Leopard: A High-Performance Framework for Updatable Private Set Intersection

Xiaowei Li
Shanghai Jiao Tong University
happy_lxw@sjtu.edu.cn

Shi-Feng Sun✉
Shanghai Jiao Tong University
shifeng.sun@sjtu.edu.cn

Yanxue Jia
Illinois Institute of Technology
jiayanxue820@gmail.com

Hao Li
Shanghai Jiao Tong University
hao.li@sjtu.edu.cn

Dawu Gu
Shanghai Jiao Tong University
dwgu@sjtu.edu.cn

Abstract

Updatable Private Set Intersection (UPSI) extends conventional PSI protocols to dynamic scenarios, where input sets evolve over time. A straightforward approach is to rerun the entire PSI protocol after each update, incurring redundant computational and communication overhead. Although several UPSI schemes have been proposed to enable updates without full protocol re-execution, existing constructions remain impractical.

Building on the high efficiency of unbalanced PSI designs, this work introduces Leopard—a general framework that leverages unbalanced PSI to realize UPSI without complete re-execution, while handling both insertions and deletions. Leopard is compatible with any unbalanced PSI protocol and achieves further performance gains by reusing internal data structures across updates. Specifically, we observe that certain data structures in unbalanced PSI can be incrementally updated instead of fully rebuilt. We provide an FHE-based instantiation of Leopard that demonstrates substantial performance improvements through such data structure reuse.

Through comprehensive experiments, we evaluate Leopard's performance. Compared to Badrinarayanan et al. (ASIACRYPT 2024) and Alborch et al. (ACNS 2026), our instantiations reduce computational cost by up to 111.28× and at least 1.48×, and communication cost by up to 118.97× and up to 522.01×, respectively.

Keywords

Updatable Private Set Intersection (UPSI), General Framework, Data Structure Reuse.

1 Introduction

Private set intersection (PSI) enables two or more parties, each holding a private dataset, to jointly compute the intersection of their sets without revealing any additional information. PSI has found practical applications in various domains, such as collaborative medical data analysis [27] and measurement of advertising conversion rates in commercial environments [19].

The first PSI protocol was introduced by Freedman et al. [14] in 2004. Since then, numerous protocols based on diverse cryptographic primitives have been developed, including: Diffie-Hellman (DH)-based [18], circuit-based [31], oblivious pseudo-random function (OPRF)-based [6, 7, 21, 23, 29, 30, 35], oblivious transfer (OT)-based [20, 28, 33], fully homomorphic encryption (FHE)-based [8–10, 12, 26, 37], oblivious key-value store (OKVS)-based [5, 15, 17, 34], and vector oblivious linear evaluation (VOLE)-based [13, 34, 36] approaches. In addition, a range of PSI variants has been explored to meet specialized requirements, such as asymmetric PSI [8–10, 12, 21, 22, 26, 35], PSI-SUM [11, 18], circuit-PSI [6, 17, 31, 32, 36, 37], among others.

Almost all previous PSI works focus on the *static* setting, where the private sets of participants remain unchanged. In practice, however, many scenarios involve *dynamic* data, where set elements are updated over time. In such cases, participants have to repeatedly invoke conventional PSI protocols to recompute the intersection after each update. This repeated execution incurs performance overhead, particularly when *updates are minor yet frequent*.

To address the need for efficient intersection updates in the presence of small changes to the sets, Badrinarayanan et al. [4] introduced the concept of *updatable PSI with set intersection reuse*. Later, Agarwal et al. [1] proposed a general framework for UPSI that supports both element insertion and deletion, and Ling et al. [24] presented an efficient instantiation of such a scheme. Liu et al. [25] introduced a formalization of updatable 'affine set encoding' along with three symmetric-key instantiations. However, these constructions only support *two-sided* output, in which both parties learn the result, limiting their applicability in certain real-world settings—such as Microsoft Edge's password breach alert system—where only one party should receive the output. The *one-sided* output functionality is strictly stronger than its two-sided counterpart, as it grants output access to a single designated participant.

Badrinarayanan et al. [3] presented the first concrete construction for one-sided output UPSI supporting both element insertions and deletions. While their protocol achieves worst-case asymptotic computational and communication complexities that scale linearly with the update set sizes and sublinearly with the original set sizes, its practical performance remains insufficient for real-world deployment. Subsequently, Alborch et al. [2] proposed a general framework under this setting. However, their instantiation relies on unbalanced circuit-PSI and combine-and-permute sub-protocols, which inevitably incur substantial overhead. Consequently, the problem of *how to standardize the designation of two-party, one-sided output UPSI with support for both element insertions and deletions, and to instantiate it with high practical efficiency*, remains an open question.

1.1 Our Contributions

In this work, we present several positive advancements in the design of one-sided output UPSI, encompassing both general and concrete constructions. Our main contributions are summarized as follows:

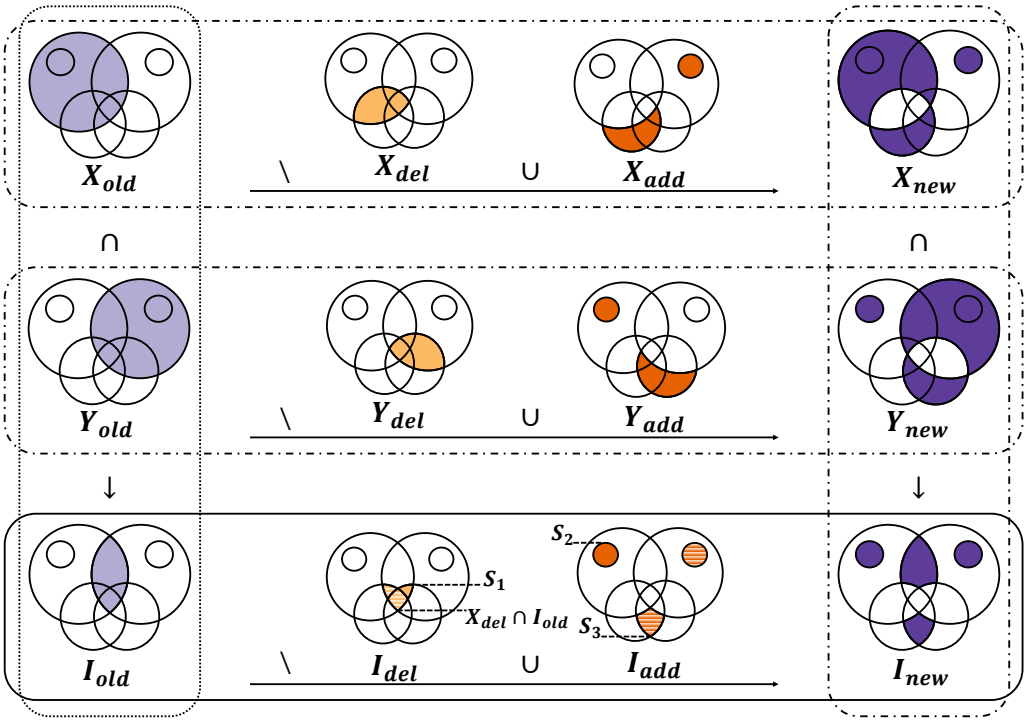

Figure 1: Updated Set Intersection Reconstruction of One-Sided Output UPSI Illustrated by Venn Diagrams. We consider the two-party one-sided output UPSI supporting both element insertions and deletions. The original set intersection is calculated by $I_{old} = X_{old} \cap Y_{old}$ through a conventional PSI protocol invocation. After set updates, $X_{new} = (X_{old} \setminus X_{del}) \cup X_{add}$ and $Y_{new} = (Y_{old} \setminus Y_{del}) \cup Y_{add}$, with proper assumptions $X_{del} \subseteq X_{old}$, $X_{add} \cap X_{old} = \emptyset$, $Y_{del} \subseteq Y_{old}$, and $Y_{add} \cap Y_{old} = \emptyset$. By traditional method, we re-invocate the conventional PSI protocol to reach the updated set intersection $I_{new} = X_{new} \cap Y_{new}$. While for updatable PSI, we reuse I_{old} and compute $I_{new} = (I_{old} \setminus I_{del}) \cup I_{add}$, with $I_{del} \subseteq I_{old}$ and $I_{add} \cap I_{old} = \emptyset$. To compute I_{del} and I_{add}, we further divide them into $X_{del} \cap I_{old}$, S_1, and S_2, S_3 by the principles illustrated in Section 3.1. Thus the computation of updated set intersection after reconstruction is $I_{new} = [I_{old} \setminus (X_{del} \cup S_1)] \cup (S_2 \cup S_3)$.

We propose Leopard, the *first* general framework for one-sided output UPSI that supports full dynamic operations (insertions and deletions) and is built entirely upon efficient (reversed) asymmetric PSI protocols in a *black-box fashion*.

We present an FHE-based instantiation under the Leopard framework, which features *data structure reuse* (obviating full protocol re-execution) and readily generalizes to *multi-update* settings.

We evaluate Leopard through comprehensive experiments. The results show that it outperforms Badrinarayanan et al. [3] by *up to 111.28×* in computation and *up to 118.97×* in communication, while against Alborch et al. [2], it achieves *at least 1.48×* computation gain and *up to 522.01×* communication gain.

1.2 Technical Overview

As illustrated by Venn diagrams in Figure 1, for the two-party one-sided output setting, denote the party who receives the output as the receiver $\mathcal{R}$, and the other party as the sender $\mathcal{S}$. Let X_{old} and Y_{old} represent the original private sets held by $\mathcal{R}$ and $\mathcal{S}$, respectively. In a subsequent update, let X_{del} and Y_{del} denote the elements to be deleted, and X_{add} and Y_{add} denote the elements to be inserted by $\mathcal{R}$ and $\mathcal{S}$, respectively. Without loss of generality, we assume deletions only affect elements already present in the original sets, and insertions only introduce new elements; that is: $X_{del} \subseteq X_{old}$, $X_{add} \cap X_{old} = \emptyset$, $Y_{del} \subseteq Y_{old}$, and $Y_{add} \cap Y_{old} = \emptyset$. Consequently, no element is simultaneously deleted and inserted in the same update, so: $X_{del} \cap X_{add} = \emptyset$ and $Y_{del} \cap Y_{add} = \emptyset$. The updated sets are then defined as: $X_{new} = (X_{old} \setminus X_{del}) \cup X_{add}$ and $Y_{new} = (Y_{old} \setminus Y_{del}) \cup Y_{add}$.

To compute the updated set intersection I_{new} efficiently via UPSI, the previous intersection $I_{old} = X_{old} \cap Y_{old}$ is reused. Only the differences between I_{new} and I_{old}—namely the elements removed from I_{old} (denoted I_{del}) and the elements newly inserted to I_{old} (denoted I_{add})—need to be computed. Thus, I_{new} can be reconstructed as: $I_{new} = (I_{old} \setminus I_{del}) \cup I_{add}$.

The core challenge is how to compute I_{del} and I_{add} securely and efficiently. Our central idea is to decompose I_{del} and I_{add} into subsets and compute them using well-established private set operations—such as local set operations, PSI, and PSU protocols. *Our set-partitioning principle ensures that no information beyond what $\mathcal{R}$ can infer from I_{new} is leaked, while minimizing the number*

of invoked set operations and reducing the use of interactive protocols for better performance.

For I_{del}, which consists of elements removed due to deletions from X_{del} and Y_{del}, we have: $I_{del} = (X_{del} \cap Y_{old}) \cup (X_{old} \cap Y_{del})$. Observe that $X_{del} \cap Y_{old} = X_{del} \cap I_{old}$, which can be computed *locally* by $\mathcal{R}$. However, revealing $X_{old} \cap Y_{del}$ directly would leak extra information—specifically, $(X_{old} \cap Y_{del}) \cap (X_{del} \cap Y_{old})$. To maintain security, we rewrite I_{del} as $I_{del} = (X_{del} \cap I_{old}) \cup \big((X_{old} \setminus X_{del}) \cap Y_{del}\big)$. Here, $(X_{old} \setminus X_{del}) \cap Y_{del}$ complements $X_{del} \cap Y_{old}$, and can be computed via a PSI protocol, while $X_{del} \cap I_{old}$ remains locally computable by $\mathcal{R}$.

Similarly, for I_{add}, we can express it as: $I_{add} = \big(X_{add} \cap (Y_{old} \setminus Y_{del})\big) \cup \big((X_{old} \setminus X_{del}) \cap Y_{add}\big) \cup (X_{add} \cap Y_{add})$. However, allowing $\mathcal{R}$ to distinguish whether an inserted element originates from Y_{old} or Y_{add} would violate our security principle. We therefore avoid separating $X_{add} \cap (Y_{old} \setminus Y_{del})$ and $X_{add} \cap Y_{add}$. Instead, we combine them and reformulate I_{add} as: $I_{add} = \big((X_{old} \setminus X_{del}) \cap Y_{add}\big) \cup (X_{add} \cap Y_{new})$, where the two subsets are disjoint: $\big((X_{old} \setminus X_{del}) \cap Y_{add}\big) \cap (X_{add} \cap Y_{new}) = \emptyset$. Thus, $\mathcal{R}$ can obtain these two subsets by executing two separate PSI protocols.

Based on the above analysis, we reconstruct I_{new} as $I_{new} = [I_{old} \setminus (X_{del} \cup S_1)] \cup (S_2 \cup S_3)$, with $S_1 = (X_{old} \setminus X_{del}) \cap Y_{del}$, $S_2 = (X_{old} \setminus X_{del}) \cap Y_{add}$, and $S_3 = X_{add} \cap Y_{new}$. In the context of UPSI, set updates are typically much smaller than the original sets. Therefore, computing S_3 requires one asymmetric PSI protocol, while S_1 and S_2 each call an asymmetric PSI protocol in which the party holding the larger set is the output receiver—we refer to such protocols as *reversed* asymmetric PSI. After receiving S_1, S_2, and S_3, $\mathcal{R}$ can locally compute the updated intersection I_{new}.

Following the updated set intersection reconstruction method described above, we propose `Leopard`, a general framework for one-sided output UPSI supporting both element insertions and deletions, with its overall structure illustrated in Figure 2. Further details are provided in Section 3.

`Leopard` can be instantiated with *arbitrary* asymmetric PSI and reversed asymmetric PSI protocols in a *black-box manner*. Asymmetric PSI protocols can be broadly categorized into three families: DH/OPRF-based, OKVS-based, and FHE-based. We observe that for FHE-based asymmetric PSI, when the larger input set undergoes only minor changes between successive invocations, the associated data structure for that set can be reused with small updates corresponding to the changed elements, rather than being fully rebuilt. This property aligns well with the typical usage patterns in UPSI, where updates are small and frequent. Accordingly, we instantiate `Leopard` using FHE-based asymmetric PSI, and construct its reversed counterpart trivially from the same FHE primitives, with *data structure reuse* applied to both variants. A symmetric-key version is also briefly mentioned. Further details on these instantiations are deferred to Section 4.

