 4 Gbps, and the average RTT is set to be 0.1 ms. In the WAN settings, we fix RTT as 80 ms and test on various network bandwidths. WAN 1, WAN 2, WAN 3 networks have bandwidths equaling 100 Mbps, 10 Mbps, 1 Mbps, respectively. Our experiments use multiple threads for the sake of parallelization. Detailed benchmarks for primitives and protocols are presented in Appendix B.

6.3 Evaluation

We aim to look for optimal primitives based on the results in Table 2, 3, 4, and 5. The inputs of primitives are fixed to 128-bit long. The best-performed cases are annotated in the tables. In the hash tests, we have found that the 160-bit hash function BLAKE2s in Bouncy Castle performs better when hashing into short messages. SHA-512 of SHA-2 and SHA-3 performs better when hashing into long messages. In the cipher test, the CTR-mode AES-Light in Bouncy Castle encrypts fastest when the expected output length is within 2^{10}, and the CTR-mode AES implemented in JDK library performs faster as the output length grows. Different PRG implementations based on CTR-mode AES, SM4, and stream cipher ISSAC, ZUC-256 all perform well when generating a bit string shorter than 2^{14}, but the PRG based on CTR-mode AES in JDK library performs significantly better when longer output bit strings are expected. As for PRF, the implementation based on the Highway hash function performs better when the expected output length is shorter than 2^8, and the one based on AES in Bouncy Castle performs better in a range from 2^{12} to 2^{16}. The one based on the SipHash hash function performs better when expecting even longer outputs.

We have applied the strategies of choosing primitives in our base OT implementations, and the results are presented in Table 6, which demonstrates that there are two base OT protocols performing better considering both communication and latency, CO15 [8] and NP01 [37] based on ECC. They have similar performances and can be regarded as a minor trade-off of communication and latency.

CO15 has slightly less communication, and NP01 has slightly less latency. We have further adopted CO15 as base OT to implement OTE protocols. The results are presented in Table 7 and 8. Different OTE protocols have different security assumptions, and thus they need to be discussed separately based on the upper-layer PSI protocols. The results of 1-out-of-2 OTE in Table 7 demonstrate that ALSZ13 [1] achieves a significant performance improvement compared with IKNP03 [20], and thus we recommend to use ALSZ13 for 1-out-of-2 OTE under semi-honest assumptions. Also, KOS15 [23] costs more in both communication and latency, but it achieves malicious security. Note that PSI protocols using 1-out-of-n OTE are mostly assigned to adopt a specified protocol, such as [28,47,41], which is different from 1-out-of-2 OTE. Only PSZ14 [44] protocol does not assign the OTE protocol and only requires 1-out-of-256 OTE instances. Thus we evaluate our 1-out-of-n OTE implementations when n is fixed to be 256. The results are presented in Table 8, which demonstrates that KK13 [27] runs fastest, and OOS17 requires less communication as well as achieving malicious security at the same time.

We provide a detailed evaluation of PSI protocols in our framework based on the optimal choosing strategies as above, and the results are proposed in Table 9. The results demonstrate that CM20 [7] performs best considering both communication and latency among semi-honest secure PSI protocols. KKRT16 [28] and PRTY19-Fast [40] also achieve good performance. PRTY20 [41] performs the best among malicious-secure PSI protocols, especially when the scale of input sets grows larger. Note that when the scale of PSI is too small, e.g. 2^8, public-key cryptographic PSI [18] is still the best choice. The reason is that the number of required base OT instances is similar to the private set size, and thus OT-based PSI cannot effectively optimize the performance with the advantages of OTE.

7 Which Protocol to Choose

A guideline for constructing efficient PSI protocols is presented in this section. To construct an optimal PSI in a semi-honest case, we recommend ALSZ13 [1] to be as 1-out-of-2 OTE and adopt CO15 [8] or ECC-based NP01 [37] as its dependent base OT instances according to the scenarios. In malicious settings, we recommend KOS15 [23] to be a 1-out-of-2 OTE with the same dependencies as the semi-honest case. 1-out-of-n OTE is always determined by specified protocols, and thus there is no need to provide recommendation strategies. As for bottom-layer primitives, we recommend BLAKE2s-160 and CTR-mode AESLight in Bouncy Castle to be as hashes and ciphers when expecting a short output. The PRG instances are constructed using CTR-mode SM4 in Bouncy Castle, and the PRF instances are constructed using the Highway hash function. When the primitives are expected to output long bit strings, for example, the PRG used by OTE for matrix extension, we recommend SHA-512 and CTR-mode AES in the JDK library to be as optimal hashes and ciphers. Then the PRG instances are constructed using this CTR-mode AES, and the PRF instances are constructed using SipHash or AES in the Bouncy Castle library. For practical PSI instances, we recommend CM20 [47] and PRTY20 [41] for semi-honest and malicious cases respectively, considering both communication and latency.

8 Related Works

PSI is an open MPC problem and has lots of other custom-tailored protocols besides the OT-based ones, and these PSI protocols of other species are considered as our future work of extending our work into a generic PSI framework. We provide a brief description of these protocols.

Another big branch of PSI is circuit-based PSI. Garbled circuit (GC) is a natural choice to compute functions on the intersection like other generic MPC problems. Any problems that can be converted into a circuit can use GC for secure computation, and circuit-based PSI can be easily extended to a private set analysis by connecting with certain analysis circuits, which is one of the biggest advantages of circuit-based PSI protocols. Huang et al. [17] proposed a direct solution of circuit-based PSI. The parties construct a sort-compare-shuffle circuit and then input their elements to the circuit. which later sorts and compares to obtain the intersection, and the sorting information is hidden by shuffling. The SCS circuit can directly concatenate with subsequent computing circuits to evaluate specific functions. However, handling PSI entirely based on the circuit is inefficient, because of the complexity. In [45], Pinkas et al. attempted to adopt cuckoo hashing to aid circuit-based PSI. Similar with the hashing strategies in KKRT16 [28], the elements are inserted into simple and cuckoo hash tables respectively, and the element in cuckoo hashing entry only needs to compare the element inserted in two candidate entries of the simple hashing, and thus the comparisons of the circuit can be greatly reduced. OPRF can also be constructed based on circuits. A garbled PRF circuit is an natural instance of OPRF when the garbler inputs key k, and the evaluator inputs x. Note that circuit-based PSI in [26,21] adopts this OPRF construction. In detail, [26] adopts garbled AES circuit, and [21] adopts garbled LowMC circuit.

