Maliciously Secure Circuit Private Set Intersection via SPDZ-Compatible Oblivious PRF

Yaxi Yang
Singapore University of Technology and Design, Singapore
yaxi_yang@sutd.edu.sg

Xiaojian Liang
Ant International, Ant Group, China
im.liangxj@gmail.com

Xiangfu Song*
National University of Singapore & Guangzhou University, China
songxf@comp.nus.edu.sg

Ye Dong
Singapore University of Technology and Design, Singapore
ye_dong@sutd.edu.sg

Linting Huang
Guangzhou University, China
linting_huang@e.gzhu.edu.cn

Hongyu Ren
Guangzhou University, China
lubricantrhy@gmail.com

Changyu Dong*
Guangzhou University, China
changyu.dong@gmail.com

Jianying Zhou
Singapore University of Technology and Design, Singapore
jianying_zhou@sutd.edu.sg

Abstract

Circuit Private Set Intersection (Circuit-PSI) allows two parties to compute a function f on items in the intersection of their input sets without revealing items in the intersection set. It is a well-known variant of PSI and has numerous practical applications. However, existing Circuit-PSI protocols only provide security against *semi-honest* adversaries. A straightforward approach to constructing a maliciously secure Circuit-PSI is to extend a pure garbled-circuit-based PSI (NDSS'12 [23]) to a maliciously secure circuit-PSI, but it will not be concretely efficient. Another is converting state-of-the-art semi-honest Circuit-PSI protocols (EUROCRYPT'21 [54]; PoPETS'22 [10]) to be secure in the malicious setting. However, it will come across *the consistency issue* (EUROCRYPT'11 [56]) since parties can not guarantee the inputs of the function f stay unchanged as obtained from the last step.

This paper tackles the previously mentioned issue by presenting the first maliciously secure Circuit-PSI protocol. Our key innovation, the Distributed Dual-key Oblivious Pseudorandom Function (DDOPRF), enables the oblivious evaluation of secret-shared inputs using dual keys within the SPDZ MPC framework. Notably, this construction seamlessly ensures fairness within the Circuit-PSI. Compared to the state-of-the-art semi-honest Circuit-PSI protocol (PoPETS'22), experimental results demonstrate that our malicious Circuit-PSI protocol not only reduces around 5x communication costs but also enhances efficiency, particularly for modest input sets ($\leq 2^{14}$) in the case of the WAN setting with high latency and limited bandwidth.

Proceedings on Privacy Enhancing Technologies 2025(2), 680–696

https://doi.org/10.56553/popets-2025-0082

Keywords

privacy set intersection, secure multiparty computation, oblivious pseudorandom function, malicious, secret sharing

1 Introduction

Private set intersection (PSI) has emerged as a powerful multi-party computation paradigm that enables two parties to compute the intersection $\mathbf{x} \cap \mathbf{y}$ of their input sets $\mathbf{x}$ and $\mathbf{y}$ without revealing raw input data [18, 23, 30]. Serving as a silver bullet for achieving privacy during data analysis, PSI is poised to revolutionize myriad data-driven applications, such as contact tracing [19], advertising conversion [63], and genomic sequence testing [62], etc. In parallel, big data companies, in turn, develop numerous open-source PSI-related projects, including Private-Join-and-Compute[1] from Google and SecretFlow[2] from Antgroup, to further facilitate their investigation.

However, generic PSI is not applicable to accommodate subsequent computations of function at the intersection itself, which is an emerging demand in practice. For instance, in the Private-Join-and-Compute project, Google aims to compute the sum of expense values over the intersection of two databases, while not revealing the intersection. Driven by practical needs, Huang et al. [23] introduce the notion of *Circuit-PSI*, which can support arbitrary secure computations over the intersection of private sets, i.e., $f(\mathbf{x}\cap\mathbf{y})$. It captures an extensive imagination that outputs $f(\mathbf{x}\cap\mathbf{y})$, where the intermediate result, i.e., the intersection $\mathbf{x} \cap \mathbf{y}$, is kept private and sent to a circuit for a customized function f for further computation.

Existing works for Circuit-PSI [10, 23, 47, 51, 54] are under *a semi-honest model*, whose security properties may not hold in the presence of malicious adversaries. In real-world applications, designing a Circuit-PSI protocol in the *malicious model* is very meaningful, as it captures many realistic scenarios where the parties may

*Corresponding author.
[1]https://github.com/google/private-join-and-compute
[2]https://github.com/secretflow

take arbitrary strategies to break the security of a protocol. To address this problem, a trivial but less efficient solution is to integrate Huang's Circuit-PSI protocol [23] with general constructions for maliciously secure garbled circuits [35, 60]. Since Huang's protocol is a *purely garbled-circuits-based* Circuit-PSI, intuitively, two parties can use it to design a circuit that implements the function $f(\mathbf{x} \cap \mathbf{y})$ and convert this circuit into a maliciously secure version by using cut-and-choose [35] or the authenticated garbling method [60]. However, the communication complexity is exacerbated by the intricacy of the function f and the size of the initial input sets, which would lead to quadratic complexity.

Compared to Huang's protocol, the more efficient and tailored Circuit-PSI protocols in the semi-honest setting [10, 25, 26, 47, 51] leverage the *align-randomize-compare* framework to efficiently compute $f(\mathbf{x} \cap \mathbf{y})$. The process is described as follows: 1) Both parties hash their input sets, $\mathbf{x}$ and $\mathbf{y}$, into bins using hashing-to-bin techniques, to **align** the items for further processing and better efficiency. 2) Each party invokes a specialized protocol called Oblivious Programmable Pseudorandom Functions (OPPRF) to **randomize** the items within the same bins, mapping them to specific PRF values. 3) A secure comparison protocol is then employed to **compare** the PRF values bin-by-bin. Instead of directly revealing the intersection set, the parties receive shares of the intersection set $\{\mathbf{x} \cap \mathbf{y}\}$. So both parties can use the shares of $\{\mathbf{x} \cap \mathbf{y}\}$ as input to the following function f to compute $f(\mathbf{x} \cap \mathbf{y})$.

To enhance the security of these Circuit-PSI protocols against malicious adversaries, parties need to *i) substitute all underlying protocols with their corresponding malicious secure counterparts against malicious behavior, and ii) commit to the shares obtained in Step 3) before inputting them into the function f*. However, verifying the integrity of these commitments presents a challenge, as a malicious party could alter the shares locally and commit the fake shares before they are input to f. If the malicious party is able to discard or alter the results, another honest party cannot benefit from the commitment verification, as the input value for the commits is falsified. This challenge is known as *the consistency issue* [56]. Therefore, the following open problem remains:

Can we solve the consistency issue and construct a maliciously secure Circuit-PSI protocol?

We answer this question affirmatively. To address *the consistency issue*, each party needs to commit their items in the input sets and compute the PRF values with the commitments. Our initial strategy involves adopting the general malicious MPC framework, the SPDZ framework [14, 29], known for its efficient secret-shared computation paired with Message Authentication Code (MAC) capabilities. This framework facilitates the verification of secret-shared secure addition and multiplication over a finite field, ensuring that parties can confirm the accuracy of computations while maintaining the privacy of their input values. Consequently, our objective is to transform the *align-random-compare* routine, recognized for its better efficiency, into a secret-shared format utilizing the SPDZ framework. By integrating MAC authentication, we can ensure the correctness of each computation, thereby achieving a maliciously secure Circuit-PSI protocol.

However, the existing OPRF protocol in *randomize* phase of semi-honest Circuit-PSI protocols, based on IKNP-style OT [10] or Silent OT protocols [54], lacks a linear structure that is suitable for transformation into a secret-shared form. So we redesign an OPRF protocol and propose a Distributed Dual-key Oblivious Pseudorandom Function (DDOPRF) protocol based on SPDZ, which can compute a PRF of a secret-shared value with commitment. Then, the PRF values are compared to obtain the secret-shared intersection of $\{\mathbf{x} \cap \mathbf{y}\}$. This secret-shared value, combined with a MAC, can be sent to a function f, which resolves the commitment issue.

Roughly, our protocol for achieving a maliciously secure Circuit-PSI avoids the use of complex commitment protocols, as used in [40] and instead employs secret shares and MAC from SPDZ, which are as follows: 1) the input sets $\mathbf{x}$ and $\mathbf{y}$ are secret-shared with authentication between two parties and use a secret shared shuffle with malicious security to shuffle those shares. 2) Next, two parties employ our specially designed SPDZ-compatible OPRF protocol (DDOPRF) to compute the OPRF values for items in $\mathbf{x}$ and $\mathbf{y}$ in a secret shared way. 3) Ultimately, two parties compare the OPRF values and select the corresponding shares as inputs for the subsequent computation of f.

Initially, in the second phase, we intended to design a distributed OPRF protocol compatible with SPDZ (DOPRF) to compute the PRF value of the secret shared inputs. And it fulfilled our intention as expected. However, we then encountered a security flaw as the simulator can not simulate an adversary's view within this framework. To overcome this issue, we implemented a "dual-key" mechanism in the DOPRF, evolving it into DDOPRF. Remarkably, this adjustment not only addressed the security flaw, but also augmented our protocol with the feature of *two-party fairness* effortlessly. The two-party fairness ensures that if one party learns the intersection, the other must also learn it; otherwise, neither party gains any information about the intersection set [17]. It is frequently tackled in two-party PSI contexts using inefficient methods or depending on a trusted third party [15, 17]. This makes our solution more significant as it addresses these challenges more effectively.

Our Contributions. In this work, we introduce the first maliciously secure circuit-PSI protocol, designated Π_{mcPSI}, which is based on a Distributed Dual-key Oblivious Pseudorandom Function (DDOPRF) protocol that is SPDZ-compilable. In more detail, we summarise important features of our protocol as follows:

- **Malicious Security.** Based on the MAC authentication method of SPDZ, Π_{mcPSI} is the first Circuit-PSI protocol that provides malicious security property. In Π_{mcPSI}, we propose a malicious OPRF protocol based on secret sharing, named DDOPRF, which can compute PRF values in a secret-shared format. We believe that DDOPRF is of independent interest in its own right.
- **Two-party Fairness.** Surprisingly, we find that our malicious Circuit-PSI protocol can support the two-party fairness. Some fair PSI protocols have been proposed [1, 15, 17]. However, those PSI protocols are subjected to low efficiency or need a trusted third party. We achieve built-in two-party fairness with a "dual-key" mechanism and use it as a module in Π_{mcPSI}. The augmentation of the Circuit-PSI protocol with fairness incurs constant computational overhead.
- **Linear Complexity.** We build our Circuit-PSI protocol based on the DDOPRF protocol, which is succinct and only needs two-round communication to get PRF results for the input

values. Inheriting the inherent efficiency of SPDZ, Π_{mcPSI} achieves linear online efficiency. Compared to the state-of-the-art semi-honest Circuit-PSI protocol [10], our malicious circuit-PSI Π_{mcPSI} offers around 5x better communication overhead and provides better efficiency with modest input datasets ($\leq 2^{14}$) under WAN conditions (10Mbps with an echo latency of 0.02s).

Applications. The limitation of previous PSI-related works is that they are specifically designed for intersection set computations. In contrast, our Circuit-PSI protocol, based on a secret-shared computation framework, offers greater versatility and applicability. First, Circuit-PSI plays a crucial role in database operations, enabling privacy-preserving processes such as database joins [63] and queries [19]. For example, two companies that own separate databases may want to perform joint operations on the items within these databases. Using Circuit-PSI, they can align items based on indexes, select common items from both databases, and then carry out operations such as summation or variance computation on the selected items.

Another application of Circuit-PSI is in Vertical Federated Learning (VFL) [37], a privacy-preserving machine learning framework where the training dataset is vertically partitioned. In this setup, items in the datasets share the same ID but contain different features [13]. Our Circuit-PSI protocol facilitates ID alignment within a VFL system, allowing parties to securely retrieve all items with the same ID while keeping unrelated items private, enabling the execution of subsequent computations. With minimal modifications, Circuit-PSI can also be adapted for privacy-preserving telegram computation in some industrial systems [52, 64]. It facilitates the secure identification and counting of items distributed between different parties while preserving privacy. This makes it a valuable tool in various data analysis applications [34, 38].

Organization. We first present the preliminaries in § 2. Then, the protocol construction is illustrated in § 3. Next, we analyze the security and fairness of our protocols in § 4. The implementation and experimental results are shown in § 5. To demonstrate the capabilities of our protocol, we also evaluate the performance of PSI-SUM and PSI-Variant in § 5.4. These represent common function computations applied in various practical applications. We summarize the related work in § 6 and conclude our work in § 7.

2 Preliminaries

We use $\mathbb{G}$ to denote an abelian group, and $\mathbb{F}$ denotes a finite field (*e.g.*, $\mathbb{F} = \mathbb{F}_{p^k}$ for some prime p) with items of ℓ bits. $[n]$ denotes the set $\{1, ..., n\}$ and $[l, r]$ to denote $\{l, l+1, ..., r-1, r\}$. Given a set $\mathbf{x}$, we use $x \xleftarrow{\$} \mathbf{x}$ to denote x is uniformally sampled from $\mathbf{x}$. We use $a||b$ to denote strings concatenation of a and b. For an ℓ-bits string $x \in \{0,1\}^\ell$, we use b_i to denote its i-th bit of x, and $x = \sum_{i=1}^{\ell} b_i \cdot 2^{\ell-i}$. For a share of x over $\mathbb{G}$, the bit decomposition operation is a protocol for converting a share $\langle x \rangle$ into ℓ shares $\langle b_1 \rangle, ..., \langle b_\ell \rangle$, where $\langle b_1 \rangle$ represents high order bit share of x.

2.1 Security Model and Fairness Definition

We consider a two-party model in Π_{mcPSI}. Any one of the two parties can be corrupted by a malicious adversary. We prove the security of our protocols in the ideal/real world paradigm [36]. To begin with, we testify that our protocol is secure in the semi-honest setting. Then, we compile our protocol to be secure in the malicious setting. To prove the security of our circuit-PSI protocol, the standard functionality of each sub-protocol used in our protocol is presented for access as a trusted party, and to function as a sub-functionality.

We follow the simulation-based security model [36] with malicious security and static corruption. The security goals are defined as an ideal functionality $\mathcal{F}$. This ideal functionality works as a trusted entity that receives inputs from parties, performs the defined computation, and outputs results to parties. In the real world, an adversary A who represents a corrupted party C will run the protocol with the other honest parties. In the ideal world, a simulator S will interact with $\mathcal{F}$.

Definition 1. *A protocol* Π *securely computes functionality* $\mathcal{F}$ *in the presence of a malicious adversary if for every PPT adversary* A *there exists a PPT simulator* S*, such that*

$$Real_{\Pi, A(z), C}(1^\kappa, 1^\lambda, x_{i, i \notin C}) \stackrel{c}{\equiv} Ideal_{\Pi, S(z), C}(1^\kappa, 1^\lambda, x_{i, i \notin C}).$$

The left side of the equation represents the joint output from the honest parties and adversity A, and x_i represents the input from a party P_i and z is the auxiliary input from A. Similarly, the right side denotes the joint output of the honest parties and simulator S. We say that Π can securely compute functionality $\mathcal{F}$ with less than statistical error 2^λ under the malicious model.

In terms of fairness, we follow and extend the definition of two-party fairness in [5, 22, 44] as follows.