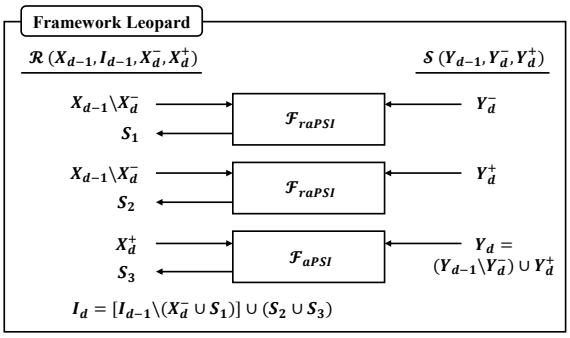

Figure 2: Overall Structure of Framework `Leopard`. Since the update set sizes are smaller than the original ones by orders of magnitude, S_3 corresponds to one invocation of an asymmetric PSI protocol, while S_1 and S_2 two invocations of the reversed asymmetric PSI protocol—all realized in an arbitrary black-box manner. Detailed introduction of `Leopard`, our one-sided output UPSI general framework please refer to Section 3.1.

1.3 Related Works

In this section, we briefly review several representative UPSI works relevant to `Leopard`. A concise summary is provided in Table 1.

In 2022, Badrinarayanan et al. [4] first introduced the notion of set intersection reuse in UPSI. They proposed a one-sided output addition-only UPSI protocol based on the Diffie–Hellman assumption, Oblivious Random Access Machine (ORAM), and Additive Homomorphic Encryption (AHE), achieving amortized asymptotic computational and communication complexities that scale linearly with the size of the update sets and logarithmically with the size of the larger original sets. They also presented a two-sided output weak deletion UPSI protocol, which only supports deletion of the entire set that was inserted t days earlier.

Subsequently, Badrinarayanan et al. [3] further improved their design [4] and proposed a one-sided output UPSI protocol supporting both element insertions and deletions, with extensions to functionalities including PSI-CA, PSI-SUM, and circuit-PSI. Their protocol attains worst-case asymptotic complexities that grow linearly with the update set size and logarithmically with the original set size, representing the state-of-the-art in UPSI with strong asymptotic guarantees. However, its practical performance is limited by the heavy overhead of ORAM, the use of non-optimized AHE, and multiple auxiliary hiding mechanisms.

Agarwal et al. [1], a contemporary work with [3], proposed the first general framework for two-sided output UPSI that supports both element insertions and deletions, along with a construction paradigm based on ORAM, structured encryption (StE), and private set union (PSU). Ling et al. [24] later refined the framework of [1] to be compatible with arbitrary asymmetric PSI and conventional PSU sub-protocols, and provided a concrete instantiation with strong practical performance.

Liu et al. [25] aimed to implement UPSI using purely symmetric-key techniques. Their approach followed the technical route of constructing PSI via VOLE-to-OPRF, and they defined the required data structure as 'Affine Set Encoding' (ASE). They implemented an updatable version of ASE using OKVS, hash tables, and Path ORAM, respectively, and realized three UPSI instantiations from it: one addition-only and two supporting both insertion and deletion,

Table 1: Summary of Leopard and Related UPSI Works. We briefly summary Leopard and prior UPSI works from the following perspectives: support of addition and deletion of elements, one-sided or two-sided output, concrete construction or general framework, and computational and communication complexities. For asymptotic complexities: n denotes the set size of small update sets; N denotes the set size of large original sets; t denotes the time window size in the weak deletion scheme of [4], where each party updates its set by inserting new elements and removing those inserted t days ago, so that its private set always consists of the insertion batches from the most recent t updates; for '$O(.); O(.)$', the first $O(.)$ denotes the amortized complexity over many epochs, and the second $O(.)$ denotes the worst-case complexity; $\tilde{O}(.)$ omits polylogarithmic factors; for UPSI supporting both addition and deletion, we consider the scenario which allows arbitrary additions and deletions of the same element; the upper row of [25] represents their 'okvs' protocol, and the lower row for their 'tree' and 'cuckoo' ones; for [40], the stated asymptotic complexities refer solely to the online phase, whereas the offline phase incurs a cost of $O(N^2)$; for Leopard, the upper row lists the asymptotic complexities of our symmetric-key based instantiation, while the lower row corresponds to the public-key based one.

Work	Addition / Deletion	Output Mode	Construction Mode	Comp. Complexity	Comm. Complexity
[BMX22] [4]	Addition-Only	Two-Sided	Concrete Construction	$O(n)$	$O(n)$
		One-Sided	Concrete Construction	$O(n \cdot \log N); O(n \cdot N)$	$O(n \cdot \log N); O(n \cdot N)$
	Weak Deletion	Two-Sided	General Framework & Concrete Construction	$O(n \cdot t)$	$O(n \cdot t)$
[BMS+24] [3]	Addition-Only	One-Sided	Concrete Construction	$O(n \cdot \log N)$	$O(n \cdot \log N)$
	Addition & Deletion	One-Sided	Concrete Construction	$O(n \cdot \log^2 N)$	$O(n \cdot \log^2 N)$
[ACG+26] [1]	Addition & Deletion	Two-Sided	General Framework	$O(n \cdot \log N)$	$O(n \cdot \log N)$
[LTSQ26] [24]	Addition & Deletion	Two-Sided	General Framework & Concrete Construction	$O(n \cdot \log N + \sqrt{N})$	$O(n + \log^2 N)$
[LMR+26] [25]	Addition-Only	Two-Sided	Concrete Construction	$O(n \cdot \log N); O(n \cdot N)$	$O(n \cdot \log N); O(n \cdot N)$
	Addition & Deletion	Two-Sided	Concrete Construction	$O(n \cdot \log N)$	$O(n \cdot N)$
[ACF+26] [2]	Addition & Deletion	One-Sided	General Framework & Concrete Construction	$O(n \cdot \log n + N)$	$O(n \cdot \log n + N)$
[ZWS+26] [40]	Addition-Only	One-Sided	General Framework & Concrete Construction	$O(n \cdot N)$	$O(n \cdot N)$
Leopard	Addition & Deletion	One-Sided	General Framework & Concrete Construction	$O(N)$	$O(N \cdot \log N)$
				$\tilde{O}(n \cdot N)$	$O(n \cdot \log N)$

all in the two-sided output setting. In addition, they introduced the security notion of 'adaptively chosen inputs', which was explicitly considered in their schemes.

Alborch et al. [2] identified several potential privacy leakages in UPSI executions. They further proposed a general framework for one-sided output UPSI that supports both insertion and deletion, with its security guarantees built upon unbalanced circuit-PSI and combine-and-permute protocols. They also extended the framework and construction to support the cardinality functionality.

Zhang et al. [40] delivered a novel PSI protocol based on distributed multi-point functions (DMPF). They further formalized a general framework for one-sided output addition-only UPSI and instantiated it using their DMPF-based PSI construction with a trusted third party.

2 Preliminaries

In this section, we introduce the notations, the security model for semi-honest adversaries, the definitions of private set intersection (PSI) and its variants, as well as relevant cryptographic primitives used in this paper.

2.1 Notations

PPT stands for probabilistic polynomial time. The symbol $\overset{c}{\approx}$ denotes computational indistinguishability between the left-hand side and right-hand side. For a set S, $|S|$ represents its cardinality. $\complement_U S$ represents the complement of set S with respect to the universal set U.

2.2 Security Model

We adopt the standard security notion in the presence of semi-honest adversaries [16].

Definition 1. *Let* $\mathtt{View}^{\Pi}_{\mathcal{R}}(X, Y)$ *and* $\mathtt{View}^{\Pi}_{\mathcal{S}}(X, Y)$ *be the views of receiver* $\mathcal{R}$ *and sender* $\mathcal{S}$*, respectively, and let* $\mathtt{Out}^{\Pi}_{\mathcal{R}}(X, Y)$ *and* $\mathtt{Out}^{\Pi}_{\mathcal{S}}(X, Y)$ *be their corresponding outputs. A protocol* Π *securely computes an ideal functionality* $\mathcal{F}$ *with outputs* $\mathcal{F}_{\mathcal{R}}$ *and* $\mathcal{F}_{\mathcal{S}}$ *for* $\mathcal{R}$ *and* $\mathcal{S}$ *in the semi-honest model if, for every PPT adversary* $\mathcal{A}$*, there exist PPT simulators* $\mathtt{Sim}_{\mathcal{R}}$ *and* $\mathtt{Sim}_{\mathcal{S}}$ *such that for all inputs* X *from* $\mathcal{R}$ *and* Y *from* $\mathcal{S}$*, the following holds:*

$$\{\mathtt{View}^{\Pi}_{\mathcal{R}}(X, Y),\ \mathtt{Out}^{\Pi}_{\mathcal{S}}(X, Y)\} \overset{c}{\approx} \{\mathtt{Sim}_{\mathcal{R}}(X, \mathcal{F}_{\mathcal{R}}(X, Y)),\ \mathcal{F}_{\mathcal{S}}(X, Y)\},$$

$$\{\mathtt{View}^{\Pi}_{\mathcal{S}}(X, Y),\ \mathtt{Out}^{\Pi}_{\mathcal{R}}(X, Y)\} \overset{c}{\approx} \{\mathtt{Sim}_{\mathcal{S}}(Y, \mathcal{F}_{\mathcal{S}}(X, Y)),\ \mathcal{F}_{\mathcal{R}}(X, Y)\}.$$

Functionality $\mathcal{F}_{\mathsf{PSI}}$

Parameters:
- The functionality interacts with two parties: the receiver $\mathcal{R}$ and the sender $\mathcal{S}$, and the simulator Sim.

Functionality:
0. Initialize an ideal state $\mathtt{state}_{\mathsf{U}} = \emptyset$ for party U where $\mathsf{U} \in \{\mathcal{R}, \mathcal{S}\}$. If U is corrupted, Sim is allowed to access U's state $\mathtt{state}_{\mathsf{U}}$.
1. Wait for input X from $\mathcal{R}$. Update $\mathtt{state}_{\mathcal{R}} = \langle X \rangle$, and send $\langle \mathtt{Request}, \mathcal{R} \rangle$ to Sim.
2. Wait for input Y from $\mathcal{S}$. Update $\mathtt{state}_{\mathcal{S}} = \langle Y \rangle$, and send $\langle \mathtt{Request}, \mathcal{S} \rangle$ to Sim.
3. The ideal functionality computes $I = X \cap Y$.
4. Upon receiving $\langle \mathtt{Response}, \mathtt{OK} \rangle$ from Sim, add $\langle I \rangle$ to $\mathtt{state}_{\mathcal{R}}$, output I to $\mathcal{R}$.

Figure 3: The Ideal Functionality of Private Set Intersection.

Functionality $\mathcal{F}_{\mathsf{aPSI}}$ ($\mathcal{F}_{\mathsf{raPSI}}$)

Parameters:
- The functionality interacts with two parties: the receiver $\mathcal{R}$ and the sender $\mathcal{S}$, and the simulator Sim.
- The set size for $\mathcal{R}$ is n_r, and the set size for $\mathcal{S}$ is n_s, with $n_r \ll n_s$ ($n_r \gg n_s$).

Functionality:
0. Initialize an ideal state $\mathtt{state}_{\mathsf{U}} = \emptyset$ for party U where $\mathsf{U} \in \{\mathcal{R}, \mathcal{S}\}$. If U is corrupted, Sim is allowed to access U's state $\mathtt{state}_{\mathsf{U}}$.
1. Wait for input X from $\mathcal{R}$, abort if $|X| \neq n_r$. Update $\mathtt{state}_{\mathcal{R}} = \langle X \rangle$, and send $\langle \mathtt{Request}, \mathcal{R} \rangle$ to Sim.
2. Wait for input Y from $\mathcal{S}$, abort if $|Y| \neq n_s$. Update $\mathtt{state}_{\mathcal{S}} = \langle Y \rangle$, and send $\langle \mathtt{Request}, \mathcal{S} \rangle$ to Sim.
3. The ideal functionality computes $I = X \cap Y$.
4. Upon receiving $\langle \mathtt{Response}, \mathtt{OK} \rangle$ from Sim, add $\langle I \rangle$ to $\mathtt{state}_{\mathcal{R}}$, output I to $\mathcal{R}$.

Figure 4: Ideal Functionality of (Reversed) Asymmetric PSI.

2.3 PSI and Its Variants

PSI. In one-sided output model, each party inputs its private set, and only one of the two parties obtains the set intersection as output. Ideal functionality is depicted in Figure 3.

Asymmetric PSI and Reversed Asymmetric PSI. Traditional PSI typically assumes that the two parties' input sets are of similar or comparable size; we refer to such scenarios as *balanced* PSI. When the sizes differ substantially, we denote it as *unbalanced* PSI. In unbalanced settings, if the party with the smaller set receives the output, the protocol is termed *asymmetric PSI*. Conversely, if the party holding the larger set obtains the output, we introduce the notion of *reversed asymmetric PSI*. Formally, let the receiver $\mathcal{R}$ and sender $\mathcal{S}$ possess input sets of sizes n_r and n_s, respectively. The setting is: balanced if $n_r \simeq n_s$; unbalanced if $n_r \ll n_s$ or $n_r \gg n_s$, where: $n_r \ll n_s$ corresponds to the asymmetric case, and $n_r \gg n_s$ corresponds to the reversed asymmetric case. The ideal functionality for (reversed) asymmetric PSI is illustrated in Figure 4.

Functionality $\mathcal{F}_{\mathsf{UPSI}}$

Parameters:
- The functionality interacts with two parties: the receiver $\mathcal{R}$ and the sender $\mathcal{S}$, and the simulator Sim.
- Starting set size for $\mathcal{R}$ and $\mathcal{S}$ is N. The size of both the deletion set and the insertion set for $\mathcal{R}$ and $\mathcal{S}$ is bounded above by n.

Functionality:

Day 1**:**
0. Initialize an ideal state $\mathtt{state}_{\mathsf{U}} = \emptyset$ for party U where $\mathsf{U} \in \{\mathcal{R}, \mathcal{S}\}$. If U is corrupted, Sim is allowed to access U's state $\mathtt{state}_{\mathsf{U}}$.
1. Wait for input starting set X_1 from $\mathcal{R}$, abort if $|X_1| \neq N$. Update $\mathtt{state}_{\mathcal{R}} = \langle X_1 \rangle$, and send $\langle \mathtt{Request}, \mathcal{R} \rangle$ to Sim.
2. Wait for input starting set Y_1 from $\mathcal{S}$, abort if $|Y_1| \neq N$. Update $\mathtt{state}_{\mathcal{S}} = \langle Y_1 \rangle$, and send $\langle \mathtt{Request}, \mathcal{S} \rangle$ to Sim.
3. The ideal functionality computes the starting set intersection $I_1 = X_1 \cap Y_1$.
4. Upon receiving $\langle \mathtt{Response}, \mathtt{OK} \rangle$ from Sim, add $\langle I_1 \rangle$ to $\mathtt{state}_{\mathcal{R}}$, output I_1 to $\mathcal{R}$.