Pinkas et al. [43] proposed another circuit-based PSI with linear communication. Their optimization mainly lies in their novel OPRF construction. Note that it [45], every entry of cuckoo hash table is required to compare with all elements lying in its entry in the simple hash table. In [43], the comparison is reduced to once for each hash entry. They presented a new primitive named Oblivious Programmable Pseudo Random Function (OPPRF). OPPRF is kind like a correlated OPRF. If one party inputs specific points $\{x\}_n$, the PRF results is set to specific values $\{t\}_n$ respectively, determined by the other party. While the outputs on the other input points acts like a random string. OPPRF in [43] is constructed with BaRK-OPRF [28] and polynomials. The brief description lies as follows. Alice holds a cuckoo hash table, and Bob holds a simple hash table. They apply an OPRF instance for each entry. Alice inputs the element in the entry x_A and gets $PRF_k(x_A)$, while Bob gets the key k. Then Bob applies a polynomial interpolation to the point set $\{(x_{B,i}, t_i \oplus PRF(x_{B,i}))\}$, where $i \in [|\{x_B\}|]$. The polynomial p is sent to Alice, who computes $PRF_k(x_A) \oplus p(x_A)$ locally as the OPPRF result y_A. It is interesting that if all t_i equal to the same t, obviously we have $y_A = t$ when $x_A \in \{x_B\}$. If $x_A \notin \{x_B\}$, y_A seems to be a random string. Thus with the help of OPPRF, Alice and Bob determine whether $x_A \in \{x_B\}$ by only comparing y_A and t, and the subsequent comparison circuit can be reduced to only one. Note that although an independent OPRF instance is required for each entry, multiple entries can share a common polynomial by interpolating the point sets of different entries into the single polynomial. The entries sharing a common polynomial consist of a mega-bin. They set the optimal number of mega-bin as $(N/logN)$ to get optimal communication cost, where N is the number of the parties' elements. Note that their PSI still achieves a superlinear communication when handling stash, because the elements in Alice's cuckoo stash still need to compare with all elements of Bob. The authors proposed two methods to efficiently handle stash [43]. One is dual execution, which requires Bob to reinsert his elements in cuckoo hashing, and Alice inserts her stash elements in simple hashing. They then apply a reverse OPPRF and input PRF results into comparison circuits, and finally compare the elements in Bob's stash with Alice's stash one by one. The other is the adoption of stashless cuckoo hashing, by adjusting the parameters to guarantee that the stash is not empty in a negligible probability, like [11]. Both two methods achieve linear communication, but stashless cuckoo hashing performs better according to their proposed evaluation results. Another advantage of their PSI is that it is extendable to compute on the payloads

of the intersection, by applying another OPPRF instance for each entry, whose results will be input to the circuit together with y_A and t.

PSI in [21] also adopts circuits, and it aims at an unbalanced case of PSI. It improves two protocols in [26]. The idea of [21] follow the naive approach. In their circuit-based PSI, the server builds and garbles a PRF circuit, and sends it to the client, who obtains its input labels with OT, and locally evaluates the circuit to output PRF results. Then the server computes plaintext PRF with its set and inserts PRF results into a cuckoo filter. The client checks the existence of each set element in the cuckoo filter and obtains the intersection. Their NR-PSI is similar to GC-PSI, except that the PRF circuit is replaced by NR-PRF. Their protocols' efficiency comes from their pre-computation because the computationally expensive tasks are performed in the pre-computation phase. OT operations are placed into the precomputing phase, where the server and client apply OT extension [20] on pairs of correlated random masks, and the masks will be used to compute the circuit wire labels. Thus the online phase-only consists of efficient symmetric primitives. The cuckoo filter is managed by the server and can be inserted or deleted conveniently. Another optimization itekales2019mobile is the adoption of the LowMC circuit. The cost of the online phase lies in label transferring and circuit evaluation, and a simpler circuit achieves better performance.

Demmler et al. [11] also proposed an interesting PSI protocol aimed at unbalanced cases. Their protocol can be regarded as a scale reducing the extension of PSI. It first reduces the scale of the original unbalanced PSI into a two-party PSI with a smaller scale, and then it can be handled with efficient protocols such as [28].

Another interesting direction of PSI is to utilize a third party. A naive approach of a third party-aided PSI is that Alice and Bob locally apply PRF to their input sets, and send the outputs to the third party. The third-party compares these PRF outputs that seem random to himself because he learns nothing about the PRF. The comparison results will be returned to Alice and Bob, who later obtains the intersection from the intersection's PRF results. However, this naive scheme requires the third party to be semi-honest. A malicious third party may manipulate the intersection by modifying the PRF results' intersection, even though he gets no access to the original set elements of the participants. Le et al. [30] proposed a PSI protocol aided by an untrusted third party. Their protocol applies an additional cardinality verification to the naive solution. The untrusted third-party is required to prove that the intersection cardinality equals to a certain τ he commits, to convince the participants that he applies no manipulation on the expected intersection. The cardinality verification is based on polynomial interpolation. Note that at least n points can interpolate a certain $(n-1)$-degree polynomial. In [30], Alice and Bob first apply PRF_{k_1} to their private sets X and Y. The third party, named Carlos, compares $\hat{X} = PRF_{k_1}(X)$ and $\hat{Y} = PRF_{k_1}(Y)$ to obtain the PRF results' intersection, and returns the cardinality τ to Alice and Bob afterwards. Alice and Bob generate two random polynomials, where p_1 has a degree of $(\tau - 1)$, and p_2 has a degree of $(|X| + |Y| - \tau - 1)$. They evaluate p_1 and p_2 on $\hat{X}$ and $\hat{Y}$, and choose another PRF PRF_{k_2} to map $\hat{X}$ and $\hat{Y}$. Then Alice computes $V_1 \leftarrow \{(\hat{x}, PRF_{k_2}(\hat{x}))|\hat{x} \in \hat{X}\}$, and $W_1 \leftarrow \{(\hat{x}, p_2(\hat{x}))|\hat{x} \in \hat{X}\}$. Bob computes $V_2 \leftarrow \{(\hat{(}y), p_1(\hat{y}) \oplus PRF_{k_2}(\hat{y}))|\hat{y} \in \hat{Y}\}$, and $W_2 \leftarrow \{(\hat{y}, p_2(\hat{y}))|\hat{y} \in \hat{Y}\}$. (V_1, V_2, W_1, W_2) are sent to Carlos, who is required to interpolate p_1 and p_2 using (V_1, V_2, W_1, W_2). If Carlos acts honestly, he can interpolate p_1 with point set $\{(a, b \oplus c)|(a, b) \in V_1 \& (a, c) \in V_2\}$, and interpolate p_2 with point set $W_1 \bigcup W_2$. If Carlos lied about the cardinality before, he cannot interpolate both p_1 and p_2 correctly. The polynomial-based cardinality verification requires low communication but more latency. Thus another circuit-based PSI was presented in [30]. PRF result sets $\hat{X}$ and $\hat{Y}$ are distributed as secret shares among three participants. Note that [30] adopts replicated sharing in [35]. After Carlos compares $\hat{X}$ and $\hat{Y}$ to obtain the intersection, he is required to generate a permutation

