Private Set Intersection in Two Extended Settings

Max Tromanhauser
Brown University
Advisor: Peihan Miao

1 Introduction

Consider a setting where two parties have related data sets and wish to find elements which are shared between them — e.g., a server contains a database of passwords that have been leaked and a client has a list of their own passwords. One or both parties want to learn the common elements, but neither wants to reveal any additional information about their set. The parties can use private set intersection (PSI), a cryptographic protocol with this security guarantee.

This project has focused on two setting within the PSI literature: updatable PSI and unbalanced PSI. In updatable PSI, parties run the protocol repeatedly, each time making small updates to their input sets. Each instance of the protocol should run proportionally to the size of the update, rather than the larger sets. Unbalanced PSI captures applications where the cardinalities of the inputs are significantly different. In this setting, the protocol should be efficient with respect to the smaller of the two sets.

In this preamble, I will first describe my contributions to these papers, provide an overview of their results, and finally comment on future directions for these lines of inquiry.

2 Contributions

This project resulted in two submissions. In Finding Balance in Unbalanced PSI, my main contribution was the experimental evaluation. This required implementing the protocol in C++ and Go, integrating it with the existing SimplePIR library, and running experiments in a controlled environment.

In Updatable Private Set Intersection Revisited, in addition to implementation work, I made several theoretical contributions as we designed our protocols. I contributed to the formalization of the constructions and wrote several of the proofs of security for the final submission. I also worked alongside two other students to implement the protocols in C++ and compare our results to previous work.

3 Results

In the unbalanced setting, we demonstrated that a framework that combines oblivious pseudorandom functions (OPRF) with private information retrieval (PIR) can achieve unbalanced PSI with a balanced trade off between offline storage and online runtime. In concrete evaluation, our advantage in runtime is shown to be strongest in settings which are extremely unbalanced — particularly when the smaller party’s set is only a single element.

In the second paper, our protocol achieved expanded functionalities that had not been demonstrated before in the updatable setting. Rather than simply outputting the intersection itself, we can output the cardinality of the intersection or a secret sharing of that intersection. Alternatively, if one party has a set of values that are associated with their input elements, the parties can compute the sum of values connected with elements in the intersection. These functionalities are often more desirable in practical applications. Additionally, we were able to support arbitrary deletion of elements, where previous works had relied on a weaker notion where elements were always deleted after $t$ days.

4 Future Work

In both settings, there are opportunities to improve PSI protocols in efficiency and functionality. Our framework for unbalanced PSI was agnostic to the underlying OPRF and PIR protocols, so improvements in either of those fields may lead to improvements to our PSI construction. Our use of cuckoo hashing to support multiple elements in the client set requires that the server’s offline phase is dependent on the size of the client set. Any technique which removes this dynamic may result in a significant improvement in efficiency in cases where the input sets are closer in size.

In updatable PSI, arbitrary deletion requires an additional overhead of $O(\log N)$ where $N$ is the size of the entire set being updated. This is the result of a loose bound on a stochastic procedure used in the protocol, so a tightening of this bound may result an asymptotic improvement.
Finding Balance in Unbalanced PSI:
A New Construction from Single-Server PIR

Chengyu Lin\(^3\), Zeyu Liu\(^2\), Peihan Miao\(^3\) and Max Tromanhauser\(^1\)

\(^1\) Brown University  
\(^2\) Yale University  
\(^3\) Espresso Systems

Abstract. Private set intersection (PSI) enables two parties to jointly compute the intersection of their private sets without revealing any extra information to each other. In this work, we focus on the unbalanced setting where one party (a powerful server) holds a significantly larger set than the other party (a resource-limited client). We present a new protocol for this setting that achieves a better balance between low client-side storage and efficient online processing.

We first formalize a general framework to transform Private Information Retrieval (PIR) into PSI implicitly used in prior works. Building upon recent advancements in Private Information Retrieval (PIR), specifically the SimplePIR construction (Henzinger et al., USENIX Security’23), combined with our tailored techniques, our construction shows a great improvement in online efficiency. Concretely, when the client holds a single element, our protocol achieves more than 100× faster computation and over 4× lower communication compared to the state-of-the-art unbalanced PSI based on leveled fully homomorphic encryption (Chen et al., CCS’21). The client-side storage is only in the order of tens of megabytes, even for a gigabyte-sized set on the server. Moreover, since the framework is generic, any future improvement in PIR can further improve our construction.

1 Introduction

Consider two parties, each holding a private set of elements, who want to learn the intersection of the two sets without revealing any other information to each other. For example, two companies may want to identify their common customers, or an ad platform and an advertiser may want to determine which consumers who viewed an ad ended up making a purchase.

The above problem can be formulated as **private set intersection (PSI)**, which refers to a specialized secure two-party computation protocol that takes two private sets \(X, Y\) as input and outputs their intersection \(X \cap Y\) to one or both of the participating parties. Over the years, PSI has found numerous applications in practice, including DNA testing and pattern matching [TKC07], remote diagnostics [BPSW07], online advertising [IKN\(^+\)20, MPR\(^+\)20], password breach monitoring [TPY\(^+\)19, Ali18, LKLM21, APP21], mobile private contact discovery [DRRT18, KRS\(^+\)19, Mar14, HWS\(^+\)21], privacy-preserving contact tracing for infectious diseases [TSS\(^+\)20, CCF\(^+\)20], and many more. Tremendous progress has been made towards realizing PSI efficiently [KKRT16, RR17, CLR17, PSWW18, PRTY19, PSTY19, CM20, PRTY20, GPR\(^+\)21, CMdG\(^+\)21, RS21].

Most work in PSI focuses on the **balanced** setting where the input sets are similarly sized, which is not the best fit for some real-world use cases such as password breach monitoring and private contact discovery on mobile devices. In these applications, the input set of the service provider is significantly larger than the input set of the user,
Finding Balance in Unbalanced PSI:
A New Construction from Single-Server PIR

sometimes by a factor of millions if not billions. If we apply the PSI protocols designed for
the balanced setting, both the computation and communication complexity would grow
linearly with the size of the larger set. This can be highly prohibitive, especially for a user
with limited resources (e.g., a mobile phone or a wearable device).

To accommodate these applications, techniques have been developed for the unbal-
canced setting with one-sided output. In particular, a server holding a large set \( X \)
interacts with a resource-limited client holding a small set \( Y \) in a PSI protocol, where only
the client learns the intersection \( X \cap Y \) and the server learns nothing.

The existing work on unbalanced PSI with one-sided output can be categorized into two
approaches: those based on oblivious pseudorandom function (OPRF) [FIPR05, PSSW09,
KLS+17, RA18, KRS+19] and those based on leveled fully homomorphic encryption (FHE)
[CLR17, CHLR18, CMdG+21]. They both follow a common paradigm, where the server
first performs a one-time, offline pre-processing step on its set \( X \) and possibly sends
some pre-processed data to the client, which is stored on the client side. Once the client
determines its set \( Y \), it can initiate the online phase by sending a PSI query to the server.
For practicality, the online computation and communication costs are typically much lower
compared to the pre-processing phase.

When evaluating the practical efficiency of PSI protocols in this paradigm, two important
metrics to consider are 1) the client’s storage requirement after the pre-processing phase,
and 2) the online processing speed. Looking at prior work, the OPRF-based constructions
achieve fast computation and low communication in the online phase. However, they
usually require the client to have a large storage of size \( O(|X|) \) from pre-processing. In
contrast, most constructions based on FHE do not require client offline storage, but the
online processing speed is much slower due to the heavy online computation and high
communication overhead, especially on the server side.

Can we achieve a balanced approach that has both sublinear client storage and
fast online processing?

1.1 Our Results

In this work, we make positive progress towards addressing the above question by presenting
a new PSI protocol that achieves a better balance between these two extremes. Our
protocol for unbalanced PSI with one-sided output strikes a favorable trade-off, requiring
low (although non-zero) local storage on the client side, while achieving significantly lower
online costs compared to the FHE-based constructions.

Approaching unbalanced PSI through the lens of PIR. Private information retrieval
(PIR) [CGKS95] is another important cryptographic primitive that shares similarities
with unbalanced PSI in terms of the unbalanced input size. Informally, PIR allows
a client to retrieve a particular entry from a database stored on the server without
revealing any information about its query to the server. Notably, a recent line of work
[CK20, SACM21, CHK22, LMW22, LP23] adopts the offline/online paradigm, where the
server pre-processes the database in the offline phase to enable efficient online query
processing.

In this work, we leverage these recent advancements in PIR to achieve similar trade-offs
in unbalanced PSI. We summarize the technical ideas behind our results below and give a
more detailed construction overview in section 3.

Generic construction using PIR and OPRF. There are two key challenges in
constructing PSI from PIR.

The first is the difference in their functionalities. In PSI, the client wants to learn if
a particular element \( y \) is in the server’s set \( X \), while in PIR, the client wants to retrieve
a particular entry from the server’s database. This gap can be bridged by a variant of
PIR known as PIR by Keyword or Keyword PIR [CGN98], where the server holds a set of elements $X$ and the client holds a single element $y$. The client wants to learn whether $y \in X$ without revealing any information about $y$ to the server, which is precisely what we need for PSI.\footnote{Note that in the recent work by Patel et al. [PSY23], Keyword PIR refers to the setting where the client wants to retrieve a data entry associated with $y$ without revealing $y$ to the server, differing from the original definition in [CGN98]. For simplicity, we use the original definition.}

The second challenge comes from the difference in the security guarantees: while PSI requires privacy for both parties, PIR only protects client privacy. This can be solved by leveraging an oblivious pseudorandom function (OPRF) as in [FIPR05]. The server first samples a key $k$ for a PRF $F_k(\cdot)$ and evaluates the PRF on all its elements to obtain $X' := \{F_k(x) \mid x \in X\}$. Next, the client engages in an OPRF protocol with the server to learn all the PRF evaluations on the client’s elements, namely $Y' = \{F_k(y) \mid y \in Y\}$, without leaking any information about $Y$ to the server. Now the original PSI problem is reduced to a new PSI problem for $X' \cap Y'$, but we no longer need to protect sender privacy due to the security guarantees of OPRF.

**Concrete instantiations and practical efficiency.** To leverage recent advancements in PIR, we build our construction using the state-of-the-art single-server offline/online PIR construction SimplePIR [HHC+22] and the Diffie-Hellman-based OPRF [HFH99, JL10]. Asymptotically, our construction requires an offline storage of $O(\sqrt{|X|})$ on the client side and an online communication of $O(\sqrt{|X|})$, which follows from the SimplePIR construction.

To further enhance the practicality of our construction, we construct keyword PIR on OPRF values in a way that introduces essentially no overhead to the underlying SimplePIR protocol. Notably, we are able to construct our keyword PIR from SimplePIR using only a simple hash to avoid pre-processing overhead. Moreover, we provide various techniques tailored for SimplePIR to optimize our concrete efficiency, including re-arranging the database, modulus switching, and tight parameter analysis.

Our protocol is particularly efficient when the server’s set is significantly larger than the client’s set (e.g., when the ratio $|X|/|Y| \geq 2^{20}$). Concretely, when $|Y| = 1$ and $|X|$ is in the range of $2^{20} - 2^{28}$, our construction is more than 100× faster than the state-of-the-art FHE-based PSI [CMdG+21] in terms of online computation and also achieves more than 4× improvement in online communication. Our offline computation remains comparable to prior works, but requires an extra offline communication and client storage. However, this requirement is small: only tens of megabytes for a database that is thousands of megabytes large. For $|Y| > 1$, our protocol does not have an as obvious advantage as the $|Y| = 1$ case, but the online computation is still about one order of magnitude faster than the prior work. We additionally give an estimation of how our construction can be applied to applications such as password breach checkup. Concretely, it costs more than two orders of magnitude less than prior constructions.

**Achieving malicious security almost for free.** Our basic construction is secure against semi-honest adversaries, who follow the protocol description honestly while trying to extract more information from the protocol execution. We show how to strengthen the security guarantees to protect against malicious adversaries, who may arbitrarily deviate from the protocol description, with minimal modifications to our semi-honest protocol. To achieve this, we develop proof techniques against a malicious server via a novel use of rewinding.

**Extensions.** We highlight a few extensions to our construction. First, we explore techniques to remove offline storage as well as reduce the round complexity in the online phase. Second, we can extend our construction to achieve labeled PSI [CHLR18], where the server holds associated values $v_i$ for each element $x_i \in X$, and the client learns the values associated with the elements in the intersection, namely $\{v_i \mid x_i \in X \cap Y\}$. The server holds associated values $v_i$ for each element $x_i \in X$, and the client learns the values associated with the elements in the intersection, namely $\{v_i \mid x_i \in X \cap Y\}$. The second challenge comes from the difference in the security guarantees: while PSI requires privacy for both parties, PIR only protects client privacy. This can be solved by leveraging an oblivious pseudorandom function (OPRF) as in [FIPR05]. The server first samples a key $k$ for a PRF $F_k(\cdot)$ and evaluates the PRF on all its elements to obtain $X' := \{F_k(x) \mid x \in X\}$. Next, the client engages in an OPRF protocol with the server to learn all the PRF evaluations on the client’s elements, namely $Y' = \{F_k(y) \mid y \in Y\}$, without leaking any information about $Y$ to the server. Now the original PSI problem is reduced to a new PSI problem for $X' \cap Y'$, but we no longer need to protect sender privacy due to the security guarantees of OPRF.

**Concrete instantiations and practical efficiency.** To leverage recent advancements in PIR, we build our construction using the state-of-the-art single-server offline/online PIR construction SimplePIR [HHC+22] and the Diffie-Hellman-based OPRF [HFH99, JL10]. Asymptotically, our construction requires an offline storage of $O(\sqrt{|X|})$ on the client side and an online communication of $O(\sqrt{|X|})$, which follows from the SimplePIR construction.

To further enhance the practicality of our construction, we construct keyword PIR on OPRF values in a way that introduces essentially no overhead to the underlying SimplePIR protocol. Notably, we are able to construct our keyword PIR from SimplePIR using only a simple hash to avoid pre-processing overhead. Moreover, we provide various techniques tailored for SimplePIR to optimize our concrete efficiency, including re-arranging the database, modulus switching, and tight parameter analysis.

Our protocol is particularly efficient when the server’s set is significantly larger than the client’s set (e.g., when the ratio $|X|/|Y| \geq 2^{20}$). Concretely, when $|Y| = 1$ and $|X|$ is in the range of $2^{20} - 2^{28}$, our construction is more than 100× faster than the state-of-the-art FHE-based PSI [CMdG+21] in terms of online computation and also achieves more than 4× improvement in online communication. Our offline computation remains comparable to prior works, but requires an extra offline communication and client storage. However, this requirement is small: only tens of megabytes for a database that is thousands of megabytes large. For $|Y| > 1$, our protocol does not have an as obvious advantage as the $|Y| = 1$ case, but the online computation is still about one order of magnitude faster than the prior work. We additionally give an estimation of how our construction can be applied to applications such as password breach checkup. Concretely, it costs more than two orders of magnitude less than prior constructions.

**Achieving malicious security almost for free.** Our basic construction is secure against semi-honest adversaries, who follow the protocol description honestly while trying to extract more information from the protocol execution. We show how to strengthen the security guarantees to protect against malicious adversaries, who may arbitrarily deviate from the protocol description, with minimal modifications to our semi-honest protocol. To achieve this, we develop proof techniques against a malicious server via a novel use of rewinding.

**Extensions.** We highlight a few extensions to our construction. First, we explore techniques to remove offline storage as well as reduce the round complexity in the online phase. Second, we can extend our construction to achieve labeled PSI [CHLR18], where the server holds associated values $v_i$ for each element $x_i \in X$, and the client learns the values associated with the elements in the intersection, namely $\{v_i \mid x_i \in X \cap Y\}$. The server holds associated values $v_i$ for each element $x_i \in X$, and the client learns the values associated with the elements in the intersection, namely $\{v_i \mid x_i \in X \cap Y\}$.
Finally, we emphasize that further usage of the generic PSI construction may be of independent interest since it can be instantiated with any OPRF and PIR constructions. Therefore, future advancements in these primitives can directly improve the efficiency of our PSI construction. To show this, we make an estimation for PSI from other keyword PIR protocols [PSY23] using the framework we formalize. It also shows an advantage over the prior constructions, while having different trade-offs compared to our construction.

Our contributions. To summarize, we

- formalize the framework for constructing unbalanced PSI with one-sided output from OPRF and PIR;
- instantiate our PSI construction with SimplePIR and develop novel techniques for concrete efficiency;
- implement our protocol and demonstrate performance improvement compared with prior work, showing that for applications like password breach checkup, our construction offers an appealing solution (only seconds to check against a database with \(2^{32}\) passwords, compared to hundreds of seconds using prior constructions);
- provide formal security proofs for our protocols, introducing new techniques for proving malicious security through minimal modifications to our semi-honest protocol; and
- present various extensions to our protocol achieving achieving expanded functionalities and better flexibility.

1.2 Related Work

Unbalanced PSI from OPRF. The first PSI protocol based on the Oblivious Pseudorandom Function (OPRF) was proposed by Freedman et al. [FIPR05]. In their work, they instantiated the OPRF using the renowned Noar-Reingold (NR) pseudorandom function [NR97]. Subsequently, Pinkas et al. [PSSW09] utilized an AES-based OPRF and garbled circuits (GC) [Yao86] to construct another PSI protocol. Building on these developments, Kiss et al. [KLS+17] and Davi Resende and Aranha [RA18] made further contributions, and the state-of-the-art OPRF-based protocol was presented by Kales et al. [KRS+19].

To provide an overview of this line of research, we explain the high-level idea here. Initially, the server generates a secret OPRF key. During the offline/pre-processing stage, the server transmits the OPRF values of all the elements in its larger set to the client through a hash table, usually a Cuckoo filter [FAKM14]. In the subsequent online phase, the client determines the intersection by first evaluating the OPRF values of all the elements in its smaller set and then verifying them against the hash table (or a Cuckoo filter) received earlier. It is important to note that such protocols typically involve linear communication in the server’s set during the pre-processing phase and linear communication in the client’s set during the online phase.