Day d**:**
1. Wait for inputs deletion set X_d^-, insertion set X_d^+ from $\mathcal{R}$, abort if $X_d^- \not\subseteq X_{d-1}$, $X_d^+ \cap X_{d-1} \neq \emptyset$, $|X_d^-| > n$, or $|X_d^+| > n$. Add $\langle X_d^-, X_d^+ \rangle$ to $\mathtt{state}_{\mathcal{R}}$, and send $\langle \mathtt{Request}, \mathcal{R} \rangle$ to Sim.
2. Wait for inputs deletion set Y_d^-, insertion set Y_d^+ from $\mathcal{S}$, abort if $Y_d^- \not\subseteq Y_{d-1}$, $Y_d^+ \cap Y_{d-1} \neq \emptyset$, $|Y_d^-| > n$, or $|Y_d^+| > n$. Add $\langle Y_d^-, Y_d^+ \rangle$ to $\mathtt{state}_{\mathcal{S}}$, and send $\langle \mathtt{Request}, \mathcal{S} \rangle$ to Sim.
3. The ideal functionality computes the updated private sets $X_d = (X_{d-1} \setminus X_d^-) \cup X_d^+$, $Y_d = (Y_{d-1} \setminus Y_d^-) \cup Y_d^+$, and the updated set intersection $I_d = X_d \cap Y_d$.
4. Upon receiving $\langle \mathtt{Response}, \mathtt{OK} \rangle$ from Sim, add $\langle X_d, I_d \rangle$ to $\mathtt{state}_{\mathcal{R}}$, and $\langle Y_d \rangle$ to $\mathtt{state}_{\mathcal{S}}$, output I_d to $\mathcal{R}$.

Figure 5: The Ideal Functionality of Updatable PSI.

Updatable PSI. For two-party one-sided output UPSI, the ideal functionality is shown in Figure 5. It accepts the initial and the update private sets from both parties as input, and outputs the set intersections to the receiver $\mathcal{R}$.

2.4 Cryptographic Primitives

Polynomial Interpolation. Polynomial interpolation represents the most basic form of OKVS, achieving optimal rate but exhibiting higher encoding and decoding complexity compared to modern OKVS schemes such as RB-OKVS [5] or 3H-GCT [15]. Given a set of n key-value pairs $S = \{(k_i, v_i) : i \in [n]\}$, polynomial interpolation constructs a polynomial $P(x)$ of degree $\deg(P) < n$ that satisfies $P(k_i) = v_i$ for every $i \in [n]$.

Fully Homomorphic Encryption. A Fully Homomorphic Encryption (FHE) scheme allows arbitrary arithmetic circuits to be evaluated directly on ciphertexts. Formally, an FHE scheme is defined by the following set of algorithms:

Functionality $\mathcal{F}_{\mathsf{PEQT}}$

Parameters:
- Two parties: the receiver $\mathcal{R}$ and the sender $\mathcal{S}$.

Functionality:
1. Wait for input x from $\mathcal{R}$.
2. Wait for input y from $\mathcal{S}$.
3. Generate b as 1 if $x = y$, and 0 otherwise.
4. Send $\langle \mathtt{Request} \rangle$ to the simulator $\mathtt{Sim}$.
5. Upon receiving $\langle \mathtt{Response}, \mathtt{OK} \rangle$ from $\mathtt{Sim}$, output b to $\mathcal{R}$.

Figure 6: The Ideal Functionality of PEQT.

Functionality $\mathcal{F}_{\mathsf{OT}}$

Parameters:
- Two parties: the receiver $\mathcal{R}$ and the sender $\mathcal{S}$.

Functionality:
1. Wait for input $b \in \{0, 1\}$ from $\mathcal{R}$.
2. Wait for input (x_0, x_1) from $\mathcal{S}$.
3. Send $\langle \mathtt{Request} \rangle$ to the simulator $\mathtt{Sim}$.
4. Upon receiving $\langle \mathtt{Response}, \mathtt{OK} \rangle$ from $\mathtt{Sim}$, output x_b to $\mathcal{R}$.

Figure 7: The Ideal Functionality of OT.

- $\mathtt{FHE.Setup}(1^\lambda)$: On input the statistical security parameter λ, outputs a set of encryption parameters $\mathtt{params}$.
- $\mathtt{FHE.KeyGen}(\mathtt{params})$: On input $\mathtt{params}$, outputs a secret key $\mathtt{sk}$ and a public key $\mathtt{pk}$.
- $\mathtt{FHE.Enc}(m, \mathtt{pk})$: Inputs message m and $\mathtt{pk}$, outputs ciphertext c.
- $\mathtt{FHE.Dec}(c, \mathtt{sk})$: On input a ciphertext c and $\mathtt{sk}$, outputs the corresponding message m.

Private Equality Test. A private equality test (PEQT) protocol allows two participants to test whether their respective elements are equal without revealing the elements to each other. Concretely, a receiver $\mathcal{R}$ with element x and a sender $\mathcal{S}$ with element y engage in PEQT; $\mathcal{R}$ obtains a bit b where $b = 1$ if $x = y$ and $b = 0$ otherwise. Throughout the protocol, neither party learns any additional information about the other's input. The ideal functionality of PEQT is illustrated in Figure 6.

Oblivious Transfer. In a 1-out-of-2 oblivious transfer (OT) protocol, the sender $\mathcal{S}$ inputs a pair (x_0, x_1) and the receiver $\mathcal{R}$ inputs a choice bit b. $\mathcal{R}$ learns only x_b and nothing about x_{1-b}, while $\mathcal{S}$ learns nothing about b. The ideal functionality of OT is depicted in Figure 7.

3 Leopard: A New Framework for One-Sided Output UPSI

We present Leopard, a novel general framework for one-sided output UPSI. Section 3.1 provides a detailed exposition of its construction (illustrated in Figure 8), while Section 3.2 contains formal correctness analysis and security proof of the Leopard theorem.

3.1 Construction

Venn diagrams that visually illustrate the reconstruction process of the updated set intersection for one-sided output UPSI is provided in Figure 1. Initially, the receiver $\mathcal{R}$ and the sender $\mathcal{S}$ invoke a conventional one-sided output PSI protocol on their original private sets X_{old} and Y_{old}, respectively, after which $\mathcal{R}$ obtains the set intersection $I_{old} = X_{old} \cap Y_{old}$. Throughout the following description, we assume that the input sets of both parties are of roughly the same order of magnitude.

When at least one party updates its private set, the intersection must be recomputed. Without loss of generality, we consider the case in which both participants update their sets. Let $\mathcal{R}$'s deletion and insertion sets be X_{del} and X_{add}, respectively, and let $\mathcal{S}$'s deletion and insertion sets be Y_{del} and Y_{add}. The updated sets are then $X_{new} = (X_{old} \cup X_{add}) \setminus X_{del}$ and $Y_{new} = (Y_{old} \cup Y_{add}) \setminus Y_{del}$. We assume, as is natural, that only elements present in the original set can be deleted and only elements not originally present can be inserted; i.e., $X_{del} \subseteq X_{old}$, $X_{add} \cap X_{old} = \emptyset$, $Y_{del} \subseteq Y_{old}$, and $Y_{add} \cap Y_{old} = \emptyset$. Consequently, no element can be both deleted and inserted in a single update, i.e., $X_{del} \cap X_{add} = \emptyset$ and $Y_{del} \cap Y_{add} = \emptyset$. Hence the updated sets may also be written as $X_{new} = (X_{old} \setminus X_{del}) \cup X_{add}$ and $Y_{new} = (Y_{old} \setminus Y_{del}) \cup Y_{add}$. A straightforward method to obtain the updated intersection I_{new} is to rerun a conventional PSI protocol on X_{new} and Y_{new}.

In the setting of updatable PSI that supports both element deletion and insertion, the update sets are typically much smaller than the original sets. Consequently, the differences between the updated and original sets—and between the updated and original intersections—are minor. Formally, $|X_{del}| \simeq |Y_{del}| \simeq |X_{add}| \simeq |Y_{add}| \ll |X_{old}| \simeq |Y_{old}|$, which implies $X_{new} \approx X_{old}$, $Y_{new} \approx Y_{old}$, and $I_{new} \approx I_{old}$. To improve efficiency, the existing intersection I_{old} can be *reused* to avoid redundant computation. The updated intersection is thus derived as $I_{new} = (I_{old} \setminus I_{del}) \cup I_{add}$, where I_{del} and I_{add} are, respectively, the set of elements to be deleted from and inserted to I_{old}, satisfying $I_{del} \subseteq I_{old}$, $I_{add} \cap I_{old} = \emptyset$, $I_{del} \cap I_{add} = \emptyset$.

Computing I_{del} and I_{add} must adhere to the design principles:

- **Security.** In the one-sided output model, no information beyond what $\mathcal{R}$ can legitimately infer from receiving I_{new} may be leaked.
- **Performance.** Minimize the computational operations. Local set operations (e.g., unions, intersections) are preferred over interactive protocol invocations such as PSI or PSU.

I_{del} contains elements in X_{del} and Y_{del} which are deleted from I_{old}, thus we represent I_{del} as $(X_{del} \cap I_{old}) \cup (I_{old} \cap Y_{del})$. As X_{del} and I_{old} are held by $\mathcal{R}$, $\mathcal{R}$ can learn $X_{del} \cap I_{old}$ by local computation. For the remaining part $(I_{old} \cap Y_{del}) \setminus (X_{del} \cap I_{old})$, it can be equivalently computed as $(I_{old} \setminus X_{del}) \cap Y_{del}$, which requires a single PSI invocation, with the result obtained by $\mathcal{R}$. The reconstruction of I_{del} thus proceeds as follows, consistent with our stated principles.

$$\begin{aligned} I_{del} &= (X_{del} \cap I_{old}) \cup (I_{old} \cap Y_{del}) \\ &= (X_{del} \cap I_{old}) \cup [(I_{old} \cap Y_{del}) \setminus (X_{del} \cap I_{old})] \\ &= (X_{del} \cap I_{old}) \cup [(I_{old} \setminus X_{del}) \cap Y_{del}] \end{aligned}$$

I_{add} consists of the elements newly inserted into I_{old} due to X_{add} and Y_{add}. Formally, we can write it as: $I_{add} = [X_{add} \cap (Y_{old} \setminus$

$Y_{del})] \cup [(X_{old} \setminus X_{del}) \cap Y_{add}] \cup (X_{add} \cap Y_{add})$. Note that X_{del} and Y_{del} are removed from X_{old} and Y_{old}, respectively, because any element deleted during the update cannot appear in the updated intersection. The terms $X_{add} \cap (Y_{old} \setminus Y_{del})$ and $X_{add} \cap Y_{add}$ need to be combined as $X_{add} \cap [(Y_{old} \setminus Y_{del}) \cup Y_{add}]$, as they should not be distinguished by $\mathcal{R}$ who holds the same input X_{add}. Moreover, since $(X_{old} \setminus X_{del}) \cap X_{add} = \emptyset$, the two components $(X_{old} \setminus X_{del}) \cap Y_{add}$ and $X_{add} \cap [(Y_{old} \setminus Y_{del}) \cup Y_{add}]$ are disjoint. Both can be obtained by $\mathcal{R}$ via two independent PSI invocations, which aligns with our principle. Therefore, I_{add} can be reconstructed as:

$$\begin{aligned} I_{add} &= [X_{add} \cap (Y_{old} \setminus Y_{del})] \cup [(X_{old} \setminus X_{del}) \cap Y_{add}] \\ &\quad \cup (X_{add} \cap Y_{add}) \\ &= [(X_{old} \setminus X_{del}) \cap Y_{add}] \cup \{X_{add} \cap [(Y_{old} \setminus Y_{del}) \\ &\quad \cup Y_{add}]\} \end{aligned}$$

Combining the reconstructions of I_{del} and I_{add}, we can express I_{new} as: $I_{new} = [I_{old} \setminus (X_{del} \cup S_1)] \cup (S_2 \cup S_3)$, where $S_1 = (I_{old} \setminus X_{del}) \cap Y_{del}$, $S_2 = (X_{old} \setminus X_{del}) \cap Y_{add}$, $S_3 = X_{add} \cap [(Y_{old} \setminus Y_{del}) \cup Y_{add}]$. The derivation is shown in the formulas below. Consequently, I_{new} can be obtained through three PSI invocations (to compute S_1, S_2, and S_3) followed by local operations performed by $\mathcal{R}$.

$$\begin{aligned} I_{new} &= (I_{old} \setminus I_{del}) \cup I_{add} \\ &= I_{old} \setminus \{(X_{del} \cap I_{old}) \cup [(I_{old} \setminus X_{del}) \cap Y_{del}]\} \\ &\quad \cup \{[(X_{old} \setminus X_{del}) \cap Y_{add}] \cup [X_{add} \cap \big((Y_{old} \setminus Y_{del}) \\ &\quad \cup Y_{add}\big)]\} \\ &= I_{old} \setminus (X_{del} \cap I_{old}) \setminus S_1 \cup (S_2 \cup S_3) \\ &= I_{old} \setminus (X_{del} \cup S_1) \cup (S_2 \cup S_3) \end{aligned}$$

For UPSI, the update sets are typically much smaller than the original or updated sets: $|X_{del}| \simeq |Y_{del}| \simeq |X_{add}| \simeq |Y_{add}| \ll |X_{old}| \simeq |Y_{old}| \simeq |X_{new}| \simeq |Y_{new}|$. Thus, computing S_3 requires one invocation of an asymmetric PSI protocol, while S_1 and S_2 each require a reversed asymmetric PSI protocol.

Furthermore, we note that $(I_{old} \setminus X_{del}) \cap Y_{del}$ can be rewritten as $(X_{old} \setminus X_{del}) \cap Y_{del}$ (as shown in the derivation below). This allows an optimization: since both $S_1 = (X_{old} \setminus X_{del}) \cap Y_{del}$ and $S_2 = (X_{old} \setminus X_{del}) \cap Y_{add}$ depend on the same intermediate set $X_{old} \setminus X_{del}$ held by $\mathcal{R}$, they can be computed more efficiently by *reusing* the intra-protocol operations related to $X_{old} \setminus X_{del}$, provided that the same reversed asymmetric PSI protocol is employed for both computations. This corresponds to the concept of 'data structure reuse' introduced in Section 4.

$$\begin{aligned} S_1 = (I_{old} \setminus X_{del}) \cap Y_{del} &= [(X_{old} \cap Y_{old}) \setminus X_{del}] \cap Y_{del} \\ &= (X_{old} \setminus X_{del}) \cap Y_{old} \cap Y_{del} = (X_{old} \setminus X_{del}) \cap Y_{del} \end{aligned}$$

We conceptualize each update as a distinct day and generalize the updated set intersection computation of one-sided output UPSI over multiple days. This yields our general framework for one-sided output UPSI, called `Leopard`, which is depicted in Figure 8.