π which switches elements belonging to the intersection to front and switches the sorted leftovers to end. Then the participants shuffle the shares of $\hat{X}||\hat{Y}$ using π. Then a 3PC circuit is applied to compare the first 2τ elements of shuffled $\hat{X}||\hat{Y}$, to check whether there are τ equaling pairs. For the leftovers behind, another 3PC circuit is applied to check whether they are arranged in sorted order. The protocol adopts if any verification fails. Another advantage of their PSI protocol is its extendibility to secure computation of the intersection payloads.

9 Conclusion

In this paper, we make integrated research of up-to-date OT-based PSI protocols and have introduced a Java-based PSI framework, in which we implement these PSI protocols together with their dependent building blocks and primitives. Our framework is modular and extendable and thus enables us to analyze, compare, and optimize these protocols. We present several optimizations in our framework, including filters and paralization, which significantly reduce the latency and communication costs. We present a detailed performance analysis of these protocols and provide a guideline as to which primitives and building blocks to choose to construct an optimal OT-based PSI protocol. We believe that our framework benefits a lot for future PSI researches.

References

1. Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 535–548, 2013.
2. Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. pages 673–701, 2015.
3. Donald Beaver. Precomputing oblivious transfer. CRYPTO '95, page 97–109, Berlin, Heidelberg, 1995. Springer-Verlag.
4. M Bellare. Non-interactive oblivious transfer and applications. In *International Cryptology Conference on Advances in Cryptology*, 1989.
5. Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of The ACM*, 13(7):422–426, 1970.
6. Alex Breslow and Nuwan Jayasena. Morton filters: fast, compressed sparse cuckoo filters. *The VLDB Journal*, 08 2019.
7. Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious prf. Cryptology ePrint Archive, Report 2020/729, 2020. `https://eprint.iacr.org/2020/729`.
8. Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In *International Conference on Cryptology and Information Security in Latin America*, 2015.
9. Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography & Data Security, International Conference, Fc, Tenerife, Canary Islands, January, Revised Selected Papers*, 2010.
10. Emiliano De Cristofaro and Gene Tsudik. Experimenting with fast private set intersection. 2012.
11. Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. Pir-psi: Scaling private contact discovery. 2018(4):159–178, 2018.
12. Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.
13. Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. pages 789–800, 2013.

14. Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo filter: Practically better than bloom. pages 75–88, 2014.
15. M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. Sok: General purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1220–1237, 2019.
16. Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *Theory of Cryptography*, pages 155–175, 2008.
17. Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? 2012.
18. Bernardo A. Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, page 78–86, New York, NY, USA, 1999. Association for Computing Machinery.
19. Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. *IACR Cryptol. ePrint Arch.*, 2017:738, 2017.
20. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. pages 145–161, 2003.
21. Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. pages 1447–1464, 2019.
22. Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. Cryptology ePrint Archive, Report 2020/521, 2020. `https://eprint.iacr.org/2020/521`.
23. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure ot extension with optimal overhead. pages 724–741, 2015.
24. Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 549–560. ACM, 2013.
25. Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009.
26. Agnes Kiss, Jian Liu, Thomas Schneider, N Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. 2017(4):177–197, 2017.
27. Vladimir Kolesnikov and Ranjit Kumaresan. Improved ot extension for transferring short secrets. 8043:54–70, 2013.
28. Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious prf with applications to private set intersection. pages 818–829, 2016.
29. Mikkel Lambaek. Breaking and fixing private set intersection protocols. Cryptology ePrint Archive, Report 2016/665, 2016. `https://eprint.iacr.org/2016/665`.
30. Phi Hung Le, Samuel Ranellucci, and S Dov Gordon. Two-party private set intersection with an untrusted third party. pages 2403–2420, 2019.
31. C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376, 2015.
32. Daniel Mansy and Peter Rindal. Endemic oblivious transfer. In *the 2019 ACM SIGSAC Conference*, 2019.
33. Peihan Miao, Sarvar Patel, Mariana Raykova, Karn Seth, and Moti Yung. Two-sided malicious security for private intersection-sum with cardinality. Cryptology ePrint Archive, Report 2020/385, 2020. `https://eprint.iacr.org/2020/385`.
34. R Moenck and A Borodin. Fast modular transforms via division. pages 90–96, 1972.
35. Payman Mohassel and Peter Rindal. Aby 3: a mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 35–52. ACM, 2018.

36. Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. Cryptology ePrint Archive, Report 2018/403, 2018. https://eprint.iacr.org/2018/403.
37. Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, page 448–457, USA, 2001. Society for Industrial and Applied Mathematics.
38. Michele Orru, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of- n ot extension with application to private set intersection. pages 381–396, 2017.
39. Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. pages 121–133, 2001.
40. Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse ot extension. pages 401–431, 2019.
41. Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Psi from paxos: Fast, malicious private set intersection. Cryptology ePrint Archive, Report 2020/193, 2020. https://eprint.iacr.org/2020/193.
42. Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. *IACR Cryptology ePrint Archive*, 2015:634, 2015.
43. Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based psi with linear communication. pages 122–153, 2019.
44. Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on ot extension. pages 797–812, 2014.
45. Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *IACR Cryptology ePrint Archive*, 2016:930, 2016.
46. Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. Cryptology ePrint Archive, Report 2016/746, 2016. https://eprint.iacr.org/2016/746.
47. Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. Cryptology ePrint Archive, Report 2017/769, 2017. https://eprint.iacr.org/2017/769.
48. Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters. *Proc. VLDB Endow.*, 13(2):197–210, October 2019.
49. Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. https://eprint.iacr.org/2015/1153.