Unbalanced PSI from FHE. Another line of work on unbalanced PSI is based on leveled FHE [CLR17, CHLR18, CMdG+21]. These works achieve linear communication in the client’s set and logarithmic in the server’s set. Thus, the local storage requirement of the client is minimized. All of these works are based on the BFV/BGV homomorphic encryption schemes [Bra12, FV12, BGV14] and thus result in a relatively large overhead in terms of online computation and communication.

Unbalanced PSI from PIR. To the best of our knowledge, [DRRT18] is the only work that directly constructs unbalanced PSI from PIR. However, they rely on two-server PIR, which requires two non-colluding servers holding the same database. Their construction uses the underlying PIR in a non-black-box way. Thus, they have a stronger environmental
assumption and are less generic (as switching their underlying PIR construction requires a non-black-box change). Furthermore, their online communication is relatively large: for a database of size hundreds of megabytes, their online communication for a single PSI query is tens of megabytes.

**PIR schemes.** Achieving concretely efficient PIR constructions for practical applications has been an active area of research [DC14, KLL+15, GLM16, ABFK16, ACLS18, GH19, PT20, ALP+21a, MCR21, MW22]. Classic protocols have no offline phase, and thus have relatively limited online efficiency.

A recent line of work on offline/online PIR [CK20, SACM21, KC21, CHK22, LP22, LMW22, HHC+22, ZLTS23, LP23, ZPSZ23] take advantage of offline pre-processing together with client storage. The server pre-processes the database and sends some processed data called hint to the client. Later, the client uses the hint to query for the data entry and achieves high online efficiency. SimplePIR [HHC+22] along this line of work achieves the best concrete online efficiency for single-server PIR.

Doubly efficient PIR [BIM00, CHR17, BIPW17, LMW22] is another related line of work, where the server also performs an offline pre-processing step but does not require client-side storage. Instead, it takes advantage of extra storage on the server side to achieve better online efficiency. Recent work by Lin et al. [LMW22] has shown that with a pre-processing step of \(\tilde{O}(N)\) time, where \(N\) is the size of the database, a client can retrieve the entry with computation and communication cost both being \(\text{polylog}(N)\). However, since this work does not provide concrete efficiency but only serves as an asymptotic result (see [OPPW23] for its concrete performance), we do not employ it to realize our PSI protocol.

**Keyword PIR.** A generic transformation from PIR to keyword PIR was introduced along with its definition [CGN98] but required overhead proportional to the underlying search data structure. Recent progress has introduced a transformation without any overhead [PSY23] but that requires either \(O(N^2)\) extra pre-processing time for the server (where \(N\) is the database size) or \(O(N)\) extra pre-processing time along with enlarged client or server storage. We provide an estimation of how their keyword PIR construction performs compared to ours and prior works, when used for PSI, in section 5.

**OPRFs for server privacy.** Using OPRFs to achieve server-side privacy was introduced in [FIPR05] to construct a fully-private keyword search protocol from keyword PIR. In keyword search, a client wants to learn the value associated with a keyword in a server’s database (or \(\perp\) if it is not in the database), which can be reframed as labeled PSI.

## 2 Preliminaries

**Notation.** Let \([N]\) denote \(\{1, \ldots, N\}\). We use \(\kappa, \lambda\) to denote the computational and statistical security parameters, respectively. The logarithm always has base 2 unless otherwise specified. \(\negl(\cdot)\) denotes a negligible function, i.e., a function \(f\) such that \(f(n) < 1/p(n)\) holds for any polynomial \(p(n)\) and sufficiently large \(n\). \(\text{poly}(\cdot)\) denotes a polynomial function. \(\text{PPT}\) stands for “probabilistic polynomial time.” \(\log(\cdot)\) denotes a logarithmic function. \(\text{polylog}(\cdot)\) denotes a poly-logarithmic function. We omit a \(\text{polylog}(N)\) factor in \(O(\cdot)\), namely \(O(N) = O(N\text{polylog}(N))\). \(|x|\) denotes the size of \(x\), where \(x\) can be a set or a vector. Let \(x\) be a vector, \(x[i]\) denotes the \(i\)-th element of the vector. Let \(x \leftarrow \mathbb{Z}_q\) denote \(x\) being sampled uniformly at random from \(\mathbb{Z}_q\).

**Private Set Intersection (PSI).** PSI is a specialized secure two-party computation [Yao86]. We follow the standard ideal/real-world paradigm for defining secure two-party computation against semi-honest or malicious adversaries (see e.g., [Lin16] for the formal definitions). The ideal functionality of PSI is formalized in fig. 1.
Public Parameters. The honest server and client have respective set sizes $N$ and $M$. If the server is maliciously corrupted, then its set size is $N'$.

Inputs. The server $S$ inputs a set $X$ where $|X| = N$ if $S$ is honest and $|X| = N'$ otherwise. The client $C$ inputs a set $Y$ where $|Y| = M$.

Output. The client $C$ receives the set intersection $I = X \cap Y$ and the server $S$ receives $\perp$.

Figure 1: Ideal functionality for private set intersection.

Private Information Retrieval (PIR). We formalize an offline/online PIR protocol as follows. A server holds a database $T$ of $n$ data entries, and a client wants to access $T[i]$ for some $i \in [N]$. The server can pre-process the database during the offline phase, and send the pre-processed data hint to the client. During the online phase, the client sends some query $qry$ to the server. The server replies with $\text{rsp}$.

The correctness of PIR guarantees that with hint and $\text{rsp}$, the client can correctly recover $T[i]$. The receiver privacy of PIR guarantees that for any $i \neq i' \in [N]$, $qry$ for $i$ is indistinguishable from $qry$ for $i'$ to the server.

Decisional Diffie-Hellman (DDH) Assumption. Let $g$ be a generator of a group $\mathbb{G}$ of order $q$. The DDH problem is hard in $\mathbb{G}$ if for any PPT adversary $A$, $|\Pr[A(g^a, g^b, g^{ab}) = 1] - \Pr[A(g^a, g^{bh}, g^{b})]| \leq \text{negl}(\kappa)$, for the probability over the random $a, b, c \leftarrow \mathbb{Z}_q$.

One-More Gap Diffie-Hellman (OMGDH) Assumption. Let $g$ be a generator of a group $\mathbb{G}$ of order $q$. We say $(N, Q)$-OMGDH is hard in $\mathbb{G}$ if for any PPT adversary $A$, $|\Pr[(g_i, h_i)_{i \in [q + 1]} \leftarrow A^{\text{DDH}(\cdot \cdot \cdot)}(g_1, \ldots, g_N)] - \text{negl}(\kappa)|$, where the probability is over random $(g_1, \ldots, g_N) \leftarrow \mathbb{G}^N$ and $k \leftarrow \mathbb{Z}_q$; $(\cdot)^k$ is an oracle that takes any $h \in \mathbb{G}$ and returns $h^k$, and $A$ can call this oracle at most $Q$ times in parallel; $\text{DL}_k(\cdot)$ is an oracle that on input tuple $(g, h)$ returns $1$ if $h = g^k$ and $0$ otherwise.

Learning with Error (LWE). Let $n, q, \sigma$ and distribution $\mathcal{D}$ be LWE parameters, and let $\chi_\sigma$ denote a discrete Gaussian distribution with mean 0 and standard deviation of $\kappa$. LWE is hard if for any PPT adversary $A$, $|\Pr[A(\tilde{a}, u) = 1] - \Pr[A(\tilde{a}, (\tilde{a}, \tilde{s}) + e)]| \leq \text{negl}(\kappa)$, where probability is over $\tilde{a} \leftarrow \mathbb{Z}_q, u \leftarrow \mathbb{Z}_q, \tilde{s} \leftarrow \mathcal{D}$, and $e \leftarrow \chi_\sigma$.

Regev Encryption. The LWE Regev encryption scheme has an additional parameter $p$ as plaintext modulus. The encryption of a $\mathbb{Z}_p$ element $m$ under secret key $\tilde{s} \leftarrow \mathcal{D}$ is $(\tilde{a}, b \leftarrow (\tilde{a}, \tilde{s}) + e + m \cdot \Delta)$ where $\Delta = \lfloor q/p \rfloor, e \leftarrow \chi_\sigma$ and $\tilde{a} \leftarrow \mathbb{Z}_q$. With all but negligible probability, it can be correctly decrypted to $m = \lfloor \frac{b - \langle \tilde{a}, \tilde{s} \rangle}{\Delta} \rfloor$ as long as $\Pr[|e| > \lfloor \Delta/2 \rfloor] \leq \text{negl}(\kappa)$, which means $\text{err}(\frac{\Delta/2}{\sqrt{2\Delta}}) \leq \text{negl}(\kappa)$, where $\text{err}(\cdot)$ is the Gauss error function.

The LWE Regev encryption is linearly homomorphic. Let $(\tilde{a}, b)$ be the encryption of $m$ and $(\tilde{a}', b')$ be the encryption of $m'$. The encryption of $c \cdot m$ for any plaintext $c \in \mathbb{Z}_p$ can be obtained by a scalar multiplication $c \cdot (\tilde{a}, b) = (c \cdot \tilde{a}, c \cdot b)$. The encryption of $m + m'$ can be obtained by the entry-wise addition of the ciphertext vectors $(\tilde{a}, b) + (\tilde{a}', b') = (\tilde{a} + \tilde{a}', b + b')$. Note that both operations require the resulting error to remain sufficiently small.

3 Our PSI Protocol

In this section, we present our unbalanced PSI protocol with one-sided output. We give a construction overview in section 3.1 and discuss various optimization techniques when instantiating our protocol with SimplePIR in section 3.2. The protocols for clients holding a single element and multiple elements are presented in fig. 2 and fig. 3, respectively.
3.1 Construction Overview

Starting point. We start with the extremely unbalanced PSI problem where the client’s set contains a single element. Specifically, the server \( S \) holds a large set \( X \) of size \( N \) and the client \( C \) holds a single element \( y \). The client wants to learn whether \( y \in X \).

We first follow the OPRF-based PSI paradigm [FIPR05]. Specifically, the server \( S \) generates a secret key \( k \) for a pseudorandom function (PRF) \( F_k(\cdot) \) and sends all the PRF evaluations of its elements, \( X' := \{ F_k(x) | x \in X \} \), to the client \( C \). Afterwards, \( S \) and \( C \) engage in an OPRF protocol, which is a specialized secure two-party computation protocol, where \( C \) learns \( y' = F_k(y) \) and \( S \) learns nothing. Finally, \( C \) simply checks whether \( y' \in X' \).

By the security guarantees of OPRF, \( S \) learns nothing about \( y \) while \( C \) learns nothing about \( k \) beyond \( F_k(y) \), hence \( X' \setminus \{ y' \} \) is computationally indistinguishable from a random set. Therefore, \( C \) learns nothing other than whether \( y \in X \).

However, this protocol requires \( O(N) \) communication from the server to the client, which can be impractical for a large set \( X \). Moreover, if the OPRF evaluations \( X' \) are sent in the pre-processing phase, it would require significant storage on the client side.

Embedding keyword PIR. To address the issue above, we can utilize a variant of PIR named PIR by Keyword or Keyword PIR [CGN98] instead of requiring the server to send the entire set \( X' \). In keyword PIR, the server holds \( N \) elements \( S = \{ s_1, \ldots, s_N \} \) and the client holds a single element \( w \). The client wants to learn whether \( w = s_j \) for some \( j \leq N \) without revealing any information about \( w \) to the server. This primitive directly serves our purpose. In more detail, after \( S \) computes \( X' \) and \( C \) obtains \( y' \) from OPRF, \( C \) can make a keyword PIR query to learn whether \( y' \in X' \) without revealing \( y' \) to \( S \). The remaining challenge lies in constructing an efficient keyword PIR protocol.

Constructing keyword PIR from PIR. We can now plug any generic keyword PIR protocol into our framework. In section 5.2, we provide a performance estimation for unbalanced PSI from other keyword PIR constructions [PSY23], following our framework.

Nevertheless, we introduce an alternative approach to constructing keyword PIR from PIR, which is particularly tailored for optimal performance when instantiated with SimplePIR (see section 3.2). This approach can also achieve malicious security almost for free. Specifically, we construct keyword PIR from PIR in a black-box way using a hash function \( H : \{0,1\}^* \rightarrow [\tau] \) that maps elements into a hash table of size \( \tau \).

The server \( S \) first creates a hash table \( T \) of size \( \tau \), putting every element \( x' \in X' \) into the hash bin \( T[H(x')] \), namely \( T[\ell] := \{ x' | x' \in X' \wedge H(x') = \ell \} \) for all \( \ell \in [\tau] \). We can bound the maximum number of elements in any hash bin (with overwhelming probability), denoted by \( \gamma \). Then the server pads each hash bin with dummy elements to reach a size of \( \gamma \). We can now view \( T \) as a database consisting of \( \tau \) entries, where each entry contains \( \gamma \) elements. The client \( C \) then simply computes \( \ell^C := H(y') \) and makes a PIR query for \( T[\ell^C] \). Finally, \( C \) can conclude \( y' \in X' \) if and only if \( y' \in T[\ell^C] \). By the receiver security of PIR, \( S \) does not learn anything about \( y' \).

Regarding the parameters, we first set \( \tau = N = |X| \), which results in \( \gamma = O(\log N \log \log N) \). These parameters can be further tuned for improved performance, as discussed in section 3.2. Given any PIR with sublinear communication complexity in \( N \), we can achieve sublinear communication in our protocol as well.

Optimization: padding to the largest bin. In the above keyword PIR construction, we observe that the size of each hash bin in \( T \) does not reveal any information to the client. This is because the elements in the hash table are PRF values and hash locations are computed based on these PRF values, which can be sent directly to the client as in the OPRF-based PSI protocol. Therefore, there is no need for any padding in terms of security guarantees. For the PIR protocol to go through, it suffices to pad each bin to the

\[^2\text{As long as } \gamma = \omega(\log N), \text{ the probability that any bin exceeds the size of } \gamma \text{ is negligible.}\]
size of the actual largest bin in $T$ instead of the theoretical upper bound on the maximum size of any bin, which drastically reduces the size of the database for PIR. Moreover, the server can pad with 0-strings to further reduce the computational cost in PIR.

**The OPRF construction.** The missing component in our construction is the realization of OPRF. Prior works on PSI have proposed various types of OPRF constructions, including Noar-Reingold-based [FIPR05], Diffie-Hellman-based [HFH99, JL10], garbled circuit-based [PSSW09, KLS+17, RA18, KRS+19], and OT-based [KKRT16, PRTY19, CM20]. Our construction is generic and can work with any OPRF, but to best serve our PSI purpose, we look for OPRF constructions that satisfy the following properties: 1) the server’s OPRF key $k$ can be reused across multiple clients, 2) the protocol can easily be made maliciously secure, and 3) the protocol is practically efficient, especially in the online phase. Considering these factors, we choose the OPRF construction presented in [JL10].

In this construction, $S$ and $C$ agree on two hash functions $H_1 : \{0,1\}^* \rightarrow \mathbb{G}$ and $H_2 : \mathbb{G} \times \mathbb{G} \rightarrow \{0,1\}^k$. The PRF is computed as $F_k(x) := H_2(H_1(x), H_1(x)^k)$ for a randomly sampled key $k$. To jointly compute $F_k(y)$, $C$ randomly samples $k_c$ and sends $z := H_1(y)^{k_c}$ to $S$. $S$ then replies with $z' := z^k$. After getting $z'$ back, $C$ computes $z'' := H_2(z, (z')^{k_c^{-1}})$, which gives $F_k(y)$.

**Achieving malicious security.** The above protocol is semi-honest secure in the random oracle model assuming DDH is hard in $\mathbb{G}$. To enhance its security against malicious adversaries, $S$ only needs to attach a proof of knowledge (PoK) for the key $k$ along with its response $z'$, assuming OMGDH is hard in $\mathbb{G}$. Looking ahead, we can get rid of this PoK in our protocol when $|Y| = 1$. At a high level, the purpose of the PoK is to ensure that $S$ computes the response $z'$ correctly. We observe that $C$ outputs $y$ in the intersection if and only if the corresponding OPRF value $z''$ matches an OPRF value retrieved from $S$, irrespective of whether $z''$ is computed correctly or not. By leveraging this property, we can remove the overhead associated with the PoK while still maintaining security against a malicious server. The detailed proof is presented in section 4.2.2.

**Handling multiple elements in the client’s set.** Now we discuss the scenario where the client $C$ has multiple elements in its set, namely $C$ holds a set $Y$ of size $M \geq 1$. One straightforward approach is to apply the single-element PSI on every element in $Y$. However, this approach can be computationally expensive on the server side if the server’s online computational complexity in PIR grows linearly with the database size, which is the case in most PIR protocols.

To reduce the server’s computation cost, we adopt the technique of Cuckoo hashing [PR04]. Specifically, $S$ and $C$ agree on three hash functions $h_1, h_2, h_3 : [0,1]^* \rightarrow [m]$ to map elements into a hash table of size $m$. The client $C$ first creates a hash table of size $m$ and puts each element $y_i \in Y$ into one of three bins located at $\{h_1(y_i), h_2(y_i), h_3(y_i)\}$, ensuring that each hash bin contains at most one element. The Cuckoo hashing parameter $m$ is chosen such that this step fails with negligible probability. On the server side, $S$ also creates a hash table of size $m$ and puts each element $x_i \in X$ into all three bins located at $\{h_1(x_i), h_2(x_i), h_3(x_i)\}$. We can then bound the maximum number of elements in any hash bin and have the server pad each hash bin with dummy elements to reach that size. Finally, $S$ and $C$ run a single-element PSI protocol for each hash bin. As a result, the server’s total online computation remains linear in the database size.