Definition 2. *A two-party secure protocol* Π *that achieves the functionality* $\mathcal{F}(x, y)$ *is* (c, ϵ)*-fair if: For any working time* t*, an adversary* A *runs the protocol* Π *for computing* $\mathcal{F}$*. Whenever* A *aborts the protocol and attempts to recover* $\mathcal{F}(x, y)$*, let* q_0 *denote the probability of success of* A*. Then, the other party* C *can run in the working time* $c \cdot t$ *for computing* $\mathcal{F}(x, y)$ *after the protocol is aborted by* A*, such that* q_1 *is the probability of success of* C*. It holds that* $|q_0 - q_1| \leq \epsilon$*.*

In this paper, we consider *partial fairness*, a relaxation of two-party complete fairness, which means that the adversary has *one-bit* privilege as the upper bound to recover the results of the protocol. Looking ahead, whenever the protocol aborts, the possibility of one party infers the results $\mathcal{F}(x, y)$ with a one-bit advantage over the other party during the same working time.

2.2 Dodis-Yampolskiy PRF

The Dodis-Yampolskiy PRF (DY-PRF) [16] requires a cyclic group $\mathbb{G}$ with prime-order p, and is defined as

$$F_{\text{DY}}(k, x) = g^{\frac{1}{k+x}}, \quad (1)$$

where g is a generator of $\mathbb{G}$, and $k \xleftarrow{\$} \mathbb{F}_p^*$. The pseudorandomness is guaranteed by the Decisional q-Diffie-Hellman Inversion Assumption (q-DDHI) [40, 41]. We define q-DDHI as follows.

The computation q-DHI problem in a group $\mathbb{G}$ with generator g and order p is to compute $g^{\frac{1}{k}}$, given $(g, g^k, \cdots, g^{k^q})$ for k randomly picked in $\mathbb{F}_p^*$. The hardness of q-DDHI for any fixed constant q is as follows. We assume gGen is an algorithm that inputs a security parameter 1^λ and outputs a modulus p and a generator g of a group

$\mathbb{G}$ of order p. q-DDHI assumption holds on group $\mathbb{G}$ and a random choice R if for every efficient algorithm $\mathcal{A}$,

$$\Big|\Pr[\mathcal{A}(g, g^k, \cdots, g^{k^q}, g^{\frac{1}{k}}) = 1 | (g,p) \leftarrow \mathrm{gGen}(1^\lambda); k \xleftarrow{\$} \mathbb{F}_p^*] - \Pr[\mathcal{A}(g, g^k, \cdots, g^{k^q}, R) = 1 | (g,p) \leftarrow \mathrm{gGen}(1^\lambda); k \xleftarrow{\$} \mathbb{F}_p^*; R \leftarrow \mathbb{G}]\Big| \leq \mathsf{negl}(\lambda). \quad (2)$$

2.3 Authenticated Secret Sharing

Linear secret sharing. We use $[\![x]\!]$ to denote an additive linear secret sharing (LSS) for $x \in \mathbb{F}$ shared between n parties such that each P_i has a random share $[\![x]\!]_i \in \mathbb{F}$ with $\sum_{i\in[n]} [\![x]\!]_i = x$. The secret x can be constructed iff all parties reveal their shares and then sum them up. Therefore, LSS preserves perfect privacy against $n-1$ corrupted parties [3]. If x and y are two values shared between n parties, LSS supports the following linear operations:

- $[\![z]\!] \leftarrow [\![x]\!] + [\![y]\!]$: P_i computes $[\![z]\!]_i \leftarrow [\![x]\!]_i + [\![y]\!]_i$;
- $[\![z]\!] \leftarrow c + [\![x]\!]$: P_0 computes $[\![z]\!]_0 \leftarrow c + [\![x]\!]_0$ and P_i computes $[\![z]\!]_i \leftarrow [\![x]\!]_i$ for all $i \in [n] \setminus \{0\}$;
- $[\![z]\!] \leftarrow c \cdot [\![x]\!]$: P_i computes $[\![z]\!]_i \leftarrow c \cdot [\![x]\!]_i$,

where we can verify $[\![x+y]\!] = [\![x]\!] + [\![y]\!]$, $[\![c+x]\!] = c + [\![x]\!]$, and $[\![c \cdot x]\!] = c \cdot [\![x]\!]$. All the operations mentioned above do not need interaction between parties. In particular, if we want to compute multiplication operation as $[\![z]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$ and verify $z = xy$, then the parties require interaction. One commonly used approach to achieve the multiplication operation is Beaver's method [50]. In detail, suppose the parties pre-share a *Beaver Triple* ($[\![a]\!], [\![b]\!], [\![c]\!]$) with $a \cdot b = c$. The parties can perform the following interaction to compute $[\![x \cdot y]\!]$ from $[\![x]\!]$ and $[\![y]\!]$:

- The parties compute $[\![e]\!] \leftarrow [\![x]\!] - [\![a]\!]$ and $[\![f]\!] \leftarrow [\![y]\!] - [\![b]\!]$;
- The parties open $[\![e]\!]$ and $[\![f]\!]$ to obtain e and f;
- The parties compute $[\![z]\!] \leftarrow [\![c]\!] + f \cdot [\![a]\!] + e \cdot [\![b]\!] + e \cdot f$,

where we can verify that $z = xy$ is just as required.

Authenticated secret sharing. Authenticated secret sharing (ASS) ensures the integrity of shared secrets. A typical SPDZ-style ASS [14] relies on *information-theoretic message authentication codes* (IT-MACs) for integrity. To be specific, the parties will additionally share $[\![\xi]\!]$ for a secret MAC key $\xi \xleftarrow{\$} \mathbb{F}$. For a share $[\![x]\!]$, the parties also share its MAC share $[\![\gamma(x)]\!]$ such that $\gamma(x) = \xi \cdot x$. We call $\langle x \rangle = ([\![x]\!], [\![\gamma(x)]\!])$ as an *authenticated secret sharing* for a secret x, and $\langle x \rangle_i = ([\![x]\!]_i, [\![\gamma(x)]\!]_i) \in \mathbb{F}^2$ as an *authenticated share* held by P_i. Since the soundness error is proportional to the inverse of the field size, we require $\mathbb{F}$ to be sufficiently large (*i.e.*, $|\mathbb{F}| > 2^\kappa$), this is crucial to detect errors with overwhelming probability. ASS supports the following local computation:

- $\langle z \rangle \leftarrow \langle x \rangle + \langle y \rangle$: $\langle z \rangle \leftarrow ([\![x]\!] + [\![y]\!], [\![\gamma(x)]\!] + [\![\gamma(y)]\!])$;
- $\langle z \rangle \leftarrow c + \langle x \rangle$: $\langle z \rangle \leftarrow (c + [\![x]\!], c \cdot [\![\xi]\!] + [\![\gamma(x)]\!])$;
- $\langle z \rangle \leftarrow c \cdot \langle x \rangle$: $\langle z \rangle \leftarrow (c \cdot [\![x]\!], c \cdot [\![\gamma(x)]\!])$,

where we can verify $\langle x + y \rangle = \langle x \rangle + \langle y \rangle$, $\langle c + x \rangle = c + \langle x \rangle$, and $\langle c \cdot x \rangle = c \cdot \langle x \rangle$.

Commonly, the parties may reveal an ASS $\langle x \rangle$ when using ASS for computation, and the parties have to make sure that x is opened correctly. To securely open an ASS share $\langle x \rangle$, the parties can leverage the embedded MAC to detect any introduced error. Specifically, the parties compute

$$[\![d]\!] \leftarrow [\![\gamma(x)]\!] - x \cdot [\![\xi]\!]. \quad (3)$$

The parties then each commit to its share of d followed by opening to check if $d = 0$ and abort it is not the case.

Computing multiplication between ASS shares $\langle x \rangle$ and $\langle y \rangle$ can be done using an *Authenticated Beaver Triple* ($\langle a \rangle, \langle b \rangle, \langle c \rangle$) satisfying $a \cdot b = c$. The parties can perform the following interaction to compute $\langle x \cdot y \rangle$ from $\langle x \rangle$ and $\langle y \rangle$:

- The parties compute $\langle e \rangle \leftarrow \langle x \rangle - \langle a \rangle$ and $\langle f \rangle \leftarrow \langle y \rangle - \langle b \rangle$;
- The parties *partically* open $\langle e \rangle$ and $\langle f \rangle$ (not their MACs) to obtain e and f;
- The parties compute $\langle z \rangle \leftarrow \langle c \rangle + f \cdot \langle a \rangle + e \cdot \langle b \rangle + e \cdot f$.

In the malicious setting, the corrupted parties may tamper their values when opening e and f. Thus, the parties must check the correct opening of e and f, using the previous method in Eq. (3). Note that the above definitions for LSS and ASS generally work over vectors. We use $[\![\mathbf{x}]\!]$ to denote a vector shares of $\mathbf{x}$, and $\gamma(\mathbf{x})$ to denote its MAC vector shares where $\gamma(\mathbf{x}_i) = \xi \cdot \mathbf{x}_i$.

Arithmetic black-box. We define the functionality of arithmetic black-box to capture the commands over ASS shares used in Π_{mcPSI} as shown in Fig. 1. We refer to well-known instantiations from existing SPDZ-style protocols [14, 27–29, 43]. For completeness, we also provide the details in §A.

Functionality $\mathcal{F}_{\mathrm{ABB}}$

Parameters: a prime p.

The ABB functionality contains the following commands:

- $\langle r \rangle \leftarrow \mathsf{Rand}()$: Output an ASS share $\langle r \rangle$ for $r \in \mathbb{Z}_p$.
- $\langle x \rangle \leftarrow \mathsf{Input}(x)$: Output a randomly ASS share $\langle x \rangle$ for the input value x.
- $(\langle a \rangle, \langle b \rangle, \langle c \rangle) \leftarrow \mathsf{RandomMul}()$: Output three ASS shares $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ such that $a \cdot b = c$.
- $\langle z \rangle \leftarrow \mathsf{Mul}(\langle x \rangle, \langle y \rangle)$: On input $\langle x \rangle$ and $\langle y \rangle$, output $\langle z \rangle$ such that $z = x \cdot y$.
- Linear combination: Given $\langle x \rangle, \langle y \rangle$ and $a, b, c \in \mathbb{Z}_p$, the parties can compute $\langle z \rangle = a \cdot \langle x \rangle + b \cdot \langle y \rangle + c$ for free with communication.
- Bit decomposition: Given $\langle x \rangle \in \mathbb{Z}_p$, the parties can get the share of a sequence $\langle b_1 \cdot 2^{\ell-1} \rangle \cdots \langle b_\ell \cdot 2^0 \rangle$, where $x = \sum_{t=0}^{\ell-1} b_t \cdot 2^{\ell-t}$.
- $x \leftarrow \mathsf{Open}(\langle x \rangle)$: On input an ASS share $\langle x \rangle$, open x to all the parties, and check the MAC value of x.

Figure 1: The arithmetic black-box functionality.

Secure Two-Party Computation. Based on the secret sharing input and all the commands over ASS, we can define a secret shared secure two-party computation functionality $\mathcal{F}_{\mathrm{2PC}}$ as in Fig. 2.

2.4 Secret-shared Shuffle

A Secret-shared Shuffle (SSS) allows shareholders to jointly permutate one secret-shared vector $\langle \mathbf{x} \rangle$ using a random permutation π known by neither party [11], where a permutation π is a bijective function $\pi : [n] \mapsto [n]$. We use $\mathbf{S}_n$ to denote a symmetric group containing all $[n] \mapsto [n]$ permutations. For a vector $\mathbf{x} = \{x_1, ..., x_n\}$,

Functionality $\mathcal{F}_{2PC}$

Parameters: two parties P_0 and P_1, a function f parameterized by a circuit C.

Functionality:
Upon receiving sharings $\langle x \rangle$ and $\langle y \rangle$ from P_0 and P_1.
The functionality computes f parameterized by the circuit C and returns the outputs $f(x, y)$ to P_0 and P_1.

Figure 2: Ideal functionality $\mathcal{F}_{2PC}$ of generic two-party computation.

when a permutation function π is applied over $\mathbf{x}$, the value x_i ($i \in [n]$) is moved to the position $\pi(i)$ as

$$\mathbf{y} = \pi(\mathbf{x}) = (\mathbf{x}_{\pi(1)}, \cdots, \mathbf{x}_{\pi(n)}). \tag{4}$$

Then, we use π^{-1} to denote the inverse of a permutation π. Therefore, $\mathbf{y}_i = \mathbf{x}_{\pi(i)}$, or equivalently, $\mathbf{x}_i = \mathbf{y}_{\pi^{-1}(i)}$. We denote by $\pi \circ \rho$ the composition of two permutations π and ρ such that $\pi \circ \rho(i) = \pi(\rho(i))$.

This paper will rely on a maliciously secure SSS ideal functionality $\mathcal{F}_{SSS}$ [57] over ASS. The functionality is formally defined in Fig. 3. The detail of the SSS protocol [57] used in our circuit-PSI protocol is described in Appendix B.

Functionality $\mathcal{F}_{SSS}$

Parameters: a prime p; n denotes the dimension of the shared vector to be shuffled.

Functionality: On input $\langle \mathbf{x} \rangle$ with $\mathbf{x} \in \mathbb{Z}_p^n$, sample a random permutation $\pi \xleftarrow{\$} \mathrm{S}_n$. Compute $\mathbf{x}' \leftarrow \pi(\mathbf{x})$ with $\mathbf{x}'_i = \mathbf{x}_{\pi(i)}$ and reshare $\langle \mathbf{x}' \rangle$ between the parties.

Figure 3: The ideal secret-shared shuffle functionality.

3 Construction

In this section, we first provide an overview and intuition of our proposed malicious circuit-PSI protocol Π_{mcPSI} in § 3.1. Next, in § 3.2, we introduce our proposed sub-protocol, DDOPRF, and explain how DDOPRF is used as the main building block of Π_{mcPSI} to fulfill privacy requirements. Then, we compile DDOPRF with malicious security § 3.3.

3.1 Workflow Overview

For two parties P_0 and P_1, their input sets are $\mathbf{x} = \{x_1, ..., x_n\}$ and $\mathbf{y} = \{y_1, ..., y_n\}$. Our malicious circuit-PSI protocol, Π_{mcPSI}, works in four phases as follows:

Phase 1: Secret Shared Shuffle. In the first phase, P_0 and P_1 will share each item in the input set using ASS, and then P_0 and P_1 will obtain $\langle \mathbf{x} \rangle$ and $\langle \mathbf{y} \rangle$. Next, the parties will take their authenticated shares as input of the functionality $\mathcal{F}_{SSS}$ to obtain the shuffled shares $\langle \pi(\mathbf{x}) \rangle$ and $\langle \rho(\mathbf{y}) \rangle$. The permutations π and ρ are random and remain unknown to any party.

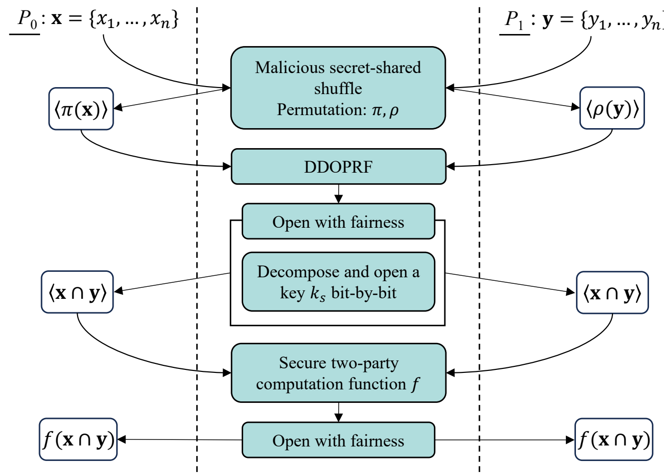

Figure 4: An overview of Π_{mcPSI}.