A Security Proofs for Filters

As mentioned in the filter optimization section, most OT-based PSI protocols require the sender to send the PRF result set to the receiver for comparison. We replace this step with filter insertions and checks, and this optimization achieves a great communication reduction. However, the security of using filters is also worth consideration. In this section, we provide a formal security analysis to prove that the protocols optimized by filters still maintain identical security.

We define an ideal filter functionality $\mathcal{F}_{filter}$ with two oracles. One is the inserting oracle $\mathcal{O}_{ins}$, and the other is the checking oracle $\mathcal{O}_{chk}$. The functionalities of $\mathcal{O}_{ins}$ and $\mathcal{O}_{chk}$ lie as follows:

$\mathcal{O}_{ins} : (S, (M, \{H\})) \rightarrow V$

$\mathcal{O}_{chk} : (x, (M, \{H\}), V) \rightarrow b$

$\mathcal{O}_{ins}$ performs like a PRF. It maps an input set S ($S \in \mathbb{Z}$) to a pseudo-random vector V with a public mapping strategy M and a set of random oracles $\{H\}$, where $V = \{v_i\}$. $\{v_i\}$ is pseudo-random bits in Bloom filters and is pseudo-random strings in cuckoo filters. $\mathcal{O}_{chk}$ inputs $(x, (M, \{H\}), V)$ where $x \in \mathbb{Z}$ is a element in the field of S. It returns a bit of $true$ if $in S$. If $x \notin S$, it returns $true$ with a negligible probability of ϵ, or returns $false$ with the probability of $(1 - \epsilon)$. ϵ is the false positive rate of the filter which is determined by $(M, \{H\})$.

Simulator overview The simulator utilizes the oracles of original non-filter protocols to simulate other messages during execution. In the sender's view, the simulator invokes $\mathcal{O}_{chk}$ to recover the

corrupted sender's elements based on the filter vector V, and the recovered elements are injected into the ideal PSI function to obtain the intersection. In the receiver's view, the simulator first simulates a list of OPRF results using the oracles from the original non-filter protocols, and then invokes $\mathcal{O}_{ins}$ to insert the results into a filter, which will be later sent to the corrupted receiver.

Theorem 1. *Assume that $\mathcal{P}$ is an OT-based PSI protocol with the final step of comparing OPRF results in plaintext, and $\mathcal{P}'$ is our new protocol using filters. If $\mathcal{P}$ is secure against a semi-honest/malicious adversary, and $N\epsilon < 2^{-\lambda}$, then $\mathcal{P}'$ is also secure against a semi-honest/malicious adversary. λ indicates the statistic security parameter of $\mathcal{P}$, and N indicates the upper bound of the insertion number of the filter.*

Proof. As $\mathcal{P}$ is proven to be secure against the adversary, there exists a set of oracles $\{\mathcal{O}_P\}$ to simulate all possible messages of $\mathcal{P}$. If $\{\mathcal{O}_P\}$ is secure, then the simulators for the sender and receiver of $\mathcal{P}$ must exist, and we can use them as subroutines when constructing simulators for $\mathcal{P}'$.

Sender's view We start from a corrupted sender. The simulator for a corrupted sender observes the sender's inputs and outputs, and also observes all of the sender's queries to random oracles in $\{H\}$ as well as other observations of the non-filter simulator $Sim_S^{\mathcal{P}}$. The simulator collects an OPRF result set $\tilde{Y} = \{y | y \text{ was queried to } \{H\}\}$. The front phases of $\mathcal{P}'$ are identical to those of $\mathcal{P}$, thus $Sim_S^{\mathcal{P}'}$ is also identical to $Sim_S^{\mathcal{P}}$ in these phases. The difference lies in the final phase of $\mathcal{P}$ and $\mathcal{P}'$. After $Sim_S^{\mathcal{P}'}$ receives V' from the corrupted sender, it applies $\mathcal{O}_{chk}$ on every OPRF result in $\tilde{Y}$, and obtains $Y' = \{y \in \tilde{Y} | \mathcal{O}_{chk}(y, (M, \{H\}), V') = true\}$. The simulator then sends Y' to $Sim_S^{\mathcal{P}}$ to finish subsequent simulations.

We then prove the indistinguishability of this simulator. The only difference in the front phases is to record the queries and has nothing to do with the outputs and the intermediate messages. Thus the messages in the front phases are directly simulated by $Sim_S^{\mathcal{P}}$, which has been proven to be indistinguishable. When $\mathcal{O}_{chk}$ is involved, the simulator $Sim_S^{\mathcal{P}'}$ aborts in two cases. One is that the honest receiver holds $x \in X_R \backslash X_S$, but $F_{OPRF}(x) \in F_{OPRF}(X_S)$, and however, this case can be handled by $Sim_S^{\mathcal{P}}$, which is not taken into our consideration. The other case is that the honest receiver holds $x \in X_R$, whose OPRF result satisfies $F_{OPRF}(x) \notin \tilde{Y}$, but $\mathcal{O}_{chk}(F_{OPRF}(x), (M, \{H\}), V') = true$. In that case, x appears in the intersection output of $\mathcal{P}'$ by mistake. This case happens with the probability of ϵ for every individual x based on the above definitions, and thus the existence probability of this kind of x is less than $n\epsilon$, where n indicates the honest receiver's set size. If the probability is bounded by $n\epsilon < 2^{-\lambda}$, it is indistinguishable from the ideal functionality.

Receiver's view We also construct another simulator for a corrupted receiver $Sim_R^{\mathcal{P}'}$. $Sim_R^{\mathcal{P}'}$ observes the receiver's inputs and outputs, and also observes all of the receiver's queries to random oracles in $\{H\}$, as well as other observations of $Sim_R^{\mathcal{P}}$. The front phases of $\mathcal{P}'$ are identical to those of $\mathcal{P}$, thus $Sim_R^{\mathcal{P}'}$ is identical to $Sim_R^{\mathcal{P}}$ in these phases. Then we focus on the final phase. The task for $Sim_R^{\mathcal{P}'}$ is to simulate an indistinguishable filter V' without obtaining the input of the honest sender, and we define $Sim_R^{\mathcal{P}'}$ to behave as follows. $Sim_R^{\mathcal{P}'}$ invokes $Sim_R^{\mathcal{P}'}$ to generate a simulated OPRF results' set $\tilde{Y}$, and inputs $(\tilde{Y}, (M, \{H\}))$ to $\mathcal{O}_{ins}$ to construct the filter vector V', which will be later sent to the corrupted receiver.