**Optimization: no need for padding.** Consider the padding step in the above multi-element protocol. Now it is necessary, for security reasons, to pad each hash bin to the theoretical upper bound since the size of each bin could reveal information about the server’s set $X$. However, if we apply the Cuckoo hashing on the PRF values instead of the original elements, the same idea applies, and padding is no longer required.
We instantiate the above PSI construction based on SimplePIR \cite{hいかがし22} with an LWE Regev encryption scheme using preprocessing to boost online performance.

**Online Phase:** Communication does not directly fit the PSI applications we have in mind. We can view \texttt{qry} as a $\sqrt{N}$-by-$(n + 1)$ matrix where each row is given in plaintext form. Although the local storage of the client is not linear in $N$, this linear pre-processing communication does not directly fit the PSI applications we have in mind.

**Online Phase:**
1. C randomly samples $kc \leftarrow \mathbb{Z}_q$, computes $z := H_1(y)^{kc}$, and sends $z$ back to S.
2. Upon receiving $z$, S computes $z' := z^k$ and sends it back to C.
3. Upon receiving $z'$ back, C does the following:
   (a) Compute $z'' := H_2\left(H_1(y), (z')^{k\ell} \right)$ and $\ell^c := H_3(z'')$.
   (b) Prepare a PIR query for the $\ell^c$-th entry of the database $T$ and send it to S.
4. Upon receiving the PIR query, S computes the PIR response and sends it back to C.
5. Upon receiving the PIR response, C does the following:
   (a) Recover the entire $\ell^c$-th entry of $T$ as a set of strings $R = \{r_1, \ldots, r_N\}$.
   (b) Output $\{y\}$ if $z'' \in R$ and $\emptyset$ otherwise.

**3.2 Tailoring SimplePIR**

We instantiate the above PSI construction based on SimplePIR \cite{hいかがし22}, which is the fastest single-server PIR scheme known to date with sublinear communication.\footnote{While Piano \cite{zhenga23} is faster in terms of online time, the communication cost in pre-processing is linear in $N$. Although the local storage of the client is not linear in $N$, this linear pre-processing communication does not directly fit the PSI applications we have in mind.} We now open this black box for better performance.

**SimplePIR.** SimplePIR essentially realizes the square root PIR introduced in \cite{Kobayashi97} with an LWE Regev encryption scheme using preprocessing to boost online performance. Specifically, for a database of size $N$ with each data entry in $\mathbb{Z}_p$, SimplePIR models it as a matrix $D \in \mathbb{Z}_p^{\sqrt{N}} \times \mathbb{Z}_p^{\sqrt{N}}$. Retrieval of a single element is done by retrieving the entire column of $D$ where the target element lies.

The client sends an encrypted indicator vector $\texttt{qry} = (ct_1, \ldots, ct_{\sqrt{N}})$ where $ct_i$ encrypts 1 if the queried entry is in the $i$-th column and $ct_i$ encrypts 0 otherwise. Since the LWE Regev encryption scheme is linear homomorphic, the server replies with the result from a homomorphic matrix-vector multiplication $\texttt{rsp} \leftarrow D \times \texttt{qry}$, which encrypts the $i$-th column of the matrix $D$. The underlying homomorphic operations are either homomorphic addition or scalar multiplication, since the database $D$ is given in plaintext form.

**Dive into more details.** We can view $\texttt{qry}$ as a $\sqrt{N}$-by-$(n + 1)$ matrix where each row

<table>
<thead>
<tr>
<th>\textbf{Inputs:}</th>
<th>The server $S$ holds a large set $X = {x_1, \ldots, x_N}$ where $x_i \in {0, 1}^<em>$ for each $i \in [N]$ (assume $X$ is randomly shuffled). The client $C$ holds a single element $y \in {0, 1}^</em>$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{Setup:}</td>
<td>$S$ and $C$ agree on the security parameters $\kappa, \lambda$, protocol parameters $N, \delta, \tau$, a cyclic group $G$ of prime order $q$ with generator $g$, three hash functions $H_1 : {0, 1}^* \rightarrow G$, $H_2 : G \times G \rightarrow {0, 1}^\delta$, and $H_3 : {0, 1}^\delta \rightarrow [\tau]$.</td>
</tr>
</tbody>
</table>
| \textbf{Pre-processing Phase:} | $S$ does the following:
1. Randomly sample $k_3 \leftarrow \mathbb{Z}_q$.
2. Initialize an empty table $T$ of size $\tau$, namely $T[i] := \emptyset$ for all $i \in [\tau]$.
3. For each $i \in [N]$:
   (a) Compute $u_i := H_2(H_1(x_i), H_1(x_i)^{k_3})$ and $\ell_i := H_3(u_i)$.
   (b) Let $T[\ell_i] := T[\ell_i] \cup \{u_i\}$.
4. Let $\gamma$ denote the size of the largest entry in $T$, namely $\gamma := \max_{\ell \in [\tau]} |T[\ell]|$. For each $i \in [\tau]$, if $|T[i]| < \gamma$, then pad it with dummy strings of length $\delta$ (e.g., $0^\delta$) to reach a size of $\gamma$.
5. View $T$ as a database with $\tau$ entries, each entry containing a set of $\gamma$ strings of length $\delta$.
   Perform the pre-processing step of PIR, and send the pre-processed data hint to $C$ together with $\gamma$.
| \textbf{Online Phase:} | $C$ randomly samples $kc \leftarrow \mathbb{Z}_q$, computes $z := H_1(y)^{kc}$, and sends $z$ to $S$.
2. Upon receiving $z$, $S$ computes $z' := z^k$ and sends it back to $C$.
3. Upon receiving $z'$ back, $C$ does the following:
   (a) Compute $z'' := H_2\left(H_1(y), (z')^{k\ell} \right)$ and $\ell^c := H_3(z'')$.
   (b) Prepare a PIR query for the $\ell^c$-th entry of the database $T$ and send it to $S$.
4. Upon receiving the PIR query, $S$ computes the PIR response and sends it back to $C$.
5. Upon receiving the PIR response, $C$ does the following:
   (a) Recover the entire $\ell^c$-th entry of $T$ as a set of strings $R = \{r_1, \ldots, r_N\}$.
   (b) Output $\{y\}$ if $z'' \in R$ and $\emptyset$ otherwise.

Figure 2: Our PSI protocol where the client has a single element.
**Inputs:** The server $S$ holds a large set $X = \{x_1, \ldots, x_N\}$ where $x_i \in \{0, 1\}^*$ for each $i \in [N]$ (assume $X$ is randomly shuffled). The client $C$ holds a small set $Y = \{y_1, \ldots, y_M\}$ where $y_i \in \{0, 1\}^*$ for each $i \in [M]$.

**Setup:** $S$ and $C$ agree on the security parameters $\kappa, \lambda$, protocol parameters $N, M, m, \delta$, a cyclic group $G$ of prime order $q$ with generator $g$, five hash functions $H_1 : \{0, 1\}^* \rightarrow G$, $H_2 : G \times G \rightarrow \{0, 1\}^s$, and $h_1, h_2, h_3 : \{0, 1\}^s \rightarrow [m]$.

**Pre-processing Phase:** $S$ does the following:
1. Randomly sample $k_S \leftarrow Z_q^n$.
2. Initialize $m$ empty hash bins $X_1, \ldots, X_m := \emptyset$.
3. For each $i \in [N]$:
   (a) Compute $u_i := H_2(H_1(x_i), H_1(x_i)^{k_S})$.
   (b) Let $X_i := X_i \cup \{u_i\}$ for each $j \in \{h_1(u_i), h_2(u_i), h_3(u_i)\}$.
4. For each $j \in [m]$, let $N_j = |X_j|$ and denote $X_j = \{u_{j,1}, \ldots, u_{j,N_j}\}$. Proceed as in the single-element pre-processing phase:
   (a) Choose a parameter $\tau_j$ and a hash function $H_{3,j} : \{0, 1\}^* \rightarrow [\tau_j]$.
   (b) Initialize an empty table $T_j$ of size $\tau_j$, namely $T_j[i] := \emptyset$ for all $i \in [\tau_j]$.
   (c) For each $i \in [N_j]$, compute $H_{3,j}(u_{j,i})$ and let $T_j[i] := T_j[i] \cup \{u_{j,i}\}$.
   (d) Let $\gamma_j$ denote the size of the largest entry in $T_j$, namely $\gamma_j = \max_{i \in [\tau_j]} |T_j[i]|$. For each $i \in [\tau_j]$, if $|T_j[i]| < \gamma_j$, then pad it with dummy strings of length $\delta$ to reach a size of $\gamma_j$.
   (e) View $T_j$ as a database with $\tau_j$ entries, each entry containing a set of $\gamma_j$ strings of length $\delta$. Perform the pre-processing step of PIR to obtain $\hat{h}_j$.
5. Send $\{(N_j, \tau_j, H_{3,j}, \gamma_j, \hat{h}_j)\}_{j \in [m]}$ to $C$.

**Online Phase:**
1. $C$ randomly samples $k_C \leftarrow Z_q^n$, computes $z_i := H_2(y_i)^{k_C}$ for all $i \in [M]$, and sends $\{z_i\}_{i \in [M]}$ to $S$.
2. Upon receiving $\{z_i\}_{i \in [M]}$, $S$ computes $z_i' := z_i^{k_S}$ for all $i \in [M]$ and sends $\{z_i'\}_{i \in [M]}$ back to $C$.
3. Upon receiving $\{z_i'\}_{i \in [M]}$, $C$ does the following:
   (a) For each $i \in [M]$, compute $z_i'' := H_2(H_1(y_i), (z_i')^{k_C^{-1}})$.
   (b) Initialize $m$ empty hash bins $Y_1, \ldots, Y_m := \emptyset$.
   (c) Perform Cuckoo hashing using $h_1, h_2, h_3$ on $\{z_i''\}_{i \in [M]}$ and put $(y_i, z_i'')$ into one of the hash bins $Y_{h_1(z_i'')}, Y_{h_2(z_i'')}, Y_{h_3(z_i'')}$ such that each bin contains exactly one tuple. Pad each empty bin with a dummy random tuple.
4. Upon receiving the PIR queries, $S$ computes the response $\text{rsp}_j$ for $\text{qry}_j$ using $T_j$ for each $j \in [m]$, and sends $\{\text{rsp}_j\}_{j \in [m]}$ back to $C$.
5. Upon receiving the PIR responses, $C$ does the following:
   (a) For each $j \in [m]$, recover the entire $\ell_j$-th entry of $T_j$ as a set of strings
      \[ R_j = \{r_{j,1}, \ldots, r_{j,\gamma_j}\}. \]
   (b) Output the intersection $I := \{y_j \mid z_j'' \in R_j, j \in [m]\}$.

**Figure 3:** Our PSI protocol where the client has multiple elements.

corresponds to an LWE Regev ciphertext of the form $(\vec{a}, b) \in \mathbb{Z}_q^{n+1}$ where $n$ is the LWE dimension, $q$ is the ciphertext modulus and $\vec{a} \in \mathbb{Z}_q^n$ is sampled uniformly.
The homomorphic matrix-vector multiplication $D \times qr_y$ can be viewed as a matrix multiplication of a $\sqrt{N}$-by-$\sqrt{N}$ matrix $D$ and a $\sqrt{N}$-by-$(n + 1)$ matrix $qr_y$, whose each row corresponds to a ciphertext $ct_x = (a_i, b_i)$. Now let $qr_y = [a_1^T, a_2^T, \ldots, a_{\sqrt{N}}^T]^T$ be the first $n$ columns of query matrix $qr_y$ and $qr_y = [b_1, b_2, \ldots, b_{\sqrt{N}}]^T$ be the last column of $qr_y$. The result ciphertext vector, or viewed as the result matrix, is $D \times qr_y = D \times [qr_y, qr_y] = [D \times qr_y, D \times qr_y]$. A significant insight of SimplePIR is that the first part, $D \times qr_y$, only involves multiplying the database $D$ with a uniformly random matrix $qr_y$, which is independent of the client’s input. Exploiting this property, the server $S$ can generate a random matrix $qr_y$ by uniformly sampling it from a short seed $s$. During the pre-processing phase, the server sends both the seed $s$ and hint $= D \times qr_y$ to the client. In the subsequent online phase, the client reconstructs $qr_y$ using the seed $s$ and generates the final column $qr_y$ of the query ciphertext matrix. The server’s computation is significantly reduced to performing a smaller matrix-vector multiplication, $rsp \leftarrow D \times qr_y$, during the online phase. The server then sends $rsp$ back to the client. Finally, the client decrypts the combined result, considering both the received hint $= D \times qr_y$ and $rsp$.

The hint is thus of size $\sqrt{N} \cdot n \cdot \log q$ (which is the pre-processing communication cost), and the online upload and download communication are both $\sqrt{N} \cdot \log q$ (which together is the online communication cost). The server needs to evaluate $N \cdot n$ $Z_q$-multiplications and additions during the pre-processing phase, and $N Z_q$-multiplications and additions during the online phase.

With this background, we can proceed to find more balances when instantiating the underlying PIR protocol with SimplePIR.

**Re-arrange the database.** SimplePIR arranges its database into a square $\sqrt{N}$-by-$\sqrt{N}$ matrix $D$. To find a better balance between the hint size and the $rsp$ size, we can re-arrange $D$ to be rectangular $\in Z_{p^\alpha}^{\alpha \times \beta}$, without changing the pre-processing and online computation cost. The hint size (i.e. the size of $D \times qr_y$) is then $\alpha \cdot n \cdot \log q$. The upload cost (i.e. the size of $qr_y$) is $\beta \cdot \log q$. The download cost (i.e. the size of $D \times qr_y$) is $\alpha \cdot n \cdot \log q$. The parameters $\alpha$ and $\beta$ can be chosen according to the application. We discuss our choices below.

**Retrieving multiple elements for free.** A key observation of SimplePIR is that we are retrieving back an entire column of database matrix $D$, which contains $\alpha Z_q$ elements. Thus, while an entry in hash table $T$ has $\gamma \cdot \delta$ bits, where $\gamma$ is the number of elements per data entry, by setting $\alpha = C \cdot \gamma \cdot \delta / [\log p]$, for some $C \in Z^+$, we can retrieve the entire hash table entry with a single PIR query.

**Tuning the hash table size $\tau$.** Recall that our table $T$ has $\tau \cdot (\gamma \cdot \delta)$ bits, where $\gamma$ is the size of $T$’s largest entry. Every entry with fewer than $\gamma$ elements needs to be padded to $\gamma$ elements (with zeros). Thus, as we decrease the table size $\tau$, the variance in the entry sizes in $T$ becomes smaller, hence the database size $\tau \cdot (\gamma \cdot \delta)$ also decreases.

As mentioned above, we need $\alpha = C \cdot \gamma \cdot \delta / [\log p]$ to retrieve the entire entry with one query and thus $\beta = \tau / C$, for some $C \in Z^+$. However, as discussed, efficiency grows when $\tau$ decreases. Thus, if we set $\beta = \tau / C$ for some $C > 1$, we can instead simply set a new $\tau' \leftarrow \tau / C, \beta \leftarrow \tau'$, and use $\tau'$ as the size of $T$ for better efficiency. Thus, we set $\beta = \tau$ and $\alpha = \gamma \cdot \delta / [\log p]$ (i.e., $C = 1$), and then directly adjust $\tau$ for better efficiency.

This results in $[hint] = \gamma \cdot \delta / [\log p] \cdot n \cdot [\log q]$ bits, $[qr_y] = \tau \cdot [\log q]$ bits, and $[rsp] = [hint] / n$ bits. Thus, we can adjust $\tau$ according to the desired hint, $qr_y$, $rsp$ sizes. For our purpose, we set $\tau$ such that $[hint] \approx \sqrt{|X| \cdot \delta}$, where $|X| \cdot \delta$ is the cost of sending the entire $X'$ (recall that $X' := \{F_p(x) | x \in X\}$).

\footnote{Note that reusing the same matrix for polynomial amount of queries is still secure as shown in [PVW08].}
Modulus switching. We introduce an additional technique to further reduce the communication of SimplePIR, called modulus switching [DM15]. Recall that \( \text{hint} \in \mathbb{Z}_q^{\alpha \times n}, \text{rsp} \in \mathbb{Z}_q^{\alpha \times 1} \) and \((\text{hint}, \text{rsp})\) together forms an LWE ciphertext.

Recall that an LWE ciphertext \((\vec{a}, b) \in \mathbb{Z}_q^{n+1}\) with respect to secret key \(\vec{s} \in \mathbb{Z}_q^n\) satisfies the following: \(b = \langle \vec{a}, \vec{s} \rangle = e + m \cdot \Delta\) where \(e \in \mathbb{Z}_q\) is a small error, \(m \in \mathbb{Z}_p\) is a message, and \(\Delta = \left\lfloor \frac{q}{p} \right\rfloor\). There is usually a big gap between the ciphertext modulus \(q\) and the plaintext modulus \(p\) (i.e., \(p \ll q\)). Thus, we can work on a smaller ciphertext modulus \(q' < q\) to reduce the communication, while preserving the LWE ciphertext structure. A modulus switching procedure for LWE ciphertexts from \(q\) to \(q'\) is defined as \((\vec{a}', b') \leftarrow \text{round}(\frac{q}{q'}(\vec{a}, b)) \in \mathbb{Z}_{q'}^{n+1}\). The resulting ciphertext \((\vec{a}', b')\) satisfies \(b' - \langle \vec{a}', \vec{s} \rangle = e' + m \cdot \Delta', \text{ where } e' \in \mathbb{Z}_{q'}\) is a new error term (to be bounded), and \(\Delta' = \left\lfloor q'/p \right\rfloor\).

This means that when sending \(\text{hint} \in \mathbb{Z}_q^{\alpha \times n}\) and \(\text{rsp} \in \mathbb{Z}_q^{\alpha \times 1}\), we can modulus switch them down to some \(q' \ll q\) before sending them back. The communication cost can thus be reduced by a factor of \(\log q' / \log(q')\). The correctness holds as long as \(\Pr[|e'| \leq \Delta/2] \geq 1 - \text{negl}(\lambda)\).