Day 1 *(Initialization).* In the first day, the receiver $\mathcal{R}$ with initial set X_1 and the sender $\mathcal{S}$ with initial set Y_1 invoke the conventional PSI functionality $\mathcal{F}_{\mathsf{PSI}}$. As a result, $\mathcal{R}$ obtains the initial intersection $I_1 = X_1 \cap Y_1$.

Day d (Update, $d \geq 2$). At the beginning of day d, $\mathcal{R}$ holds its previous set X_{d-1} and the previous intersection I_{d-1}, while $\mathcal{S}$ holds its previous set Y_{d-1}. To update their sets, $\mathcal{R}$ provides a deletion set X_d^- and an insertion set X_d^+, producing the updated set $X_d = (X_{d-1} \setminus X_d^-) \cup X_d^+$. Similarly, $\mathcal{S}$ provides Y_d^- and Y_d^+, yielding $Y_d = (Y_{d-1} \setminus Y_d^-) \cup Y_d^+$. To compute the updated intersection I_d, $\mathcal{R}$ and $\mathcal{S}$ interactively obtain three intermediate intersections S_1, S_2, and S_3 through the following PSI invocations, after which $\mathcal{R}$ performs the final combination locally: $\mathcal{R}$ and $\mathcal{S}$ invoke the reversed

Framework Leopard: One-Sided Output UPSI

Public Parameters:
- Two parties, the receiver $\mathcal{R}$ and the sender $\mathcal{S}$.
- The conventional PSI functionality $\mathcal{F}_{\mathsf{PSI}}$, the asymmetric PSI functionality $\mathcal{F}_{\mathsf{aPSI}}$, and the reversed asymmetric PSI functionality $\mathcal{F}_{\mathsf{raPSI}}$.

Protocol:

Day 1**:**

Inputs:
- $\mathcal{R}$ inputs initial set X_1. $\mathcal{S}$ inputs initial set Y_1.

Public Parameters:
- The set size of initial sets X_1, Y_1.

Intersection Calculation (Output Generation):
1. $\mathcal{R}$ and $\mathcal{S}$ together invoke $\mathcal{F}_{\mathsf{PSI}}$, with $\mathcal{R}$ inputs X_1, $\mathcal{S}$ inputs Y_1, and $\mathcal{R}$ receives the initial set intersection $I_1 = X_1 \cap Y_1$ as output.

Day d**:**

Inputs:
- $\mathcal{R}$ inputs deletion set X_d^-, $X_d^- \subseteq X_{d-1}$, with elements to be deleted from X_{d-1}, and insertion set X_d^+, $X_d^+ \cap X_{d-1} = \emptyset$, with elements to be inserted to X_{d-1}. Similarly, $\mathcal{S}$ inputs its deletion set Y_d^- with $Y_d^- \subseteq Y_{d-1}$, and insertion set Y_d^+ with $Y_d^+ \cap Y_{d-1} = \emptyset$.

Public Parameters:
- The set size of deletion sets X_d^-, Y_d^-, and the set size of insertion sets X_d^+, Y_d^+.

Intermediate Intersections Calculation:
1. $\mathcal{R}$ and $\mathcal{S}$ together invoke $\mathcal{F}_{\mathsf{raPSI}}$, with $\mathcal{R}$ inputs $(X_{d-1} \setminus X_d^-)$, $\mathcal{S}$ inputs Y_d^-, and $\mathcal{R}$ receives $S_1 = (X_{d-1} \setminus X_d^-) \cap Y_d^-$ as output.
2. $\mathcal{R}$ and $\mathcal{S}$ together invoke $\mathcal{F}_{\mathsf{raPSI}}$, with $\mathcal{R}$ inputs $(X_{d-1} \setminus X_d^-)$, $\mathcal{S}$ inputs Y_d^+, and $\mathcal{R}$ receives $S_2 = (X_{d-1} \setminus X_d^-) \cap Y_d^+$ as output.
3. $\mathcal{R}$ and $\mathcal{S}$ together invoke $\mathcal{F}_{\mathsf{aPSI}}$, with $\mathcal{R}$ inputs X_d^+, $\mathcal{S}$ inputs $(Y_{d-1} \setminus Y_d^-) \cup Y_d^+$, and $\mathcal{R}$ receives $S_3 = X_d^+ \cap [(Y_{d-1} \setminus Y_d^-) \cup Y_d^+]$ as output.

Outputs Generation:
- $\mathcal{R}$ locally computes the updated set $X_d = (X_{d-1} \setminus X_d^-) \cup X_d^+$. $\mathcal{S}$ locally computes the updated set $Y_d = (Y_{d-1} \setminus Y_d^-) \cup Y_d^+$.
- $\mathcal{R}$ locally computes the updated set intersection $I_d = [I_{d-1} \setminus (X_d^- \cup S_1)] \cup (S_2 \cup S_3)$.

Figure 8: Leopard: A General Framework for One-Sided Output UPSI Supporting Both Element Insertions and Deletions.

asymmetric PSI functionality $\mathcal{F}_{\texttt{raPSI}}$ on inputs $X_{d-1} \setminus X_d^-$ (from $\mathcal{R}$) and Y_d^- (from $\mathcal{S}$), and $\mathcal{R}$ receives $S_1 = (X_{d-1} \setminus X_d^-) \cap Y_d^-$; $\mathcal{R}$ and $\mathcal{S}$ invoke the reversed asymmetric PSI functionality $\mathcal{F}_{\texttt{raPSI}}$ on inputs $X_{d-1} \setminus X_d^-$ (from $\mathcal{R}$) and Y_d^+ (from $\mathcal{S}$), and $\mathcal{R}$ receives $S_2 = (X_{d-1} \setminus X_d^-) \cap Y_d^+$; $\mathcal{R}$ and $\mathcal{S}$ invoke the asymmetric PSI functionality $\mathcal{F}_{\texttt{aPSI}}$ on inputs X_d^+ (from $\mathcal{R}$) and $(Y_{d-1} \setminus Y_d^-) \cup Y_d^+$ (from $\mathcal{S}$), and $\mathcal{R}$ receives $S_3 = X_d^+ \cap [(Y_{d-1} \setminus Y_d^-) \cup Y_d^+]$; $\mathcal{R}$ computes the updated intersection as $I_d = [I_{d-1} \setminus (X_d^- \cup S_1)] \cup (S_2 \cup S_3)$.

The overall structure of Leopard is illustrated in Figure 2. Its core consists of one asymmetric PSI and two reversed asymmetric PSI sub-protocols, which together enable efficient, secure updates of the intersection across multiple days.

3.2 Correctness and Security Proofs

Choice of PSI Protocols. For Leopard, the initial intersection I_1 can be instantiated with *any* conventional PSI protocol, S_3 with *any* asymmetric PSI protocol, and S_1, S_2 with *any* reversed asymmetric PSI protocol, as long as both participants agree on the choice [1]. We denote the underlying protocols used in Leopard as $\Pi_{\texttt{PSI}}$ (conventional PSI), $\Pi_{\texttt{aPSI}}$ (asymmetric PSI), and $\Pi_{\texttt{raPSI}_1}$, $\Pi_{\texttt{raPSI}_2}$ (two reversed asymmetric PSI instances). Formal correctness and security proofs for Leopard are presented below.

Correctness. We prove correctness by induction.

Base Case ($d = 1$). On day 1, the goal is to compute $I_1 = X_1 \cap Y_1$. By the correctness of $\Pi_{\texttt{PSI}}$, the protocol outputs exactly I_1.

Inductive Step ($d \geq 2$). Assume that by day $d-1$, $\mathcal{R}$ correctly holds $I_{d-1} = X_{d-1} \cap Y_{d-1}$. On day d, the updated sets are $X_d = (X_{d-1} \setminus X_d^-) \cup X_d^+$, and $Y_d = (Y_{d-1} \setminus Y_d^-) \cup Y_d^+$. The protocol computes the following three intermediate results via the chosen sub-protocols: $S_1 = (X_{d-1} \setminus X_d^-) \cap Y_d^-$ (via $\Pi_{\texttt{raPSI}_1}$), $S_2 = (X_{d-1} \setminus X_d^-) \cap Y_d^+$ (via $\Pi_{\texttt{raPSI}_2}$), and $S_3 = X_d^+ \cap [(Y_{d-1} \setminus Y_d^-) \cup Y_d^+]$ (via $\Pi_{\texttt{aPSI}}$). By the correctness of $\Pi_{\texttt{aPSI}}$, $\Pi_{\texttt{raPSI}_1}$, and $\Pi_{\texttt{raPSI}_2}$, these values are computed accurately. Finally, $\mathcal{R}$ outputs $I_d = [I_{d-1} \setminus (X_d^- \cup S_1)] \cup (S_2 \cup S_3)$. We now expand I_d step by step to verify its equality with the reconstructed form:

$$
\begin{aligned}
I_d &= X_d \cap Y_d = [(X_{d-1} \setminus X_d^-) \cup X_d^+] \cap [(Y_{d-1} \setminus Y_d^-) \cup Y_d^+] \\
&= \{X_d^+ \cap [(Y_{d-1} \setminus Y_d^-) \cup Y_d^+]\} \cup \{(X_{d-1} \setminus X_d^-) \cap [(Y_{d-1} \\
&\quad \setminus Y_d^-) \cup Y_d^+]\} \\
&= S_3 \cup \{[(X_{d-1} \setminus X_d^-) \cap Y_{d-1}] \setminus [(X_{d-1} \setminus X_d^-) \cap Y_d^-] \\
&\quad \cup [(X_{d-1} \setminus X_d^-) \cap Y_d^+]\} \\
&= S_3 \cup [(X_{d-1} \cap Y_{d-1}) \setminus X_d^- \setminus S_1 \cup S_2] \\
&= [I_{d-1} \setminus (X_d^- \cup S_1)] \cup (S_2 \cup S_3)
\end{aligned}
$$

This concludes the proof of correctness.

Security. We now prove the semi-honest security of $\Pi_{\texttt{UPSI}}$ in the $\{\mathcal{F}_{\texttt{aPSI}}, \mathcal{F}_{\texttt{raPSI}}\}$-hybrid model.

[1] If the subprotocols invoked within the Leopard framework are constructed using DH/OPRF techniques, particular attention must be paid to the security issues that might arise from repeated use of DH/OPRF (which exhibits a streaming characteristic), as detailed in Section 4.3. Where feasible, a fresh uniformly random key should be selected for each use.

Theorem 1. *The protocol $\Pi_{\texttt{UPSI}}$ described in Figure 8 securely realizes $\mathcal{F}_{\texttt{UPSI}}$ in the $\{\mathcal{F}_{\texttt{aPSI}}, \mathcal{F}_{\texttt{raPSI}}\}$-hybrid model against semi-honest adversaries.*

Proof. We construct simulators $\texttt{Sim}_{\mathcal{S}}$ and $\texttt{Sim}_{\mathcal{R}}$ that simulate the views of corrupted sender $\mathcal{S}$ and receiver $\mathcal{R}$ respectively.

Corrupted Sender. The simulator $\texttt{Sim}_{\mathcal{S}}$ simulates a real execution in which the sender $\mathcal{S}$ is corrupted. Since the adversary $\mathcal{A}$ is semi-honest, $\texttt{Sim}_{\mathcal{S}}$ can directly obtain $\mathcal{S}$'s inputs Y_d^-, Y_d^+ and forward them externally to $\mathcal{F}_{\texttt{UPSI}}$. Once $\mathcal{A}$ provides the input Y_d^- for $\mathcal{F}_{\texttt{raPSI}}$, $\texttt{Sim}_{\mathcal{S}}$ randomly chooses input $X_{d-1} \setminus X_d^-$ for $\mathcal{R}$ and invokes the simulator $\texttt{Sim}_{\texttt{raPSI}_1}$ to simulate $\mathcal{F}_{\texttt{raPSI}}$. When $\mathcal{A}$ supplies the input Y_d^+ for $\mathcal{F}_{\texttt{raPSI}}$, $\texttt{Sim}_{\mathcal{S}}$ randomly picks $\mathcal{R}$'s input $X_{d-1} \setminus X_d^-$ and invokes the simulator $\texttt{Sim}_{\texttt{raPSI}_2}$ to simulate $\mathcal{F}_{\texttt{raPSI}}$. Finally, upon receiving from $\mathcal{A}$ the input $Y_d = (Y_{d-1} \setminus Y_d^-) \cup Y_d^+$ intended for $\mathcal{F}_{\texttt{aPSI}}$, $\texttt{Sim}_{\mathcal{S}}$ randomly selects input X_d^+ for $\mathcal{R}$ and invokes the simulator $\texttt{Sim}_{\texttt{aPSI}}$ to simulate the execution of $\mathcal{F}_{\texttt{aPSI}}$.

We argue that the outputs of $\texttt{Sim}_{\mathcal{S}}$ are computationally indistinguishable from the real view of $\mathcal{S}$ via the following sequence of hybrids:

$\texttt{Hyb}_0$: $\mathcal{S}$'s view in the real protocol.

$\texttt{Hyb}_1$: Same as $\texttt{Hyb}_0$, except that upon receiving Y_d^- from $\mathcal{A}$, $\texttt{Sim}_{\mathcal{S}}$ invokes the $\mathcal{F}_{\texttt{raPSI}}$ simulator $\texttt{Sim}_{\texttt{raPSI}_1}$ to generate the simulated view for $\mathcal{S}$. This hybrid is computationally indistinguishable from $\texttt{Hyb}_0$ due to the security of protocol $\Pi_{\texttt{raPSI}}$.

$\texttt{Hyb}_2$: Same as $\texttt{Hyb}_1$, except that upon receiving Y_d^+ from $\mathcal{A}$, $\texttt{Sim}_{\mathcal{S}}$ invokes the $\mathcal{F}_{\texttt{raPSI}}$ simulator $\texttt{Sim}_{\texttt{raPSI}_2}$ to generate the simulated view for $\mathcal{S}$. This hybrid is computationally indistinguishable from $\texttt{Hyb}_1$ due to the security of protocol $\Pi_{\texttt{raPSI}}$.