We then prove the indistinguishability of this simulator. The sender's message in the real world is $V = \mathcal{O}_{ins}(H_{OPRF}(X_S), (M, \{H\}))$. $Sim_R^{\mathcal{P}'}$ aborts in two cases. One case is that the corrupted receiver holds an element $x \in \tilde{X} \backslash X_S$, but $F_{OPRF}(x) \in F_{OPRF}(X_S)$, and this case can be handled by $Sim_R^{\mathcal{P}}$. The other case is that the corrupted receiver holds $x \in \tilde{X}$, where $F_{OPRF}(x) \notin F_{OPRF}(X_S)$, but $\mathcal{O}_{chk}(H_{OPRF}(x), (M, \{H\}), V') = true$. Then x appears in the output intersection by mistake.

This case happens with the probability of ϵ for every individual $x \in \tilde{X}$ based on the oracle definition, and thus the probability is less than $N\epsilon$, where N' is the upper bound of the size of $\tilde{X}$, which indicates the maximum of the filter the corrupted receiver is allowed to check. Obviously, the check times are less than the upper bound of the filter's insertion number N. Thus if the probability is bounded by $N\epsilon < 2^{-\lambda}$, it is indistinguishable from the ideal functionality.

Overall, as an OT-based PSI protocol, $\mathcal{P}$ always has $n < N$, and thus the condition of the two views can be rewritten as, $n\epsilon < N\epsilon < 2^{-\lambda}$. Then we conclude that the simulators of $\mathcal{P}$ is indistinguishable under the condition of $N\epsilon < 2^{-\lambda}$.

B Experimental Results

Table 2: Running time for different hash implementations to output an l-bit value. (μs) The input length is fixed to be 128-bit.

$log_2 l$	0	2	4	6	8	10	12	14	16	18	20
JDK-SHA-256	4.02	1.47	1.33	1.38	1.52	3.55	12.1	33.6	99.3	369	1407
BC-BLAKE2b-160	1.44	1.12	1.05	1.07	4.37	6.41	18.5	54.2	197	765	3079
BC-BLAKE2b-256	0.591	0.639	0.542	0.546	0.596	2.04	7.84	31.2	127	504	2015
BC-BLAKE2b-384	0.592	0.618	0.531	0.535	0.643	1.63	5.81	22.6	90.6	361	1449
BC-BLAKE2b-512	0.680	0.689	0.632	0.641	0.701	1.28	4.74	18.2	74.6	294	1173
BC-BLAKE2s-128	1.07	1.04	0.889	1.01	1.59	5.75	19.0	50.9	198	787	3139
BC-BLAKE2s-160	**0.503**	**0.479**	**0.436**	**0.444**	0.847	2.81	10.3	40.4	163	648	2585
BC-BLAKE2s-224	0.511	0.491	0.534	0.460	0.871	2.13	7.84	30.4	123	484	1937
BC-BLAKE2s-256	0.831	0.735	0.716	0.753	0.638	2.16	7.56	29.9	113	443	1766
BC-SHA2-SHA256	2.01	1.66	1.58	1.59	1.70	5.15	9.00	32.1	102	378	1510
BC-SHA2-SHA384	1.94	1.77	1.52	0.746	0.722	1.79	6.64	22.3	89.7	350	1407
BC-SHA2-SHA512	5.69	0.608	0.555	0.672	0.593	1.21	4.72	**17.9**	**71.2**	**279**	**1116**
BC-SHA2-SHA512/224	0.688	0.601	0.589	0.626	1.08	2.69	10.0	40.3	153	606	2424
BC-SHA2-SHA512/256	0.614	0.594	0.664	0.558	**0.558**	2.30	8.43	33.6	136	540	2173
BC-SHA3-224	1.50	0.902	0.763	0.832	1.36	3.01	10.3	39.2	158	623	2479
BC-SHA3-256	0.616	0.594	0.568	0.559	0.603	2.19	8.51	34.0	137	546	2188
BC-SHA3-384	0.606	0.619	0.557	0.565	0.637	1.69	6.03	23.5	93.2	380	1513
BC-SHA3-512	0.640	0.591	0.548	0.555	0.587	**1.19**	**4.56**	18.4	73.4	291	1161

Table 3: Running time for different cipher implementations to output an l-bit value. (μs) The input length is fixed to be 128-bit.

$log_2 l$	0	2	4	6	8	10	12	14	16	18	20
JDK-CTR-AES	3.40	2.92	2.82	2.96	3.05	3.53	**4.88**	11.9	**32.7**	**54.0**	**223**
BC-Stream-ISAAC	6.67	5.79	6.08	6.69	10.1	7.88	6.86	**10.0**	40.9	163	653
BC-CTR-AESLight	**1.52**	**1.20**	**1.01**	**1.16**	**1.43**	**2.77**	6.23	23.0	84.1	324	1291

Table 4: Running time for different PRG implementations to output an l-bit value. (μs) The input length is fixed to be 128-bit.

$log_2 l$	0	2	4	6	8	10	12	14	16	18	20
JDK-CTR-AES	3.35	2.93	2.76	2.79	2.73	3.09	4.76	10.9	**18.6**	**61.3**	**401**
BC-CTR-AESLight	1.67	1.30	0.968	1.02	1.46	2.56	5.83	21.0	82.5	316	1225
BC-CTR-AES	1.46	1.17	0.876	**0.974**	1.22	2.95	5.47	16.7	66.4	273	1100
BC-CTR-SM4	**1.43**	**1.04**	**0.812**	1.14	**1.19**	2.48	6.93	29.8	115	416	1668
BC-Stream-CHACHA	1.55	1.84	1.92	1.71	1.37	3.41	5.36	11.4	40.2	156	633
BC-Stream-ISAAC	8.74	6.41	6.52	6.71	4.10	2.55	**3.65**	**8.87**	31.4	122	513
BC-Stream-ZUC128	1.91	1.73	1.77	2.66	2.54	2.40	5.13	16.6	64.0	272	940
BC-Stream-ZUC256	3.01	2.78	2.71	1.71	1.41	**1.92**	4.50	16.1	60.4	236	919
JDK-Digest-SHA1	1.99	1.50	1.89	1.29	1.94	3.10	7.59	26.9	101	392	1573
BC-Digest-BLAKE2b160	3.40	2.59	2.49	2.76	2.81	4.83	11.5	35.2	135	521	2086
BC-Digest-SHA2SHA256	6.74	2.39	1.35	1.59	2.24	5.74	15.3	45.4	181	712	2873

Table 5: Running time for different PRF implementations to output an l-bit value. (μs) The input length is fixed to be 128-bit.