Ternary LWE keys and randomized rounding. To make sure \(e'\) is small, we employ two additional techniques. The first is using ternary keys for LWE secret keys, namely sample the secret key as \(\vec{s} \in \{0, 1, -1\}^n\). The second is randomized rounding. Specifically, to round a decimal value \(d\), we round it to \(c + 1\) with probability \(d\) and round it to \(c\) with probability \(1 - d\). By using these two techniques, \(e' = O(\frac{q}{q'} c + \sqrt{\frac{q}{q'}})\) [DM15]. Concretely, if \(e\) has a standard deviation of \(\sigma\) (for Gaussian distribution \(\chi_\sigma\)), then \(e'\) has a standard deviation of \(\sigma' = \frac{q}{q'} \sigma + \sigma_{MS}\) such that \(\sigma_{MS} = \sqrt{\frac{n+1}{\pi}}\) (for Gaussian distribution \(\chi_{\sigma'}\)) [LMP22, Sec 6.5].

Error analysis. Recall that \(\text{qry}\) is essentially LWE ciphertexts with some initial error, \(D \times \text{qry}\) thus results in new LWE ciphertexts with larger errors. As shown in [HHC+22, Sec C.2], to guarantee the correctness of SimplePIR, we need to choose LWE parameters \(\sigma\) (i.e., the error distribution standard deviation for the initial error generation) to satisfy:

\[
2 \exp\left(-\pi \cdot \left(\frac{\Delta}{2 \sigma \sqrt{2\pi} \sqrt{\frac{q}{p}}}ight)^2\right) \leq 2^{-\lambda}, \text{ where } \Delta = \left\lfloor \frac{q}{p} \right\rfloor.
\]

Combining with modulus switching, we choose \(\Delta_1 + \Delta_2 = \Delta, \sigma\) such that they minimize \(q'\) and satisfy \(2 \exp\left(-\pi \cdot \left(\frac{\Delta_1}{2 \sigma \sqrt{2\pi} \sqrt{\frac{q}{p}}}ight)^2\right) \leq 2^{-\lambda/2}, \text{ and } \text{erf}(\frac{\sqrt{q'}}{\sqrt{2\sigma_{MS}}}) \leq 2^{-\lambda/2}\). By union bound, we have PIR correctness with probability \(\geq 1 - 2^{-\lambda/2} - 2^{-\lambda/2} = 1 - 2^{-\lambda}\).

3.3 Parameters

In this section, we summarize the parameters required in our single-element PSI protocol (fig. 2) and multi-element PSI protocol (fig. 3).

- Computational security parameter \(\kappa\) and statistical security parameter \(\lambda\).
- Server set size \(N\) and client set size \(M\) (in fig. 3).
- \(H_2\)’s output length \(\delta\) such that \((N \cdot M)/2^{\delta} \leq 2^{-\lambda}\).
- \(m\) (in fig. 3) such that Cuckoo hashing \(M\) elements into \(m\) bins fails with probability \(\text{negl}(\lambda)\).
- Hash table size \(\tau = N\) (in fig. 2) and \(\tau_j = N_j\) for each \(j \in [m]\) (in fig. 3) (these parameters can be further tuned for optimized performance, as discussed in section 3.2).
- PIR parameters such that the underlying PIR protocol satisfies both correctness and receiver privacy.

\[\text{Recall that } e \leftarrow \chi_\sigma, \Pr[e \geq \Delta/2] \leq \text{erf}(\frac{\Delta}{\sqrt{2\sigma}}).\]
4 Security Guarantees

4.1 Corrupted Client

In the existence of a corrupted client, our PSI protocols (fig. 2 for $M = 1$ and fig. 3 for an arbitrary $M$) achieve semi-honest security in the standard PSI definition shown in fig. 1 and malicious security in the adaptive variant of the PSI functionality [JL10], which allows adaptive queries from $C$. In more detail, the ideal functionality takes a set $X$ from $S$ as input, and for each query on input $y_i$ made by $C$, for $i \in [M]$, the ideal functionality returns yes or no for whether $y_i \in X$. Although it is secure for the weaker adaptive PSI functionality, we can in fact show that a malicious client cannot change its input set after sending $\{z_i\}_{i \in [M]}$ in the online phase Step 1.

We state the theorems below and give the security proofs in appendix A.1 and appendix A.2, respectively. Note that theorem 2 achieves stronger malicious security by relying on a stronger computational assumption, namely OMGDH. The proofs for the single-element protocol in fig. 2 follow similarly.

**Theorem 1.** If $H_1, H_2$ are modeled as random oracles and DDH is hard in $G$, then our protocol in fig. 3 securely computes the PSI functionality in fig. 1 against a semi-honest client when the protocol parameters are chosen as described in section 3.3.

**Theorem 2.** If $H_1, H_2$ are modeled as random oracles, the OMGDH problem is hard in $G$, then our protocol in fig. 3 securely computes the adaptive PSI functionality against a malicious client when the protocol parameters are chosen as described in section 3.3.

4.2 Corrupted Server

In the existence of a corrupted server, our PSI protocols (fig. 2 for $M = 1$ and fig. 3 for an arbitrary $M$) achieve simulation-based security against a semi-honest server and client privacy against a malicious server in the standard PSI definition.

Furthermore, with small modifications to our protocol for a single client with $M = 1$, we can achieve simulation-based security against a malicious server, as demonstrated in section 4.2.2. Finally, we discuss extensions to serving multiple clients with multiple elements in their sets in section 4.2.3, as well as the challenges in proving full simulation-based security.

4.2.1 Original Protocol

**Semi-honest server.** We state the semi-honest security for an arbitrary $M$ below and give the security proof in appendix A.3. The proof for $M = 1$ follows similarly.

**Theorem 3.** If $H_1, H_2$ are modeled as random oracles, the DDH problem is hard in $G$, and the underlying PIR protocol satisfies correctness and receiver privacy, then our protocol in fig. 3 securely computes the PSI functionality in fig. 1 against a semi-honest server when the protocol parameters are chosen as described in section 3.3.

**Client privacy against malicious server.** We can achieve client privacy against a malicious server without making any changes to our protocol. At a high level, it means that the server cannot learn anything about the client’s input from the interaction transcript. This security guarantee is the same as the one achieved in [CHLR18, CMdG+21]. We state the theorem below and skip the proof as it follows the exact same structure as the proof of theorem 3 in arguing for client privacy.

**Theorem 4.** If $H_1$ is modeled as a random oracle, the DDH problem is hard in $G$, and the underlying PIR protocol satisfies receiver privacy, then our protocol in fig. 3 achieves client privacy [HL08, Def 2.2] against a malicious server when the protocol parameters are chosen as described in section 3.3.
4.2.2 Full Security Against Malicious Server for \( |Y| = 1 \)

In addition to client privacy, prior works \cite{CHLR18,CMdG21} on FHE-based unbalanced PSI proposed techniques to achieve simulation-based security with leakage against a malicious server. At a high level, the server in their protocols needs to homomorphically compute and return an encrypted \( H(z) \) for a public hash function \( H \) and an OPRF value \( z \) encrypted by the client. Their assumption is that the server cannot homomorphically compute an encryption of \( H(z) \) given an encryption of \( z \) and some pre-determined list of encryptions of powers of \( z \), when \( H \) is a sufficiently complex hash function such as SHA256.

The heuristic argument of this assumption comes from the difficulty of evaluating such a high-depth circuit using leveled HE, where the parameters are chosen to support a smaller multiplicative depth. However, the server is still able to make the intersection indirectly depend on the set \( Y \setminus X \), which is modeled as a leakage circuit \textit{leakage}(\cdot) in their ideal functionality for security \textit{with leakage}.

Nevertheless, this issue does \textit{not} arise in our construction, and we can indeed achieve simulation-based security with minor modifications to our protocol for \( |Y| = 1 \). In this section, we focus on the setting where the server runs the protocol with a single client with one element and does not reuse the result of the pre-processing phase with multiple clients. Under this model, we use novel techniques to prove security under a corrupted server as follows.\footnote{Note that this can be enforced in practice by simply letting the preprocessing randomness come from the client (e.g., the hash of the client’s public keys or a random seed chosen by the client).}

Recall that the OPRF-based PSI protocol \cite{JL10} achieves simulation-based security against a malicious server. Our protocol in fig. 2 differs from \cite{JL10} in that the client does not receive all the OPRF values of the server, but rather a small number of them from the PIR query. To address this issue, we let the simulator rewind the PIR query, which is indistinguishable to the server due to the receiver privacy of PIR. Then, the simulator can extract all the OPRF values in the database. This serves as our starting point.

Furthermore, in \cite{JL10}, the server provides a proof of knowledge (PoK) that the OPRF response is computed correctly. To reduce the overhead, notably, we eliminate the need for this PoK by switching the order of the OPRF and PIR queries. Intuitively speaking, the client concludes that its element is in the intersection if and only if the corresponding OPRF value \( z'' \) matches an OPRF value from the database, irrespective of whether \( z'' \) is computed correctly or not. Therefore, after extracting the elements from the PIR query, the simulator can play the role of an honest client to compute the OPRF values of all elements (via rewinding techniques) and check if they match the values retrieved from the PIR query. With these two techniques, we can modify our protocol accordingly to achieve malicious security against the server.

\textbf{Modifications to our protocol.} Since we need to switch the order of the OPRF and PIR queries, we can no longer use the OPRF value to compute the PIR location. Instead, we compute it as \( \ell := H_3(x) \). The server now uses \( \ell_i := H_3(x_i) \) instead of \( \ell_i := H_3(u_i) \) to construct \( T \) in the pre-processing phase. However, after making this change, the size of the largest entry in \( T \) could leak information about \( X \). To resolve this issue, we let \( \mathcal{S} \) and \( \mathcal{C} \) agree on a pre-determined \( \gamma \) such that hashing \( \ell_i \) elements per bin with all but negligible probability. The server then pads each entry of \( T \) with dummy random elements to reach a size of \( \gamma \), and randomly shuffles all these elements.\footnote{One optimization we can do is to sort the elements in each entry instead of randomly permuting them. This allows the client to check whether \( z'' \in R \) more efficiently.} All the rest remain the same as the original protocol.

One disadvantage, compared to our original protocol, is that the entry size now grows with the pre-determined \( \gamma \), leading to extra communication overhead. Moreover, since the padded elements cannot be 0-strings but must be random, the PIR scheme cannot skip those elements, thus causing extra computation overhead.
Inputs: The server $S$ holds a large set $X = \{x_1, \ldots, x_N\}$ where $x_i \in \{0,1\}^*$ for each $i \in [N]$ (assume $X$ is randomly shuffled). The client $C$ holds a single element $y \in \{0,1\}^*$.

Setup: $S$ and $C$ agree on the security parameters $\kappa, \lambda$, protocol parameters $N, \delta, \tau, \gamma$, a cyclic group $G$ of prime order $q$ with generator $g$, three hash functions $H_1 : \{0,1\}^* \to G$, $H_2 : G \times G \to \{0,1\}^\delta$, and $H_3 : \{0,1\}^\delta \to [\tau]$.

Pre-processing Phase: $S$ does the following:
1. Randomly sample $k_S \stackrel{\$}{\leftarrow} Z_q$.
2. Initialize an empty table $T$ of size $\tau$, namely $T[i] := \emptyset$ for all $i \in [\tau]$.
3. For each $i \in [N]$:
   
   (a) Compute $u_i := H_2(H_1(x_i), H_1(x_i)^{k_S})$ and $\ell_i := H_3(x_i)$.
   
   (b) Let $T[\ell_i] := T[\ell_i] \cup \{u_i\}$.
4. For each $i \in [\tau]$, if $|T[i]| < \gamma$, then pad it with dummy random strings of length $\delta$ to reach a size of $\gamma$. Randomly shuffle all the strings in $T[i]$.
5. View $T$ as a database with $\tau$ entries, each entry containing a set of $\gamma$ strings of length $\delta$.

Perform the pre-processing step of PIR, and send the pre-processed data hint to $C$.

Online Phase:
1. $C$ computes $\ell^C := H_3(y)$, prepares a PIR query for the $\ell^C$-th entry of the database $T$, and sends it to $S$.
2. Upon receiving the PIR query, $S$ computes the PIR response and sends it back to $C$.
3. $C$ randomly samples $k_C \stackrel{\$}{\leftarrow} Z_q$, computes $z := H_1(y)^{k_C}$, and sends $z$ to $S$.
4. Upon receiving $z$, $S$ computes $z' := z^{k_S}$ and sends it back to $C$.
5. Upon receiving $z'$ back, $C$ does the following:
   
   (a) Compute $z'' := H_2\left(H_1(y), (z')^{k_C^{-1}}\right)$.
   
   (b) Recover the entire $\ell^C$-th entry of $T$ as a set of strings $R = \{r_1, \ldots, r_\gamma\}$.
   
   (c) Output $\{y\}$ if $z'' \in R$ and $\emptyset$ otherwise.

Figure 4: Our PSI protocol against a malicious server where the client has a single element.

We present the modified protocol in fig. 4, where the differences from fig. 2 are highlighted. The modified protocol remains secure against semi-honest and malicious clients (under different assumptions), which we show in appendix A.4. Additionally, it is secure against a malicious server, as proved in appendix A.5.

4.2.3 Serving Multiple Clients for Arbitrary $|Y|$ 

Now we turn our attention to the setting where the server reuses the pre-processing phase with multiple clients. The proof technique for theorem 7 no longer works even for $|Y| = 1$. The problem is that when rewinding the PIR queries made by different clients, the server might not give consistent responses. The same inconsistency issue arises when rewinding the OPRF queries.

Reusing pre-processing phase. We start by discussing how to reuse the pre-processing phase with multiple clients for the $|Y| = 1$ case. To resolve the aforementioned inconsistency issues, we can make the following modifications to the protocol in fig. 2. In the pre-processing phase, the server commits to the database $T$ and gives a proof of knowledge (PoK) that hint is correctly computed on $T$. Additionally, the server provides $g^{k_S}$ along with a PoK for $k_S$. In the online phase, the server provides a PoK for $k_S$ when generating the OPRF response, as well as a PoK for the committed $T$ when generating the PIR response. Again, these changes do not affect the security against corrupted clients. Note that, however, this would greatly affect the performance. For example, proving that
the PIR response is correctly computed can be several times slower than generating the response itself.

We sketch a security proof for this modified protocol against a malicious server. First, the simulator is able to extract both $k_S$ and the database $T$ in the pre-processing phase from the PoKs provided by the server. Since the simulator also keeps track of all the queries to $H_1$ and $H_2$, it can extract the set $X$ using a similar approach as in [JL10]. The simulator then sends the extracted set $X$ to the ideal functionality. In the online phase, the two PoKs provided by the server ensure that the PIR and OPRF responses are consistent with the $X$ committed (extracted) in the pre-processing phase. This consistency guarantees that the responses align with the set $X$ sent to the ideal functionality, thus concluding the proof.

Handling arbitrary $|Y|$. The main challenge in handling multiple elements in a client’s set comes from Cuckoo hashing. Specifically, the server is supposed to put each element $x_i \in X$ into three hash bins. However, if the malicious server decides to put it into only one of the three bins, then it becomes unclear whether the simulator should include $x_i$ in the set sent to the ideal functionality. This is because whether the same element $x_i$ will be put into that same bin on the client’s side depends on the other elements in the client’s set. Thus, achieving a simulation-based proof seems challenging unless we incorporate a more sophisticated proof of correctness for Cuckoo hashing.

Nevertheless, notice that our protocol in fig. 3 can be viewed as performing a single-element PSI protocol for each hash bin. This observation opens up the possibility of leveraging the malicious security of single-element PSI discussed in section 4.2.2, to achieve malicious security for a weaker ideal functionality. In particular, the ideal functionality takes a set $Y$ from the client and applies Cuckoo hashing to it, putting the set elements into $m$ bins each with at most one element. On the server side, the malicious server can choose what elements to put into each hash bin $X_i$ for $\forall i \in [m]$. Then the ideal functionality applies a single-element PSI on each hash bin. If the single-element PSI protocol is maliciously secure against the server, then we can extend the security guarantee to such a weaker ideal functionality. We leave the construction of a maliciously secure protocol for the standard ideal functionality to future study.

5 Experimental Results

We implement our single-element PSI protocol in fig. 2 and multi-element PSI protocol in fig. 3 in a C++ library. We use the SimplePIR [HHC+22] implementation in a Go library directly and optimize upon it using the techniques we have mentioned in section 3.2. All benchmarks are running on an Amazon AWS c5.metal instance with Intel Xeon Platinum 8275L CPU with 96 virtual cores and 192 GB of RAM.

5.1 Parameter Setting

For single-element PSI, we ran benchmarks for $|X| = 2^{20}, 2^{22}, 2^{24}, 2^{26}$, and $2^{28}$ using computational security parameter $\kappa = 128$, statistical security parameter $\lambda = 40$, and $H_3 : G \times G \rightarrow \{0,1\}^{\delta}$ output size $\delta = 80$ (to guarantee $\lambda = 40$ given the $|X|$).

Following prior work [CLR17, ACLS18], we used experimental analyses to choose the number of hashes and bins for our Cuckoo hashing based multi-element protocol in fig. 3. Our experiments found, for $|Y| = 16$ and only three hashes, $\sim 46$ bins were required to run $2^{20}$ trials without error ($m \approx 2.875|Y|$) and $\sim 105$ bins were required for $|Y| = 64$ ($m \approx 1.64|Y|$). Using four hashes and $m = 1.5|Y|$, both 16 and 64 are able to run $2^{20}$ trials without error, so these were chosen as the increase in the server set size was more desirable than increasing the number of PIR instances. Using these, we ran benchmarks for $|X| = 2^{26}$ with $|Y| = 16$ and 64.
In all benchmarks, $\tau$ was chosen to strike a balance between the offline and online phases (note the size estimation discussed in section 3.2). We choose the error bound according to our error analysis in section 3.2, and choose other LWE parameters according to [APS15] to guarantee 128-bit computational security.