Phase 2: DDOPRF. In the second phase, P_0 and P_1 execute our proposed Distributed Dual-key OPRF protocol Π_{DDOPRF}, which takes the secret-shared shuffled sets generated in Phase 1 as inputs and generates the pseudorandom values for each item in the input sets without revealing the secret keys and shared values. Therefore, P_0 and P_1 will learn and open the pseudorandom values of the permuted input set $F(k, \pi(\mathbf{x}))$ and $F(k, k_s, \rho(\mathbf{y}))$, respectively.

Phase 3: Fair Comparison. In this phase, P_0 and P_1 will transform the share $\langle k_s \rangle$ over the finite $\mathbb{F}_p$ to the bit share sequence. So parties can recover k_s bit by bit to compute pseudorandom values $F(k, k_s, \pi(\mathbf{x}))$ and then compare the sets $F(k, k_s, \pi(\mathbf{x}))$ and $F(k, k_s, \rho(\mathbf{y}))$.

Phase 4: Function Computation. In this phase, P_0 and P_1 record the equal items in $F(k, k_s, \pi(\mathbf{x}))$ and $F(k, k_s, \rho(\mathbf{y}))$, and then take the corresponding shares in $\langle \pi(\mathbf{x}) \rangle$ and $\langle \rho(\mathbf{y}) \rangle$ with MACs, i.e., $\langle \mathbf{x} \cap \mathbf{y} \rangle$ as the input of predefined function f. After that, P_0 and P_1 will obtain $\langle f(\mathbf{x} \cap \mathbf{y}) \rangle$. Then P_0 and P_1 will use a similar trick in Phase 3 to recover $f(\mathbf{x} \cap \mathbf{y})$ while ensuring two-party fairness. P_0 and P_1 will generate and share a result key $\langle k_r \rangle \xleftarrow{\$} \mathbb{F}_p$. This key is used to encrypt and randomize $\langle f(\mathbf{x} \cap \mathbf{y}) \rangle$, enabling the computation of $F(k_r, \langle f(\mathbf{x} \cap \mathbf{y}) \rangle)$. Next, they use a bit decomposition protocol to change the secret shared key $\langle k_r \rangle$ bit-wise and open it bit-by-bit. Finally, P_0 and P_1 can decrypt and obtain the final results $f(\mathbf{x} \cap \mathbf{y})$.

Intuition: (1) Malicious. In Phase 4, when parties select the corresponding shares $\langle \mathbf{x} \cap \mathbf{y} \rangle$ and send them to f, any party that attempts to revise or discard the shares can be detected by checking the correctness and integrity of the MAC value of $f(\mathbf{x} \cap \mathbf{y})$. In this way, we solve the consistency issue. (2) Fairness. In Phase 2, if two parties use a plain OPRF protocol (without "dual-key") to compute the PRF results of their input sets and open those results, one dishonest party may abort at any time during the process of opening the PRF values. Although the dishonest party cannot learn the input items of another party, the adversary can learn how many items in the intersection set have been opened, i.e. $\leq |\mathbf{x} \cap \mathbf{y}|$. Then the simulator cannot simulate this adversary behavior, as it cannot learn when the adversary will abort. Therefore, we need to ensure that the adversary will learn all or nothing, as will the other party. To achieve this, we design a secret shared OPRF protocol with a dual key to bring fully malicious security. It is worth noting that the "dual-key" mechanism with bit decomposition, inherently ensures two-party fairness.

3.2 DDOPRF Protocol from DY-PRF

We propose a Distributed Dual-key OPRF (DDOPRF) protocol based on DY-PRF. In particular, the protocol starts with parties sharing a PRF key $\llbracket k \rrbracket$, a secondary key k_s and an input $\llbracket x \rrbracket$. At the end of the protocol, the parties output $F(k, x)$ or $F(k, k_s, x)$.

Before giving the details of DDOPRF, we first introduce a *multiplication secret-sharing (MSS)* over $\mathbb{G}$, which resembles LSS over $\mathbb{F}_p$. Then we design *authenticated multiplication secret sharing (AMSS)* over $\mathbb{G}$, borrowing authentication mechanisms from SPDZ-like ASS over $\mathbb{F}_p$. By carefully combining AMSS with ASS, we design maliciously secure DDOPRF with low overhead.

Multiplicative secret sharing over $\mathbb{G}$. Let $\mathbb{G}$ be a prime-order cyclic group of order p, where g is the group generator. We use $\llparenthesis x \rrparenthesis$ to denote a multiplicative secret sharing (MSS) over $\mathbb{G}$, where P_i holds a share $\llparenthesis x \rrparenthesis_i$ such that $\prod_{i\in[n]} \llparenthesis x \rrparenthesis_i = g^x$. Namely, the parties share a secret in the exponent. The above multiplicative secret sharing over $\mathbb{G}$ supports the following computation:

- $\llparenthesis z \rrparenthesis \leftarrow \llparenthesis x \rrparenthesis \cdot \llparenthesis y \rrparenthesis$: P_i computes $\llparenthesis z \rrparenthesis_i \leftarrow \llparenthesis x \rrparenthesis_i \cdot \llparenthesis y \rrparenthesis_i$;
- $\llparenthesis z \rrparenthesis \leftarrow \llparenthesis x \rrparenthesis^c$: Given a public $c \in \mathbb{Z}_p$, P_i computes $\llparenthesis z \rrparenthesis_i \leftarrow \llparenthesis x \rrparenthesis_i^c$,

where we can verify that $\llparenthesis x + y \rrparenthesis = \llparenthesis x \rrparenthesis \cdot \llparenthesis y \rrparenthesis$ and $\llparenthesis c \cdot x \rrparenthesis = \llparenthesis x \rrparenthesis^c$. Namely, multiplication between $\llparenthesis x \rrparenthesis$ and $\llparenthesis y \rrparenthesis$ corresponds to addition in the exponent, and $\llparenthesis x \rrparenthesis^c$ corresponds to scalar multiplication in the exponent.

Authenticated multiplicative secret sharing over $\mathbb{G}$. Similarly, we define authenticated multiplicative secret sharing (AMSS) $\langle\!\llbracket x \rrbracket\!\rangle = (\llparenthesis x \rrparenthesis, \llparenthesis \gamma(x) \rrparenthesis)$ over $\mathbb{G}$, where $\gamma(x) = \xi \cdot x \pmod{p}$. We assume the parties share the MAC key ξ using an LSS share $\llbracket \xi \rrbracket$. AMSS supports the following local computation:

- $\langle\!\llbracket z \rrbracket\!\rangle \leftarrow \langle\!\llbracket x \rrbracket\!\rangle \cdot \langle\!\llbracket y \rrbracket\!\rangle$: $\langle\!\llbracket z \rrbracket\!\rangle \leftarrow (\llparenthesis x \rrparenthesis \cdot \llparenthesis y \rrparenthesis, \llparenthesis \gamma(x) \rrparenthesis \cdot \llparenthesis \gamma(y) \rrparenthesis)$,
- $\langle\!\llbracket z \rrbracket\!\rangle \leftarrow \langle\!\llbracket x \rrbracket\!\rangle^c$: $\langle\!\llbracket z \rrbracket\!\rangle \leftarrow (\llparenthesis x \rrparenthesis^c, \llparenthesis c \cdot \gamma(x) \rrparenthesis)$; here $c \in \mathbb{Z}_p$.

We can verify that $\langle\!\llbracket x + y \rrbracket\!\rangle = \langle\!\llbracket x \rrbracket\!\rangle \cdot \langle\!\llbracket y \rrbracket\!\rangle$ and $\langle\!\llbracket c \cdot x \rrbracket\!\rangle = \langle\!\llbracket x \rrbracket\!\rangle^c$.

Functionality $\mathcal{F}_{\mathsf{ABB+}}$

Parameters: a prime p; a cyclic group $\mathbb{G}$ of order p, where g is the generator of $\mathbb{G}$.

The ABB functionality contains the following commands:

- Rand, RandMul, Mul, Open defined as in $\mathcal{F}_{\mathsf{ABB}}$.
- $\langle\!\llbracket x \rrbracket\!\rangle \leftarrow \mathsf{Convert}(\langle x \rangle)$: On input a ASS share $\langle x \rangle$, output an AMSS sharing $\langle\!\llbracket x \rrbracket\!\rangle$.
- $g^x \leftarrow \mathsf{Open}(\langle\!\llbracket x \rrbracket\!\rangle)$: On input a group ASS share $\langle\!\llbracket x \rrbracket\!\rangle$, output g^x to all the parties, and check the MAC value of g^x.

Figure 5: The extended arithmetic black-box functionality.

Share conversion from $\langle x \rangle$ to $\langle\!\llbracket x \rrbracket\!\rangle$. We note that an MSS share $\llparenthesis x \rrparenthesis$ over $\mathbb{Z}_p$ can be non-interactively converted from an LSS share $\llbracket x \rrbracket$ over $\mathbb{G}$ of order p. In particular, each party locally computes $\llparenthesis x \rrparenthesis_i \leftarrow g^{\llbracket x \rrbracket_i}$. Similarly, the parties can obtain an AMSS share $\langle\!\llbracket x \rrbracket\!\rangle$ over $\mathbb{G}$ from an ASS share $\langle x \rangle$ over $\mathbb{Z}_p$, by simply running the above conversion for $\llbracket x \rrbracket$ and $\llbracket \gamma(x) \rrbracket$, respectively. In this paper, we use $\langle\!\llbracket x \rrbracket\!\rangle \leftarrow \mathsf{Convert}(\langle x \rangle)$ to denote the conversion.

Secretly open g^x from $\langle\!\llbracket x \rrbracket\!\rangle$. To open g^x from $\langle\!\llbracket x \rrbracket\!\rangle$ correctly, similar to the trick used in ASS, the parties can leverage the MAC share $\llparenthesis \gamma(x) \rrparenthesis$ to detect any possible error. In particular, the parties run the following open protocol $\mathsf{Open}(\langle\!\llbracket x \rrbracket\!\rangle)$ to detect possible errors during opening:

1. Each party P_i reveals its share $\llparenthesis x \rrparenthesis_i$. By combining all parties' shares, the parties obtain $g^{x'}$, and $g^{x'}$ may not equate to g^x due to additive errors.
2. Each parties P_i computes $d_i \leftarrow (g^{x'})^{\llbracket \gamma \rrbracket_i} / \llparenthesis \gamma(x) \rrparenthesis_i$.
3. After each party committing to d_i, all the parties decommit d_i and check whether $\prod_i d_i = 1$ over $\mathbb{G}$. Abort if the check fails.

The above check resembles the MAC check for SPDZ ASS in Equation 3, despite the check being evaluated in the exponent. Correctness is easy to check:

$$\begin{aligned}\prod_i d_i &= \prod_i (g^{x'})^{\llbracket \xi \rrbracket_i} / \prod_i \llparenthesis \gamma(x) \rrparenthesis_i \\ &= g^{\sum_i (x' \llbracket \xi \rrbracket_i)} / g^{\gamma(x)} \\ &= g^{x' \xi} / g^{x \cdot \xi},\end{aligned}$$

which equals to 1 over $\mathbb{G}$ *iff* $x = x'$.

The enhanced ABB+ functionality $\mathcal{F}_{\mathsf{ABB+}}$. We formalize an enhanced ABB functionality called ABB+ in Figure 5, which captures not only commands for SPDZ ASS over $\mathbb{F}_p$ but also the commands for AMSS over $\mathbb{G}$; here $|\mathbb{G}| = p$. In the following, whenever we require operation over ASS and AMSS authenticated shares, we will directly resort to this ABB+ functionality. This modular formalization enables a clear and easy-understanding design.

Given the ABB+ functionality, we can construct the DDOPRF protocol.

A semi-honest DDOPRF protocol. We design a semi-honest DDOPRF protocol with one PRF key as follows:

- The parties generate a share $\llbracket r \rrbracket$ for a random secret $r \in \mathbb{Z}_p$ and a beaver triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$.
- The parties compute $\llbracket d \rrbracket \leftarrow \llbracket r \rrbracket \cdot (\llbracket k \rrbracket + \llbracket x \rrbracket)$ using $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$. The parties open $\llbracket d \rrbracket$ to obtain d.
- The parties compute $\llbracket e \rrbracket \leftarrow \llbracket r \rrbracket \cdot d^{-1}$.
- The parteis locally run $\llparenthesis e \rrparenthesis \leftarrow \mathsf{Convert}(\llbracket e \rrbracket)$. The parties open $\llparenthesis e \rrparenthesis$.

The correctness of the above protocol is easy to check:

$$\begin{aligned}\llbracket e \rrbracket &= \llbracket r \rrbracket \cdot d^{-1} = \llbracket r \rrbracket \cdot (r \cdot (k + x))^{-1} \\ &= \llbracket r \cdot (r \cdot (k + x))^{-1} \rrbracket \\ &= \llbracket (k + x)^{-1} \rrbracket\end{aligned}$$

From the definition of Convert, for two parties, P_0 can compute $g^{\llbracket e \rrbracket_0}$ and P_1 can compute $g^{\llbracket e \rrbracket_1}$. Clearly,

$$g^{\llbracket e \rrbracket_0} \cdot g^{\llbracket e \rrbracket_1} = g^e = g^{\frac{1}{k+x}}.$$

Efficiency properties. The above semi-honest DDOPRF protocol only has a low communication round and supports secret shared data. Specifically, parties only perform one secret-shared multiplication for computation $[\![d]\!]$ followed by two openings: one is for opening d and another for opening g^e. It only requires two rounds to compute the OPRF output. As we discussed in related work, previous constructions of OT-based OPRF protocols suffer from diverse shortcomings, including high communication complexity, or not supporting secret-shared data structure.

Protocol Π_{DDOPRF}

Parameters: The DY-PRF $F(k, x) = g^{\frac{1}{k+x}}$; an ASS share $\langle k \rangle$ for the PRF secret key k; an optional input ASS share $\langle k_s \rangle$ for the secondary PRF secret key k_s;

Protocol: On input $\langle x \rangle$ and $\langle k \rangle, \langle k_s \rangle$, do the following:

1. $\langle r \rangle \leftarrow \mathcal{F}_{\text{ABB}}.\mathsf{Rand}()$
2. $\langle d \rangle \leftarrow \langle r \rangle \cdot (\langle k \rangle + \langle x \rangle)$
3. open $d \leftarrow \mathcal{F}_{\text{ABB+}}.\mathsf{Open}(\langle d \rangle)$
4. $\langle e \rangle \leftarrow d^{-1} \cdot \langle r \rangle$
5. If $\langle k_s \rangle$ is provided as an input, $\langle e \rangle \leftarrow \langle e \rangle \cdot \langle k_s \rangle$
6. $\langle\!\langle e \rangle\!\rangle \leftarrow \mathcal{F}_{\text{ABB}^+}.\mathsf{Convert}(\langle e \rangle)$
7. open $g^e \leftarrow \mathcal{F}_{\text{ABB}^+}.\mathsf{Open}(\langle\!\langle e \rangle\!\rangle)$, e can be $g^{\frac{1}{k+x}}$ or $g^{\frac{k_s}{k+x}}$.

Figure 6: The malicious DDOPRF protocol.

3.3 Compile DDOPRF with Malicious Security

This semi-honest DDOPRF enjoys low communication costs. Unfortunately, compiling it to be maliciously secure using generic techniques (*e.g.*, zero-knowledge proof, GMW compiler) [8, 21] will introduce highly expensive costs. In this section, we show how to achieve the compilation with very low overhead.