$\texttt{Hyb}_3$: Same as $\texttt{Hyb}_2$, except that upon receiving Y_d from $\mathcal{A}$, $\texttt{Sim}_{\mathcal{S}}$ invokes the simulator $\texttt{Sim}_{\texttt{aPSI}}$ for $\mathcal{F}_{\texttt{aPSI}}$ to generate the simulated view for $\mathcal{S}$. This hybrid is computationally indistinguishable from $\texttt{Hyb}_2$ due to the security of protocol $\Pi_{\texttt{aPSI}}$. $\texttt{Hyb}_3$ is exactly the view produced by $\texttt{Sim}_{\mathcal{S}}$.

Corrupted Receiver. Simulator $\texttt{Sim}_{\mathcal{R}}$ simulates a real execution in which the receiver $\mathcal{R}$ is corrupted. It begins by invoking the semi-honest adversary $\mathcal{A}$ that corrupts $\mathcal{R}$, and directly obtains $\mathcal{R}$'s inputs X_d^-, X_d^+. Then, $\texttt{Sim}_{\mathcal{R}}$ forwards these sets to the ideal functionality $\mathcal{F}_{\texttt{UPSI}}$ and waits for the output I_d from $\mathcal{F}_{\texttt{UPSI}}$. Upon receiving from $\mathcal{A}$ the input $X_{d-1} \setminus X_d^-$ intended for $\mathcal{F}_{\texttt{raPSI}}$, $\texttt{Sim}_{\mathcal{R}}$ computes $S_1 = (I_{d-1} \setminus I_d) \setminus X_d^-$. It then randomly selects a set $Y_d^{-\prime}$ such that $|Y_d^{-\prime}| = |Y_d^-| - |S_1|$ and $Y_d^{-\prime} \cap (X_{d-1} \setminus X_d^-) = \emptyset$, and constructs $\mathcal{S}$'s input as $Y_d^- = Y_d^{-\prime} \cup S_1$. Next, $\texttt{Sim}_{\mathcal{R}}$ invokes the simulator $\texttt{Sim}_{\texttt{raPSI}_1}$ to simulate $\mathcal{F}_{\texttt{raPSI}}$ and sends S_1 to $\mathcal{A}$. Upon receiving from $\mathcal{A}$ the input $X_{d-1} \setminus X_d^-$ intended for $\mathcal{F}_{\texttt{raPSI}}$, $\texttt{Sim}_{\mathcal{R}}$ computes $S_2 = (I_d \setminus I_{d-1}) \cap X_{d-1}$ (or $S_2 = (I_d \setminus I_{d-1}) \setminus X_d^+$). It then randomly selects a set $Y_d^{+\prime}$ such that $|Y_d^{+\prime}| = |Y_d^+| - |S_2|$ and $Y_d^{+\prime} \cap (X_{d-1} \setminus X_d^-) = \emptyset$, and constructs $\mathcal{S}$'s input as $Y_d^+ = Y_d^{+\prime} \cup S_2$. Next, $\texttt{Sim}_{\mathcal{R}}$ invokes the simulator $\texttt{Sim}_{\texttt{raPSI}_2}$ to simulate $\mathcal{F}_{\texttt{raPSI}}$ and sends S_2 to $\mathcal{A}$. Upon receiving from $\mathcal{A}$ the input X_d^+ intended for $\mathcal{F}_{\texttt{aPSI}}$, $\texttt{Sim}_{\mathcal{R}}$ computes $S_3 = I_d \cap X_d^+$ (or $S_3 = I_d \setminus X_{d-1}$). It then randomly selects a set Y_d' such that $|Y_d'| = |Y_d| - |S_3|$ and $Y_d' \cap X_d^+ = \emptyset$, and constructs $\mathcal{S}$'s input as $Y_d = Y_d' \cup S_3$. Next, $\texttt{Sim}_{\mathcal{R}}$ invokes the simulator $\texttt{Sim}_{\texttt{aPSI}}$ to simulate $\mathcal{F}_{\texttt{aPSI}}$ and sends S_3 to $\mathcal{A}$.

We argue that the outputs of $\mathsf{Sim}_{\mathcal{R}}$ are computationally indistinguishable from the real view of $\mathcal{R}$ via the following sequence of hybrids:

Hyb_0 : $\mathcal{R}$'s view in the real protocol.

Hyb_1 : Same as Hyb_0, except that upon receiving the input $X_{d-1} \setminus X_d^-$ for $\mathcal{F}_{\mathsf{raPSI}}$ from $\mathcal{A}$, $\mathsf{Sim}_{\mathcal{R}}$ computes $S_1 = (I_{d-1} \setminus I_d) \setminus X_d^-$, replaces the output of $\mathcal{F}_{\mathsf{raPSI}}$ with S_1, and invokes the $\mathcal{F}_{\mathsf{raPSI}}$ simulator $\mathsf{Sim}_{\mathsf{raPSI}_1}$ to generate the simulated view for $\mathcal{R}$. This hybrid is computationally indistinguishable from Hyb_0 due to the security of protocol Π_{raPSI}. Note that the set relationships satisfy the following: $S_1 = \complement_{[(I_{d-1} \setminus X_d^-) \cup (S_2 \cup S_3)]} I_d$, $(I_{d-1} \setminus I_d) \setminus X_d^- = S_1 = (X_{d-1} \setminus X_d^-) \cap Y_d^-$.

Hyb_2 : Same as Hyb_1, except that upon receiving the input $X_{d-1} \setminus X_d^-$ for $\mathcal{F}_{\mathsf{raPSI}}$ from $\mathcal{A}$, $\mathsf{Sim}_{\mathcal{R}}$ computes $S_2 = (I_d \setminus I_{d-1}) \cap X_{d-1}$, replaces the output of $\mathcal{F}_{\mathsf{raPSI}}$ with S_2, and invokes the $\mathcal{F}_{\mathsf{raPSI}}$ simulator $\mathsf{Sim}_{\mathsf{raPSI}_2}$ to generate the simulated view for $\mathcal{R}$. This hybrid is computationally indistinguishable from Hyb_1 due to the security of protocol Π_{raPSI}. Note that the set relationships satisfy the following: $S_2 = \complement_{I_d}\{[I_{d-1} \setminus (X_d^- \cup S_1)] \cup S_3\}$, $(I_d \setminus I_{d-1}) \cap X_{d-1} = S_2 = (X_{d-1} \setminus X_d^-) \cap Y_d^+$.

Hyb_3 : Same as Hyb_2, except that upon receiving the input X_d^+ for $\mathcal{F}_{\mathsf{aPSI}}$ from $\mathcal{A}$, $\mathsf{Sim}_{\mathcal{R}}$ computes $S_3 = I_d \cap X_d^+$, replaces the output of $\mathcal{F}_{\mathsf{aPSI}}$ with S_3, and invokes the $\mathcal{F}_{\mathsf{aPSI}}$ simulator $\mathsf{Sim}_{\mathsf{aPSI}}$ to generate the simulated view for $\mathcal{R}$. This hybrid is computationally indistinguishable from Hyb_2 due to the security of protocol Π_{aPSI}. Note that the set relationships satisfy the following: $S_3 = \complement_{I_d}\{[I_{d-1} \setminus (X_d^- \cup S_1)] \cup S_2\}$, $I_d \cap X_d^+ = S_3 = X_d^+ \cap Y_d$. The hybrid is exactly the view produced by $\mathsf{Sim}_{\mathcal{R}}$.

□

4 Instantiation of One-Sided Output UPSI Based on Leopard

This section begins by introducing the core design framework for asymmetric PSI and its derived versions for reversed asymmetric PSI. We then recall representative paradigms of asymmetric PSI that fall under this framework. For the instantiation of UPSI under Leopard, we select an FHE-based asymmetric PSI as the underlying sub-protocol, as it naturally supports *the reuse of data structures*—secure updates to participants' transformed representations of private sets. The corresponding FHE-based reversed asymmetric PSI is then straightforwardly derived to form a complete protocol pair.

4.1 Design Frameworks of Asymmetric PSI and Reversed Asymmetric PSI

We summarize the design framework for asymmetric PSI as illustrated in Figure 9. In this setting, the receiver $\mathcal{R}$ holds the private set Y, and the sender $\mathcal{S}$ holds the private set X, with $|Y| \ll |X|$. The framework consists of three main steps: *Step 1.* Transform the elements of $\mathcal{R}$'s set Y from y to $\hat{y}$ (if required). *Step 2.* Transform $\mathcal{S}$'s set X from its set representation into an appropriate data structure D_X. *Step 3.* Perform private membership tests (PMT) on the values $\hat{y}$ with the participation of $\mathcal{S}$ using D_X; $\mathcal{R}$ then obtains the final output indicating whether each original element y corresponding to $\hat{y}$ belongs to the set intersection. *Step 1* and *Step 3* can be processed

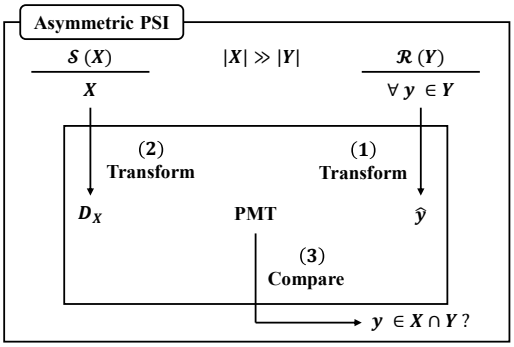

Figure 9: Design Framework of Asymmetric PSI.

in batches according to the batching granularity of the transformed values $\hat{y}$ in Y.

The design frameworks for reversed asymmetric PSI are derived from that of asymmetric PSI, as shown in Figure 10 and Figure 11. One straightforward way to obtain reversed asymmetric PSI is by directly swapping the roles of the parties, as depicted in Figure 10; however, this approach tends to incur high overhead. A more efficient framework transforms asymmetric PSI into reversed asymmetric PSI by substituting the core building block—private membership test (PMT)—with a reversed private membership test (RPMT) [2], illustrated in Figure 11. The construction of RPMT can draw upon existing works on asymmetric PSI and PSU.

4.2 Representative Paradigms of Asymmetric PSI under the Design Framework

Several representative paradigms of asymmetric PSI under the design framework shown in Figure 9 are outlined below.

DH/OPRF-Based Paradigm. In DH/OPRF-based designs [21, 22, 30, 35], $\mathcal{S}$ holds a secret key $\mathsf{k}_{\mathcal{S}}$. *Step 1.* $\mathcal{R}$ obtains $\hat{y} = \mathsf{Enc}(y, \mathsf{k}_{\mathcal{S}})$ for every $y \in Y$, either via Diffie-Hellman-style computation or with the help of an OPRF provided by $\mathcal{S}$. *Step 2.* $\mathcal{S}$ locally computes the data structure $D_X = \mathsf{Enc}(X, \mathsf{k}_{\mathcal{S}})$. *Step 3.* $\mathcal{S}$ sends D_X to $\mathcal{R}$, who then locally compares each $\hat{y}$ against D_X to determine the set intersection.

OKVS-Based Paradigm. For OKVS-based designs [17, 34], the simplest instantiation does not require *Step 1. Step 2.* $\mathcal{S}$ locally encodes its private set X into the data structure $D_X = \mathsf{OKVS.Enc}(X, AD)$, where AD denotes the associated data of X. *Step 3.* $\mathcal{S}$ sends D_X to $\mathcal{R}$, who locally decodes it with each element $y \in Y$ to obtain $\mathsf{OKVS.Dec}(D_X, y)$ and compares the result with the associated data of y to obtain the intersection.

[2] Private membership test (PMT) is a fundamental cryptographic primitive in which $\mathcal{R}$ holds an element $x \in X$, $\mathcal{S}$ holds a set Y, and $\mathcal{R}$ learns a single bit indicating whether $x \in Y$—equivalently, whether $x \in X \cap Y$—while ensuring that no additional information is revealed beyond this membership outcome, as illustrated in Figure 10. The corresponding notion of reversed private membership test (RPMT) reverses the roles: $\mathcal{R}$ holds a set X, $\mathcal{S}$ holds an element $y \in Y$, and $\mathcal{R}$ obtains a bit denoting whether $y \in X$ (or equivalently, whether $y \in X \cap Y$), again with no other leakage, as illustrated in Figure 11.

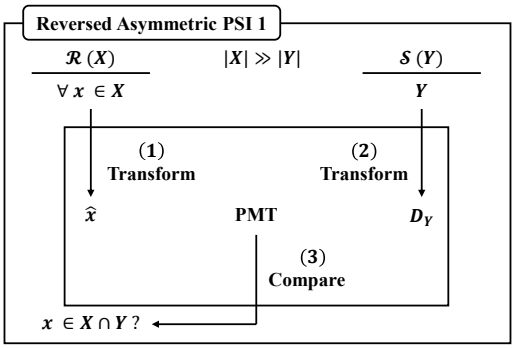

Figure 10: Design Framework 1 of Reversed Asymmetric PSI.

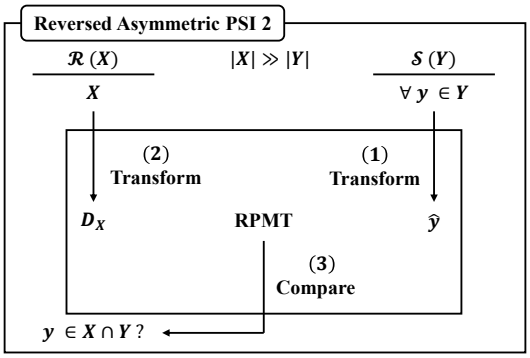

Figure 11: Design Framework 2 of Reversed Asymmetric PSI.

FHE-Based Paradigm. In FHE-based designs [8, 9, 12], $\mathcal{R}$ generates a key pair $(\mathsf{pk}, \mathsf{sk})$ and makes pk known to $\mathcal{S}$. *Step 1.* $\mathcal{R}$ locally computes the ciphertext set $\hat{Y} = \mathsf{FHE.Enc}(\mathsf{pk}, Y)$. *Step 2.* $\mathcal{S}$ encodes its private set X into a polynomial-form data structure $D_X = P(z) = \prod_{i=1}^{N}(z - x_i)$ where $x_i \in X$ and $|X| = N$. *Step 3.* $\mathcal{R}$ sends $\hat{Y}$ to $\mathcal{S}$; For each encrypted element $\hat{y} \in \hat{Y}$, $\mathcal{S}$ picks a random non-zero mask r, computes the masked encrypted evaluation $\widehat{P'(y)} = \mathsf{FHE.Enc}(\mathsf{pk}, r \cdot P(y))$, and returns $\widehat{P'(y)}$ to $\mathcal{R}$; finally, $\mathcal{R}$ locally checks whether $\mathsf{FHE.Dec}(\mathsf{sk}, \widehat{P'(y)}) \stackrel{?}{=} 0$ to determine the intersection.