5.2 Benchmark Comparison

Single-element PSI. In table 1, we compare the main efficiency metrics with APSI [CMdG+21], the state-of-the-art unbalanced PSI protocol with sublinear client storage (zero for APSI). We use green to highlight the efficiency metrics at which our protocol is better and red to indicate the ones at which our protocol is worse.

| $|X|$ | $|Y|$ | $n$ | $\tau$ | $\phi$ | $\delta$ | $\sigma$ |
|---|---|---|---|---|---|---|
| $2^{26}$ | 0.028 | 0.028 | 0.028 | 0.028 | 0.028 | 0.028 |
| $2^{27}$ | 0.042 | 0.042 | 0.042 | 0.042 | 0.042 | 0.042 |
| $2^{28}$ | 0.055 | 0.055 | 0.055 | 0.055 | 0.055 | 0.055 |

As shown in the table, our online runtime with $D = 1$ is about two orders of magnitude faster than APSI with $D = 1$, and about one magnitude faster than APSI with $D = 32$. Thus, we believe that multi-threading is not needed for our online time for most applications. However, note that our construction can be easily multi-threaded: a SimplePIR query is simply $n$ LWE ciphertexts for some $n = O(\sqrt{|X|})$. Thus, we can simply divide the $n$ LWE ciphertexts into $T$ threads and process them separately. We believe setting $D = 32$ gives us a similar speed up as APSI, if not more. Moreover, our online communication cost is also at least 4x smaller.

On the other hand, our overall offline server computation time is slightly worse than APSI’s but is still comparable (mainly due to the performance for $D = 1$). Recall that this is a one-time process and can be reused for all clients, and so this extra cost has relatively small impact. Also note that our multi-threaded offline time outperforms APSI, and this is because our offline phase can be easily multi-threaded without much overhead. However, their offline phase contains computation that is not easily multi-threadable (e.g., large polynomial evaluation).

The only major drawback of our protocol when compared to APSI is offline communication. We require the client to store a hint for the underlying SimplePIR protocol. However, as shown in the table, the hint is relatively small (only $<10x$ larger than the online communication of APSI), and can be amortized over multiple queries. Moreover, it grows with $O(\sqrt{|X|})$ instead of being linear to $|X|$, and thus grows relatively slowly.

Multi-element PSI. We compare our multi-element protocol with PSI in table 2. We also add a naive use of our single-element protocol in table 1 for a more comprehensive comparison. We use bold texts to highlight the best of the three for a given efficiency metric. It is easy to see that for $|Y| > 1$, our construction has less advantage compared to APSI. However, both of our constructions’ online time still greatly outperforms APSI, for the $|Y|$’s we test. Note that when $|Y|$ grows larger, our offline communication may grow too large and becomes impractical. A similar argument applies to our online communication.

---

5.2 Benchmark Comparison

Table 1: Efficiency comparison with APSI [CMdG+21] for our single-element PSI in fig. 2. $D$ is the number of threads. $X$ is the server set. $C$ is the client. For $|X| = 2^{26}, 2^{27}, 2^{28}$, we use the APSI default parameters. For $|X| = 2^{29}, 2^{30}, 2^{31}$, since APSI does not provide an interface to choose the optimal parameters, we tested different parameters they provide and chose the optimal one.

| $|X|$ | $|Y|$ | $n$ | $\tau$ | $\phi$ | $\delta$ | $\sigma$ |
|---|---|---|---|---|---|---|
| $2^{26}$ | - | - | 224 | 335 | 0.15 | 18 |
| $2^{27}$ | - | - | 994 | 1380 | 0.86 | 57 |
| $2^{28}$ | - | - | 505 | 630 | 4.9 | 109 |
| $2^{29}$ | - | - | 229 | 101 | 12.6 | 15 |
| $2^{30}$ | - | - | 57.1 | 23.0 | 3.27 | 3.7 |

---

*Our offline communication consists of multiple SimplePIR hints. The number of hints grows with $|Y|$, while each hint size decreases as $|Y|$ increases, and the former grows faster than the latter decreases.*
Finding Balance in Unbalanced PSI: A New Construction from Single-Server PIR

Table 2: Efficiency comparison with APSI [CMdG+21] for our multi-element PSI and variables are defined in the same way as table 1. “Naive” means we simply repeat single-element described in fig. 2 for \(|Y|\) times and “Cuckoo hashing” means that we use the Cuckoo hashing protocol described in fig. 3. We choose \(|X| = 2^{26}\), as APSI with \(|X| = 2^{28}\) and \(|Y| > 1\) does not run on our instance (we conjecture the reason to be out of memory).

| \(|X| = 2^{26}\) | \(|Y|\) | \begin{tabular}{c} \text{Server Offline} \\ \text{Time (s)} \end{tabular} & \begin{tabular}{c} \text{Offline} \\ \text{Comm} \\ \text{(MB)} \end{tabular} & \begin{tabular}{c} \text{Server Online} \\ \text{Time (s)} \end{tabular} & \begin{tabular}{c} \text{Server Online} \\ \text{Time (s)} \end{tabular} & \(C \rightarrow S\) & \(S \rightarrow C\) |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| Ours            | 16 (Naive)      | 524             | 15.7            | 2.35            | 0.168           | 8.38            | 0.240           |
| Ours            | 16 (Cuckoo hashing) | 542             | 75.5            | 0.553           | 0.064           | 12.6            | 0.070           |
| APSI            | 64 (Naive)      | 524             | 0               | 40.5            | 2.1             | 3.39            | 2.30            |
| Ours            | 64 (Cuckoo hashing) | 542             | 188             | 0.928           | 0.059           | 25.2            | 0.167           |
| APSI            | 64              | 520             | 0               | 41              | 2.2             | 3.39            | 2.30            |

Comparing with other schemes. In addition to comparing our benchmarks to the APSI protocol, we also evaluated the works of Davi Resende and Aranha in [RA18] and Kales et al. [KRS+19] as shown in table 3. Asymptotically, the online server time of our construction and APSI are \(O(|X|)\) while other works are \(O(|Y|)\). The offline client communication cost and storage in other works are \(O(|X|)\), while ours is \(O(\sqrt{|X|})\) and APSI is \(O(1)\). Our protocol demonstrates a better balance, e.g. for \(|X| = 2^{26}\) and \(|Y| = 1\), our server online time is as low as 0.15s, while the offline communication is only 15.7MB, significantly less than the OPRF-based protocols. It should be noted, as mentioned in [CMdG+21], that the protocol proposed in [RA18] utilized extremely aggressive Cuckoo filter parameters, for its exceptionally high performance during the online phase, resulting in an impractical high false-positive rate of \(2^{-13}\). We demonstrated that, for small \(|Y|\), our protocol is computationally very efficient during the online phase, while also keeping the offline communication low as compared to other OPRF-based protocols. However, when the client’s size \(|Y|\) grows, we could suffer from high communication costs.

Cost estimation for password breach checkup. As mentioned in the introduction, one essential application of unbalanced PSI is password breach checkup [Ali18, LKLM21]. Essentially, a central server holds a database containing a large amount of leaked passwords due to data breaches. The users themselves have one or more passwords that they want to check whether it is already insecure against this database. Of course, the users do not want to leak their own passwords, and the server does not want to share other leaked passwords with the users performing the checkup.

An unbalanced PSI (especially our extremely unbalanced setting) is perfectly suited for such a setting. As suggested in [TPY+19, ALP+21b], a database may contain \(2^{32}\) passwords, and a user may have one or several passwords to check against such a database. Since the online runtime for prior works remains prohibitively large (e.g., checking \(2^{32}\) passwords can take more than 400 seconds), [ALP+21b] suggests dividing the password into buckets, each with size, say \(2^{20}\) passwords. This leaks extra information as the server learns which bucket the client is checking against.

However, with our construction, such online cost is no longer unaffordable. For a single-core server, it takes only about 4 seconds to check for all the \(2^{32}\) passwords against a single password as illustrated in table 1. This efficiency comes at the expense of the user needing to store a hint of size \(\sim 240\)MB. However, this hint can be reused for future checkups, making it a favorable trade-off.
Table 3: Comparisons to prior works. LowMC and ECNR are two protocols described in [KRS+19]. APSI is the one in [CMG+21]. For $|Y| > 1$, we use the Cuckoo hashing protocol in fig. 3. All protocols are running in a single thread.

| $|X|$ | $|Y|$ | Protocol | Offline | Online |
|------|------|---------|---------|--------|
| $2^{20}$ | 1 | [RA18] | 13.5 | 0.013 | < 0.001 |
|      |     | LowMC   | 5      | 0.212 | 0.059 |
|      |     | ECNR    | 151    | 0.257 | 0.04  |
|      |     | APSI    | 23     | 0.18  | 1.525 |
|      |     | Ours    | 57     | 0.002 | 0.07  |
| $2^{24}$ | 1 | [RA18] | 217    | 0.013 | < 0.001 |
|      |     | LowMC   | 80.5   | 0.22  | 0.059 |
|      |     | ECNR    | 2,403  | 0.250 | 0.04  |
|      |     | APSI    | 435    | 6    | 2.49  |
|      |     | Ours    | 925    | 0.034 | 0.27  |
| $2^{26}$ | 16 | [RA18] | 870    | 0.013 | < 0.001 |
|      |     | LowMC   | 323    | 0.220 | 0.059 |
|      |     | ECNR    | 9,617  | 0.202 | 0.04  |
|      |     | APSI    | 4,375  | 0     | 3.09  |
|      |     | Ours    | 3,750  | 0.15  | 0.54  |
|      | 64 | [RA18] | 870    | 0.013 | < 0.001 |
|      |     | LowMC   | 323    | 0.177 | 0.406 |
|      |     | ECNR    | 9,617  | 0.160 | 0.13  |
|      |     | APSI    | 4,680  | 0     | 40.5  |
|      |     | Ours    | 5,257  | 0.553 | 12.5  |
|      | 64 | [RA18] | 870    | 0.015 | 0.002 |
|      |     | LowMC   | 323    | 0.179 | 1.54  |
|      |     | ECNR    | 9,616  | 0.173 | 0.41  |
|      |     | APSI    | 4,680  | 0     | 5.69  |
|      |     | Ours    | 5,623  | 0.928 | 25.3  |

**PSI from Other Keyword PIR Constructions.** We also estimate the performance of unbalanced PSI by plugging other state-of-the-art keyword PIR schemes into our framework. The estimation with the keyword PIR in [PSY23] is based on their numbers in Fig. 12. For $|X| = 2^{20}$, $|Y| = 1$, as in table 1, their online server runtime is about $\sim 3$ seconds, with online upload cost being $\sim 0.014$ MB and online download cost being $\sim 0.021$ MB. Compared to ours, the runtime is about 20x slower and the online download cost is slightly larger. However, their online upload cost is a lot smaller and requires no offline storage of the client. Alternatively, they could re-parametrize their keyword PIR to reduce the runtime to $\sim 1.5$ seconds, while increasing their download cost to $\sim 0.086$ MB. Either way, their construction provides different trade-offs compared to our construction. Similar results hold for other sizes of $|X|$, hence we skip the details of the estimation.

### 6 Extensions

**Remove the offline communication.** In the pre-processing phase of the single-element PSI protocol (fig. 2), S sends the pre-processed data (i.e., the hint) to C in Step 5. This requires the client to keep some local storage. While this is usually fine as hint is much smaller than the set $X$, there may be cases where this needs to be avoided.

Thankfully this is a simple adjustment: one can send the hint during the online phase along with the OPRF results. Since hint now is included in the online communication, it should be as small as possible. To ensure this, simply choose $\alpha, \beta$ such that $\alpha \cdot (n+1) \cdot \log q + \beta \cdot \log q$ is minimized.
Finding Balance in Unbalanced PSI: A New Construction from Single-Server PIR

For multi-element PSI, a slightly different approach is required since the knowledge of \( \tau_i \) for each \( T_i \) is necessary to perform a PIR query. \( C \) and \( S \) can agree on \( \tau_i \) in advance (e.g., let \( \tau_i \) being the expected size for each \( T_i \)). Then, everything else remains the same as for our single-element PSI protocol.

**Reduce the round complexity.** Our single-element PSI protocol (fig. 2) has two rounds in the online phase: \( C \) first sends OPRF request in Step 1, and then uses the OPRF result to prepare the PIR query in Step 4. Recall that this is because our PIR (by keyword) database is constructed from \( X' \) instead of \( X \) itself.\(^9\) If \( S \) uses \( X \) to construct the database instead of \( X' \), the client can prepare the OPRF request and PIR query at the same time. However, simply replacing \( X' \) with \( X \) leaks information. Recall our starting point in section 3, \( S \) can send the entire \( X' \) to the client, but it is insecure to send \( X \) directly to \( C \).

Thus, \( S \) needs to construct \( T \) using \( X \) in the following way. We start with the case where \( |Y| = 1 \). \( S \) and \( C \) share a hash function \( H : \{0,1\}^\delta \rightarrow [\tau] \), where \( \tau = N \) (\( \tau \) can be optimized, discussed below). First, \( S \) initializes an empty table \( T \) of size \( \tau \). Then, \( S \) computes \( T[H(x)] \leftarrow T[H(x)] \cup F_k(x) \), for all \( x \in X \). We can bound that each entry in the table has at most \( \gamma = \omega(\log(N)) \) elements with overwhelming probability. \( S \) thus pads random elements to each entry that has \( < \gamma \) elements and randomly permutes each entry. Lastly, \( S \) uses \( T \) as the PIR database. To query for element \( y \), \( C \) queries entry \( H(y) \) without needing \( F_k(y) \). Thus, the PIR query can be prepared together with the OPRF query. After receiving \( T[H(y)], F_k(y) \) in response, \( C \) simply checks whether \( F_k(y) \in T[H(y)] \).

However, one major issue is that now \( T \) is of size \( \tau \cdot \gamma \geq N \) for some fixed \( \tau \), and the padded elements are random instead of zeros. Thus, the computation, instead of being \( O(N) \), becomes \( O(\tau \cdot \gamma) \). As \( \tau = N \), the cost is \( \tau \cdot \gamma = \omega(N \log(N)) \). Moreover, recall that the PIR scheme needs to download one entry at a time, which means that the download cost is \( O(\gamma) = \omega(\log(N)) \). In contrast, in the original construction, \( \gamma \) is chosen dynamically after hashing which is much smaller with high probability.

Therefore, the trade-off is worse computation and communication complexity for a better round complexity. For \( |Y| \geq 1 \), again, \( C \) and \( S \) need to agree on \( \tau_i \) for each \( T_i \) in advance. Everything else follows the exact same way.

**Tuning \( \tau \) and \( \gamma \).** As mentioned, \( \tau \) can be further tuned in the alternative above. Recall that the smaller \( \tau \) is, the smaller \( \tau \cdot \gamma \) gets. Thus, instead of setting \( \tau = N \), we can set \( \tau < N \). Let \( Z \) be the random variable representing the size of a randomly populated entry in \( T \) with \( \tau \) entries. Then use Chernoff bound we have \( v < \gamma / \mu - 1 \geq 0 \), we have \( \Pr[Z > (1 + v)\mu] \leq \left( \frac{e^v}{1 + v}\right)^\mu \), where \( \mu = N / \tau \). Then, we set \( \gamma \) to be the smallest integer such that \( \Pr[Z > (1 + v)\mu] \leq 2^{-\lambda} \). One can then reduce the computation and communication costs by fine-tuning \( \tau \) and \( \gamma \).

**Extension to labeled PSI.** Labeled PSI introduced in [CHLR18] does not directly return \( X \cap Y \), but returns the payloads attached with \( X \cap Y \). In other words, for each \( x_i \in X \), there is a payload \( p_i \) associated with it. For each \( x_i \) that is also in \( Y \), output \( p_i \) to the client \( C \). Our construction can be extended to labeled PSI in a straightforward way. Instead of returning \( F_k(x_i) \) during the PIR query, the server returns \( (F_k(x_i), p_i \oplus F_k(x_i)) \) where \( F_k(\cdot) \) is another PRF. The OPRF query, therefore, outputs both \( F_k(y_i) \) and \( F_k(\cdot) \). \( C \) first check whether \( F_k(y_i) \) is in the decoded PIR answer, and then use \( F_k(\cdot) \) to decode the payloads. With similar analysis in appendix A, we achieve the functionality of labeled PSI.

**Using doubly efficient PIR.** Since PIR to PSI construction is generic, one can replace SimplePIR with arbitrary PIR constructions. A recent work [LMW22] shows that with \( O(|X|^{1+\omega(1)}) \) server offline computation and storage, the client can retrieve an entry with \( \text{polylog}(|X|) \) time and communication. However, as mentioned, this work is not yet practical, and thus we use SimplePIR in our concrete instantiation.

\(^9\)Recall that \( X' := \{F_k(x) | x \in X\} \), and the database \( T \) has entry \( T[\ell] := \{x' | x' \in X' \land H(x') = \ell\} \).
References


Finding Balance in Unbalanced PSI: 
A New Construction from Single-Server PIR


Finding Balance in Unbalanced PSI:
A New Construction from Single-Server PIR

[HWS+21] Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko, and Thomas Schneider. All the numbers are US: Large-scale abuse of contact discovery in mobile messengers. In NDSS 2021. The Internet Society, February 2021.