$log_2 l$	0	2	4	6	8	10	12	14	16	18	20
SipHashPRF	1.27	0.864	0.765	0.772	1.67	3.66	13.0	31.6	76.2	**218**	**926**
HighwayHashPRF	1.57	**0.517**	**0.399**	**0.493**	1.92	3.88	13.1	46.6	190	832	2915
NaiveHashPRF-JDKSHA1	**1.19**	0.566	0.588	0.577	**1.15**	3.13	11.2	37.6	132	507	2036
NaiveHashPRF-JDKSHA256	4.06	2.75	2.67	1.60	1.20	**2.08**	11.1	38.4	141	403	1602
NaiveHashPRF-BCSM3	1.92	1.69	3.30	3.14	3.17	7.47	13.7	36.7	138	403	1481
BCBlockCipherPRF-AESLight	5.12	3.92	3.31	1.89	2.82	2.90	6.53	21.7	83.7	304	1205
BCBlockCipherPRF-AES	5.68	1.86	1.74	1.64	1.80	2.50	**5.23**	**16.6**	**62.1**	279	1125
BCBlockCipherPRF-SM4	2.32	1.79	1.51	1.80	2.16	3.79	10.6	39.0	110	421	1690

Table 6: Running time (ms) and communication (KB) of the sender (Snd) and the receiver (Recv) for base OT implementations. LAN setting has a 4 kGbps bandwidth and a 0.1 ms RTT. WAN1 setting has a 100 Mbps bandwidth and a 80 ms RTT. WAN2 setting has a 10 Mbps bandwidth and a 80 ms RTT. WAN3 setting has a 1 Mbps bandwidth and a 80 ms RTT.

n	Protocol	Comm.			Running time (LAN)			Running time (WAN1)			Running time (WAN2)			Running time (WAN3)		
		Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total
2^7	NP01(FFG) [37]	0.25	16	16.25	27	17	44	187	96	283	198	96	294	311	97	408
	NP01(ECC) [37]	0.064	4.13	4.19	10	6	**16**	169	85	254	172	85	257	201	85	286
	CO15 [8]	0.032	4.13	**4.16**	13	7	20	173	86	259	175	87	262	207	87	294
	MR19(ECC) [32]	4.13	8.25	12.38	18	12	30	97	91	**188**	102	95	**197**	160	124	**284**
	MR19(FFG) [32]	16	48	64	38	32	70	118	110	228	147	114	261	485	227	712
2^8	NP01(FFG) [37]	0.25	32	32.25	52	33	85	213	112	325	235	113	348	460	113	573
	NP01(ECC) [37]	0.064	8.25	8.31	20	10	**30**	178	89	267	184	90	274	243	90	**333**
	CO15 [8]	0.032	8.25	**8.28**	22	11	33	184	92	276	191	94	285	248	92	340
	MR19(ECC) [32]	8.25	16	24.25	34	24	58	115	103	**218**	125	110	**235**	241	167	408
	MR19(FFG) [32]	32	96	128	74	64	138	155	139	294	215	149	364	890	375	1265
2^9	NP01(FFG) [37]	0.25	64	64.25	102	66	168	265	145	410	308	144	452	761	146	907
	NP01(ECC) [37]	0.064	16.5	16.56	36	20	**56**	196	100	296	209	101	310	325	101	**426**
	CO15 [8]	0.032	16.5	**16.53**	37	20	57	199	100	299	211	100	311	327	100	427
	MR19(ECC) [32]	16.5	33	49.5	66	45	111	149	125	**274**	168	138	**306**	400	253	653
	MR19(FFG) [32]	64	192	256	148	116	264	229	198	427	350	219	569	1701	670	2371
2^{10}	NP01(FFG) [37]	0.25	128	128.25	201	130	331	366	206	572	457	208	665	1356	207	1563
	NP01(ECC) [37]	0.064	33	33.06	68	39	**107**	232	121	**353**	254	119	**373**	487	120	**607**
	CO15 [8]	0.032	33	**33.03**	72	38	110	236	119	355	257	119	376	492	120	612
	MR19(ECC) [32]	33	66	99	136	94	230	218	173	391	254	194	448	720	425	1145
	MR19(FFG) [32]	128	384	521	294	216	510	363	366	729	337	368	705	322	1261	1583

Table 7: Running time (s) and communication (KB) for 1-out-of-2 OTE implementations. "Snd" and "Recv" denote communications the sender and receiver send respectively, and "SH", "M" denote the security level of "Semi-honest", and "Malicious" respectively. LAN setting has a 4 kGbps bandwidth and a 0.1 ms RTT. WAN1 setting has a 100 Mbps bandwidth and a 80 ms RTT. WAN2 setting has a 10 Mbps bandwidth and a 80 ms RTT. WAN3 setting has a 1 Mbps bandwidth and a 80 ms RTT.

n	Protocol	Security	Comm.			Running time (LAN)			Running time (WAN1)			Running time (WAN2)			Running time (WAN3)		
			Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total
2^8	IKNP03 [20]	SH	4.13	8.03	12.2	0.02	0.02	0.4	0.26	0.17	0.43	0.27	0.18	0.45	0.35	0.21	0.56
	ALSZ13 [1]		4.13	4.03	8.26	0.03	0.02	0.05	0.26	0.18	0.44	0.27	0.18	0.45	0.32	0.21	0.53
	KOS15 [23]	M	4.13	6.7	10.8	0.05	0.04	0.09	0.44	0.35	0.79	0.46	0.36	0.82	0.53	0.44	0.97
2^{12}	IKNP03 [20]	SH	4.13	128	132	0.07	0.04	0.11	0.31	0.20	0.51	0.41	0.20	0.61	1.34	0.23	1.57
	ALSZ13 [1]		4.13	64	68	0.07	0.04	0.11	0.31	0.20	0.51	0.36	0.20	0.56	0.84	0.23	1.07
	KOS15 [23]	M	4.13	66.5	70.5	0.35	0.21	0.56	0.73	0.52	1.25	0.81	0.57	1.38	1.29	1.07	2.36
2^{16}	IKNP03 [20]	SH	4.13	2048	2052	1.11	0.55	1.66	1.34	0.70	2.04	1.32	0.68	2.00	1.34	0.72	2.06
	ALSZ13 [1]		4.13	1024	1028	1.08	0.51	1.59	1.35	0.71	2.06	1.32	0.68	2.00	1.33	0.71	2.04
	KOS15 [23]	M	4.13	1027	1031	5.13	3.00	8.13	5.82	3.53	9.35	5.46	3.45	8.91	5.39	3.41	8.80
2^{20}	IKNP03 [20]	SH	4.13	32768	32772	17.7	8.83	26.5	17.4	8.78	26.2	17.1	8.22	25.3	18.2	9.32	27.5
	ALSZ13 [1]		4.13	16384	16388	18.0	8.64	26.6	17.3	8.28	25.6	18.3	8.71	27.0	18.4	8.4	26.8
	KOS15 [23]	M	4.13	16387	16391	80.7	49.1	129.8	79.3	49.3	128.6	80.5	51.0	131.5	82.6	50.5	133.1