4.3 Discussion on Data Structure Reusability

Asymmetric PSI schemes can be broadly categorized into two abstract paradigms, illustrated in Figure 12. In both paradigms, one party encrypts and transmits its set elements to the other party for comparison; the key difference lies in which party performs the encryption and transmission. The DH/OPRF-based and OKVS-based paradigms belong to the first category, while the FHE-based paradigm falls into the second.

The second paradigm is generally preferable. In this setting, the data structure D_X derived from $\mathcal{S}$'s larger set X does not need to be

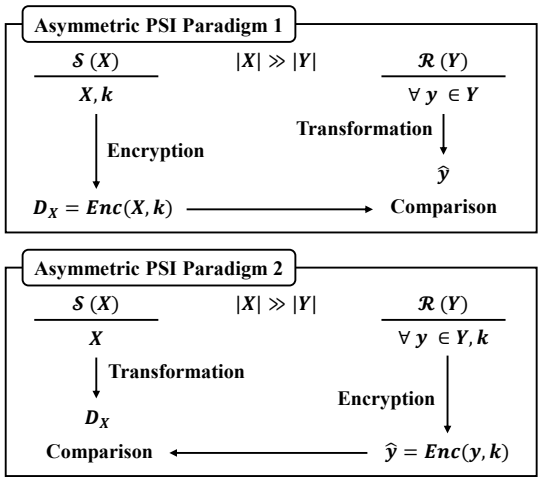

Figure 12: Abstract Paradigms of Asymmetric PSI.

encrypted and transferred, thereby reducing both communication and computation overhead. Moreover, D_X can be reused when $\mathcal{S}$'s input set X undergoes minor changes across repeated protocol invocations, as its structure naturally supports element-wise updates. Consequently, the second paradigm is well-suited for constructing efficient UPSI protocols.

In contrast, the DH/OPRF-based and OKVS-based paradigms do not inherently support such data structure reuse. For the DH/OPRF-based paradigm, to enable reuse, $\mathcal{S}$ would have to keep its secret key fixed so that each element is encrypted to the same value deterministically. After encrypting and transmitting the updated elements in each invocation, $\mathcal{R}$ could then identify and process the changes to update its local copy of $\mathcal{S}$'s database. However, as updated ciphertexts are transmitted upon every invocation, $\mathcal{R}$ may infer the evolution of X by observing the history of received ciphertexts—an issue that clearly violates security (referred to as forward privacy in [39]). For the OKVS-based paradigm, D_X typically does not easily support fine-grained element updates [3].

Upon closer examination, the FHE-based schemes [8, 9, 12] employ a polynomial-interpolation structure for D_X that readily accommodates element insertion and deletion. Specifically, given $D_X = P(z) = \prod_{i=1}^{N}(z - x_i)$ where $x_i \in X$ and $|X| = N$: Inserting an element x^+ updates D_X by multiplying the polynomial by $(z - x^+)$; whereas deleting an element x^- updates D_X by dividing the polynomial by $(z - x^-)$. Due to its natural support for data-structure reuse, we adopt the state-of-the-art FHE-based asymmetric PSI scheme [12] as the sub-protocol for constructing `Leopard`.

4.4 Leopard Instantiations

Using the asymmetric PSI protocol from [12] as the sub-protocol, an instantiation of the UPSI protocol under `Leopard` still requires a reversed asymmetric PSI sub-protocol. Since the literature on reversed asymmetric PSI remains sparse, we propose a new scheme

[3] [25] also mentioned in their work that it is still difficult to define the updatable OKVS supporting the operation of deletion.

based on [38], which currently represents the state-of-the-art FHE-based asymmetric PSU derived from [12].

Below is a brief summary of the core construction of [38]. Let the receiver $\mathcal{R}$ hold the private set X and the sender $\mathcal{S}$ hold the private set Y, with $|X| \gg |Y|$. $\mathcal{S}$ generates a key pair $(\mathsf{pk}, \mathsf{sk})$ for a FHE scheme and makes the public key pk available to $\mathcal{R}$. First, $\mathcal{S}$ locally encrypts Y as $\hat{Y} = \mathsf{FHE.Enc}(\mathsf{pk}, Y)$. Meanwhile, $\mathcal{R}$ encodes its set $X = \{x_1, \cdots, x_N\}$ into the data structure of polynomial form $D_X = P(z) = \prod_{i=1}^{N}(z - x_i)$. Second, $\mathcal{S}$ sends $\hat{Y}$ to $\mathcal{R}$. For each encrypted element $\hat{y} \in \hat{Y}$, $\mathcal{R}$ picks a random mask r and homomorphically evaluates $P(y)$ under encryption to obtain $\widehat{P'(y)} = \mathsf{FHE.Enc}(\mathsf{pk}, r + P(y))$. $\mathcal{R}$ sends these ciphertexts back to $\mathcal{S}$, who then decrypts each one: $r' = \mathsf{FHE.Dec}(\mathsf{sk}, \widehat{P'(y)})$. Third, $\mathcal{R}$ and $\mathcal{S}$ together invoke a PEQT sub-protocol, with inputs r and r', respectively. PEQT outputs a bit b to $\mathcal{R}$, where $b = 1$ if $r = r'$ and $b = 0$ otherwise. Finally, $\mathcal{R}$ and $\mathcal{S}$ together invoke an OT protocol: $\mathcal{R}$ uses b as its choice bit, and $\mathcal{S}$ provides $(y, \perp)$ as the two messages. If $b = 1$, $\mathcal{R}$ receives $\perp$; otherwise, $\mathcal{R}$ receives y. After processing all elements, $\mathcal{R}$ collects the received values to form $Y \setminus X$, and thus obtains the union $X \cup Y = X \cup (Y \setminus X)$.

The reversed asymmetric PSI scheme derived from [38] adheres to the second design framework in Figure 11. It can be obtained by modifying the final OT step in the original scheme: either flipping the choice bit b of $\mathcal{R}$ (from 0 to 1 and from 1 to 0) , or swapping the input tuple of $\mathcal{S}$ from $(y, \perp)$ to $(\perp, y)$. This adjustment shifts the output from set union to set intersection.

Communication overhead in FHE-based schemes is generally low, with asymptotic communication complexity scaling linearly with the size of the smaller set Y. Moreover, the interaction between parties can be completed in a single round. Computationally, however, encryption, decryption, and homomorphic operations remain resource-intensive. To mitigate this, multiple FHE-oriented engineering optimizations are employed. For a comprehensive discussion of these techniques, we refer readers to the original works [12, 38].

For data structure reuse, two key optimizations can be applied. First, as shown in the framework Leopard in Figure 2, the receiver's input remains identical across the first and second calls to the reversed asymmetric PSI sub-protocols. If the same reversed asymmetric PSI protocol is used in both calls, its associated data structure can be generated (and transmitted) only once. Beyond reusing data structures across sub-protocols within a single UPSI execution, further improvements can be achieved by reusing data structures across multiple UPSI invocations. After initial creation, the data structures for the asymmetric and two reversed asymmetric PSI sub-protocols can be retained and reused. In subsequent runs of the UPSI protocol, they only need to be updated to reflect changes by incorporating both insertions and deletions of elements.

Since the original schemes [12, 38] incorporate OPRF techniques, the PRF key must be fixed to support data structure reuse. To this end, we follow the approach of [24], as illustrated in its protocol instantiation, to carry out the OPRF operations explicitly: namely, we reuse the PRF values of the respective set elements held by both parties, and perform local PRF computation and Multi-Round OPRF (MROPRF) evaluation on newly added elements.

The basic protocol of our FHE-based UPSI instance under framework Leopard is provided in Appendix A.

To further evaluate Leopard, we also give a symmetric-key based instantiation via black-box invocations of the state-of-the-art one-sided output conventional PSI protocol [34], which can be directly extended to unbalanced settings to meet our requirements for (reversed) asymmetric PSIs. To distinguish this instantiation from the FHE-based one, we refer to the latter as the public-key based.

5 Performance Evaluation

This section presents the performance evaluation of our general framework Leopard and two instantiations under Leopard, combining theoretical analysis with experimental validation. Both confirm that Leopard provides noticeable performance improvements over the state-of-the-art one-sided output UPSI works [2, 3].

5.1 Theoretical Analysis

Framework Comparison. We first compare our general framework Leopard with the upper-layer structure of protocol [3] and the generic protocol in [2]. In Leopard, only one asymmetric PSI and two reversed asymmetric PSI sub-protocols are invoked. In contrast, the protocol in [3] requires an additional invocation of underlying asymmetric PSI, while the generic protocol in [2] needs additional invocations of one circuit-PSI and one combine-and-permute sub-protocols, as they did not exploit local set operations to further simplify the overall flow.

Asymptotic Complexity Comparison. Let n denotes the set size of the small update sets and N the set size of the large original sets. For [34] in the balanced setting, the computational and communication complexities of the one-sided output conventional PSI protocol are $O(N)$ and $O(N \cdot \log N)$. While for [34] under unbalanced setting, the computational and communication complexities for asymmetric PSI are $O(N)$ and $O(N \cdot \log N)$, and both $O(N)$ for the reversed asymmetric PSI. Thus for Leopard instance based on symmetric-key primitives, the overall computational and communication complexities are $O(N)$ and $O(N \cdot \log N)$. For [12], the computational and communication complexities are $\tilde{O}(n \cdot N)$ and $\tilde{O}(n \cdot (\log \log N)^2)$ respectively, and for [38] they are $O(N + n \cdot \log N)$

Table 2: Asymptotic Complexities of Two Instantiations under Leopard. 'Leopard SKP' represents the instantiation of framework Leopard based on symmetric-key primitives, and 'Leopard PKP' denotes the public-key based one. The computational and communication complexities of lower-layer sub-protocols and the composed instance are presented. n denotes the set size of small update sets, and N for the large original sets. $\tilde{O}(.)$ denotes the asymptotic complexity ignores polylogarithmic factors.

Instance	Breakdown	Comp. Complexity	Comm. Complexity
Leopard SKP	aPSI SKP [34]	$O(N)$	$O(N \cdot \log N)$
	raPSI SKP [34]	$O(N)$	$O(N)$
	Overall	$O(N)$	$O(N \cdot \log N)$
Leopard PKP	aPSI PKP [12]	$\tilde{O}(n \cdot N)$	$\tilde{O}(n \cdot (\log \log N)^2)$
	raPSI PKP [38]	$O(N + n \cdot \log N)$	$O(n \cdot \log N)$
	Overall	$\tilde{O}(n \cdot N)$	$O(n \cdot \log N)$

Table 3: Efficiency of Data Structure Reuse for Public-Key Based Leopard Instantiation. The unoptimized version is denoted as 'Leopard PKP (no reuse)', and the version with data structure reuse is denoted as 'Leopard PKP (reuse)'.

N	n	Protocol	Comm. (MB)	Total Running Time (s)							
				LAN		200Mbps		50Mbps		5Mbps	
				T=1	T=8	T=1	T=8	T=1	T=8	T=1	T=8
2^{18}	2^4	Leopard PKP (no reuse)	3.611	12.237	9.907	18.036	15.628	18.364	16.270	28.923	26.924
		Leopard PKP (reuse)	3.610	8.379	6.945	12.525	11.099	13.021	11.625	24.140	23.265
	2^5	Leopard PKP (no reuse)	3.606	12.267	9.892	17.984	15.878	17.851	16.310	28.808	27.662
		Leopard PKP (reuse)	3.607	8.257	7.038	12.714	11.176	13.257	11.599	24.157	23.253
	2^6	Leopard PKP (no reuse)	3.601	12.225	9.991	17.540	15.831	18.111	16.039	28.881	27.681
		Leopard PKP (reuse)	3.610	8.398	7.220	12.764	11.370	13.222	11.785	24.513	23.483
2^{20}	2^4	Leopard PKP (no reuse)	6.080	57.887	31.686	61.789	35.448	60.755	36.073	75.252	52.672
		Leopard PKP (reuse)	6.079	14.237	9.154	18.362	13.705	18.725	14.432	36.094	32.628
	2^5	Leopard PKP (no reuse)	6.078	57.666	31.385	61.052	35.698	60.766	36.204	75.010	52.607
		Leopard PKP (reuse)	6.078	14.102	9.078	18.188	13.861	18.536	14.598	36.008	32.750
	2^6	Leopard PKP (no reuse)	6.077	57.817	31.619	61.072	35.866	60.323	35.867	75.570	52.500
		Leopard PKP (reuse)	6.078	14.175	9.094	18.212	13.642	18.625	14.480	35.916	32.665
2^{22}	2^4	Leopard PKP (no reuse)	11.163	245.469	126.751	243.104	125.097	242.782	126.051	254.509	152.667
		Leopard PKP (reuse)	11.163	43.840	23.216	48.017	27.837	49.162	29.615	74.787	69.461
	2^5	Leopard PKP (no reuse)	11.163	250.713	128.156	247.374	126.208	247.599	127.443	257.004	154.149
		Leopard PKP (reuse)	11.163	44.391	23.632	48.954	28.178	50.099	29.984	75.759	69.739
	2^6	Leopard PKP (no reuse)	11.163	249.464	128.740	248.733	127.241	249.378	127.545	258.079	154.693
		Leopard PKP (reuse)	11.163	44.222	23.733	48.953	28.161	50.159	30.135	76.035	69.933

and $O(n \cdot \log N)$ [4]. Consequently, the Leopard instance based on public-key primitives possesses overall complexities of $\tilde{O}(n \cdot N)$ in computation and $O(n \cdot \log N)$ in communication. The statistics are summarized as shown in Table 2.

The communication efficiency inherent to the FHE-based sub-protocols [12, 38] straightly results in the strong performance of Leopard instance when instantiated with public-key primitives—achieving favorable outcomes in both asymptotic communication complexity and practical communication cost. Considering the comparative works, the protocol in [3] attains asymptotic complexities of $O(n \cdot \log^2 N)$ in both computation and communication, and [2] of $O(n \cdot \log N)$, which both scale linearly with n and sub-linearly with N. Although these prior works achieve remarkable asymptotic complexities via careful design, our Leopard-based instantiations offer compelling practical advantages. A more detailed comparison of the asymptotic complexities of existing UPSI instantiations is provided in Table 1. As for practical performance, please refer to the next Section 5.2 for comprehensive experimental results.

5.2 Experimental Evaluation

5.2.1 Experimental Setup. All experiments were conducted on an Aliyun ECS instance of model `ecs.r9i.6xlarge`, equipped with *24 vCPUs and 192 GiB of RAM*. The operating system was *Ubuntu 24.04 (64-bit)*.

We evaluate the practical computational and communication performance of our two instantiations under Leopard, comparing with the state-of-the-art one-sided output UPSI protocols [2, 3].

The size N of the large original sets is chosen as $2^{18}, 2^{20}, 2^{22}$, and the size n of the small update sets as $2^4, 2^5, 2^6$.