Finding Balance in Unbalanced PSI:
A New Construction from Single-Server PIR


[SACM21] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce M. Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In Tal Malkin and Chris Peikert, editors,
A Security Proofs

A.1 Proof of theorem 1

We construct a simulator $\text{Sim}$ that simulates $\text{C}$’s view as follows. $\text{Sim}$ is given $\text{C}$’s input set $Y$ and the output $I = X \cap Y$ (but no information about $X \setminus Y$). $\text{Sim}$ runs the honest $\text{C}$’s protocol to generate its view, playing the role of an honest server $S$ with the following exceptions:

1. For each element $t \in Y$, sample a random $v_t \xleftarrow{\$} \mathbb{G}$ as $H_1(t)^{k_S}$.
2. In the pre-processing phase, $\text{Sim}$ follows the protocol execution of an honest $S$ except that it skips Step 1 in sampling $k_S$. When computing $u_i$’s in Step 3a, $\text{Sim}$ computes $|I|$ of them by $\{H_2(H_1(t), v_t)\}_{t \in I}$, samples the remaining $(N - |I|)$ of them randomly from $\{0, 1\}^k$, and then randomly shuffles all these $u_i$’s.
3. In the online phase, upon receiving $\{z_i = H_1(y_i)^{k_S}\}_{i \in [M]}$ in Step 1, $\text{Sim}$ computes $\{z'_i := (v_{y_i})^{k_S}\}_{i \in [M]}$ and sends it back to $\text{C}$.
4. $\text{Sim}$ follows the rest of the protocol execution honestly and outputs $\text{C}$’s view.
Since the ideal-world server gets $\perp$ from the ideal functionality and the real-world server also outputs $\perp$, we only need to argue that C's view in the real-world protocol execution with the honest server $S$ is indistinguishable from its view when interacting with $\text{Sim}$ in the ideal world. We sketch a hybrid argument below.

$\mathcal{H}_0$ C's view in the real world.

$\mathcal{H}_1$ Same as $\mathcal{H}_0$ except that for each element $t \in X \cup Y$, sample a random $v_t \overset{\$}{\leftarrow} \mathbb{G}$ as $H_1(t)^{k_3}$. In particular, in the pre-processing phase Step 3a, $u_i := H_2(H_1(x_i), v_{x_i})$; in the online phase Step 2, $z'_i$ is computed as $z'_i := (v_y)^{k_c}$. This hybrid is computationally indistinguishable from $\mathcal{H}_0$ because $H_1$ is modeled as a random oracle and that DDH holds in $\mathbb{G}$. In more detail, we can construct a sequence of hybrids from $\mathcal{H}_0$ to $\mathcal{H}_1$ to change from $H_1(t)^{k_3}$ to $v_t$ one by one. To argue indistinguishability between every pair of intermediate consecutive hybrids, we can construct a reduction $\text{Red}$ to break the DDH assumption of $\mathbb{G}$. On receiving a DDH challenge tuple $(g_1, g_2, g_3)$, $\text{Red}$ sets $H_1(t) := g_1, g_3^{k_3} := g_2$, and $H_1(t)^{k_3} := g_3$. $\text{Red}$ builds a table $T_1 = \{(x, \phi)\}$ to answer all the hash queries to $H_1$, where $(t, g_1)$ is first added to $T_1$. To answer an $H_1$ query on $x$ that has never been queried before, $\text{Red}$ picks a random $\alpha_x \overset{\$}{\leftarrow} \mathbb{Z}_q$, adds an entry $(x, \phi = g^{\alpha_x})$ to $T_1$, and returns $\phi$ as $H_1(x)$. Whenever $\text{Red}$ needs to compute $H_1(x)^{k_3}$ for some $x \neq t$, it computes it as $(g_2)^{\alpha_x}$. Distinguishing between $H_1(t)^{k_3}$ and $v_t$ directly corresponds to distinguishing a DDH tuple from a random tuple for $(g_1, g_2, g_3)$.

$\mathcal{H}_2$ Same as $\mathcal{H}_1$ except that in the pre-processing phase Step 3a, for each $x_i \notin I$, we replace its $u_i$ by a random string from $\{0, 1\}^\delta$. We can again construct a sequence of hybrids from $\mathcal{H}_1$ to $\mathcal{H}_2$ to change the elements one by one from $u_i := H_2(H_1(x_i), v_{x_i})$ to $u_i \overset{\$}{\leftarrow} \{0, 1\}^\delta$. The only way to distinguish between the two intermediate consecutive hybrids is if C makes an $H_2$ query on $(H_1(x_i), v_{x_i})$. Let $q_2$ be the number of queries that C makes to $H_2$. Since $v_{x_i}$ is randomly sampled from $\mathbb{G}$, the probability that C makes such a query is at most $q_2/q$, which is negligible. This hybrid is exactly the output view of Sim.

A.2 Proof of theorem 2

We construct a simulator $\text{Sim}$ that interacts with the malicious client $C^*$ as follows and outputs whatever $C^*$ outputs in the end.

1. $\text{Sim}$ builds two tables $T_1 = \{(x, \phi)\}$ and $T_2 = \{((h, t), \psi)\}$ to answer the hash queries to $H_1$ and $H_2$ respectively. To answer an $H_1$ query on $x$ that has never been queried before, $\text{Sim}$ picks a random $\phi \overset{\$}{\leftarrow} \mathbb{G}$, adds an entry $(x, \phi)$ to $T_1$, and returns $\phi$ as $H_1(x)$. To answer an $H_2$ query on the pair $(h, t)$ which has never been queried before, $\text{Sim}$ samples a random $\psi \overset{\$}{\leftarrow} \{0, 1\}^\delta$, adds an entry $((h, t), \psi)$ to $T_2$, and returns $\psi$ as $H_2(h, t)$. For the queries that have been queried before, maintain consistency and return whatever was returned already.

2. In the pre-processing phase, $\text{Sim}$ follows the protocol execution of an honest server with the following exceptions: skip Step 1 in sampling $k_5$, and randomly sample each $u_i \overset{\$}{\leftarrow} \{0, 1\}^\delta$ in Step 3a. Let $U := \{u_i\}_{i \in [N]}$. $\text{Sim}$ also answers requests to $H_1, H_2$ as in item 1 above.

3. In the online phase, upon receiving $\{z_i\}_{i \in [M]}$ in Step 1, $\text{Sim}$ first samples $k_5 \overset{\$}{\leftarrow} \mathbb{Z}_q$ and then sends $\{z'_i\}_{i \in [M]}$ back to $C^*$ where $z'_i := z_i^{k_5}$ for all $i \in [M]$.

4. $\text{Sim}$ checks if $\exists((h, t), \cdot) \in T_2$ such that $t = h^{k_3}$. If so, $\text{abort}_1$. Finding Balance in Unbalanced PSI: A New Construction from Single-Server PIR
5. After sending \( \{z'_i\}_{i \in [M]} \), Sim initializes empty sets \( Z, V := \emptyset \) and answers \( H_1, H_2 \) queries as follows:

- For the queries that have been queried before, maintain consistency and return whatever was returned already.
- For each query \( x \) to \( H_1 \) that has not been queried before, randomly sample \( \phi \leftarrow \mathcal{G} \) and then check if \( \exists ((h, t), \cdot) \in T_2 \) such that \( h = \phi \) and \( t = h^{k_5} \). If so, \texttt{abort}_2; otherwise, add \((x, \phi)\) to \( T_1 \) and return \( \phi \) as \( H_1(x) \).
- For each query \((h, t)\) to \( H_2 \) that is not yet queried, check if \( \exists (x, \phi) \in T_1 \) such that \( h = \phi \) and \( t = h^{k_5} \). If not, answer the query as in item 1 above. Otherwise, add \( x \) to \( Z \). If \( |Z| > M \), \texttt{abort}_3. Otherwise, send \( x \) to the ideal functionality.
  - If the functionality returns \texttt{yes}, then Sim picks a random \( u \leftarrow \mathcal{G} \setminus U \setminus V \), adds \( u \) to \( V \), and adds \((h, t), u) \) to \( T_2 \), and returns \( u \) as \( H_2(h, t) \). Otherwise, as the ideal functionality returns \texttt{no}, Sim samples a random \( \psi \leftarrow \{0, 1\}^\delta \), adds \((h, t), \psi)\) to \( T_2 \), and returns \( \psi \) as \( H_2(h, t) \).

6. Sim answers the PIR queries following the protocol execution of an honest server.

Sim also answers queries to \( H_1, H_2 \) as in item 5 above.

Since the ideal-world server gets \( \bot \) from the ideal functionality and the real-world server also outputs \( \bot \), we only need to argue that \( C^* \)'s view in the real-world protocol execution with the honest server \( S \) is indistinguishable from its view when interacting with Sim in the ideal world. The only difference between \( C^* \)'s views in the real world and ideal world is how \( H_1 \) and \( H_2 \) queries are answered. Since \( H_1, H_2 \) are modeled as random oracles, it is easy to see that the two views only differ when Sim aborts. Next, we argue the three aborts happen with negligible probability. Let \( q_1, q_2 \) be the number of queries that \( C^* \) makes to \( H_1, H_2 \) respectively.

- \texttt{abort}_1 happens if \( C^* \) queries \( H_2 \) with \((h, h^{k_5}) \) before receiving any information about \( k_5 \). Since \( k_5 \) is sampled randomly from \( \mathbb{Z}_q \), this happens with probability at most \( q_2/q \), which is negligible.
- \texttt{abort}_2 happens if \( C^* \) queries \( H_1(x) \) which returns \( \phi \), while the entry \((\phi, h^{k_5}), \cdot)\) already exists in \( T_2 \). In other words, \( C^* \) makes a query \((\phi, h^{k_5}) \) for \( H_2 \) before knowing that \( H_1(x) = \phi \). This happens with probability at most \( q_1 \cdot q_2/q_2 \), which is negligible.
- If \texttt{abort}_3 happens, we can construct a reduction \( \text{Red} \) to break the \((q_1, M)\)-OMGDH assumption with challenges \((g_1, \ldots, g_9)\) as the challenge instance. \( \text{Red} \) works in the same way as Sim with the following exceptions: (1) \( \text{Red} \) uses \((g_1, \ldots, g_9)\) to reply to the \( H_1 \) queries from \( C^* \); (2) upon receiving \( \{z_i\}_{i \in [M]} \) in Step 2 of the online phase, \( \text{Red} \) queries the \((\cdot)^k \) oracle and gets back \( \{z'_i\}_{i \in [M]} \), which it sends back to \( C^* \); (3) whenever Sim needs to check if \( t = h^{k_5} \) for some \((h, t)\), \( \text{Red} \) calls the DDH oracle to decide whether \( t = h^{k_5} \). Finally, \( \text{Red} \) outputs \( Z \) if \( |Z| > M \). Note that \( \text{Red} \) decides to add an \( x \) to \( Z \) only if \( C^* \) makes an \( H_2 \) query on \((h, t)\) for which \( \exists (x, \phi) \in T_1 \) such that \( h = \phi \) and \( t = h^{k_5} \). Hence, the probability that \texttt{abort}_3 happens is bounded by the probability to break the \((q_1, M)\)-OMGDH assumption.

The committing property. We showed that our protocol is secure against a malicious client for the adaptive PSI functionality. Nevertheless, we can also show that a malicious client \( C^* \) cannot change its input set after sending \( \{z_i\}_{i \in [M]} \) in the online phase Step 1. In other words, even though the protocol achieves only an adaptive version of the PSI functionality, the adversary \( C^* \) is committed to all its inputs in Step 1, and hence it is not clear what advantage \( C^* \) could obtain by not making all these queries in later steps, in which case the adaptive functionality is equivalent to the standard functionality. The proof is more involved and we refer the reader to [JL10, Thm 2] for details.
A.3 Proof of theorem 3

We first prove the correctness of the protocol. In the online phase, for each $y_i \in Y$, its corresponding $z''_i$ is computed as $z''_i = H_2(H_1(y_i), ((H_1(y_1))^{k_c})^{k_c-1}) = H_2(H_1(y_i), (H_1(y_1))^{k_c})$. Given the fact that $H_2$ is modeled as a random oracle and the parameter choice of $\delta$, collisions of $z''_i$ happen with negligible probability. Furthermore, the parameter choice of $m$ guarantees that Cuckoo hashing fails with negligible probability in Step 3c of the online phase. $y_i$ is then put into the hash bin $Y_j$ for some $j \in \{h_1(z''_i), h_2(z''_i), h_3(z''_i)\}$. If $y_i \in X$, then $z''_i$ is put into all three bins of $X_k$ for each $k \in \{h_1(z''_i), h_2(z''_i), h_3(z''_i)\}$ in the pre-processing phase Step 3b, hence $z''_i \in X_j$, and $z''_i$ is put into the $H_{j,3}(z''_i)$-th entry of the table $T_j$ in pre-processing Step 4c. If $y_i \notin X$, then with overwhelming probability $z''_i \neq H_2(H_1(x), (H_1(x))^{k_c})$ for any $x \in X$ given the fact that $H_2$ is modeled as a random oracle and the parameter choice of $\delta$, thus $z''_i$ does not appear in $X_j$ (and hence not in $T_j$ either). In other words, $z''_i \in T_j[H_{j,3}(z''_i)]$ iff $y_i \in X$ with all but negligible probability. Finally, $z''_i \in R$, in Step 5a iff $z''_i \in T_j[H_{j,3}(z''_i)]$, which follows from the correctness of the underlying PIR protocol. This concludes the correctness proof.

To prove privacy, we construct a simulator $Sim$ that simulates $S$’s view as follows. $Sim$ runs the honest $S$’s protocol to generate its view, playing the role of an honest client $C$ with the following exceptions. In the online phase Step 1, send randomly sampled group elements to $S$. In Step 3c, send PIR queries for the first entry of each database. $S$’s view in the real-world protocol execution is indistinguishable from its view when interacting with $Sim$ in the ideal world. We sketch a hybrid argument below.

$H_0$ $S$’s view in the real world.

$H_1$ Same as $H_0$ except that in the online phase Step 3c, send PIR queries for the first entry of each database. This is indistinguishable from $H_0$ because of the security of the underlying PIR protocol.

$H_2$ Same as $H_1$ except that for each element $t \in Y$, sample a random $v_t \xleftarrow{\$} G$ as $H_1(t)^{k_c}$. This hybrid is exactly the output view of $Sim$. $H_2$ is computationally indistinguishable from $H_1$ because $H_1$ is modeled as a random oracle and that DDH holds in $G$. In more detail, we can construct a sequence of hybrids from $H_1$ to $H_2$ to change from $H_1(t)^{k_c}$ to $v_t$ one by one. To argue indistinguishability between every pair of intermediate consecutive hybrids, we can construct a reduction $Red$ to break the DDH assumption of $G$. On receiving a DDH challenge tuple $(g_1, g_2, g_3)$, $Red$ sets $H_1(t) := g_1, g^{k_c} := g_2$, and $H_1(t)^{k_c} := g_3$. $Red$ builds a table $T_1 = \{(x, \phi)\}$ to answer all the hash queries to $H_1$, where $(t, g_1)$ is first added to $T_1$. To answer an $H_1$ query on $x$ that has never been queried before, $Red$ picks a random $\alpha \xleftarrow{\$} Z_q$, adds an entry $(x, \phi = g^{\alpha x})$ to $T_1$, and returns $\phi$ as $H_1(x)$. Whenever $Red$ needs to compute $H_1(y)^{k_c}$ for some $y \neq t$, it computes it as $(g_2)^{\alpha \phi}$. Distinguishing between $H_1(t)^{k_c}$ and $v_t$ directly corresponds to distinguishing a DDH tuple from a random tuple for $(g_1, g_2, g_3)$.

A.4 Corrupted Client in fig. 4

Theorem 5. If $H_1, H_2$ are modeled as random oracles and the DDH problem is hard in $G$, then our protocol in fig. 4 securely computes the PSI functionality in fig. 1 for $M = 1$ against a semi-honest client when the protocol parameters are chosen as described in section 3.3.

Proof sketch. We sketch the construction of a simulator $Sim$ that simulates $C$’s view below, and skip the hybrid argument as it follows a similar structure as the proof of theorem 1. $Sim$ is given $C$’s input $y$ and $I$. $Sim$ runs the honest $C$’s protocol to generate its view, playing the role of an honest server $S$ with the following exceptions:
1. For the element \( y \), sample a random \( v_y \xleftarrow{\$} \mathbb{G} \) as \( H_1(y)^k \).

2. In the pre-processing phase, \( \text{Sim} \) samples a random table \( T \) with \( \tau \) entries, each entry containing a set of \( \gamma \) random strings of length \( \delta \). If \( y \in I \), then randomly replace a random string in the entry \( T[H_3(y)] \) by \( H_2(H_1(y), v_y) \). Perform the pre-processing step of PIR and send the pre-processed data hint to \( C \).

3. In the online phase, upon receiving \( z = H_1(y)^k \) in Step 3, \( \text{Sim} \) computes \( z' := (v_y)^k \) and sends it back to \( C \).

4. \( \text{Sim} \) follows the rest of the protocol execution honestly and outputs \( C \)'s view.

\section*{Theorem 6.} If \( H_1, H_2 \) are modeled as random oracles, the OMGDH problem is hard in \( \mathbb{G} \), then our protocol in Fig. 4 securely computes the PSI functionality in Fig. 1 for \( M = 1 \) against a malicious client when the protocol parameters are chosen as described in section 3.3.

\textbf{Proof sketch.} We sketch the construction of a simulator \( \text{Sim} \) that interacts with the malicious client \( C^* \) as follows and outputs whatever \( C^* \) outputs in the end. We skip the indistinguishability proof as it follows a similar structure as the proof of theorem 2.

1. \( \text{Sim} \) builds two tables \( T_1 = \{(x, \phi)\} \) and \( T_2 = \{((h, t), \psi)\} \) to answer the hash queries to \( H_1 \) and \( H_2 \) respectively. To answer an \( H_1 \) query on \( x \) that has never been queried before, \( \text{Sim} \) picks a random \( \phi \xleftarrow{\$} \mathbb{G} \), adds an entry \( (x, \phi) \) to \( T_1 \), and returns \( \phi \) as \( H_1(x) \). To answer an \( H_2 \) query on the pair \( (h, t) \) which has never been queried before, \( \text{Sim} \) samples a random \( \psi \xleftarrow{\$} \{0,1\}^\delta \), adds an entry \( ((h, t), \psi) \) to \( T_2 \), and returns \( \psi \) as \( H_2(h, t) \). For the queries that have been queried before, maintain consistency and return whatever was returned already.

2. In the pre-processing phase, \( \text{Sim} \) samples a random table \( T \) with \( \tau \) entries, each entry containing a set of \( \gamma \) random strings of length \( \delta \). Perform the pre-processing step of PIR and send the pre-processed data hint to \( C^* \).