Using $\mathcal{F}_{\text{ABB+}}$, we design a maliciously secure DDOPRF protocol Π_{DDOPRF} in a modular fashion, as illustrated in Figure 6. The idea is to authenticate the secure computation using prior authenticated mechanisms from ASS and AMSS. In detail, the first key is to randomize the input value x and get a PRF value of x, and the secondary key is to re-randomize the PRF value. The ASS share $\langle k \rangle$ of the PRF secret key k is generated to compute the pseudorandom value of the input x. In steps 1-4, P_0 and P_1 can compute an ASS share $\langle e \rangle = \langle (k + x)^{-1} \rangle$. Instead of directly converting it to AMSS share, P_0 and P_1 add a secondary PRF secret key k_s on the PRF value e as in step 5, we can get a PRF value associated with two keys and convert it to AMSS share $\langle\!\langle e \rangle\!\rangle = \langle\!\langle g^{\frac{k_s}{k+x}} \rangle\!\rangle$ in step 6. If not, P_0 and P_1 will directly convert e into AMSS share. Finally, P_0 and P_1 open the final OPRF value in Step 7. Specifically, the "dual-key" mechanism in DDOPRF brings the built-in *fairness* of the following computation. We will present how our proposed DDOPRF with correlated keys can guarantee *fairness* of Π_{mcPSI} in Section. 3.4.

Compared to the semi-honest DDOPRF, the malicious version also features a low-round property and low communication. Specifically, Π_{DDOPRF} requires three rounds of communication: The first round is from computing $\langle e \rangle$ using a beaver triple, the second round opens $\langle\!\langle e \rangle\!\rangle$, and the third round checks the opened result g^e.

Besides the efficiency, we capture the correctness and security of Π_{DDOPRF} in Theorem 1.

Theorem 1. *In the* $\{\mathcal{F}_{\text{ABB}}, \mathcal{F}_{\text{ABB}^+}\}$*-hybrid model, the protocol* Π_{DDOPRF} *implements* $\mathcal{F}_{\text{DDOPRF}}$ *correctly and securely against malicious adversary.*

The ideal functionality and full proof of Theorem 1 are in Appendix .C.

Functionality $\mathcal{F}_{\text{mcPSI}}$

Parameters: The party P_0 inputs $\mathbf{x} = \{x_1, \cdots, x_n\}$, and another party P_1 has an input set $\mathbf{y} = \{y_1, \cdots, y_n\}$;

$\mathcal{F}_{\text{mcPSI}}$:

1. On receiving $(\mathcal{F}_{\text{mcPSI}}, \mathbf{x})$ from P_0 and $(\mathcal{F}_{\text{mcPSI}}, \mathbf{y})$ from P_1, the functionality stores $\mathbf{x}$ and $\mathbf{y}$ and waits. If any party aborts, outputs $\perp$ to P_0 and P_1. Otherwise, continue.
2. On receiving (compute) from P_0 and P_1, the functionality outputs the computation results $f(\mathbf{x} \cap \mathbf{y})$ and size of the intersection set $|\mathbf{x} \cap \mathbf{y}|$ to both P_0 and P_1 if it does not abort. Otherwise, $\perp$ is output to P_0 and P_1.

Figure 7: Ideal functionality of $\mathcal{F}_{\text{mcPSI}}$.

3.4 Our Circuit-PSI from DDOPRF

In this section, we show the construction of Π_{mcPSI}, which uses Π_{DDOPRF} as a core building block. In addition, we illustrate some other common PSI computations based on Π_{mcPSI}, including PSI with payload computation (each item in the input set has a corresponding payload, with a compute function f applied to the payloads of the items in the intersection set).

We define the ideal functionality of $\mathcal{F}_{\text{mcPSI}}$ as in Fig 7. And Π_{mcPSI} is shown as Fig 8. In the two-party setting, P_0 and P_1 have two input sets $\mathbf{x}$ and $\mathbf{y}$, respectively. We will introduce Π_{mcPSI} in more detail in the following.

Phase 1: Secret Shared Shuffle. As shown in Step 1 in Fig 8, on the input sets $\mathbf{x}$ and $\mathbf{y}$, P_0 and P_1 invoke the functionality $\mathcal{F}_{\text{ABB}}.\mathsf{Input}$ to get the ASS shares $\langle \mathbf{x} \rangle$ and $\langle \mathbf{y} \rangle$ of their input sets. Next, in Step 2, P_0 and P_1 will use Π_{SSS} to secretly shuffle their input shares $\langle \mathbf{x} \rangle$ and $\langle \mathbf{y} \rangle$, i.e., $\langle \mathbf{x}' \rangle = \pi(\langle \mathbf{x} \rangle) \leftarrow \mathcal{F}_{\text{SSS}}(\langle \mathbf{x} \rangle)$, $\langle \mathbf{y}' \rangle = \rho(\langle \mathbf{y} \rangle) \leftarrow \mathcal{F}_{\text{SSS}}(\langle \mathbf{y} \rangle)$. Neither P_0 or P_1 know the permutation methods π and ρ.

Phase 2: DDOPRF. After P_0 and P_1 shuffle their input shares, they can not learn anything from the shuffled shares $\langle \mathbf{x}' \rangle$ and $\langle \mathbf{y}' \rangle$. Then, in Steps 3-5, P_0 and P_1 will invoke protocol Π_{DDOPRF} as defined in Fig. 6 to generate the pseudorandom values for the permuted shares. Specifically, let $\langle k \rangle, \langle k_s \rangle \leftarrow \mathcal{F}_{\text{ABB}^+}.\mathsf{Rand}()$ be the PRF keys shares for DDOPRF protocol. P_0 and P_1 run DDOPRF protocol over $\langle \mathbf{x}' \rangle$ and $\langle \mathbf{y}' \rangle$ using key shares $\langle k \rangle$ and $\langle k_s \rangle$. In Step 4, P_0 and P_1 output the OPRF values of $\mathbf{x}'$ under the PRF key share $\langle k \rangle$, denoted as $\{F(k, \mathbf{x}'_i)\}_{i \in [1,n]}$. In Step 5, P_0 and P_1 output the OPRF values of $\mathbf{y}'$ under the PRF key share $\langle k \rangle$ and secondary key share $\langle k_s \rangle$, denoted as $\{F(k, k_s, \mathbf{y}'_i)\}_{i \in [1,n]}$.

Phase 3: Fair Comparison. To make sure that P_0 and P_1 can compare the OPRF values at the same time, and they can get all the

Protocol Π_{mcPSI}

Parameters: Party P_b $(b \in \{0,1\})$; $\mathbf{x} = \{x_1, \cdots, x_n\}$ and $\mathbf{y} = \{y_1, \cdots, y_n\}$ denote two sets with n values; an authenticated vector $\mathbf{x}$ where $\mathbf{x} \in \mathbb{Z}_p^n$; the length of each item in $\mathbf{x}$ and $\mathbf{y}$ is ℓ; an ASS share $\langle k \rangle$ for the PRF secret key.

Protocol:

1. For $i \in [1, n]$: P_0 and P_1 generate ASS shares of their inputs $\langle x_i \rangle \leftarrow \mathcal{F}_{\text{ABB}}.\mathsf{Input}(x_i)$, $\langle y_i \rangle \leftarrow \mathcal{F}_{\text{ABB}}.\mathsf{Input}(y_i)$;
2. P_0 and P_1 use Π_{SSS} to shuffle their shares $\langle \mathbf{x}' \rangle \leftarrow \mathcal{F}_{\text{SSS}}(\langle \mathbf{x} \rangle)$, $\langle \mathbf{y}' \rangle \leftarrow \mathcal{F}_{\text{SSS}}(\langle \mathbf{y} \rangle)$;
3. Let $\langle k \rangle, \langle k_s \rangle \leftarrow \mathcal{F}_{\text{ABB}^+}.\mathsf{Rand}()$ be the PRF key share for Π_{DDOPRF};
4. Run DDOPRF protocol over $\langle \mathbf{x}' \rangle$ using key share $\langle k \rangle$. Denote the output as $F(k, \mathbf{x}')$, where each $F(k, x'_i) = g^{\frac{1}{k+x'_i}}$;
5. Run DDOPRF protocol over $\langle \mathbf{y}' \rangle$ using key share $\langle k \rangle, \langle k_s \rangle$. Denote the output as $F(k, k_s, \mathbf{y}')$,where each $F(k, k_s, y'_i) = g^{\frac{k_s}{k+y'_i}}$;
6. Use the bit decomposition operation over $\langle k_s \rangle$, and get the share of sequence $\langle b_1 \cdot 2^{\ell-1} \rangle \cdots \langle b_\ell \cdot 2^0 \rangle$, where b_t $(t \in [1, \ell])$ is the t-th bit of k_s (left most first);
7. P_0 and P_1 open $\langle b_1 \cdot 2^{\ell-1} \rangle \cdots \langle b_\ell \cdot 2^0 \rangle$ one by one, and reconstruct k_s locally;
8. P_0 and P_1 locally compute $(F(k, x'_i))^{k_s}$ to get $F(k, k_s, x'_i)$;
9. $P_{b, b \in \{0,1\}}$ prepares two empty sets $\mathbf{R}_{X_b}$ and $\mathbf{R}_{Y_b}$;
10. For $i \in [1, n], j \in [1, n]$:
    (a) P_0 and P_1 compare $F(k, k_s, \mathbf{x}'_i)$ and $F(k, k_s, \mathbf{y}'_j)$;
    (b) If $F(k, k_s, \mathbf{x}'_i) = F(k, k_s, \mathbf{y}'_j)$, $P_{b, b \in \{0,1\}}$ picks out the matched record $\mathbf{R}_{X_b} = \langle \mathbf{x}'_i \rangle \cup \mathbf{R}_{X_b}$ and $\mathbf{R}_{Y_b} = \langle \mathbf{y}'_j \rangle \cup \mathbf{R}_{Y_b}$;
11. For $f(\mathbf{x} \cap \mathbf{y})$: P_0 and P_1 call $\mathcal{F}_{\text{2PC}}$ with the input shares $\mathbf{R}_{X_0}$ and $\mathbf{R}_{Y_0}$, $\mathbf{R}_{X_1}$ and $\mathbf{R}_{Y_1}$, and this $\mathcal{F}_{\text{2PC}}$ achieves the function f;
12. When P_0 and P_1 open the computation results by invoking $\mathcal{F}_{\text{ABB}}.\mathsf{Open}$, they will check the corresponding MAC values. If an error happens, the protocol will abort.

Figure 8: Protocol Π_{mcPSI} using DDOPRF.

comparison results or nothing, P_0 and P_1 will run a bit decomposition protocol on the secondary key share $\langle k_s \rangle$ in Step 6. Then, they can open the bit composition of k_s one by one to reconstruct k_s. After that, P_0 and P_1 can compute $(F(k, x'_i))^{k_s} = g^{\frac{k_s}{k+x'_i}}$ with $g^{\frac{k_s}{k+y'_i}}$ to learn which items are in the intersection set.

Phase 4: Function Computation. In the following, P_0 and P_1 intend to find the corresponding shares in the intersection set. In Step 9, P_0 and P_1 will prepare two empty sets to store the shares. In Step 10, the parties can find matches over $\{F(k, k_s, \mathbf{x}'_i)\}_{i \in [1,n]}$ and $\{F(k, k_s, \mathbf{y}'_j)\}_{j \in [1,n]}$ and store the corresponding matched shares in $\mathbf{R}_{X_b}$ and $\mathbf{R}_{Y_b}$. Therefore, in Step 11, P_0 and P_1 use the stored shares as inputs to the following function f to compute $f(\mathbf{x} \cap \mathbf{y})$. This function can then be securely evaluated by $\mathcal{F}_{\text{2PC}}$. Since the input values have been shuffled before the Π_{DDOPRF}, P_0 and P_1 cannot correlate their original input values $\mathbf{x}$ and $\mathbf{y}$ with the pseudorandom results. As we can see, the consistency issue we mentioned before will be solved since the shares sent into the circuit are with MAC shares. P_0 and P_1 can access the final results to verify whether any modifications were made to the shares prior to the circuit computation.

Specifically, as mentioned in [47, 48], some circuit-based PSI protocols require the function f to be symmetric. Namely, the function's output must not depend on the order of its inputs. For non-symmetric functions, the circuit computing the intersection must shuffle its output to ensure each item of the intersection is placed in a location independent of the other values. In Π_{McPSI}, the two input sets are secret-shared and shuffled in the first phase, enabling support for non-symmetric functions as well. (However, it is challenging to identify practical examples of interesting non-symmetric functions related to the intersection, aside from the intersection itself [48]).

Malicious PSI with payload computation (labeled circuit-PSI). We present how to use Π_{mcPSI} to achieve PSI with payload computation as follows. Let us assume that two secret-shared tables $\langle X \rangle$ and $\langle Y \rangle$, and X and Y are both two-column tables of P_0 and P_1, where the first column is the ID column and the second is the payload column. P_0 and P_1 want to perform an intersection over two ID columns for X and Y and then select out all the payload values associated with the IDs in the intersection.

Similarly, P_0 and P_1 will compute the ASS for their input matrices, shown as $\langle X \rangle, \langle Y \rangle$. Next, P_0 and P_1 perform row-wise secret-shared shuffle over $\langle X \rangle$ and $\langle Y \rangle$. Let us denote the shuffled table as $\langle X' \rangle = \langle \pi(X) \rangle$ and $\langle Y' \rangle = \langle \rho(Y) \rangle$ for some random permutation methods π and ρ. Neither of the parties learns about the permutation methods. Then, P_0 and P_1 invoke Π_{DDOPRF}. For Π_{DDOPRF}, they first sample a random share $\langle k \rangle$ as the ASS key share of the DY-PRF. Parse $\langle X' \rangle$ as $(\langle X'^{(0)} \rangle, \langle X'^{(1)} \rangle)$ and $\langle Y' \rangle$ as $(\langle Y'^{(0)} \rangle, \langle Y'^{(1)} \rangle)$, where $X'^{(0)}$ and $Y'^{(0)}$ are the ID columns. The parties run Π_{DDOPRF} over the ID column of $\langle X' \rangle$ using PRF key share $\langle k \rangle$, and run Π_{DDOPRF} over $\langle Y' \rangle$ with PRF key share $\langle k \rangle$ and secondary key share $\langle k_s \rangle$. Then, they invoke the bit decomposition protocol on $\langle k_s \rangle$ to open this key bit by bit. So parties can compute $\{F(k, k_s, X'^{(0)}_i)\}_{i \in [1,n]}$ to do the following comparison. At the end of this protocol, the parties can learn the pseudo-random values of those ID columns, denoted as $\{F(k, k_s, X'^{(0)}_i)\}_{i \in [1,n]}$ and $\{F(k, k_s, Y'^{(0)}_i)\}_{i \in [1,n]}$.

For $i, j \in [n]$, if $F(k, k_s, X'^{(0)}_i) = F(k, k_s, Y'^{(0)}_j)$, P_0 (P_1) picks out the matched records $R_{X_0} = \langle X'_i \rangle \cup R_{X_0}$ ($R_{X_1} = \langle X'_i \rangle \cup R_{X_1}$) and $R_{Y_0} = \langle Y'_j \rangle \cup R_{Y_0}$ ($R_{Y_1} = \langle Y'_i \rangle \cup R_{Y_1}$). For payload computation, the parties take their shares $R^{(1)}_{X_0}$ and $R^{(1)}_{Y_0}$, $R^{(1)}_{X_1}$ and $R^{(1)}_{Y_1}$ as inputs to the following payload computation. Note that, due to the secret-shared shuffle, the parties do not know which records are matched, and they only learn the number of matched records at the end of the protocol.