Table 8: Running time (s) and communication (KB) for 1-out-of-n OTE implementations, where $n = 256$. "Snd" and "Recv" denote communications the sender and receiver send respectively, and "SH", "M" denote the security level of "Semi-honest", and "Malicious" respectively. LAN setting has a 4 kGbps bandwidth and a 0.1 ms RTT. WAN1 setting has a 100 Mbps bandwidth and a 80 ms RTT. WAN2 setting has a 10 Mbps bandwidth and a 80 ms RTT. WAN3 setting has a 1 Mbps bandwidth and a 80 ms RTT.

n	Protocol	Security	Comm.			Time (LAN)			Time (WAN1)			Time (WAN2)			Time (WAN3)		
			Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total
2^8	KK13 [27]	SH	8.25	16.05	24.3	0.04	0.04	0.08	0.30	0.21	0.51	0.32	0.22	0.54	0.50	0.28	0.78
	KKRT16 [28]		13.7	26.6	40.3	0.08	0.07	0.15	0.38	0.29	0.67	0.41	0.30	0.71	0.75	0.45	1.20
	OOS17 [38]	M	8.25	10.6	18.9	0.04	0.04	0.08	0.45	0.37	0.82	0.45	0.37	0.82	0.67	0.58	1.25
2^{12}	KK13 [27]	SH	8.25	256	263	0.19	0.13	0.32	0.44	0.28	0.72	0.63	0.30	0.93	2.50	0.37	2.87
	KKRT16 [28]		14.0	432	446	0.32	0.22	0.54	0.64	0.46	1.10	0.66	0.49	1.15	1.19	0.65	1.84
	OOS17 [38]	M	8.25	131	139	0.19	0.18	0.37	0.61	0.52	1.13	0.70	0.62	1.32	2.09	1.99	4.08
2^{16}	KK13 [27]	SH	8.25	4.10×10^3	4.11×10^3	3.06	2.10	5.16	3.21	2.14	5.35	3.28	2.18	4.46	5.63	2.39	8.02
	KKRT16 [28]		14.2	7.05×10^3	7.06×10^3	5.56	3.64	9.20	5.86	3.99	9.85	5.83	3.87	9.70	10.1	4.27	14.4
	OOS17 [38]	M	8.25	2.05×10^3	2.06×10^3	2.97	2.95	5.92	3.61	3.50	7.11	3.38	3.27	6.65	10.3	10.1	20.4
2^{20}	KK13 [27]	SH	8.25	6.55×10^4	6.55×10^4	49.3	33.4	82.7	46.7	31.2	77.9	50.5	33.8	84.3	54.3	35.9	90.2
	KKRT16 [28]		14.5	1.15×10^5	1.15×10^5	93.1	62.5	155.6	93.9	63.0	156.9	91.1	60.4	151.5	98.5	64.8	163.3
	OOS17 [38]	M	8.25	3.28×10^4	3.28×10^4	50.1	49.1	99.2	51.1	50.1	101.2	49.8	49.2	99.0	59.1	58.3	117

Table 9: Running time (s) and communication (KB) for PSI implementations. "Snd" and "Recv" denote communications the sender and receiver send respectively, and "N", "SH", "M" denote the security level of "Not Secure", "Semi-honest", and "Malicious" respectively. "-" denotes cases with too much communication or latency. LAN setting has a 4 kGbps bandwidth and a 0.1 ms RTT. WAN1 setting has a 100 Mbps bandwidth and a 80 ms RTT. WAN2 setting has a 10 Mbps bandwidth and a 80 ms RTT. WAN3 setting has a 1 Mbps bandwidth and a 80 ms RTT.