3. In the online phase, \( \text{Sim} \) first answers the PIR query following the protocol execution of an honest server. Upon receiving \( z \) from \( C^* \) in Step 3, \( \text{Sim} \) first samples \( k_S \xleftarrow{\$} \mathbb{Z}_q \) and then sends \( z := z^{k_S} \) back to \( C^* \).

4. \( \text{Sim} \) checks if \( \exists ((h, t), \cdot) \in T_2 \) such that \( t = h^{k_S} \). If so, \texttt{abort}_1.

5. After sending \( z' \), \( \text{Sim} \) answers \( H_1, H_2 \) queries as follows:
   - For the queries that have been queried before, maintain consistency and return whatever was returned already.
   - For each query \( x \) to \( H_1 \) that has not been queried before, randomly sample \( \phi \xleftarrow{\$} \mathbb{G} \) and then check if \( \exists ((h, t), \cdot) \in T_2 \) such that \( h = \phi \) and \( t = h^{k_S} \). If so, \texttt{abort}_2; otherwise, add \( (x, \phi) \) to \( T_1 \) and return \( \phi \) as \( H_1(x) \).
   - For each query \( (h, t) \) to \( H_2 \) that is not yet queried, check if \( \exists (x, \phi) \in T_1 \) such that \( h = \phi \) and \( t = h^{k_S} \). If not, answer the query as in item 1 above. Otherwise, check if \( \text{Sim} \) has already sent a set to the ideal functionality. If so, \texttt{abort}_3; otherwise, send \( \{x\} \) to the ideal functionality. If the functionality returns \( \{x\} \), then \( \text{Sim} \) picks a random element \( u \) from the table entry \( T[H_3(x)] \), adds \( ((h, t), u) \) to \( T_2 \), and returns \( u \) as \( H_2(h, t) \). Otherwise, as the ideal functionality returns \( \emptyset \), \( \text{Sim} \) samples a random \( \psi \xleftarrow{\$} \{0,1\}^\delta \), adds \( ((h, t), \psi) \) to \( T_2 \), and returns \( \psi \) as \( H_2(h, t) \).
A.5 Corrupted Server in fig. 4

Theorem 7. If $H_1, H_2$ are modeled as random oracles, the DDH problem is hard in $\mathbb{G}$, and underlying PIR protocol satisfies receiver privacy, then our protocol in fig. 4 securely computes the PSI functionality in fig. 1 for $M = 1$ against a malicious server when the protocol parameters are chosen as described in section 3.3.

Proof sketch. We construct a simulator $\text{Sim}$ that interacts with the malicious client $S^*$ as follows and outputs whatever $S^*$ outputs in the end. $\text{Sim}$ follows the protocol playing the role of an honest client $C$ with the following exceptions:

1. $\text{Sim}$ builds two tables $T_1 = \{(x, \phi)\}$ and $T_2 = \{((h, t), \psi)\}$ to answer the hash queries to $H_1$ and $H_2$ respectively. To answer an $H_1$ query on $x$ that has never been queried before, $\text{Sim}$ picks a random $\phi \leftarrow \mathbb{G}$, adds an entry $(x, \phi)$ to $T_1$, and returns $\phi$ as $H_1(x)$. To answer an $H_2$ query on the pair $(h, t)$ which has never been queried before, $\text{Sim}$ samples a random $\psi \leftarrow \{0, 1\}^\delta$, adds an entry $((h, t), \psi)$ to $T_2$, and returns $\psi$ as $H_2(h, t)$. For the queries that have been queried before, maintain consistency and return whatever was returned already.

2. $\text{Sim}$ initializes an empty set $Z := \emptyset$.

3. In the online phase, for each $\ell \in [\gamma]$:
   (a) Rewind to Step 1 in fig. 4 if needed.
   (b) Prepare a PIR query for the $\ell$-th entry and sends it to $S^*$ in Step 1 in fig. 4.
   (c) Upon receiving the PIR response from $S^*$, recover the $\ell$-th entry of $T$ as a set of strings $R = \{r_1, \ldots, r_{\gamma}\}$.
   (d) For each $j \in [\gamma]$, find $(x_j, \phi_j(t_j), \psi_j)$ such that $(x_j, \phi_j) \in T_1$, $((\phi_j, t_j), \psi_j) \in T_2$, and $\psi_j = r_j$. In other words, from the existing records of replies to $H_1, H_2$, find $x_j$ such that $H_1(x_j) = \phi_j$ and $H_2(\phi_j, t_j) = r_j$. If $H_3(x_j) = \ell$:
      i. Rewind to Step 3 in fig. 4 if needed.
      ii. Randomly sample $k_C \leftarrow \mathbb{Z}_q$, compute $z_j := (\phi_j)^{k_C}$, and send $z_j$ to $S^*$.
      iii. Upon receiving $z_j'$ back, compute $z''_j := H_2(\phi_j, (z_j')^{k_C^{-1}})$ (to be consistent with $T_2$).
      iv. If $z''_j = r_j$, then add $x_j$ to $Z$, namely $Z := Z \cup \{x_j\}$.

4. Feed $Z$ to the ideal functionality.

We need to argue that $S^*$’s output along with $C$’s output in the real world is indistinguishable from $\text{Sim}$’s output along with $C$’s output in the ideal world. First, it is easy to see that the view of $S^*$ when interacting with $\text{Sim}$ is indistinguishable from its view when interacting with an honest client $C$. This is because (a) the element $z$ received by $S^*$ in Step 3 is computationally indistinguishable from a random element in $\mathbb{G}$ since DDH holds in $\mathbb{G}$, and (b) the receiver privacy of the underlying PIR protocol guarantees that the PIR query in Step 1 is indistinguishable from querying for an arbitrary entry.

Now we look at the output of the client $C$. In the real world, $C$ queries for $H_1(y)$-th entry of the database $T$, and it $C$ outputs $\{y\}$ if and only if the retrieved entry $R$ contains some $r_j$ such that $r_j = H_2(H_1(y), z''_j)$ where $z''_j$ is computed in Step 5a. Note that our simulator does exactly the same thing, and $Z$ includes the element $y$ if and only if all these requirements are met. Thus, $C$’s output in the ideal world is the same as in the real world.
Updatable Private Set Intersection Revisited: Extended Functionalities, Deletion, and Worst-Case Complexity

Saikrishna Badrinarayanan  
LinkedIn

Peihan Miao  
Brown University

Xinyi Shi  
Brown University

Max Tromanhauser  
Brown University

Ruida Zeng  
Brown University

ABSTRACT

Private set intersection (PSI) allows two mutually distrusting parties each holding a private set of elements, to learn the intersection of their sets without revealing anything beyond the intersection. Recent work by Badrinarayanan et al. (PoPETS’22) initiates the study of updatable PSI (UPSI), which allows the two parties to compute PSI on a regular basis with sets that constantly get updated, where both the computation and communication complexity only grow with the size of the small updates and not the large entire sets. However, there are several limitations of their presented protocols. First, they can only be used to compute the plain PSI functionality and do not support extended functionalities such as PSI-Cardinality and PSI-Sum. Second, they only allow parties to add new elements to their existing set and do not support arbitrary deletion of elements. Finally, their addition-only protocols either require both parties to learn the output or only achieve low complexity in an amortized sense and incur linear worst-case complexity.

In this work, we address all the above limitations. In particular, we study UPSI in both the addition-only and addition-deletion settings. We present new protocols for both settings that support plain PSI as well as extended functionalities including PSI-Cardinality and PSI-Sum, achieving one-sided output with semi-honest security (which implies two-sided output). In the addition-only setting, we also present a protocol for a more general functionality Circuit-PSI that outputs secret shares of the intersection. All of our protocols have worst-case computation and communication complexity that only grow with the size of the updates instead of the entire sets (except for a polylogarithmic factor). We implement our new UPSI protocols and compare with the state-of-the-art protocols for PSI and extended functionalities. Our protocols compare favorably when the total set sizes are sufficiently large, the new updates are sufficiently small, or in networks with low bandwidth.

CCS CONCEPTS

• Security and privacy → Privacy-preserving protocols; • Theory of computation → Cryptographic protocols.

KEYWORDS

Private Set Intersection, Secure Two-Party Computation, Oblivious Data Structure

1 INTRODUCTION

Private Set Intersection (PSI) enables two distrusting parties, each holding a private set of elements, to jointly compute the intersection of their sets without revealing anything other than the intersection itself. Despite its simple functionality, PSI and its related notions have found many real-world applications including online advertising measurement (deployed by Google Ads [6, 31]), secure password breach alerting (deployed by Google Chrome [8], Microsoft Edge [3], Apple iCloud Keychain [4], etc.), mobile private contact discovery (deployed by Signal [9, 33]), privacy-preserving contact tracing in a global pandemic (jointly deployed by Google and Apple [5, 14, 51]).

The last several decades have witnessed enormous progress towards realizing PSI efficiently using various techniques achieving both semi-honest and malicious security [15, 16, 19–22, 27, 34, 40, 42, 46].

In many real-world applications such as aggregated ads measurement and privacy-preserving contact tracking, PSI is performed on a regular (e.g., daily) basis with updated sets, where the updates can be small when compared to the entire sets. However, most of the existing work requires the two parties to perform a fresh PSI protocol every time. A recent work by Badrinarayanan et al. [13] initiates the study of updatable PSI (UPSI), which allows the two parties to compute set intersections for sets that regularly get updated. Their work presents protocols for updatable PSI where both the computation and communication complexity only grow with the size of the updates and are independent of the size of the entire sets (except for a logarithmic factor). As a result, these protocols are orders of magnitudes faster than a fresh PSI protocol, especially when the updates are significantly smaller than the entire sets. Nevertheless, there are several limitations with the protocols in [13].

• Functionality: All the protocols presented in [13] are restricted to the plain PSI functionality, crucially leveraging the fact that parties learn all the elements in the intersection. However, certain real-world applications require more refined PSI functionalities that do not reveal the entire intersection but instead only provide aggregate information about the intersection or enable restricted computation on the data in the intersection. As two specific examples that model many applications, PSI-Cardinality allows two parties to jointly learn the cardinality (or size) of their set intersection; PSI-Sum allows two parties, where one party additionally holds a private integer value associated with each element in her set, to jointly compute the sum of the associated integer values for all the elements in the intersection (together with the cardinality of the intersection).

• Addition-Only: [13] mainly focuses on the addition-only setting, where both parties can only add new elements to their existing old sets, and do not support arbitrary deletion of elements from their sets. Note that they present a protocol for UPSI with weak deletion, which allows the parties to refresh their sets every t days, namely, they will add a set of elements to their sets every day, and delete elements...
that were added to their sets \( t \) days ago. However, it does not support arbitrary deletion, and the daily computation and communication complexity additionally grows with \( t \).

- **Tradeoffs of the Addition-Only Protocols:** \([13]\) presents two protocols for addition-only UPSI, each with its own tradeoffs. In particular, one protocol crucially requires both parties to learn the output (namely, two-sided UPSI), which may not be applicable in certain applications. The other protocol allows a single party to learn the output (namely, one-sided UPSI), but it only achieves low computation and communication complexity in an amortized sense over many days; the worst-case complexity can be as high as linear in the entire sets. Note that one-sided UPSI is a strictly stronger functionality in the semi-honest setting (as considered in \([13]\)) since the output-receiving party can simply send the output to the other party so as to achieve two-sided UPSI.

### 1.1 Our Results

In this work, we address all the aforementioned limitations by presenting new UPSI protocols for extended functionalities, supporting both addition and deletion of elements, achieving one-sided output and low worst-case complexity in both computation and communication. All of our protocols are secure in the semi-honest model, hence one-sided UPSI is a stronger functionality. In the setting with both addition and deletion, we achieve a slightly more general functionality than PSI-Sum as defined in \([31, 36]\), where we do not reveal the cardinality of the intersection along with the sum.

Besides the functionalities of plain PSI, PSI-Cardinality, and PSI-Sum that we discussed above, we consider a more general functionality of Circuit-PSI \([15, 16, 39, 46, 49]\), where the two parties learn the cardinality of the intersection as well as an additive secret share of each element in it. This functionality allows the two parties to perform further computation over the shares afterwards.

Note that we only consider Circuit-PSI in the addition-only setting. The challenge in achieving Circuit-PSI with both addition and deletion is as follows. Intuitively speaking, when deleting elements from the intersection, the parties must learn which existing secret shares to delete from the intersection (unless the parties update their entire secret shared intersection, where the complexity grows with the entire sets, which is undesirable). Given that they know when a particular secret share (not the element itself) was added to the intersection, this essentially reveals more information than what the ideal functionality outputs. Crucially, note that in the case of plain PSI with addition and deletion, this is not a problem since the ideal functionality’s output also reveals when a particular element was added and deleted; and in the case of PSI-Cardinality or PSI-Sum, parties only learn aggregated information and this challenge doesn’t arise in the protocol design. We summarize our results in comparison with \([13]\) in Table 1.

### Experiments

We implement all our protocols and compare their performance with the state-of-the-art protocols for PSI and extended functionalities \([16, 46]\). As our communication grows with the size of the update and not the entire input (except by a logarithmic factor), we demonstrate a significant improvement, up to orders of magnitude, when the input sets grow sufficiently large with smaller updates. Although our usage of public key operations damps the asymptotic impact on computation, in realistic WAN settings, our protocols are able to outperform prior work in end-to-end running time.

### 1.2 Related Work

There has been a long line of work towards realizing PSI efficiently using various techniques including Diffie-Hellman-based \([30, 31, 35]\), RSA-based \([12, 23]\), circuit-based \([29, 41–43]\), oblivious transfer (OT)-based \([17, 24, 34, 39, 44]\), fully homomorphic encryption (FHE)-based \([18, 19, 21]\), and vector oblivious linear evaluation (VOLE)-based \([15, 22, 27, 46, 49]\) approaches, achieving both semi-honest and malicious security \([15, 18, 20, 37, 40, 46, 48]\).

As discussed earlier, certain applications require PSI with extended functionalities that do not reveal the entire intersection but rather enable restricted computation on the elements in the intersection. PSI-Cardinality and PSI-Sum model many applications such as aggregated ads measurement \([31, 36]\) and privacy-preserving contact tracing \([14, 51]\). More generally, Circuit PSI \([15, 16, 29, 42, 46, 49]\) enables the two parties to learn secret shares of the set intersection, which can be used to securely compute any function using generic secure two-party computation protocols \([28, 53]\). However, all these approaches study PSI or PSI with extended functionalities in the standalone setting, which do not support small updates to the sets beyond running a fresh protocol after each update.

To the best of our knowledge, \([13]\) is the only work that formalizes and studies PSI in the updateable setting, which we have extensively discussed above. Another related work is \([10]\), which studies delegatable PSI with small updates. Specifically, they allow multiple clients to outsource their (encrypted) private sets and delegate PSI computation to a cloud server. Clients can perform efficient updates on their outsourced sets where the computation and communication only grow with their updates. However, both the computation and communication costs of computing PSI still grow with size of the entire sets, and their protocol crucially requires the existence of a server.

### 2 TECHNICAL OVERVIEW

We discuss the technical challenges and novelties in this work. We start with addition-only UPSI. Let \(X, Y\) denote the old sets of the two parties \(P_0, P_1\) respectively, and let \(X_d, Y_d\) denote their new added sets on Day \(d\). For simplicity, assume \(|X| = |Y| = N\) and \(|X_d| = |Y_d| = N_d\). Our constructions work for different set sizes as well. Recall that we are mostly interested in the scenario when the set updates are significantly smaller than the entire sets, namely \(N \gg N_d\). The parties have already learned \(I = X \cap Y\) of the old sets, and they would like to learn the updated intersection \(I_d = (X \cup X_d) \cap (Y \cup Y_d)\) (for one-sided UPSI, only \(P_0\) learns the output). Our goal is to make the computation and communication complexity only grow with \(N_d\) and not \(N\) (except for logarithmic factors).

**Prior Work’s Approach.** In the addition-only setting, prior work \([13]\) observes that it suffices to learn the set difference \(I_d \setminus I\), which, from \(P_0\)’s perspective, can be split into two disjoint sets, \((X \cap Y_d)\) and \((X_d \cap (Y \cup Y_d))\). The first set \((X \cap Y_d)\) can be computed by leveraging the Diffie-Hellman-based PSI \([30, 35]\), which naturally supports...
small updates to $Y$ with computation and communication complexity only growing with $N_d$. To learn the second set $(X_d \cap (Y \cup Y_d))$, [13] presents two different approaches for two-sided and one-sided UPSI. In the two-sided case, they further leverage the algebraic structures of Diffie-Hellman-based PSI coupled with the information revealed to $P_1$ from the ideal functionality. In the one-sided case, their key idea is to let $P_1$ store an encrypted version of her set on $P_0$’s side; on each day, she updates this encrypted dataset based only on her new input $Y_d$. Here, they require a data structure that allows $P_1$ to obliviously update the dataset without leaking any information about $Y_d$. Meanwhile, the data structure should allow $P_0$ to obliviously query this dataset and compute on the encrypted data (by interacting with $P_1$) to learn the intersection without leaking any information to $P_1$. [13] constructs such an oblivious data structure via a binary tree and uses additively homomorphic encryption to compute on encrypted data.

**PSI with Extended Functionalities.** The above approach crucially relies on the fact that parties learn the set intersections $I$ and $I_d$, and that $I_d \setminus I$ can be split into two disjoint sets, which can be learned individually. However, if they want to learn PSI with extended functionalities such as PSI-Cardinality, they cannot split $|I_d \setminus I|$ into $|X \cap Y_d|$ and $|X_d \setminus (Y \cup Y_d)|$ because the individual cardinalities reveal more information than the ideal functionality.

First, by carefully re-crafting the homomorphic operations on the encrypted data in the oblivious data structure, we design a method that reveals only the number of elements that are matched between $X_d$ and the encrypted dataset $(Y \cup Y_d)$. In this way, we enable $P_0$ to learn $|X_d \cap (Y \cup Y_d)|$ from the oblivious data structure, instead of $(X_d \cap (Y \cup Y_d))$.