Communication between participants is carried out via `localhost`, with different network conditions simulated using the Linux `tc` command. LAN environment is emulated with bandwidth (BW) of 1 Gbps and round-trip time (RTT) of 0.2 ms. For WAN-like conditions, we consider bandwidths of 200 Mbps, 50 Mbps, and 5 Mbps, each with RTT of 80 ms.

Since the open source implementations of both the symmetric-key [34] and public-key based [12, 38] sub-protocols used for Leopard instantiations support multi-threading, we evaluate performance with thread-level parallelism, setting thread count T to $1, 8$. [2] supports process-level parallelism with the execution of the four underlying circuit-PSI, thus we tested it under this multi-processing scenario. Both [2, 3] only support single-threaded execution in their implementations.

Communication cost is measured in megabytes (MB), and computational cost (reported as 'Total Running Time') is measured in seconds (s).

5.2.2 Implementation Details. The symmetric-key based Leopard instantiation is built upon open source code of [34], and [12, 24, 38]'s for the public-key based Leopard instantiation [5]. The implementations of [2, 3] are based on their respective official repositories [6]. Code of [2] is written in RUST, while others in C++.

Set elements are set to 128-bit binary strings, except for [3], which follows its original implementation by using 16-digit decimal

[4] As detailed in Section 4.4, the practical implementation of the reversed asymmetric PSI deviates slightly from the original protocol [38]. Nevertheless, this modification—namely, the trivial conversion of [38] from asymmetric PSU to reversed asymmetric PSI—leave the asymptotic complexities unchanged in terms of their theoretical orders.

[5] https://github.com/ladnir/volepsi for [34], https://github.com/microsoft/APSI for [12], https://github.com/real-world-cryptography/APSU for [38], and https://github.com/ShallMate/upsi for [24].

[6] https://github.com/ruidazeng/upsi-revisited for [3], and https://gitlab.eurecom.fr/project-spring-2025/PSI/ for [2].

Table 4: Practical Performance Evaluation Table for One-sided Output UPSI Instances. The state-of-the-art one-sided output UPSI protocols [2, 3] are chosen as baselines, and our symmetric-key and public-key based Leopard instantiations are evaluated. The Leopard instance under symmetric-key primitives is denoted as 'Leopard SKP', and the public-key based one under the optimization of data structure reuse is denoted as 'Leopard PKP (reuse)'. Original set size N is set to $2^{18}, 2^{20}, 2^{22}$, and both the insertion and deletion set sizes are set as n with value $2^4, 2^5, 2^6$. Network setting is simulated with (bandwidth, RTT latency)-pairs: (1 Gbps, 0.2 ms), (200 Mbps, 80 ms), (50 Mbps, 80 ms), and (5 Mbps, 80 ms). The Leopard instances are tested with multi-threading at $T = 1$ and $T = 8$. Since the baselines [2, 3] are inherently single-threaded, we fill their $T = 8$ entries with '-'. Both communication (MB) and computational (s) costs are measured. The protocol execution times exceeding 1800 seconds are ddenoted as '—'. For each parameter configuration, the top-performing result is marked with an orange fill, while the runner-up is highlighted in purple.

N	n	Protocol	Comm. (MB)	Total Running Time (s)							
				LAN		200Mbps		50Mbps		5Mbps	
				T=1	T=8	T=1	T=8	T=1	T=8	T=1	T=8
2^{18}	2^4	[BMS+24] [3]	108.595	208.591	-	220.147	-	232.234	-	376.753	-
		[ACF+26] [2]	365.034	12.440	-	28.700	-	73.635	-	—	-
		Leopard SKP	14.567	7.670	6.559	9.075	7.961	10.954	9.845	32.495	31.337
		Leopard PKP (reuse)	3.610	8.379	6.945	12.525	11.099	13.021	11.625	24.140	23.265
	2^5	[BMS+24] [3]	215.772	420.250	-	434.457	-	461.455	-	765.519	-
		[ACF+26] [2]	366.033	13.901	-	29.746	-	74.750	-	—	-
		Leopard SKP	14.568	7.668	6.558	9.071	7.962	11.007	9.888	32.499	31.367
		Leopard PKP (reuse)	3.607	8.257	7.038	12.714	11.176	13.257	11.599	24.157	23.253
	2^6	[BMS+24] [3]	429.488	826.545	-	846.119	-	915.729	-	1525.730	-
		[ACF+26] [2]	368.071	16.785	-	33.057	-	76.851	-	—	-
		Leopard SKP	14.569	7.428	6.322	8.830	7.728	10.751	9.650	32.252	31.364
		Leopard PKP (reuse)	3.610	8.398	7.220	12.764	11.370	13.222	11.785	24.513	23.483
2^{20}	2^4	[BMS+24] [3]	114.232	226.990	-	232.283	-	244.417	-	408.342	-
		[ACF+26] [2]	1456.861	47.725	-	103.194	-	280.148	-	—	-
		Leopard SKP	53.218	29.567	25.020	32.786	28.403	39.540	34.880	120.445	115.617
		Leopard PKP (reuse)	6.079	14.237	9.154	18.362	13.705	18.725	14.432	36.094	32.628
	2^5	[BMS+24] [3]	227.247	443.045	-	483.876	-	477.988	-	804.638	-
		[ACF+26] [2]	1457.861	48.009	-	104.731	-	282.962	-	—	-
		Leopard SKP	54.219	29.626	25.078	33.053	28.329	39.613	35.138	122.015	116.482
		Leopard PKP (reuse)	6.078	14.102	9.078	18.188	13.861	18.536	14.598	36.008	32.750
	2^6	[BMS+24] [3]	452.319	923.336	-	907.468	-	952.805	-	1609.738	-
		[ACF+26] [2]	1459.898	51.211	-	105.198	-	287.186	-	—	-
		Leopard SKP	54.220	28.684	24.054	31.911	27.357	38.784	34.004	120.054	115.535
		Leopard PKP (reuse)	6.078	14.175	9.094	18.212	13.642	18.625	14.480	35.916	32.665
2^{22}	2^4	[BMS+24] [3]	120.024	237.450	-	243.156	-	250.491	-	421.283	-
		[ACF+26] [2]	5824.171	188.692	-	398.233	-	1121.145	-	—	-
		Leopard SKP	211.858	118.995	100.277	129.889	110.810	155.495	136.937	478.515	459.233
		Leopard PKP (reuse)	11.163	43.840	23.216	48.017	27.837	49.162	29.615	74.787	69.461
	2^5	[BMS+24] [3]	238.683	465.389	-	482.249	-	509.407	-	849.015	-
		[ACF+26] [2]	5825.170	188.016	-	397.224	-	1118.808	-	—	-
		Leopard SKP	211.859	119.111	100.397	129.507	110.965	155.606	137.221	480.423	459.624
		Leopard PKP (reuse)	11.163	44.391	23.632	48.954	28.178	50.099	29.984	75.759	69.739
	2^6	[BMS+24] [3]	475.228	936.773	-	993.907	-	1012.401	-	1692.808	-
		[ACF+26] [2]	5827.208	190.907	-	397.185	-	1122.830	-	—	-
		Leopard SKP	211.860	115.317	96.670	125.618	106.974	153.427	133.103	475.425	456.487
		Leopard PKP (reuse)	11.163	44.222	23.733	48.953	28.161	50.159	30.135	76.035	69.933

numbers. The original sets are of size N; both insertion and deletion update sets are of size n. The statistical and computational security parameters are set to $\lambda = 40$ and $\kappa = 128$, respectively.

For public-key based Leopard instantiation, we utilize the DDH-based variant from the implementation of [38]. The design of [3] inherently supports data structure reuse, whereby the expensive generation of data structures for the original sets is treated as setup phase. Thus we also implement our data structure reused version following the construction principles outlined in Section 4.4, in which the original data structures are precomputed locally and reused across protocol executions.

5.2.3 Performance Comparison. In this Section, we first demonstrate the efficiency of data structure reuse for public-key based Leopard instantiation, then present a comprehensive performance evaluation of the two Leopard instantiations and existing one-sided output UPSI instances [2, 3].

Public-Key Based Leopard *Instantiation With or Without Data Structure Reuse.* Table 3 presents the experimental results.

The generation, storage, and update of the data structures are performed locally (offline) at the party holding the large set. Consequently, the reuse of these data structures does not affect the overall communication overhead of the protocol. This is empirically confirmed in our experimental results, where the communication costs of the no reuse and reuse variants remain nearly identical, which aligns precisely with the theoretical analysis.

Within the same UPSI round, the second invocation of the reversed asymmetric PSI sub-protocol can directly reuse the data structure generated during the first one's call. Moreover, across multiple consecutive UPSI protocol executions, the data structures from both the asymmetric and reversed asymmetric PSI sub-protocols can be reused, enabling small-scale updates that grow only linearly with the size of the update sets. As a result, the reuse version is expected to incur substantially lower computational overhead than the no reuse counterpart. Our experimental results corroborate this expectation: under all tested configurations—varying parameters N, n, T, and (BW, RTT)—the reuse version consistently and significantly outperforms the no reuse version, achieving a speedup factor ranging from 1.16× to 5.65×. The performance gain is most pronounced in compute-intensive scenarios (i.e., large N with high-bandwidth low-latency networks), where the computational bottleneck dominates. These findings firmly demonstrate that data structure reuse constitutes a highly effective performance optimization for the public-key based Leopard instantiation.

Practical Performance Evaluation for Leopard *Instantiations and Prior One-sided Output UPSI Instances.* The experimental results are summarized in Table 4, which demonstrate that the two proposed Leopard instantiations generally outperform the state-of-the-art one-sided output UPSI protocols [2, 3] in both communication and computational efficiency. The public-key based variant, constructed atop the FHE-based asymmetric PSO sub-protocols [12, 38], is especially well-suited for settings in which the cardinalities of the original sets and the update sets, denoted by N and n, differ substantially; its markedly lower communication overhead renders it particularly effective in adverse network conditions. The symmetric-key based variant, in contrast, builds upon the state-of-the-art balanced PSI construction [34] and is therefore better suited to scenarios where the disparity between N and n is small.

With respect to communication costs, the public-key based Leopard instantiation achieves reductions by factors ranging from 10.75× to 118.97× against [3], and from 101.12× to 522.01× against [2]. The symmetric-key based Leopard instantiation attains up to a 29.48× improvement over [3] and a 25.06× to 27.50× improvement over [2].

In terms of computational overhead under single-threaded execution, the public-key based Leopard instantiation yields speedups of 5.06× to 98.43× over [3] and at least 1.48× over [2]; the symmetric-key based variant achieves up to 111.28× over [3] and at least 1.58× over [2]. These advantages are further amplified when the Leopard instances are executed with multi-threading parallelism.

The excessive communication overhead of [2] leads to prohibitive runtimes under adverse network conditions. In the most challenging test environment considered (5 Mbps bandwidth with 80 ms RTT latency), all executions of [2] exceeded the 30-minute threshold, as reflected by the '—' notations in the experimental results.

5.3 Discussion

The implementation of Leopard can be further optimized through process-level parallelism. As established in the security proof of framework Leopard in Section 3.2, the invocations of the asymmetric PSI and the two reversed asymmetric PSI sub-protocols are mutually independent. Hence, by executing these sub-protocols in multi-processing scenario, the overall runtime can ideally be reduced to the maximum runtime among the three sub-protocols.

6 Conclusion

In this work, we present Leopard, a general framework for one-sided output UPSI that supports both element insertions and deletions. Our construction requires only three arbitrary black-box invocations of efficient (reversed) asymmetric PSI protocols. We instantiate Leopard with FHE-based sub-protocols and enhance the performance by reusing data structures across multiple invocations. Experimental results show that Leopard achieves up to two orders of magnitude improvement in both computational and communication costs over the state-of-the-art UPSI protocol of Badrinarayanan et al. (ASIACRYPT 2024), and also outperforms the recent scheme by Alborch et al. (ACNS 2026).

References

[1] Archita Agarwal, David Cash, Marilyn George, Seny Kamara, Tarik Moataz, and Jaspal Singh. 2026. Updatable private set intersection from structured encryption. *IACR Communications in Cryptology* 2, 4 (2026).

[2] Ferran Alborch, Tom Chauvier, Antonio Faonio, Alexandre Fontaine, Ferhat Karakoç, Alptekin Küpçü, Camille Malek, and Melek Önen. 2026. Updatable Private Set Intersection and Beyond: Efficient Constructions via Circuit PSI. In *Applied Cryptography and Network Security: 24th International Conference, ACNS 2026, Stony Brook, NY, USA, June 22–25, 2026, Proceedings, Part I* (Stony Brook, WI, USA). Springer-Verlag, Berlin, Heidelberg, 528–558.

[3] Saikrishna Badrinarayanan, Peihan Miao, Xinyi Shi, Max Tromanhauser, and Ruida Zeng. 2024. Updatable private set intersection revisited: Extended functionalities, deletion, and worst-case complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 200–233.

[4] Saikrishna Badrinarayanan, Peihan Miao, and Tiancheng Xie. 2022. Updatable private set intersection. *Proceedings on Privacy Enhancing Technologies* 2022, 2 (2022).

[5] 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}. In *32nd USENIX Security Symposium (USENIX Security 23)*. 301–318.

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

[7] Melissa Chase and Peihan Miao. 2020. Private set intersection in the internet setting from lightweight oblivious PRF. In *Annual International Cryptology Conference*. Springer, 34–63.

[8] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 2018. Labeled PSI from fully homomorphic encryption with malicious security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1223–1237.

[9] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1243–1255.

[10] Eduardo Chielle and Michail Maniatakos. 2025. Recurrent Private Set Intersection for Unbalanced Databases with Cuckoo Hashing and Leveled FHE. In *Network and Distributed System Security (NDSS) Symposium*.

[11] Wutichai Chongchitmate, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. 2022. PSI from ring-OLE. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 531–545.

[12] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. 2021. Labeled PSI from homomorphic encryption with reduced computation and communication. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1135–1150.

[13] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. 2021. Silver: silent VOLE and oblivious transfer from hardness of decoding structured LDPC codes. In *Annual International Cryptology Conference*. Springer, 502–534.

[14] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. 2004. Efficient private matching and set intersection. In *International conference on the theory and applications of cryptographic techniques*. Springer, 1–19.

[15] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2021. Oblivious key-value stores and amplification for private set intersection. In *Annual International Cryptology Conference*. Springer, 395–425.

[16] Oded Goldreich. 2001. *Foundations of cryptography: volume 2, basic applications*. Vol. 2. Cambridge university press.

[17] Meng Hao, Weiran Liu, Liqiang Peng, Hongwei Li, Cong Zhang, Hanxiao Chen, and Tianwei Zhang. 2024. Unbalanced {Circuit-PSI} from Oblivious {Key-Value} Retrieval. In *33rd USENIX Security Symposium (USENIX Security 24)*. 6435–6451.