4 Security Proof and Fairness Analysis

In this section, we will give a security proof of Π_{mcPSI} and also analysis the two-party fairness in Π_{mcPSI}.

Theorem 2. *In the* $\{\mathcal{F}_{\text{ABB}}, \mathcal{F}_{\text{ABB}^+}, \mathcal{F}_{\text{SSS}}, \mathcal{F}_{\text{DDOPRF}}\}$*-hybrid model, the protocol* Π_{mcPSI} *implements* $\mathcal{F}_{\text{mcPSI}}$ *correctly and securely against malicious adversary and achieves* $(2,0)$*-fair.*

Proof Sketch. In this part, we give an essential proof sketch of $\mathcal{F}_{\text{mcPSI}}$ to establish that it is maliciously secure and fair. First, we solve the consistent issue by designing an SPDZ-compatible OPRF protocol (i.e., DDOPRF). That is, we augment OPRF by adding the authentication mechanisms provided by SPDZ in the secret-sharing format. Instead of using heavy asymmetric-based commitment schemes, SPDZ provides the symmetric-key counterpart MAC for authentication. A MAC is a way of authenticating a value, ensuring that any revisions to the value can be detected by checking its MAC. DDOPRF perfectly integrates all the features in SPDZ, including MAC. Therefore, the computation results sent to a function f will be checked by MAC, and any changes to those results will be detected. At the end of this protocol, the correctness of $\mathcal{F}_{\text{mcPSI}}$ can be ensured. Nevertheless, the correctness and security of $\mathcal{F}_{\text{mcPSI}}$ are guaranteed by the primitives used in it. The detailed simulation-based proofs are shown in Appendix C.

Fairness of Protocol Π_{mcPSI}**.** Next, we discuss the fairness of our protocol. $\mathcal{F}_{\text{mcPSI}}$ achieves $(2,0)$-fair. According to the partial fairness definition in Definition 2, whenever the adversary aborts the protocol, the upper bound of its advantage in recovering the results is known one more bit than the other party. Therefore, the recovering time of the adversary will be half of the other party, and it can achieve the same probability of success. Besides, the fairness of $\mathcal{F}_{\text{mcPSI}}$ is guaranteed by the bit-decomposition protocol, which is also built on SPDZ and secure under the malicious model.

In Π_{mcPSI}, one party obtains PRF values of the input set $g^{\frac{1}{k+x'_i}}$ ($i \in [1,n]$), and another party gets re-randomized input set with two PRF keys $\langle g^{\frac{k_s}{k+y'_i}} \rangle$. We propose that the secondary key k_s is used to re-randomize the PRF value and ensure fairness. If two parties only compute standard PRF values with only one PRF key, i.e., $g^{\frac{1}{k+x'_i}}$ and $g^{\frac{1}{k+y'_i}}$, then they open those PRF values to each other one by one to compare each value. However, if one of the parties is corrupted, the corrupted party can quit at any time during the opening process. We can observe that, the corrupted party can learn some information, such as the intersection set having at least t items if the corrupted party finds t items in the intersection before quitting. Moreover, the corrupted party might obtain more information than another party. If the corrupted party aborts after another party shares one item (if this item belongs to the intersection set), the corrupted party can learn the intersection set at least has one item but another party learns nothing. So it would be hard to measure the leakage based on the ideal/real-world simulation method. The simulator can not simulate when the adversary would abort and define the amount of leakage.

Therefore, we construct Π_{DDOPRF}, and we find that if we construct two correlated keys for the PRF value, the problems mentioned above can be solved. Specifically, two parties will open $g^{\frac{1}{k+x'_i}}$ and $\langle g^{\frac{k_s}{k+y'_i}} \rangle$. Because the randomization of the PRF value $\langle g^{\frac{k_s}{k+y'_i}} \rangle$ is guaranteed by the secondary key k_s, two parties can not distinguish it with a random value. Afterward, they invoke a bit decomposition protocol to recover k_s bit by bit. Subsequently, the two parties can locally compute the PRF values for input sets with the same keys to get the intersection result. We can observe that, the adversary will learn the final intersection set or nothing.

If two parties intend to compute fair f(PSI), for the final result of f(PSI), two parties also can use the same trick as used in DDOPRF to guarantee *fairness.* To be specific, before the parties reveal the final shares, they will select a secret key to encrypt their shares. After they open the encrypted shares, they will open the encrypted keys bit-by-bit to decrypt the shares and get the final results.

5 Implementation and Performance

In this section, we will experimentally evaluate our circuit-PSI protocol Π_{mcPSI}. In § 5.1, we give the benchmarking environment. Then in § 5.2, we show the offline and online performance of Π_{mcPSI} and give the breakdown of computation and communication costs, and we show the details in terms of multiple-threads. In § 5.3, we compare Π_{mcPSI} with the state-of-the-art semi-honest circuit-PSI protocol [10] in single-threaded runtime on different networks, and also other representative PSI protocols [23, 40]. Additionally, to demonstrate the parallelizability and scalability of Π_{mcPSI}, we present its performance across various applications, including PSI-Sum and PSI-Variance, using different numbers of threads. These results are detailed in § 5.4. The code of our paper is available at https://github.com/mcPSI.

5.1 Benchmarking Environment

We implement Π_{mcPSI} in C++ and based on YACL [3] [39], which provides several cryptographic interfaces (e.g., pseudo-random generator, oblivious transfer, network). We run most experiments on a desktop PC equipped with 12th Gen Intel(R) Core(TM) i9-12900K at Ubuntu 20.04 LTS and 125 GB of memory and in three different network settings with the Linux tc command. One is a local host setting. Another is the local-area network (LAN) with 1 Gbps. The third setting simulates two wide-area networks (WAN): one with 100 Mbps and another with 10 Mbps, both with a 0.02s round-trip time (RTT). Except for the experiments compared with MPRS20 [40], which are not open-source and were run on the Google Cloud Platform[4] using a virtual machine equipped with an E5 processor and 3.75 GB of memory. To ensure a fair comparison, we also tested Π_{mcPSI} on the same platform. In our paper, the computational security parameter is $\kappa = 128$, the statistical security parameter is $\lambda = 64$, and the size of each element is $\ell = 128$.

5.2 Performance of Π_{mcPSI}

In this section, we show the thorough performance of Π_{mcPSI}. We give the specific numbers of Π_{mcPSI} in the different network settings

[3]https://github.com/secretflow/yacl
[4]See https://console.cloud.google.com/ for different cloud services.

Table 1: The break down and total running time (in s) and communication cost (in MB) of the online phase of Π_{mcPSI} for different set sizes ($n \in \{2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$) in LAN and WAN settings. T= $\{1, 2, 4, 8\}$ represents the number of threads. The total time includes the functionality of secret shared shuffle and DDOPRF in Phases 1, 2, and 3 of Π_{mcPSI}, and the system initialization.

Set size		Time LAN				Time WAN 100Mbps				Time WAN 10Mbps				Comm.
		T=1	T=2	T=4	T=8	T=1	T=2	T=4	T=8	T=1	T=2	T=4	T=8	
2^8	$\mathcal{F}_{\text{SSS}}$	0.002	0.002	0.002	0.002	0.005	0.005	0.005	0.005	0.051	0.051	0.051	0.051	0.024
	$\mathcal{F}_{\text{DDOPRF}}$	0.136	0.085	0.079	0.078	0.153	0.122	0.109	0.096	0.228	0.176	0.173	0.173	0.057
	Total	0.146	0.099	0.089	0.088	0.169	0.138	0.124	0.111	0.299	0.247	0.244	0.244	0.081
2^{10}	$\mathcal{F}_{\text{SSS}}$	0.004	0.004	0.004	0.004	0.022	0.022	0.022	0.022	0.206	0.206	0.206	0.206	0.098
	$\mathcal{F}_{\text{DDOPRF}}$	0.466	0.259	0.198	0.163	0.496	0.316	0.242	0.189	0.727	0.548	0.531	0.487	0.214
	Total	0.479	0.272	0.211	0.176	0.529	0.349	0.308	0.223	0.956	0.778	0.759	0.716	0.312
2^{12}	$\mathcal{F}_{\text{SSS}}$	0.010	0.008	0.007	0.007	0.083	0.081	0.081	0.079	0.747	0.746	0.746	0.746	0.393
	$\mathcal{F}_{\text{DDOPRF}}$	1.782	1.052	0.682	0.511	1.915	1.139	0.688	0.606	2.773	1.877	1.549	1.425	0.754
	Total	1.861	1.129	0.758	0.587	2.015	1.283	0.808	0.755	3.627	2.920	2.416	2.236	1.147
2^{14}	$\mathcal{F}_{\text{SSS}}$	0.039	0.037	0.034	0.034	0.318	0.316	0.315	0.315	3.015	2.964	2.962	2.962	1.573
	$\mathcal{F}_{\text{DDOPRF}}$	7.323	4.497	2.553	1.786	7.474	4.862	2.734	2.161	10.663	8.084	5.876	5.323	2.946
	Total	7.431	4.603	2.657	1.872	7.881	5.267	3.138	2.565	13.814	11.183	8.977	8.422	4.519
2^{16}	$\mathcal{F}_{\text{SSS}}$	0.134	0.131	0.129	0.129	1.226	1.208	1.203	1.203	11.904	11.864	11.863	11.863	6.291
	$\mathcal{F}_{\text{DDOPRF}}$	28.433	18.789	9.571	6.193	30.594	19.814	10.944	8.046	41.699	30.305	22.582	19.866	11.557
	Total	28.661	19.012	9.791	6.413	31.942	21.146	12.269	9.372	54.856	43.422	35.697	32.981	17.983
2^{18}	$\mathcal{F}_{\text{SSS}}$	0.583	0.572	0.564	0.555	4.808	4.808	4.806	4.798	47.241	47.206	47.169	47.166	25.166
	$\mathcal{F}_{\text{DDOPRF}}$	112.53	72.825	43.312	27.744	118.595	67.362	42.581	29.178	167.971	125.233	91.082	79.919	46.682
	Total	113.688	73.492	44.251	28.674	123.981	72.748	47.965	34.556	217.773	175.160	140.818	129.647	71.848
2^{20}	$\mathcal{F}_{\text{SSS}}$	2.354	2.317	1.289	1.277	19.950	19.936	19.916	19.916	195.471	195.437	195.402	195.398	101.664
	$\mathcal{F}_{\text{DDOPRF}}$	473.682	241.866	160.942	105.223	468.108	306.401	158.614	109.413	670.684	514.244	392.388	315.238	185.680
	Total	477.694	245.841	163.877	108.147	490.865	329.156	181.456	132.258	869.091	712.617	591.726	513.372	287.344

Table 2: Running time (in seconds) and communication cost (in MB) of online and offline in Π_{mcPSI} for different set sizes ($n \in \{2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$) in localhost setting.

Set size	Time		Comm.	
	offline	online	offline	online
2^8	3.126	0.139	7.250	0.081
2^{10}	5.799	0.461	29.711	0.312
2^{12}	10.745	1.793	117.890	1.147
2^{14}	93.290	7.126	471.781	4.519
2^{16}	1462.487	28.303	1892.081	17.983
2^{18}	25536.403	113.454	7592.139	71.848
2^{20}	398582.448	477.216	30459.672	287.334

Table 3: Running time (in seconds) and communication cost (in MB) of the online phase in Π_{mcPSI} and the semi-honest circuit-PSI protocol CGS22 [10] for different set sizes ($n \in \{2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$) in different network settings.

set size	Time LAN		Time WAN 100Mbps		Time WAN 10Mbps		Comm.	
	CGS22 [10]	Ours	CGS22 [10]	Ours	CGS22 [10]	Ours	CGS22 [10]	Ours
2^8	0.22	0.15	1.78	0.17	3.67	0.31	0.38	0.08
2^{10}	0.33	0.48	2.53	0.53	5.59	0.96	1.53	0.31
2^{12}	0.55	1.86	4.09	2.02	8.01	3.63	6.10	1.15
2^{14}	0.70	7.13	7.43	7.88	15.97	13.81	24.33	4.52
2^{16}	1.65	28.66	16.49	31.94	35.66	54.86	99.48	17.98
2^{18}	6.07	113.69	42.85	123.98	87.13	217.77	397.65	71.85
2^{20}	24.78	477.69	162.61	490.87	342.17	869.09	1700	287.34

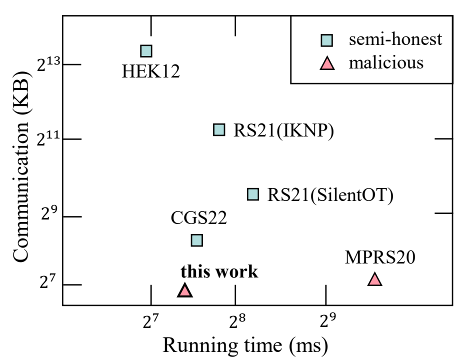

Figure 9: Time and communication for circuit-PSI protocols on $n = 256$ and LAN network setting.

in Table 1. We also break down the online running time and communication cost for the subprotocols in Π_{mcPSI}, which include the secret shared shuffling protocol $\mathcal{F}_{\text{SSS}}$ in Phase 1, as well as the DDOPRF protocol $\mathcal{F}_{\text{DDOPRF}}$ with fair comparison described in Phases 2 and 3. The total time reported encompasses the entire online time, including system initialization and offline cache loading time.

In this table, we evaluate the online costs for input set sizes of $\{2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$ and different numbers of threads (Thread = 1, 2, 4, 8). We can conclude that the running time of $\mathcal{F}_{\text{SSS}}$ is not significantly affected by the number of threads. Because our implementation employs multiple threads for various operations in

Table 4: Running time (in seconds) and communication cost (in MB) of the online phase in Π_{mcPSI} compared to the online phase of the malicious PSI-SUM protocol MPRS20 [40], for varying set sizes ($n \in 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}$) in a localhost setting on Google Cloud Platform.

Set size	**Time** (Online)		**Comm.** (Online)	
	MPRS20 [40]	Ours	MPRS20 [40]	Ours
2^{12}	141	8.51	1.89	1.15
2^{14}	553	37.50	7.56	4.52
2^{16}	2215	154.19	28.29	17.98
2^{18}	8860	629.58	111.22	71.85
2^{20}	35583	2020.43	436.72	287.34

Table 5: Running time (in seconds) and communication cost (in MB) of Π_{mcPSI} compared to the semi-honest PSI protocol HEK12 [23] for varying set sizes ($n \in 2^{12}, 2^{14}, 2^{16}, 2^{18}$) in a localhost setting on Google Cloud Platform.

Set size	**Time** (Total)		**Comm.** (Total)	
	HEK12 [23]	Ours	HEK12 [23]	Ours
2^{12}	0.61	51.67	209.92	119.04
2^{14}	3.16	357.68	1002.62	476.30
2^{16}	12.65	4019.52	4941.82	1910.06
2^{18}	58.83	62344.25	24139.47	8043.99

$\mathcal{F}_{\text{ABB}}$ and $\mathcal{F}_{\text{ABB+}}$. The online computation of $\mathcal{F}_{\text{SSS}}$ involves only one addition and permutation operation, which is not computationally intensive for the CPU. Details of the specific operations in $\mathcal{F}_{\text{SSS}}$ can be found in Appendix B and [57]. Besides, to guarantee *fairness* in Π_{DDOPRF}, we need to change the arithmetic sharings of a key to boolean sharings via bit decomposition protocol. In our implementation, we use the $0/1$ arithmetic sharings to substitute the boolean sharings to avoid a complex implementation.