n	Protocol		Comm.			Time (LAN)			Time (WAN1)			Time (WAN2)			Time (WAN3)		
			Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total	Snd	Recv	Total
2^8	Naive	N	8	0	8	0.001	0.003	0.004	0.002	0.08	0.08	0.001	0.09	0.09	0.00	0.15	0.15
	FFH99(ECC) [18]	SH	10.0	8.25	36.5	0.08	0.08	0.16	0.15	0.24	0.39	0.19	0.27	0.46	0.25	0.34	0.59
	FFH99(FFG) [18]		33.7	32.0	65.7	0.21	0.21	0.42	0.29	0.37	0.66	0.35	0.43	0.78	0.56	0.65	1.21
	CT10 [9]		66	64	130	1.68	1.69	3.37	1.96	2.05	4.01	2.04	2.18	4.22	2.25	2.84	5.09
	HL08(ECC) [16]		2.06×10^3	512	2.57×10^3	0.48	0.58	1.06	0.89	1.14	2.03	0.97	1.22	2.19	1.07	1.33	1.40
	HL08(FFG) [16]		2.10×10^3	512	2.61×10^3	0.80	0.90	1.70	1.21	1.47	2.68	1.29	1.54	2.83	1.59	1.86	3.45
	DCW13 [13]		148	232	380	0.25	0.30	0.55	0.69	0.83	1.52	0.91	1.16	2.07	2.57	3.83	6.40
	PSZ14 [44]		29.3	274	303	1.32	1.32	2.64	1.71	1.86	3.57	1.72	1.89	3.61	1.81	2.13	3.94
	PSZ14(GBF) [44]		5.66	232	238	0.15	0.16	0.31	0.56	0.65	1.43	0.79	0.88	1.67	2.44	2.55	3.99
	KKRT16 [28]		18.7	33.2	51.9	0.20	0.21	0.41	0.69	1.87	2.56	0.77	1.97	2.74	1.05	2.28	3.33
	PRTY19(Low) [40]		14.8	13.0	27.8	0.34	0.34	0.68	0.74	0.82	1.56	0.90	0.98	1.88	1.00	1.09	2.09
	PRTY19(Fast) [40]		16.3	13.5	29.8	0.27	0.27	0.54	0.69	0.84	1.53	0.82	0.96	1.78	0.91	1.09	2.00
	PRTY20 [41]		16.1	41.4	57.5	0.27	0.27	0.54	0.88	0.96	1.84	1.11	1.19	2.30	1.33	1.42	2.75
	CM20 [7]		20.4	18.3	38.7	0.40	0.41	0.81	1.02	1.10	2.12	1.22	1.31	2.53	1.21	1.30	2.51
	RR16 [46]	M	13.9	881	895	6.84	6.86	13.7	7.22	7.30	14.5	7.20	7.31	14.5	9.38	9.48	18.9
	RR17(DE) [47]		87.1	63.8	151	0.68	0.72	1.40	1.45	1.55	3.00	1.92	2.08	4.00	3.61	4.08	7.69
	RR17(EC) [47]		63.2	56.7	120	0.25	0.25	0.50	0.80	0.90	1.70	1.05	1.24	2.29	1.91	2.88	4.79
	PRTY20 [41]		26.6	72.4	99.0	0.47	0.47	0.94	1.00	1.08	2.08	1.15	1.23	2.38	1.90	2.00	3.90
2^{12}	Naive	N	128	0	128	0.005	0.01	0.02	0.005	0.10	0.11	0.03	0.22	0.25	0.007	1.09	1.10
	FFH99(ECC) [18]	SH	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	FFH99(FFG) [18]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	CT10 [9]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	HL08(ECC) [16]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	HL08(FFG) [16]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	DCW13 [13]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	PSZ14 [44]		200	4.92×10^3	5.12×10^3	24.6	24.6	49.2	23.3	23.4	46.7	21.4	21.7	43.1	23.5	25.1	48.6
	PSZ14(GBF) [44]		28.1	3.69×10^3	3.72×10^3	3.39	5.44	8.83	3.73	5.82	9.55	3.85	6.55	10.4	4.46	7.48	11.9
	KKRT16 [28]		86.2	520	606	0.40	0.40	0.80	0.87	1.53	2.40	0.91	1.65	2.56	0.98	2.22	3.20
	PRTY19(Low) [40]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	PRTY19(Fast) [40]		61.6	218	280	1.00	1.00	2.00	1.43	1.52	2.95	1.66	1.77	3.43	3.27	3.68	6.95
	PRTY20 [41]		38.5	546	585	0.61	0.61	1.22	1.14	1.22	2.36	1.35	1.46	2.81	1.48	1.76	3.24
	CM20 [7]		43.3	299	342	0.56	0.57	1.13	1.01	1.10	2.11	1.09	1.20	2.29	1.12	1.39	2.51
	RR16 [46]	M	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	RR17(DE) [47]		1.33×10^3	934	2.26×10^3	2.31	2.62	4.93	3.25	3.60	6.85	3.39	3.82	7.21	3.62	4.06	7.68
	RR17(EC) [47]		1.09×10^3	926	2.02×10^3	1.14	1.16	2.30	1.71	1.81	3.52	1.78	1.98	3.76	1.89	2.60	4.49
	PRTY20 [41]		48.8	936	985	0.90	0.90	1.80	1.47	1.55	3.12	1.62	1.73	3.35	1.86	2.14	4.00
2^{16}	Naive	N	2.05×10^3	0	2.05×10^3	0.09	0.18	0.27	0.09	0.26	0.35	0.09	0.25	0.34	0.07	0.25	0.32
	FFH99(ECC) [18]	SH	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	FFH99(FFG) [18]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	CT10 [9]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	HL08(ECC) [16]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	HL08(FFG) [16]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	DCW13 [13]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	PSZ14 [44]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	PSZ14(GBF) [44]		388	5.91×10^4	5.92×10^4	517	1.01×10^3	1.53×10^3	511	1.00×10^3	1.51×10^3	728	1.43×10^3	2.16×10^3	760	1.50×10^3	2.26×10^3
	KKRT16 [28]		1.17×10^3	8.45×10^3	9.62×10^3	3.15	3.19	6.34	3.62	3.75	7.37	3.48	3.64	7.12	3.93	4.07	8.00
	PRTY19(Low) [40]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	PRTY19(Fast) [40]		781	3.54×10^3	4.32×10^3	12.6	12.7	25.3	12.6	12.8	25.4	12.9	13.0	25.91	12.7	12.8	25.5
	PRTY20 [41]		399	8.88×10^3	9.28×10^3	5.66	5.71	11.4	6.42	6.53	13.0	6.51	6.65	13.1	6.47	6.61	13.1
	CM20 [7]		403	4.87×10^3	5.27×10^3	4.00	4.08	8.08	4.68	4.81	9.49	4.54	4.70	9.24	5.61	5.74	11.4
	RR16 [46]	M	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	RR17(DE) [47]		2.18×10^4	1.52×10^4	3.70×10^3	32.2	38.0	70.2	34.5	40.5	75.0	34.3	39.9	74.2	34.1	40.2	74.3
	RR17(EC) [47]		2.02×10^4	1.52×10^4	3.54×10^4	17.8	18.1	35.9	19.2	19.6	38.8	19.4	19.8	39.2	19.2	19.6	38.8
	PRTY20 [41]		408	1.43×10^4	1.47×10^3	8.56	8.61	17.2	9.05	9.16	18.2	9.30	9.45	18.8	10.3	10.4	20.7
2^{20}	Naive	N	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	FFH99(ECC) [18]	SH	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	FFH99(FFG) [18]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	CT10 [9]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	HL08(ECC) [16]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	HL08(FFG) [16]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	DCW13 [13]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	PSZ14 [44]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	PSZ14(GBF) [44]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	KKRT16 [28]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	PRTY19(Low) [40]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	PRTY19(Fast) [40]		1.29×10^4	5.77×10^4	7.06×10^4	210	211	421	201	202	403	199	201	400	199	200	399
	PRTY20 [41]		6.16×10^3	1.52×10^5	1.58×10^5	107	108	215	102	102	204	105	106	211	97.9	99.2	197
	CM20 [7]		6.16×10^3	7.95×10^4	8.52×10^4	90.0	90.9	180.9	92.7	93.7	186	89.1	91.7	181	90.0	91.3	181
	RR16 [46]	M	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	RR17(DE) [47]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	RR17(EC) [47]		-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	PRTY20 [41]		6.16×10^3	2.25×10^5	2.31×10^5	163	164	327	140	141	281	135	137	272	137	138	275