[18] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. 2020. On deploying secure computing: Private intersection-sum-with-cardinality. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 370–389.

[19] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. 2017. Private intersection-sum protocol with applications to attributing aggregate ad conversions. *Cryptology ePrint Archive* (2017).

[20] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending oblivious transfers efficiently. In *Annual International Cryptology Conference*. Springer, 145–161.

[21] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. 2019. Mobile private contact discovery at scale. In *28th USENIX Security Symposium (USENIX Security 19)*. 1447–1464.

[22] Ágnes Kiss, Jian Liu, Thomas Schneider, N Asokan, and Benny Pinkas. 2017. Private set intersection for unequal set sizes with mobile applications. In *Privacy Enhancing Technologies Symposium*. De Gruyter, 177–197.

[23] 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.

[24] Guowei Ling, Peng Tang, Shi-Feng Sun, and Weidong Qiu. 2026. Efficient updatable PSI from asymmetric PSI and PSU. *IEEE Transactions on Information Forensics and Security* (2026).

[25] Junxin Liu, Peihan Miao, Mike Rosulek, Xinyi Shi, and Jifeng Wang. 2026. Updatable private set intersection from symmetric-key techniques. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 376–407.

[26] Rasoul Akhavan Mahdavi, Nils Lukas, Faezeh Ebrahimianghazani, Thomas Humphries, Bailey Kacsmar, John Premkumar, Xinda Li, Simon Oya, Ehsan Amjadian, and Florian Kerschbaum. 2024. {PEPSI}: Practically Efficient Private Set Intersection in the Unbalanced Setting. In *33rd USENIX Security Symposium (USENIX Security 24)*. 6453–6470.

[27] Atsuko Miyaji, Kazuhisa Nakasho, and Shohei Nishida. 2017. Privacy-preserving integration of medical data: a practical multiparty private set intersection. *Journal of medical systems* 41, 3 (2017), 37.

[28] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2019. SpOT-light: lightweight private set intersection from sparse OT extension. In *Annual International Cryptology Conference*. Springer, 401–431.

[29] 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.

[30] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. 2009. Secure two-party computation is practical. In *International conference on the theory and application of cryptology and information security*. Springer, 250–267.

[31] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. 2019. Efficient circuit-based PSI with linear communication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 122–153.

[32] 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.

[33] 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.

[34] 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.

[35] Amanda C Davi Resende and Diego F Aranha. 2018. Faster unbalanced private set intersection. In *International Conference on Financial Cryptography and Data Security*. Springer, 203–221.

[36] 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.

[37] Yongha Son and Jinhyuck Jeong. 2023. PSI with computation or Circuit-PSI for Unbalanced Sets from Homomorphic Encryption. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. 342–356.

[38] Binbin Tu, Yu Chen, Qi Liu, and Cong Zhang. 2023. Fast unbalanced private set union from fully homomorphic encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2959–2973.

[39] Ruochen Wang, Jun Zhou, Zhenfu Cao, Xiaolei Dong, and Kim-Kwang Raymond Choo. 2024. Updatable private set intersection with forward privacy. *IEEE Transactions on Information Forensics and Security* (2024).

[40] Jiadi Zhang, Hao Wang, Ye Su, Zhi Li, and Debiao He. 2026. DMPF-PSI: Enabling High-Frequency Updatable Private Set Intersection on Dynamic Data. *IEEE Transactions on Information Forensics and Security* (2026).

A Basic Protocol for the FHE-Based UPSI Instantiation under Leopard

Figures 13 and 14 illustrate the basic protocol for the FHE-based UPSI instantiation under Leopard. The full version incorporates the permuted-matrix PEQT (pm-PEQT) building block proposed by [38], denoted as $\Pi_{pv-PEQT}$ in the basic version. Also, the full version employs several performance optimization techniques as described in [12, 38], including hash-to-bin, batching for Single Instruction Multiple Data (SIMD) [7], windowing, partitioning, and modulus switching.

Basic Protocol: FHE-Based Instantiation of Leopard

Initialization:
Receiver $\mathcal{R}$ and sender $\mathcal{S}$ generate key pairs $(\mathsf{pk}_\mathcal{R}, \mathsf{sk}_\mathcal{R}) \leftarrow \mathtt{FHE.KeyGen}(1^\lambda)$, $(\mathsf{pk}_\mathcal{S}, \mathsf{sk}_\mathcal{S}) \leftarrow \mathtt{FHE.KeyGen}(1^\lambda)$, with public keys $\mathsf{pk}_\mathcal{R}, \mathsf{pk}_\mathcal{S}$ in the open.
***Day** 1:*
Inputs:
$\mathcal{R}$ and $\mathcal{S}$ have initial input sets X_1, Y_1.
Offline Processing:
For receiver $\mathcal{R}$:
$\mathcal{R}$ locally computes initial data structure $P_{X_1}(z) = \Pi_{i=1}^{|X_1|}(z - x_i)$.
For sender $\mathcal{S}$:
$\mathcal{S}$ locally computes initial data structure $P_{Y_1}(z) = \Pi_{j=1}^{|Y_1|}(z - y_j)$.
Online Execution:
$\Pi_{\mathtt{PSI}}$ invocation: $I_1 = X_1 \cap Y_1$.
Outputs:
$\mathcal{R}$ outputs $P_{X_1}(z), I_1$. $\mathcal{S}$ outputs $P_{Y_1}(z)$.
***Day** d:*
Pre-holding:
$\mathcal{R}$ holds X_{d-1}, $P_{X_{d-1}}(z)$, I_{d-1}. $\mathcal{S}$ holds $Y_{d-1}, P_{Y_{d-1}}(z)$.
Inputs:
$\mathcal{R}$ and $\mathcal{S}$ have new input sets X_d^-, Y_d^- which contain elements to be deleted from their sets, and X_d^+, Y_d^+ of elements to be inserted.
Offline Pre-processing:
For receiver $\mathcal{R}$:
1. $\mathcal{R}$ locally computes $\widetilde{X_d^+} = \mathtt{FHE.Enc}(X_d^+, \mathsf{pk}_\mathcal{R})$.
2. $\mathcal{R}$ locally computes $X_m = X_{d-1} \setminus X_d^-, X_d = X_m \cup X_d^+$.
3. $\mathcal{R}$ locally computes:
   $P_{X_m}(z) = \Pi_{k=1}^{|X_m|}(z - x_k) = P_{X_{d-1}}(z)/P_{X_d^-}(z)$, in which
   $P_{X_d^-}(z) = \Pi_{l=1}^{|X_d^-|}(z - x_l)$;
   $P_{X_d}(z) = \Pi_{s=1}^{|X_d|}(z - x_s) = P_{X_m}(z) \cdot P_{X_d^+}(z)$, in which
   $P_{X_d^+}(z) = \prod_{t=1}^{|X_d^+|}(z - x_t)$.

For sender $\mathcal{S}$:
1. $\mathcal{S}$ locally computes
   $\widetilde{Y_d^-} = \mathtt{FHE.Enc}(Y_d^-, \mathsf{pk}_\mathcal{S})$, $\widetilde{Y_d^+} = \mathtt{FHE.Enc}(Y_d^+, \mathsf{pk}_\mathcal{S})$.
2. $\mathcal{S}$ locally computes $Y_d = (Y_{d-1} \setminus Y_d^-) \cup Y_d^+$.

Figure 13: Basic Protocol of the UPSI Instance under Leopard Based on [12, 38].

[7] The inclusion of OPRF is necessary to prevent partial item leakage under the semi-honest security model, as discussed in [12].

Basic Protocol (Cont.)

For sender $\mathcal{S}$ (Cont.):
3. $\mathcal{S}$ locally computes
   $P_{Y_d}(z) = \Pi_{f=1}^{|Y_d|}(z - y_f) = (P_{Y_{d-1}}(z)/P_{Y_d^-}(z)) \cdot P_{Y_d^+}(z)$, in which $P_{Y_d^-}(z) = \Pi_{g=1}^{|Y_d^-|}(z - y_g)$ and $P_{Y_d^+}(z) = \Pi_{h=1}^{|Y_d^+|}(z - y_h)$.

Online Execution:
1. $\Pi_{\mathtt{raPSI}}$ invocation 1:
   (a) $\mathcal{S}$ transmits $\widetilde{Y_d^-}$ to $\mathcal{R}$.
   For each $\widetilde{y_u} \in \widetilde{Y_d^-}$, carry out steps (b) and (c):
   (b) $\mathcal{R}$ randomly selects $r_u \xleftarrow{\$} \mathbb{Z}_q \setminus \{0\}$, and computes $\widetilde{r_u} = \mathtt{FHE.Enc}(r_u, \mathsf{pk}_\mathcal{S})$.
   (c) $\mathcal{R}$ homomorphically computes $\widetilde{c_u} = \widetilde{r_u} + P_{X_m}(\widetilde{y_u})$.
   (d) $\mathcal{R}$ transmits $\widetilde{\mathbf{c}} = (\widetilde{c_1}, \widetilde{c_2}, \ldots, \widetilde{c_{|Y_d^-|}})$ to $\mathcal{S}$.
   (e) $\mathcal{S}$ locally decrypts $\widetilde{\mathbf{c}}$ and obtains $\mathbf{p} = \mathtt{FHE.Dec}(\widetilde{\mathbf{c}}, \mathsf{sk}_\mathcal{S}) = (p_1, p_2, \ldots, p_{|Y_d^-|})$.
   (f) $\mathcal{S}$ locally selects a random permutation π_1 on vector $\mathbf{p}$.
   (g) $\mathcal{S}$ takes $\mathbf{p}$ and π_1 as inputs; $\mathcal{R}$ takes $\mathbf{r} = (r_1, r_2, \ldots, r_{|Y_d^-|})$ as input; they together invoke $\Pi_{\mathtt{pv-PEQT}}$; $\mathcal{R}$ obtains $\mathbf{b} = (b_1, b_2, \ldots, b_{|Y_d^-|})$.
   (h) $\mathcal{S}$ takes $((\perp, y_{\pi_1(u_1)}), (\perp, y_{\pi_1(u_2)}), \ldots, (\perp, y_{\pi_1(u_{|Y_d^-|})}))$ as input, with $\pi_1(u_n) = n$ and $y_{\pi_1(u_n)} = y_n$; $\mathcal{R}$ takes $\mathbf{b}$ as input; they together invoke $\Pi_{\mathtt{OTe}}$; $\mathcal{R}$ obtains the intermediate intersection $S_1 = \cup_{n=1}^{|Y_d^-|}\{y_{\pi_1(u_n)} : b_n = 1\}$.
2. $\Pi_{\mathtt{raPSI}}$ invocation 2:
   (a) $\mathcal{S}$ transmits $\widetilde{Y_d^+}$ to $\mathcal{R}$.
   For each $\widetilde{y_v} \in \widetilde{Y_d^+}$, carry out steps (b) and (c):
   (b) $\mathcal{R}$ randomly selects $r_v \xleftarrow{\$} \mathbb{Z}_q \setminus \{0\}$, and computes $\widetilde{r_v} = \mathtt{FHE.Enc}(r_v, \mathsf{pk}_\mathcal{S})$.
   (c) $\mathcal{R}$ homomorphically computes $\widetilde{c_v} = \widetilde{r_v} + P_{X_m}(\widetilde{y_v})$.
   (d) $\mathcal{R}$ transmits $\widetilde{\mathbf{c}} = (\widetilde{c_1}, \widetilde{c_2}, \ldots, \widetilde{c_{|Y_d^+|}})$ to $\mathcal{S}$.
   (e) $\mathcal{S}$ locally decrypts $\widetilde{\mathbf{c}}$ and obtains $\mathbf{p} = \mathtt{FHE.Dec}(\widetilde{\mathbf{c}}, \mathsf{sk}_\mathcal{S}) = (p_1, p_2, \ldots, p_{|Y_d^+|})$.
   (f) $\mathcal{S}$ locally selects a random permutation π_2 on vector $\mathbf{p}$.
   (g) $\mathcal{S}$ takes $\mathbf{p}$ and π_2 as inputs; $\mathcal{R}$ takes $\mathbf{r} = (r_1, r_2, \ldots, r_{|Y_d^+|})$ as input; they together invoke $\Pi_{\mathtt{pv-PEQT}}$; $\mathcal{R}$ obtains $\mathbf{b} = (b_1, b_2, \ldots, b_{|Y_d^+|})$.
   (h) $\mathcal{S}$ takes $((\perp, y_{\pi_2(v_1)}), (\perp, y_{\pi_2(v_2)}), \ldots, (\perp, y_{\pi_2(v_{|Y_d^+|})}))$ as input, with $\pi_2(v_o) = o$ and $y_{\pi_2(v_o)} = y_o$; $\mathcal{R}$ takes $\mathbf{b}$ as input; they together invoke $\Pi_{\mathtt{OTe}}$; $\mathcal{R}$ obtains the intermediate intersection $S_2 = \cup_{o=1}^{|Y_d^+|}\{y_{\pi_2(v_o)} : b_o = 1\}$.
3. $\Pi_{\mathtt{aPSI}}$ invocation:
   (a) $\mathcal{R}$ transmits $\widetilde{X_d^+}$ to $\mathcal{S}$.
   For each $\widetilde{x_w} \in \widetilde{X_d^+}$, carry out steps (b) and (c):
   (b) $\mathcal{S}$ randomly selects $r_w \xleftarrow{\$} \mathbb{Z}_q \setminus \{0\}$.
   (c) $\mathcal{S}$ homomorphically computes $\widetilde{c_w} = r_w \cdot P_{Y_d}(\widetilde{x_w})$.
   (d) $\mathcal{S}$ transmits $\widetilde{\mathbf{c}} = (\widetilde{c_1}, \widetilde{c_2}, \ldots, \widetilde{c_{|X_d^+|}})$ to $\mathcal{R}$.
   (e) $\mathcal{R}$ locally decrypts $\widetilde{\mathbf{c}}$ and obtains $\mathbf{p} = \mathtt{FHE.Dec}(\widetilde{\mathbf{c}}, \mathsf{sk}_\mathcal{R}) = (p_1, p_2, \ldots, p_{|X_d^+|})$.
   (f) $\mathcal{R}$ locally computes the intermediate intersection $S_3 = \cup_{w=1}^{|X_d^+|}\{x_w : p_w = 0\}$.

Offline Post-processing:
For receiver $\mathcal{R}$:
1. $\mathcal{R}$ locally computes the updated intersection $I_d = [I_{d-1} \setminus (X_d^- \cup S_1)] \cup (S_2 \cup S_3)$.

Outputs:
$\mathcal{R}$ outputs $X_d, P_{X_d}(z), I_d$. $\mathcal{S}$ outputs $Y_d, P_{Y_d}(z)$.

Figure 14: Basic Protocol of the UPSI Instance under Leopard Based on [12, 38] (Cont.).