For different thread numbers, we can conclude that when the input set is relatively small ($\leq 2^8$), varying the number of threads does not significantly enhance efficiency. However, for larger input sets, increasing the number of threads considerably reduces time consumption. For Π_{DDOPRF} in Π_{mcPSI}, the malicious OTs protocol from [55] is used as a primitive in Π_{DDOPRF}. We note that all OTs execution and the MAC check can be done in a batch in one round. Additionally, we adopt the method for generating MACs in SPDZ as described in [61].

We evaluate the runtime and communication costs for both offline and online phases across different set sizes, as shown in Table 2. The offline process involves generating large quantities of Beaver triples and correlated random values, which are essential for enabling secret sharing, shuffling, and the DDOPRF protocol. Computation in the offline phase is independent of the input sets. However, the main cost during the offline phase arises from the shuffle process. The results indicate that the offline communication cost is $O(n)$ and the computation cost is $O(n^2)$, while the online communication and computation costs scale linearly with the size of the input sets.

Table 6: The online running time (in seconds) and communication cost (in MB) of malicious sorting protocol in EMP-ag2pc for different set sizes ($n \in \{2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}\}$) in localhost setting.

Set size	2^8	2^{10}	2^{12}	2^{14}	2^{16}
Time (Online)	0.060	0.457	3.79	17.47	98.71
Comm. (Total)	48.47	294.22	1662.75	9745.52	57113

5.3 Performance Comparisons

In this section, we compare Π_{mcPSI} with the presentative two-party PSI-related protocols [10, 23, 40].

Comparison with CGS22 [10]. Compared to the state-of-the-art semi-honest circuit-PSI protocol [10] (IKNP-style OT based), Table 3 shows that Π_{mcPSI} achieves better online efficiency for smaller input set sizes in the WAN setting with higher latency. The highlighted cells represent the best efficiency under different settings. Although Π_{mcPSI} introduces some additional costs at certain levels compared to [10], it remains competitive in terms of communication costs and runtime efficiency on WAN networks. Specifically, the communication cost of Π_{mcPSI} is around 5x less than [10], and achieves better computation cost for the small input set size (2^8) in the LAN setting.

Comparison with MPRS20 [40]. To fairly compare with MPRS20 [40], we also conduct the experiments on a virtual machine on the Google Cloud Platform equipped with the same condition. Specifically, MPRS20 [40] focuses on PSI-SUM computation, and the results in Table 4 show the online computation performance of MPRS20. Thus, we consider the online computation of the same function $f = \sum \mathbf{x} \cap \mathbf{y}$, which aims to compute the sum of the items in the intersection set. Both our work and MPRS20 focus on the malicious setting and online computation time. Compared to MPRS20, the running time of Π_{mcPSI} is approximately 16 times faster, as shown in Table 4, and our communication cost is about 1.5 times lower than that of MPRS20.

Comparison with HEK12 [23]. HEK12 [23] provides a semi-honest PSI based on pure garbled circuit methods without using hashing-based optimizations. In detail, a circuit will first sort two input sets and then reduce the duplicated items, and shuffle all items to output the items in the intersection set. The comparison experiments with HEK12 were also conducted on the Google Cloud Platform. Additionally, we integrated the offline and online computation times. As shown in Table 5, compared to the semi-honest circuit-PSI protocol [23], Π_{mcPSI} (including both offline and online times) has a running time that is, on average, 2^8 times slower. Then, we compare the communication costs of Π_{mcPSI} with [23] and Π_{mcPSI}

Table 7: Additional running time (in seconds) and communication cost (in MB) of PSI-Sum and PSI-Variance in LAN and WAN network settings, with an intersection set size of 100.

intersection set size	**Time LAN**		**Time WAN 100Mbps**		**Comm. (MB)**	
	Sum	Variance	Sum	Variance	Sum	Variance
100	≈ 0	+0.003	≈ 0	+0.004	0	+0.035

Table 8: Theoretical comparison of different PSI-related protocols, using the computational security $\kappa = 128$, the length of each item ℓ, and the statistical security $\lambda = 40$. n is the size of the input set, and we consider the sizes of two input sets to be equal. χ represents the upper bound on the number of cycles in a cuckoo graph of PaXoS.

Protocol	Aims	Fairness	Comm. asymptotic	Assumption	Malicious
HEK12 [23]	PSI	✗	$(2\ell n log(2n) + ((3n-1)\ell - n) + 2\ell n log^2(2\hat{n}))\phi$	CDH	✗
CGS22 [10]	PSI, circuit-PSI	✗	$0.25\ell n\lambda + 0.5\ell\lambda + 8\ell n$		✗
RS21 [54]	PSI, circuit-PSI	✗	$(\lambda + 2log(n))n + 2^{17}\kappa n^{0.05} + \kappa n + \lvert\text{baseOT}\rvert$	LPN+CDH	✗
RS21 [54]	PSI	✗	$3\kappa n + 2^{17}\kappa n^{0.05} + \lvert\text{baseOT}\rvert$	LPN+CDH	✓
RR22 [51]	PSI	✗	$2.3\kappa n + 2^{14.5}\kappa + \lvert\text{baseOT}\rvert$		✓
PaXoS [46]	PSI	✗	$2\kappa n + \ell(2.4n + 2\lambda + \chi) + \lambda(2.4n + 2\ell) + \lvert\text{baseOT}\rvert$	CDH	✓
MPRS20 [40]	PSI-Sum	✗	$O(n)$		✓
Ours	PSI, circuit-PSI	✓	$(2\lambda + 2)\ell n$	q-DDHI	✓

features a good communication performance, which are around 3 times smaller than [23].

Intuitively, to enhance semi-honest GC components to their malicious versions and obtain a maliciously secure circuit-PSI protocol, it is crucial for P_0 and P_1 to input and sort their sets in the malicious setting. Therefore, we evaluate the malicious sorting using the state-of-the-art maliciously secure GC library in the EMP-toolkit [59]. It will transform a sorting program into a circuit file, which can then be executed by an authenticated garbling method [60] to achieve malicious security. Table 6 shows the online running time and the communication cost of the maliciously secure sorting protocol. The complexity of both running time and communication cost is $O(nlogn)$. This results in a higher computation and communication overhead, making it more impractical compared to Π_{mcPSI}. However, the detailed construction of a purely GC-based malicious circuit-PSI protocol is beyond the scope of this paper. We leave this exploration for future work.

5.4 Scalability and Practicality

In this section, we will extend our circuit-PSI protocol to achieve some specific computations: PSI-Sum and PSI-Variance, which are important metrics in database operations. We evaluate the consumption for computing a function on the intersection set, and the functions f are sum and variance computations. As shown in Table 7, we separately evaluate the running time and communication cost of secure sum and variance computation in a secret-shared way based on SPDZ. It means after parties obtain shares $\langle \mathbf{x} \cap \mathbf{y} \rangle$, the consumption of they send the shares to a function. When the intersection set size is 100, the additional running time of the sum computation is nearly zero, and no extra communication cost since the sum computation is all local. The variance computation involves sum, multiply, and square root operations. Adding the running time and communication cost of sum and variance to those of Π_{PSI} results in the total running time and communication cost for PSI-Sum and PSI-Variance. The cost of function computation is linear with the intersection set size.

5.5 Theoretical Analysis

In Table 8, we provide a thorough theoretical comparison of our protocols with other semi-honest and malicious PSI-related protocols in different security settings. HEK12 [23] is a pure circuit-based PSI protocol in a semi-honest setting. Its communication cost is linear with the number of used gates. Therefore, we use ϕ to represent the communication cost for one non-free gate. Next, we show the main communication cost of CGS22 [10], a private membership test protocol.

As for malicious protocols, although MPRS20 [40] achieves linear communication complexity, it relies on too many asymmetric operations, including the Pedersen commitments and ElGamal encryptions, resulting in low efficiency (shown in Tables 4), making it challenging to measure with uniform parameters. In Π_{mcPSI}, we consider $\ell = 2\kappa$ since the field size of the DDOPRF is 2κ. More specifically, in the shuffle protocol, the offline communication cost is $O(\kappa n \log n + \kappa n)$.l The online communication cost of the shuffle process is $2\ell n$ as it needs to shuffle the input items and their MAC values. Then, in the DDOPRF protocol, the communication cost is $2\lambda\ell n$. In Table 8, we focus solely on the online linear communication cost of Π_{mcPSI}. In conclusion, Π_{mcPSI} is not only the first malicious circuit-PSI protocol but also achieves fairness and better efficiency.

6 Related Work

In this section, we will present relevant PSI works and discuss challenges in extending the existing PSI solutions to the malicious Circuit-PSI problem.

Malicious PSI. The most common method to achieve malicious PSI protocols [7, 18, 46, 51, 53, 54, 66] is utilizing the Oblivious Key-Value Store (OKVS) structures. Dong et al. [18] propose a semi-honest two-party PSI protocol based on Garbled Bloom Filters (GBF), one of OKVS structures. Based on Dong et al.'s [18] work, Rindal et al. [53] convert it to a malicious setting via a cut-and-choose technique. Next, Pinkas et al. [46] propose the first two-party PSI protocol with linear communication and security in the malicious setting. In their work, an OKVS structure based on cuckoo hashing is proposed and achieves a constant rate. Then, Rindal et al. [54] optimize the OKVS structure by combining VOLE, and achieve the performance improvement compared to Pinkas et al. [46]. Furthermore, Bienstock et al. [7] present an RB-OKVS scheme, which achieves the best encoding rate (0.97) and the best efficiency compared to priors OKVS structures. Plugging the RB-OKVS scheme into the PSI implementation [51], it obtains the most efficient malicious PSI to date. However, if we trivially extend malicious PSI to Circuit-PSI, that is to say, the parties send the PSI results to the following functionality, it will reveal $\{\mathbf{x} \cap \mathbf{y}\}$ to parties. To extend two-party PSI in malicious settings to the multiparty case, a zero-sharing technique is proposed [24, 31]. If one item is in the intersection set, then all parties will get secret-shared values that sum up to zero. Ben et al. [4] propose the first concretely efficient maliciously secure multiparty PSI protocol, where combining results from semi-honest multiparty PSI [24] and malicious two-party PSI [53]. Afterward, Nevo et al. [42] also based on the zero-sharing idea and propose a more efficient multiparty PSI in a malicious setting. The concrete complexity of their protocol is larger only by a small factor (2-3) than the size of the input set (n), while [4] is around 200×. However, Nevo et al. [42] introduce an aided party (pivot) in their model to assist the maliciously secure computation.

Semi-honest Circuit-PSI. The functionality of Circuit-PSI is to securely compute arbitrary functions over the intersection set. Huang et al. [23] present the notion of Circuit-PSI, and use a generic garbled-circuit approach [65] to achieve it. It achieves $O(nlogn)$ complexity with small constant factors, where n is the size of the input set. Afterward, Pinkas et al. [47] propose a Circuit-PSI protocol based on OPRF and reduce the communication complexity to $O(n)$. However, the computational complexity of their protocol is super-linear $O(n(logn)^2)$. While this bottleneck is solved in [10], Chandran et al. propose a concretely efficient Circuit-PSI protocol with linear complexity. Both protocols [10, 47] are based on the IKNP-style OT extension protocol [2], and the communication cost of those can be improved by utilizing the Vector Oblivious Linear Evaluation (VOLE) style OT extensions as discussed in [51, 54]. However, it will involve more computation cost, and the concrete performance depends on the network parameters [10]. Specifically, those Circuit-PSI protocols [10, 47, 51, 54] are secure in the semi-honest setting.

The core idea of the OPRF-based Circuit-PSI protocols [10, 47] is similar to PSI protocols [30, 49] except that the intersection results are secret-shared between the parties, which can be used as inputs of the following circuit computation. In more detail, for a value v_0 (resp. v_1) in the input set of P_0 (resp. P_1), P_0 (resp. P_1) will gets an output random value a_0 (resp. a_1). If $v_0 = v_1$, and v_0 is in the intersection set, then we can get $1 = a_0 \oplus a_1$. Otherwise, $0 = a_0 \oplus a_1$. As we can see, if we adopt those Circuit-PSI protocols to the malicious setting, the main challenge is how to guarantee the consistency of secret-shared results and inputs of the following circuits. Since the intersection set results are secret-shared between two parties, a malicious party might tamper with the secret-shared results and send the tampered results to the following circuits. Then, the correctness of those Circuit-PSI results cannot be guaranteed. Next, the protocols in [33] also achieve malicious Circuit-PSI with the help of an untrusted third party, and it also reveals the size of the intersection set to the untrusted party.

Oblivious Pseudorandom Function. OPRF is an essential primitive for building PSI-related protocols. The frequently used method for building OPRF is based on OT extension protocol [2]. As we mentioned above, a line of Circuit-PSI works [12, 30, 31, 45, 47] based on OT extension are subject to the consistency issue when trivially converting those protocols into malicious Circuit-PSI. Dodis-Yampolskiy PRF (DY-PRF) [16] is another method to construct OPRF [9]. The DY-PRF-based OPRF can be combined with cryptographic commitment protocols and serve as "glue" for other parts of a Circuit-PSI protocol to solve the consistency issue in the malicious setting. Miao et al. [40] combine a DY-PRF-based OPRF protocol with a Pedersen commitment protocol and achieve a PSI-sum protocol in the malicious setting. Their DY-PRF is built by an additively homomorphic encryption scheme. Therefore, their protocol is subject to the efficiency limitation of HE. In our paper, we take advantage of the secret sharing and authentication methods [29] to avoid the costliest part of their [40] protocol.

7 Conclusion

In this work, we focus on designing the first maliciously secure circuit-PSI protocol. Specifically, we develop a distributed dual-key oblivious PRF, which is integral to our circuit-PSI protocol design. Additionally, we introduce several gadgets to enhance our protocol's efficiency, including a batched consistency check. Our approach also incorporates two-party fairness into the circuit-PSI protocol.

In terms of efficiency, our protocol is competitive with existing semi-honest circuit-PSI protocols, effectively filling a gap in the PSI field. Furthermore, our protocol employs the primitives of the SPDZ framework in a black-box manner, allowing for the substitution of these primitives with more efficient alternatives.

Future work. A drawback of our protocol is that the size of the intersection is disclosed, a vulnerability that can be mitigated using differential privacy methods. Besides, we believe that enhancing the semi-honest GC-based PSI to a pure GC-based malicious circuit-PSI protocol is not that apparent. We can also leave this as a future work.

Besides, many recent works have focused on how to extend the two-party PSI-related protocols to the multi-party setting. If we trivially extend our malicious circuit-PSI protocol into a multi-party setting, it needs a multi-party secret-shared shuffle protocol, and the DDOPRF protocol will be executed between n parties. It may result in low efficiency and non-linear communication complexity. Therefore, it remains to design a maliciously secure circuit-PSI protocol with better complexity.

Acknowledgments

This research is supported in part by the National Research Foundation, Singapore, under its National Satellite of Excellence Programme "Design Science and Technology for Secure Critical Infrastructure: Phase II" (Award No: NRF-NCR25-NSOE05-0001) and the National Natural Science Foundation of China under Grant 62302118, Grant 62261160651. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] Aydin Abadi and Steven J Murdoch. 2023. Earn While You Reveal: Private Set Intersection that Rewards Participants. *arXiv preprint arXiv:2301.03889* (2023).

[2] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 535–548.

[3] Amos Beimel. 2011. Secret-sharing schemes: A survey. In *International conference on coding and cryptology*. Springer, 11–46.

[4] Aner Ben-Efraim, Olga Nissenbaum, Eran Omri, and Anat Paskin-Cherniavsky. 2022. Psimple: Practical multiparty maliciously-secure private set intersection. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. 1098–1112.

[5] Michael Ben-Or, Oded Goldreich, Silvio Micali, and Ronald L Rivest. 1990. A fair protocol for signing contracts. *IEEE Transactions on Information Theory* 36, 1 (1990), 40–46.

[6] Václad E Beneš. 1964. Optimal rearrangeable multistage connecting networks. *Bell system technical journal* 43, 4 (1964), 1641–1656.

[7] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. 2023. Near-Optimal Oblivious Key-Value Stores for Efficient PSI, PSU and Volume-Hiding Multi-Maps. Cryptology ePrint Archive, Paper 2023/903. https://eprint.iacr.org/2023/903 https://eprint.iacr.org/2023/903.

[8] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. 2021. Sublinear GMW-style compiler for MPC with preprocessing. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II 41*. Springer, 457–485.

[9] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. 2022. SoK: Oblivious pseudorandom functions. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 625–646.

[10] Nishanth Chandran, Divya Gupta, and Akash Shah. 2022. Circuit-PSI with linear complexity via relaxed batch OPPRF. *Proceedings on Privacy Enhancing Technologies* (2022).

[11] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-shared shuffle. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26*. Springer, 342–372.

[12] Melissa Chase and Peihan Miao. 2020. Private set intersection in the internet setting from lightweight oblivious PRF. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III 40*. Springer, 34–63.

[13] Qian Chen, Zilong Wang, Hongbo Wang, and Xiaodong Lin. 2023. FedDual: Pair-Wise Gossip Helps Federated Learning in Large Decentralized Networks. *IEEE Transactions on Information Forensics and Security* 18 (2023), 335–350. https://doi.org/10.1109/TIFS.2022.3222935

[14] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*. Springer, 643–662.

[15] Sumit Kumar Debnath and Ratna Dutta. 2016. New realizations of efficient and secure private set intersection protocols preserving fairness. In *International Conference on Information Security and Cryptology*. Springer, 254–284.

[16] Yevgeniy Dodis and Aleksandr Yampolskiy. 2005. A verifiable random function with short proofs and keys. In *International Workshop on Public Key Cryptography*. Springer, 416–431.

[17] Changyu Dong, Liqun Chen, Jan Camenisch, and Giovanni Russello. 2013. Fair private set intersection with a semi-trusted arbiter. In *Data and Applications Security and Privacy XXVII: 27th Annual IFIP WG 11.3 Conference, DBSec 2013, Newark, NJ, USA, July 15-17, 2013. Proceedings 27*. Springer, 128–144.

[18] Changyu Dong, Liqun Chen, and Zikai Wen. 2013. When private set intersection meets big data: an efficient and scalable protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 789–800.

[19] Thai Duong, Duong Hieu Phan, and Ni Trieu. 2020. Catalic: Delegated PSI cardinality with applications to contact tracing. In *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26*. Springer, 870–899.

[20] Saba Eskandarian and Dan Boneh. 2021. Clarion: Anonymous communication from multiparty shuffling protocols. *Cryptology ePrint Archive* (2021).

[21] Oded Goldreich and Yair Oren. 1994. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology* 7, 1 (1994), 1–32.

[22] S Dov Gordon, Carmit Hazay, Jonathan Katz, and Yehuda Lindell. 2011. Complete fairness in secure two-party computation. *Journal of the ACM (JACM)* 58, 6 (2011), 1–37.

[23] Yan Huang, David Evans, and Jonathan Katz. 2012. Private set intersection: Are garbled circuits better than custom protocols?. In *NDSS*.

[24] Roi Inbar, Eran Omri, and Benny Pinkas. 2018. Efficient scalable multiparty private set-intersection via garbled bloom filters. In *International Conference on Security and Cryptography for Networks*. Springer, 235–252.

[25] Ferhat Karakoç and Alptekin Küpçü. 2020. Linear complexity private set intersection for secure two-party protocols. In *International Conference on Cryptology and Network Security*. Springer, 409–429.

[26] Ferhat Karakoç and Alptekin Küpçü. 2023. Enabling Two-Party Secure Computation on Set Intersection. *Cryptology ePrint Archive* (2023).

[27] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590.

[28] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 830–842.

[29] Marcel Keller, Valerio Pastro, and Dragos Rotaru. 2018. Overdrive: making SPDZ great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 158–189.

[30] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. 2016. Efficient batched oblivious PRF with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 818–829.

[31] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. 2017. Practical multi-party private set intersection from symmetric-key techniques. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1257–1272.

[32] Peeter Laud. 2021. Linear-time oblivious permutations for SPDZ. In *Cryptology and Network Security: 20th International Conference, CANS 2021, Vienna, Austria, December 13-15, 2021, Proceedings 20*. Springer, 245–252.

[33] Phi Hung Le, Samuel Ranellucci, and S Dov Gordon. 2019. Two-party private set intersection with an untrusted third party. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2403–2420.

[34] Yuxian Li, Jian Weng, Junzuo Lai, Yingjiu Li, Jianfei Sun, Jiahe Wu, Ming Li, Pengfei Wu, and Robert H. Deng. 2024. AuditPCH: Auditable Payment Channel Hub with Privacy Protection. *IEEE Transactions on Information Forensics and Security* (2024), 1–1. https://doi.org/10.1109/TIFS.2024.3515820

[35] Yehuda Lindell. 2016. Fast cut-and-choose-based protocols for malicious and covert adversaries. *Journal of Cryptology* 29, 2 (2016), 456–490.

[36] Yehuda Lindell. 2017. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich* (2017), 277–346.

[37] Yang Liu, Yan Kang, Tianyuan Zou, Yanhong Pu, Yuanqin He, Xiaozhou Ye, Ye Ouyang, Ya-Qin Zhang, and Qiang Yang. 2022. Vertical federated learning. *arXiv preprint arXiv:2211.12814* (2022).

[38] Qiyao Luo, Yilei Wang, and Ke Yi. 2022. Frequency Estimation in the Shuffle Model with Almost a Single Message. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2219–2232.

[39] Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. 2023. {SecretFlow-SPU}: A Performant and {User-Friendly} Framework for {Privacy-Preserving} Machine Learning. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 17–33.

[40] Peihan Miao, Sarvar Patel, Mariana Raykova, Karn Seth, and Moti Yung. 2020. Two-sided malicious security for private intersection-sum with cardinality. In *Annual International Cryptology Conference*. Springer, 3–33.

[41] Shigeo Mitsunari, Ryuichi Sakai, and Masao Kasahara. 2002. A new traitor tracing. *IEICE transactions on fundamentals of electronics, communications and computer sciences* 85, 2 (2002), 481–484.

[42] Ofri Nevo, Ni Trieu, and Avishay Yanai. 2021. Simple, fast malicious multiparty private set intersection. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1151–1165.

[43] Emmanuela Orsini. 2021. Efficient, actively secure MPC with a dishonest majority: a survey. In *Arithmetic of Finite Fields: 8th International Workshop, WAIFI 2020, Rennes, France, July 6–8, 2020, Revised Selected and Invited Papers 8*. Springer, 42–71.

[44] Benny Pinkas. 2003. Fair secure two-party computation. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 87–105.

[45] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2019. SpOT-light: lightweight private set intersection from sparse OT extension. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer, 401–431.

[46] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2020. PSI from PaXoS: fast, malicious private set intersection. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 739–767.

[47] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. 2019. Efficient circuit-based PSI with linear communication. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part III 38*. Springer, 122–153.

[48] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. 2018. Efficient circuit-based PSI via cuckoo hashing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 125–157.

[49] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2018. Scalable private set intersection based on OT extension. *ACM Transactions on Privacy and Security*

(TOPS) 21, 2 (2018), 1–35.
[50] Pille Pullonen et al. 2013. Actively secure two-party computation: Efficient beaver triple generation. *Instructor* (2013).
[51] Srinivasan Raghuraman and Peter Rindal. 2022. Blazing fast PSI from improved OKVS and subfield VOLE. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2505–2517.
[52] Priyanga Rajaram, Mark Goh, and Jianying Zhou. 2022. Guidelines for cyber risk management in shipboard operational technology systems. In *Journal of Physics: Conference Series*, Vol. 2311. IOP Publishing, 012002.
[53] Peter Rindal and Mike Rosulek. 2017. Improved private set intersection against malicious adversaries. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 235–259.
[54] Peter Rindal and Phillipp Schoppmann. 2021. VOLE-PSI: fast OPRF and circuit-PSI from vector-OLE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 901–930.
[55] Lawrence Roy. 2022. SoftSpokenOT: Communication–Computation Tradeoffs in OT Extension. *Cryptology ePrint Archive* (2022).
[56] Abhi Shelat and Chih-Hao Shen. 2011. Two-output secure computation with malicious adversaries. In *Advances in Cryptology–EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings 30*. Springer, 386–405.
[57] Xiangfu Song, Dong Yin, Jianli Bai, Changyu Dong, and Ee-Chien Chang. 2023. Secret-shared shuffle with malicious security. *Cryptology ePrint Archive* (2023).
[58] Abraham Waksman. 1968. A permutation network. *Journal of the ACM (JACM)* 15, 1 (1968), 159–163.
[59] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.
[60] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Authenticated garbling and efficient maliciously secure two-party computation. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 21–37.
[61] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. 2021. Wolverine: fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1074–1091.
[62] Yaxi Yang, Yao Tong, Jian Weng, Yufeng Yi, Yandong Zheng, Leo Yu Zhang, and Rongxing Lu. 2022. PriRanGe: Privacy-Preserving Range-Constrained Intersection Query Over Genomic Data. *IEEE Transactions on Cloud Computing* (2022).
[63] Yaxi Yang, Jian Weng, Yufeng Yi, Changyu Dong, Leo Yu Zhang, and Jianying Zhou. 2023. Predicate Private Set Intersection with Linear Complexity. In *International Conference on Applied Cryptography and Network Security*. Springer, 143–166.
[64] Zeyu Yang, Liang He, Hua Yu, Chengcheng Zhao, Peng Cheng, and Jiming Chen. 2022. Reverse Engineering Physical Semantics of PLC Program Variables Using Control Invariants. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*. 548–562.
[65] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets extended abstract. In *27th FOCS*. 162–167.
[66] Yongjun Zhao and Sherman SM Chow. 2017. Are you the one to share? Secret transfer with access structure. *Proceedings on Privacy Enhancing Technologies* (2017).

A Arithmetic Black-box Functionality

The detailed implementation of the functionality $\mathcal{F}_{\text{ABB}}$ is as follows.
SPDZ-style preprocessing. In the preprocessing phase, the SPDZ-style protocol will use oblivious transfer as an underlying technique for generating triples $\mathsf{RandomMul}()$ [28].
RandomMul: The pre-processing of random multiplication $\mathsf{RandomMul}()$ is to generate multiplication triples. The triple generation protocol is as follows. For two parties P_0 and P_1, each party samples $\mathbf{a}^{(0/1)}, b^{(0/1)}$, which are randomly sampled from a finite field $\mathbb{F}$. Both parties call the Random OT protocol $\mathcal{F}_{\text{ROT}}^{k\tau,k}$, where each party inputs $\mathbf{a}^{(0/1)}$ in bit form, that is, $(a_1^{(0/1)}, ..., a_{\tau k}^{(0/1)}) = \mathbf{g}^{-1}(a^{(0/1)})$, where $\mathbf{g} = (1, 2, 2^2, ..., 2^{k-1})$, integer parameter $\tau \geq 3$, k is the length of the generated triple value. Then they do the following:

1. P_1 receives $q_{0,h}^{(1,0)}, q_{1,h}^{(1,0)}$, and P_0 receives $s_h^{(0,1)} = q_{a_h^{(0)},h}^{(1)}$, for $h = 1, ..., \tau k$;
2. P_1 sends $d_h^{(1,0)} = q_{0,h}^{(1,0)} - q_{1,h}^{(1,0)} + b^{(1)}$;
3. P_0 sets $t_h^{(0,1)} = s_h^{(0,1)} + a^{(0)} \cdot d_h^{(1,0)} = q_{0,h}^{(1,0)} + a_h^{(0)} \cdot b^{(j)}$, for $h \in [1, \tau k]$. Then sets $q_h^{(1,0)} = q_{0,h}^{(1,0)}$.
4. Two parties split $(t_1^{(0,1)}, ..., t_{\tau k}^{(0,1)})$ and $(q_1^{(1,0)}, ..., q_{\tau k}^{(1,0)})$ into τ vectors of k components, denoted as $(\mathbf{t}_1, ..., \mathbf{t}_\tau)$ and $(\mathbf{q}_1, ..., \mathbf{q}_\tau)$.
5. P_0 sets $\mathbf{c}_{0,1}^{(0)} = (\mathbf{g} \cdot \mathbf{t}_1, ..., \mathbf{g} \cdot \mathbf{t}_\tau)$;
6. P_1 sets $\mathbf{c}_{0,1}^{(1)} = (\mathbf{g} \cdot \mathbf{q}_1, ..., \mathbf{g} \cdot \mathbf{q}_\tau)$.

Next, each party can locally compute $\mathbf{c}^{(0/1)} = \mathbf{a}^{(0/1)} \cdot b^{(0/1)} + \sum(\mathbf{c}_{(0,1)}^{(0/1)} + \mathbf{c}_{(1,0)}^{(0/1)})$. After each party sample a random vector $\mathbf{r}$ over a finite field, each party will sets $a^{(0,1)} = \mathbf{a}^{(0/1)} \cdot \mathbf{r}$ and $c^{(0,1)} = \mathbf{c}^{(0/1)} \cdot \mathbf{r}$. Then parties can get a valid triple (a, b, c).
SPDZ-style online evaluation. In the online phase, the SPDZ-style protocol includes the following commands.
Input: the input command takes an input x and outputs an ASS value to each party $\langle x \rangle \leftarrow \mathsf{Input}(x)$: The parties generate an ASS sharing $\langle r \rangle \leftarrow \mathcal{F}_{\text{ABB}}.\mathsf{Rand}()$, and open the value r to the party who owns the input value x. So the party will compute $\epsilon = x - r$ and broadcast ϵ. Then, all parties compute $\langle x \rangle = \langle r \rangle + \epsilon$.
Mul: On input $(\llbracket x \rrbracket, \llbracket y \rrbracket)$ from parties, the parties will take one multiplication triple $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$ and compute $\llbracket e \rrbracket = \llbracket x \rrbracket - \llbracket a \rrbracket$ and $\llbracket f \rrbracket = \llbracket y \rrbracket - \llbracket b \rrbracket$. Then, the parties compute $\llbracket c \rrbracket + f \cdot \llbracket a \rrbracket + e \cdot \llbracket b \rrbracket + e \cdot f$, which equals to $\llbracket x \cdot y \rrbracket$.
Open: on input (Open, $\langle x \rangle$) from each party, each party broadcasts $\langle x \rangle$ and recovers $x = \sum_{i \in [n]} \llbracket x \rrbracket_i$. Moreover, all parties need to run a MAC check for the opened values x in case the corrupted party opens incorrect values. The parties perform $\llbracket d \rrbracket = \llbracket \gamma(x) \rrbracket - x \cdot \llbracket \xi \rrbracket$ and check whether d equals to 0 and aborts if not. The above example is for a single evaluation. However, in implementation, the parties will perform a batch of MAC checks for better efficiency. For batch MAC check, all parties input a set of shared items $\{\langle x_1 \rangle, \ldots, \langle x_t \rangle\}$. Then, they use $\mathcal{F}_{\text{ABB}}.\mathsf{Rand}()$ to sample a vector of secret-shared random values $\{\llbracket r_1 \rrbracket, \ldots, \llbracket r_t \rrbracket\}$, and compute $\tilde{x} = \sum_{j=1}^{t} \llbracket r_j \rrbracket \cdot \llbracket x_j \rrbracket$. Next, each party computes $\llbracket \sigma \rrbracket = \sum_{j=1}^{t} \llbracket r_j \rrbracket \cdot \llbracket \gamma(x_j) \rrbracket - \tilde{x} \cdot \xi$. Therefore, parties can perform a batched MAC check by measuring whether σ equals 0.

B Secrect Shared Shuffle

The semi-honest CGP SSS protocol [11]. The CGP shuffle protocol relies on a specified correlation called *oblivious punctured matrix* (OPM). In an n-dimention OPM, a sender holds a matrix M of $n \times n$, while the receiver holds a permutation $\pi \in \mathrm{S}_n$ and a punctured matrix $\widetilde{M}$ of M, where the receiver doesn't know $M[i, \pi(i)]$ for all $i \in [n]$. From the OPM correlation, the parties can produce a correlation called *shuffle tuple*. In particular, the sender compute two n-dimension vectors $(\mathbf{a}, \mathbf{b})$ such that $\mathbf{a}_i = \sum_j M_{j,i}, \mathbf{b}_i = \sum_j M_{i,j}$ for all $i \in [n]$, and the receiver computes a n-dimension vector $\mathbf{\Delta}$ such that $\mathbf{\Delta}_i = \sum_{j \neq i} \widetilde{M}_{j,\pi(i)} - \sum_{j \neq \pi(i)} \widetilde{M}_{i,j}$.

Those *shuffle tuples* can be generated in the offline phase, and the online process of the CGP shuffling protocol is as follows. Suppose P_1 is the sender and P_0 is the receiver. Using a shuffle tuple corresponding to a permutation π, the parties can shuffle a secret-shared vector $\langle x \rangle$ as follows: P_1 sends $\delta \leftarrow \llbracket \mathbf{x} \rrbracket_1 - \mathbf{a}$ to P_0. P_0 sets $\llbracket \mathbf{y} \rrbracket_0 \leftarrow \pi(\llbracket \mathbf{x} \rrbracket_0 + \delta) + \mathbf{\Delta}$ and P_1 sets $\llbracket \mathbf{y} \rrbracket_1 \leftarrow \mathbf{b}$. Clearly, $\llbracket \mathbf{y} \rrbracket_0 + \llbracket \mathbf{y} \rrbracket_1 = \pi(\llbracket \mathbf{x} \rrbracket_0 + \llbracket \mathbf{x} \rrbracket_1 - \mathbf{a}) + \pi(\mathbf{a}) - \mathbf{b} + \mathbf{b} = \pi(\mathbf{x})$. The prior shuffling hides the underlying permutation π from P_1.

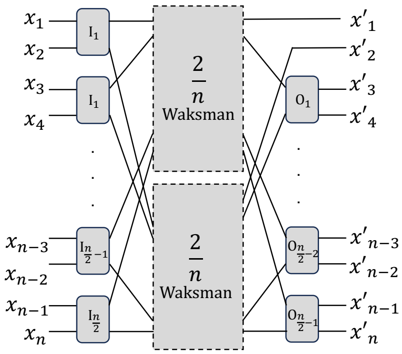

Figure 10: Waksman Network with n inputs and outputs.

During this process, instead of directly applying the permutation π to achieve SSS, the authors split the permutation π into smaller disjoint permutations via Benes permutation network [6] to improve the performance. For n elements $x_1, ..., x_n$, the Benes network will be split into two permutations, and each permutation acts on $n/2$ elements. Every wire represents whether the element is swapped or not. Then, for each $2/n$ Benes network, it will recursively call a Benes network with half the inputs and half the outputs. The Benes network for n elements contains $2\log n - 1$ layers, and each layer contains $n/2$ 2−element swappers.

Therefore, the parties run another shuffling using another shuffle tuple (corresponding to another permutation ρ known to P_1) with roles reversed. In this manner, no party learns the underlying composed permutation.

The malicious-secure SSS protocol [57]. Besides the semi-honest secure shuffle protocol, some malicious secure shuffle protocols [20, 32] are proposed. Those protocols design malicious CGP-style SSS protocol over ASS. Similarly, for an authenticated vector sharings $\langle \mathbf{x} \rangle$, the parties will perform the CGP-style shuffle protocol to get a shuffle tuple $((\pi, \delta), (\mathbf{a}, \mathbf{b}))$, where (π, δ) is for a receiver P_0, and $(\mathbf{a}, \mathbf{b})$ is for a sender P_1. Then, parties will use their ASS sharings $\langle \mathbf{x} \rangle$ to do the CGP-style shuffling process as we mentioned before. P_1 sends $\delta \leftarrow \langle \mathbf{x} \rangle - \mathbf{a}$ to P_0, The difference is that the parties can perform a MAC check to detect errors at the end of the protocol. So the checks can ensure data integrity and correct shuffling. However, those protocols are subjected to the *selective failure attack* as depicted in [57]. More concretely, a malicious sender P_1 may add errors to δ when P_1 is expected to send $\delta = \langle \mathbf{x} \rangle - \mathbf{a}$ to P_0. Instead of sending the correct message δ to P_0, P_1 will change one element in δ. Then, P_1 guesses where the changed element has been permuted to. According to the post-execution check result, P_1 can learn whether the guess is correct or not. More details about *selective failure attack* are presented in Appendix. Therefore, the authors [57] propose a malicious secure SSS protocol that is also resistant to the selective failure attack. They not only propose a correlation check to defeat an incorrect correlation attack but also design a leakage-reduction mechanism to remove possible leakage to defend against the selective failure attack. Then, all the techniques used in their protocol are combined with authenticated secret sharing to formalize a malicious secure secret-shared shuffle protocol.

Concrete Optimizations. In $\Pi_{\text{m-PSI}}$, we need to use a malicious-secure SSS protocol [57] as a building block. This work also inherits the technical components of CGP protocol, and it also utilizes the Benes network [6] as a permutation structure. Based on their work, we substitute the Benes network with the Waksman network [58]. As shown in Fig. 10, the Waksman network [58] is a realization of a permutation network using exactly $n\log n - n + 1$ 2-element swappers when n is a power of 2. The Waksman network is recursive and is built by two $n/2$−input Waksman networks. Compared to the Benes network, the Waksman network achieves better trade-offs between communication and computation costs.

C Security Proof

In this section, we give the ideal world definition of our proposed protocols and depict simulation-based proofs.

C.1 DDOPRF

Functionality $\mathcal{F}_{\text{DDOPRF}}$

Public Parameters: a prime p, a group $\mathbb{G}$ and a generator of the group g.

Private Parameter: A PRF key $k \xleftarrow{\$} \mathbb{Z}_p$ and its related ASS sharing $\langle k \rangle$; A secondary PRF key $k_s \xleftarrow{\$} \mathbb{Z}_p$ and its ASS sharing $\langle k_s \rangle$.

DDOPRF:

1. on receiving (DDOPRF, $\langle x \rangle$) from both parties, the functionality outputs $F(k, x) = g^{\frac{1}{k+x}}$ to both parties if it does not abort. Otherwise, $\perp$ is output to both parties;
2. on receiving (DDOPRF, Dual-key, $\langle x \rangle$) from both parties, the functionality outputs $F(k, k_s, x) = g^{\frac{k_s}{k+x}}$ to both parties if it does not abort. Otherwise, $\perp$ is output to both parties.

Figure 11: Ideal functionality of DDOPRF.

Theorem 3. *In the $\mathcal{F}_{\text{ABB}}, \mathcal{F}_{\text{ABB}^+}$-hybrid model, the protocol Π_{DDOPRF} implements $\mathcal{F}_{\text{DDOPRF}}$ correctly and securely against malicious adversary.*

Proof. We can construct an ideal world simulator S_{DDOPRF} as the following:

1. S_{DDOPRF} is given the public parameters $p, \mathbb{G}, g$. S_{DDOPRF} has access to $\mathcal{F}_{\text{ABB}}$ and $\mathcal{F}_{\text{ABB}^+}$, chooses $k', k'_s, x' \xleftarrow{\$} \mathbb{Z}_p$ and records $\langle k' \rangle$, $\langle k'_s \rangle$ and $\langle x' \rangle$;
2. S_{DDOPRF} invokes the adversary A with $p, \mathbb{G}, g$, and $\langle x' \rangle$;
3. S_{DDOPRF} receives a $\mathcal{F}_{\text{ABB}}$.Rand() call from the adversary, generates a random number $r \in \mathbb{Z}_p$, and returns a share of r to the adversary, and S_{DOPRF} records $\langle r \rangle$;
4. S_{DDOPRF} receives the invocations to $\mathcal{F}_{\text{ABB}}$.Mul() for computing $\langle d' \rangle = \langle r \rangle \cdot (\langle k' \rangle + \langle x' \rangle)$, return a share of d' to the adversary;
5. S_{DDOPRF} receives an invocation to $\mathcal{F}_{\text{ABB}}$.Open. S_{DDOPRF} does the MAC check of the inputs received in this step and the previous step, against stored shares $\langle d' \rangle, \langle k' \rangle, \langle x' \rangle, \langle r \rangle$. If all shares

received from the adversary are correct, send d' to the adversary; otherwise, send abort to $\mathcal{F}_{\text{DDOPRF}}$, and abort the protocol execution with the adversary;

6. S_{DDOPRF} receives an invocation to $\mathcal{F}_{\text{ABB}}$.Mul() for computing $\langle e' \rangle \leftarrow d'^{-1} \cdot \langle r \rangle$, and S_{DDOPRF} records $\langle e' \rangle$ and returns a share of e' to the adversary; If S_{DDOPRF} receives an invocation to $\mathcal{F}_{\text{ABB}}$.Mul(), it will compute $\langle e' \rangle \leftarrow \langle e' \rangle \cdot \langle k'_s \rangle$. And S_{DDOPRF} records $\langle e' \rangle$ and returns a share to the adversary;
7. S_{DDOPRF} receives an invocation to $\mathcal{F}_{\text{ABB}^+}$.Convert() for converting $\langle e' \rangle$ to $⟪e'⟫$. S_{DDOPRF} records $⟪e'⟫$ and returns a share to the adversary;
8. S_{DDOPRF} receives an invocation to $\mathcal{F}_{\text{ABB}^+}$.Open() for openning $⟪e'⟫$. S_{DDOPRF} does the MAC check on all inputs received from the adversary since step 6. If the check fails, S_{DDOPRF} sends abort to $\mathcal{F}_{\text{DDOPRF}}$ and abort the protocol execution with the adversary. Otherwise, S_{DDOPRF} sends its input $\langle x' \rangle$ to $\mathcal{F}_{\text{DDOPRF}}$, receives the output $g^{e'}$ from $\mathcal{F}_{\text{DDOPRF}}$ then passes it to the adversary.

As we can see, in this simulation: the distribution of the view of the adversary in a real execution is the same as that in the simulation because the shares are information-theoretically secure; the simulation aborts whenever an error is detected in a real execution; and the distribution of the joint output in the simulation is the same as that in a real execution. Therefore the simulation is indistinguishable from a real execution, thus $\mathcal{F}_{\text{DDOPRF}}$ can be security implemented by Π_{DDOPRF}.

C.2 $\mathcal{F}_{\text{mcPSI}}$

THEOREM 4. *In the $\mathcal{F}_{\text{ABB}}$,$\mathcal{F}_{\text{ABB}^+}$,$\mathcal{F}_{\text{SSS}}$,$\mathcal{F}_{\text{DDOPRF}}$-hybrid model, the protocol Π_{mcPSI} implements $\mathcal{F}_{\text{mcPSI}}$ correctly and securely against malicious adversary.*

Proof. We construct an ideal world simulator S_{PSI} as the following: S_{PSI} is given public parameters p, $\mathbb{G}$, g. S_{PSI} has access to $\mathcal{F}_{\text{ABB}}$, $\mathcal{F}_{\text{ABB}^+}$, $\mathcal{F}_{\text{SSS}}$, $\mathcal{F}_{\text{DDOPRF}}$.

1. S_{PSI} invokes the real-world adversary A with a simulated input $\mathbf{x_i}'$.
2. For $i \in [1, n]$: S_{PSI} receives a $\mathcal{F}_{\text{ABB}}$.Input(x'_i) call from the adversary A, then generate ASS sharing $\langle x'_i \rangle \leftarrow \mathcal{F}_{\text{ABB}}$.Input($x'_i$) and sends P_0's shares to A;
3. S_{PSI} sends (mc-PSI, $\mathbf{x}'$) to the ideal functionality $\mathcal{F}_{\text{mc-PSI}}$.
4. S_{PSI} selects a random $\mathbf{y}' = \{y'_1, \cdots, y'_n\}$, generates ASS sharings $\langle \mathbf{y}' \rangle$.
5. S_{PSI} receives invocations to $\mathcal{F}_{\text{SSS}}$ to shuffles the shares $\langle \mathbf{x}'' \rangle \leftarrow \mathcal{F}_{\text{SSS}}(\langle \mathbf{x}' \rangle)$, $\langle \mathbf{y}'' \rangle \leftarrow \mathcal{F}_{\text{SSS}}(\langle \mathbf{y}' \rangle)$, then sends P_0's shares to A;
6. S_{PSI} receives a $\mathcal{F}_{\text{ABB}^+}$.Rand() call from A, generate k, k_s and returns sharings of PRF keys $\langle k \rangle$ and $\langle k_s \rangle$ to A.
7. S_{PSI} receives an invocation to $\mathcal{F}_{\text{DDOPRF}}$ to computes $Z_x = \{F(k, \mathbf{x}''_i)\}_{i \in [1,n]}$ and $Z_y = \{F(k, k_s, \mathbf{y}''_i)\}_{i \in [1,n]}$. S_{PSI} computes the multiplicative sharing of the results and returns P_0's shares to A.
8. S_{PSI} receives an invocation to $\mathcal{F}_{\text{bDEC}}$ to get the shares of sequence $\langle b_0 \cdot 2^{\lambda-1} \rangle \cdots \langle b_{\lambda-1} \cdot 2^0 \rangle$, where b_i is the i-th bit of k_s (left most first) and returns P_0's shares to A.
9. S_{PSI} receives the shares $\langle b_0 \cdot 2^{\lambda-1} \rangle \cdots \langle b_{\lambda-1} \cdot 2^0 \rangle$ one by one from A and send P_1's shares to A, and they can reconstruct k_s locally;
10. If A aborts at any time in the previous steps, send abort to $\mathcal{F}_{\text{mcPSI}}$, otherwise sends cardinality to $\mathcal{F}_{\text{mcPSI}}$ and receives back $|\mathbf{x}' \cap \mathbf{y}'|$.
11. S_{PSI} opens Z_x and Z_y so that A can find matching elements in these two sets. S_{PSI} and A also have the sharing of $\mathbf{R}_{X_b}$, $\mathbf{R}_{Y_b}$ which corresponds to the set intersection.
12. If A aborts at any time in the previous step, send abort to $\mathcal{F}_{\text{mcPSI}}$, otherwise if A invokes $\mathcal{F}_{\text{mcPSI}}$, S_{PSI} sends compute to $\mathcal{F}_{\text{mcPSI}}$ and receives back $f(\mathbf{x}' \cap \mathbf{y})$, which is then forward to A.

Also, in this simulation, the adversary can only see the shares of the vectors generated during the simulation, which are information-theoretically secure. Thus, the distribution of the joint output in the simulation is the same as that in a real execution, and Π_{mcPSI} is secure under this situation. This situation when constructing the simulator for Π_{mcPSI} with corrupted P_1 is quite similar to the corrupted P_0 except that P_0 is acted by the simulator and P_1 is acted by the adversary.