Next, we revisit the approach to computing $X \cap Y_d$, which leverages Diffie-Hellman-based PSI in [13]. Unfortunately, it does not extend to updatable cardinality. To address this challenge, our idea is to compute $X \cap Y_d$ (in fact $|X \cap Y_d|$), not using Diffie-Hellman-based PSI, but symmetrically on $P_1$’s side using the above oblivious data structure. In particular, we let $P_0$ store an encrypted version of his set on $P_1$’s side that supports efficient and oblivious updates and queries. This way it can efficiently allow $P_1$ to learn $|X \cap Y_d|$.

The last missing piece is to learn the sum of $|X_d \cap (Y \cup Y_d)|$ and $|X \cap Y_d|$ rather than individual values. At a high level, in the above approach, $P_0$ learns the cardinality $|X_d \cap (Y \cup Y_d)|$ by decrypting a set of (homomorphically evaluated) ciphertexts and counts the number of 0’s in them. This happens similarly for $P_1$ to learn $|X \cap Y_d|$.

To enable $P_0$ to learn $|X_d \cap (Y \cup Y_d)| + |X \cap Y_d|$, we combine the two sets of ciphertexts, randomly shuffle them, and then let $P_1$ decrypt. Since we require that neither party be able to decrypt these ciphertexts, we use a 2-out-of-2 threshold homomorphic encryption scheme, and the parties will only jointly decrypt all the ciphertexts after the random shuffling. This protocol can be further extended to PSI-Sum and Circuit-PSI by attaching a payload to each element and further leveraging additive homomorphic encryption. We refer to Section 3 for more details.

**Lowering Worst-Case Complexity.** The above construction heavily relies on the oblivious data structure presented in [13]. A critical drawback of the data structure is that it only achieves low complexity in an amortized sense, namely the average complexity over many days is low. However, the worst-case complexity can be as high as linear in the entire sets on certain days.

At a high level, [13] works as follows. The encrypted dataset is maintained in a binary tree structure. When the data owner $P_1$ updates an element $x$, it defines a root-to-leaf path associated with $x$ and aims to put the ciphertext $Enc(x)$ somewhere on this path (which will be used by the querier $P_0$ when it queries for $x$). A naive approach is to refresh this designated path in the encrypted dataset on $P_0$’s storage, where the complexity only grows with the depth of the tree. However, revealing the path to $P_0$ reveals information about $x$ being added to the tree. In other words, we need a mechanism for $P_1$ to add an encrypted element to a designated path on $P_0$’s storage while hiding the path from $P_0$. In [13], this is achieved by a series of operations that requires updating an entire level of the tree rather than updating elements individually. So, the amortized complexity grows with the depth of the tree (logarithmic in $N$ and not linear). However, in the worst case, $P_1$ needs to update the leaf level of the tree, which is linear in $N$. We need a new method for $P_1$ to update the tree in an oblivious way and in which the worst-case complexity is logarithmic in $N$.

---

### Table 1: Summary of our results in comparison to [13], including functionality, one-sided or two-sided output, support of addition and deletion of elements, and computation and communication complexity. PSI-Sum\* denotes the variant of PSI-Sum that does not reveal the cardinality. $N$ denotes the size of the entire sets and $N_d$ denotes the size of the $d$-th update. $O^*(\cdot)$ denotes amortized complexity.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Functionality</th>
<th>Output</th>
<th>Addition/Deletion</th>
<th>Comp. &amp; Comm. Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>[13, UPSI-add-two]</td>
<td>PSI</td>
<td>Two-Sided</td>
<td>Addition-Only</td>
<td>$O(N_d)$</td>
</tr>
<tr>
<td>[13, UPSI-add-one]</td>
<td>PSI</td>
<td>One-Sided</td>
<td>Addition-Only</td>
<td>$O^*(N_d \cdot \log N)$</td>
</tr>
<tr>
<td>Figure 10, UPSI-Add_psi</td>
<td>PSI</td>
<td>One-Sided</td>
<td></td>
<td>$O(N_d \cdot \log N)$</td>
</tr>
<tr>
<td>Figure 4, UPSI-Add_psic</td>
<td>PSI-Cardinality</td>
<td></td>
<td>Addition-Only</td>
<td>$O(N_d \cdot \log N)$</td>
</tr>
<tr>
<td>Figure 4, UPSI-Add_psim</td>
<td>PSI-Sum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Figure 4, UPSI-Add_psit</td>
<td>Circuit-PSI</td>
<td>Secret Shared</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[13, UPSI-del]</td>
<td>PSI</td>
<td>Two-Sided</td>
<td>Weak Deletion</td>
<td>$O(N_d \cdot t)$</td>
</tr>
<tr>
<td>Figure 9, UPSI-Delete_psic</td>
<td>PSI</td>
<td>One-Sided</td>
<td>Addition &amp; Deletion</td>
<td>Single Deletion $O(N_d \cdot \log N)$ Arbitrary Deletion $O(N_d \cdot \log^2 N)$</td>
</tr>
<tr>
<td>Figure 9, UPSI-Delete_psim</td>
<td>PSI-Cardinality</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Figure 9, UPSI-Delete_psit</td>
<td>PSI-Sum*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Our solution takes inspiration from the Path ORAM construction [50]. At a high level, instead of updating the designated path, \( P_1 \) picks a random path every time, and "pushes down" the elements in that path as much as possible. The access pattern of tree updates consist of random paths, hence are oblivious to \( P_0 \). Note that Path ORAM has an additional logarithmic factor because of limited registers (or client storage) and tree recursions. We can get rid of this factor so as to achieve a single logarithmic factor since we do not have this restriction in the UPSI setting. We refer to Section 3 for more details on how to achieve this and overcome other challenges along with a full description of addition-only UPSI protocols.

Supporting Deletion. Our oblivious data structure is inspired by ORAM, but the manner in which ORAM handles deletions (or modification) of memory content does not work for us. As we discussed above, every element \( x \) has a designated path associated with it. In Path ORAM, this designated path is stored in the so-called position map. Whenever \( x \) is accessed (or modified), \( x \) will be re-allocated to a new, freshly sampled random designated path. Accessing the old designated path of \( x \) is not an issue in Path ORAM because the position map is also maintained as an ORAM recursively (and so, known only to the data owner). However, in our construction, the designated path of \( x \) is also known to the querier \( P_0 \), hence accessing it is not oblivious to \( P_0 \).

If we consider a restricted yet useful UPSI setting where every element can only be added and deleted at most once, then we can create two trees — one for addition and one for deletion, each having their respective designated paths for every element \( x \). When \( P_0 \) queries \( x \), he compares it with encrypted elements in both designated paths. However, the crucial challenge is when \( x \) is not in the intersection, we need to further hide to \( P_0 \) whether \( x \) was never added to the dataset or \( x \) was added and then deleted. To achieve this, we developed a specialized efficient secure two-party computation protocol utilizing equality testing that outputs secret shares of the actual intersection (or the respective outputs in the other functionalities) that can later be combined by both parties to compute the final output.

With this in mind, we now extend this construction to allow for unlimited additions and deletions of the same element. The key idea is that we can in fact use only one tree and assign a single designated path to \( x \). Whenever \( P_1 \) adds or deletes \( x \), she will add a new \( \text{Enc}(x) \) (or \( \text{Enc}(-x) \)) to that designated path. Recall that we can leverage the techniques of Path ORAM to update the tree by accessing a random path and "pushing down" elements to place them correctly until they intersect with their designated path, hence the access pattern remains oblivious to \( P_0 \).

There are several other challenges that arise to handle deletion. For instance, we need to bound the maximum node size of the tree when there are unlimited, repeated elements being added to the same path. A nuanced analysis shows that we can do so and this only incurs an additional logarithmic factor when compared to the addition-only protocols. When considering PSI-Cardinality and PSI-Sum, we carefully design the values being encrypted at the time of addition or deletion (i.e., not simply \( \text{Enc}(x) \) or \( \text{Enc}(-x) \)) to enable computing secret shares of the desired output.

Another challenge arises in plain UPSI with addition and deletion. Consider a scenario where \( x \in X \) and in the same day \( P_0 \) removes \( x \) and \( P_1 \) adds \( x \) to \( Y \). After these updates, \( x \) should not appear in the intersection. However, simply replicating the protocol for PSI-Cardinality — which separately processes \( X_d \cap (Y \cup Y_d) \) and \( Y_d \cap X \) before obliviously combining them — reveals that it has been added and removed. Solving this requires carefully interleaving the additions and deletions with the computation over the ciphertexts. We refer to Section 4 for more details of how to handle these challenges and the full description of the UPSI protocols with both addition and deletion.

### 3 ADDITION-ONLY UPSI

#### 3.1 Definition

In this section, we formalize the ideal functionality and security definition for addition-only UPSI. Consider two parties \( P_0 \) and \( P_1 \) who wish to run PSI on a daily basis with updated sets. In the addition-only setting, they each hold a private set and add new elements to their respective sets each day. They want to jointly compute their set intersection (or extended functionalities) on their updated sets without revealing anything beyond that. We formalize addition-only UPSI as a special case of secure two-party computation with a reactive functionality defined in Figure 1.

**Initialization:** \( X = \emptyset \) and \( Y = \emptyset \).

**Day \( d \):**

- **Public Parameters:** The number of additions that \( P_0 \) and \( P_1 \) are performing: \(|X_d| \) and \(|Y_d|\), respectively.
- **Inputs:**
  - \( P_0 \) inputs a set \( X_d \subseteq \{0, 1\}^* \) where \( X_d \cap X = \emptyset \). In \( \mathcal{F}_{\text{UPSI-Addsum}} \), \( X_d \) includes an integer value associated with each set member (i.e., \( q_i \) is associated with \( x_i \in X_d \)).
  - \( P_1 \) inputs a set \( Y_d \subseteq \{0, 1\}^* \) where \( Y_d \cap Y = \emptyset \).
- **Update:** On receiving the inputs from both parties, the ideal functionality updates \( X = X \cup X_d \) and \( Y = Y \cup Y_d \).
- **Output:**
  - In \( \mathcal{F}_{\text{UPSI-Addsum}} \), \( P_0 \) learns the intersection \( I_d = X \cap Y \).
  - In \( \mathcal{F}_{\text{UPSI-Addsum}} \), \( P_0 \) learns the cardinality of the intersection \( |I_d| = |X \cap Y| \).
  - In \( \mathcal{F}_{\text{UPSI-Addsum}} \), both parties learn \( C_d = |X \cup Y| \).
  - For new element \( z \) being added to the intersection, \( P_0 \) learns \(|z|_0 \) and \( P_1 \) learns \(|z|_1 \) as an additive secret share for \( z \).

**Figure 1: Ideal functionalities for one-sided addition-only UPSI: \( \mathcal{F}_{\text{UPSI-Addsum}} \).**

Let \( X([D]) = \{X_1, \ldots, X_D\} \) and \( Y([D]) = \{Y_1, \ldots, Y_D\} \) be the inputs for \( P_0 \) and \( P_1 \) after \( D \) days, respectively. Let \( \text{View}_b^{\Pi, D}(X([D]), Y([D])) \) and \( \text{Out}^{\Pi, D}_b(X([D]), Y([D])) \) be the view and outputs of \( P_b \) (for \( b \in \{0, 1\} \)) in the protocol \( \Pi \) at the end of \( D \) days, respectively. For a functionality \( F \), let \( \mathcal{F}_b \) be the output for \( P_b \) in the \( D \) days. Note that \( \mathcal{F}_b = I \) in all the functionalities except for \( \mathcal{F}_{\text{UPSI-Addsum}} \).

**Definition 3.1 (One-Sided Addition-Only UPSI).** A protocol \( \Pi \) is semi-honest secure with respect to ideal functionality \( F \in \{\mathcal{F}_{\text{UPSI-Addsum}} \cup \mathcal{F}_{\text{UPSI-Addsum}} \cup \mathcal{F}_{\text{UPSI-Addcirc}} \} \) if there exists PPT simulators \( \text{Sim}_0 \) and \( \text{Sim}_1 \) such that, for any \( D \in \mathbb{N}^+ \) and any inputs \( (X([D]), Y([D])) \),

\[
\left\{ \text{View}^{\Pi, D}_b(X([D]), Y([D])), \text{Out}^{\Pi, D}_b(X([D]), Y([D])) \right\}
\]
3.2 Construction

Construction Overview. As discussed in the technical overview, each party stores an encrypted version of its set on the other party’s storage. We first describe our new oblivious data structure maintained in a binary tree. Say $P_1$ is the data owner, who stores her encrypted set on $P_0$’s side. Initially, the binary tree is empty with depth 0. Each node of the tree has a maximum capacity of $\sigma$ elements. As $P_1$ adds new elements to the tree, she will gradually increase the tree depth. Each element $x$ is associated with a designated path computed by $P_2(x)$, where $P$ is a pseudorandom function and $k$ is a secret key known to both parties. When $P_1$ adds an element $x$ to the tree, she will add $x$ to the one of the nodes in the root-to-leaf path ending at leaf node $P_2(x)$, but in an oblivious way. $P_1$ first adds (encrypted) $x$ to the root node of the tree. Then she samples a random root-to-leaf path of the tree. For every element $y$ in that random path (note that this includes $x$ because $x$ was just added to the root), $y$ has a designated path $P_2(y)$, and $P_1$ will push down $y$ along its designated path, but also within the random path, as much as possible subject to the restriction that no node exceeds the maximum capacity of $\sigma$. If there are extra elements that cannot fit into the maximum capacity of the random path, $P_1$ puts them into a stash, which has maximum capacity $\rho$. Both $\sigma$ and $\rho$ are defined as part of the security parameters of the protocol. Note that this process is oblivious to $P_0$ since the access pattern for any element is a random path along with the stash. We present this subroutine formally as UpdateTree in Figure 2. This subroutine will also be used in our UPSI with both addition and deletion protocols, with slight modifications. We discuss more details in Section 4.

When $P_0$ queries for an element $x$ in the encrypted tree, he will identify the designated path $P_2(x)$ and take out all the elements from the tree, combine them with all the elements from the stash (because $x$ could potentially have been put there as well). This process is presented formally as a subroutine GetPath in Figure 3.

To compute PSI-Cardinality, $P_0$ homomorphically subtracts $x$ from each candidate encryption and then multiplies it by a random scalar, so that the encryption becomes 0 if it is a match or random otherwise. Symmetrically, $P_0$ stores an encrypted version of his set on $P_1$’s side, and $P_1$ retrieves and homomorphically computes on a set of candidate encryptions. The two parties then combine all these candidate encryptions, randomly shuffle them, and decrypt them to count the number of 0’s. Since neither party should be allowed to decrypt the encryptions by itself, we use a (2, 2)-threshold additively homomorphic encryption scheme. We can further extend the protocol to compute PSI-Sum and Circuit-PSI by attaching a payload to each element and leveraging additive homomorphism on these payload. We present the protocols $\Pi_{UPSI-Add}^*$, $\Pi_{UPSI-AddSum}$, and $\Pi_{UPSI-AddCircuit}$ in Appendix B.

Note that for addition-only plain UPSI $\mathcal{F}_{UPSI-Add}$ case, we don’t have to store two trees one on each side. Instead, we can plug our new oblivious data structure into the addition-only UPSI protocol from prior work [13, $\Pi_{UPSI-add-one}$] to achieve better concrete efficiency than the two-tree solution and much lower worst-case complexity than [13]. We present the protocol $\Pi_{UPSI-Add_{ps}}$ in Appendix B.

Notation. We use $\lambda$, $\kappa$ to denote the computational and statistical security parameters, respectively. For an integer $n \in \mathbb{N}$, $[n]$ denotes the set $\{1, \ldots, n\}$. A 2-out-of-2 additive secret share of a value $x \in \mathbb{Z}_q$ is denoted as $([x])_0$, $([x])_1$ where $([x])_0 \leftarrow \mathbb{Z}_q$ and $([x])_0 + ([x])_1 = x \mod q$. PPT stands for probabilistic polynomial time. By $\approx$ we mean two distributions are computationally indistinguishable.

Let $\Pi = (\text{KeyGen, Enc, PartDec, FullDec})$ be a $(2, 2)$-threshold additively homomorphic encryption scheme (see definition in Appendix A) over plaintext space $\mathbb{Z}_q$ for a prime $q$. Without loss of generality we assume all the set elements are in $\mathbb{Z}_q$ (if not, we can apply a collision-resistant hash function $H : \{0, 1\}^* \rightarrow \mathbb{Z}_q$ on all the elements and perform PSI on the hash outputs). Let $F : \{0, 1\}^3 \times \mathbb{Z}_q \rightarrow \{0, 1\}^3$ be a pseudorandom function (PRF). For a bit string $x \in \{0, 1\}^n$, let $s_{i,1}$ denote the prefix of $s$ of length $i$ (for $i \in [n]$).

Consider a binary tree data structure with tree height $L$ and $2^L$ leaves, let $\ell \in [0, 1, \ldots, 2^L - 1]$ denote the $\ell$-th leaf node of the tree. Any leaf node $F$ defines a unique path from the root to the leaf. We use $\mathcal{P}(\ell)$ to denote such a path, and $\mathcal{P}(\ell, k)$ to denote the node in $\mathcal{P}(\ell)$ at level $k$ of the tree (for $k \in \{0, 1, \ldots, L\}$). Let $\sigma$ denote the maximum tree node size and $\rho$ denote the stash size.

3.3 Complexity, Correctness and Security

Computation and Communication Complexity. On each day $d$, let the entire set sizes of the two parties be $N$ and $M$, respectively. Let the update set sizes be $n$ and $m$, respectively. Then both the computation and communication complexity are $O(n \log M + m \log N)$, assuming $\sigma$ and $\rho$ are both $O(1)$.

Correctness and Security. We state the theorem below and defer its proof to Appendix C.

Theorem 3.2. Assuming $\Pi$ is a secure (2, 2)-threshold additively homomorphic encryption scheme, $F$ is a pseudorandom function, the protocols $\Pi_{UPSI-Add}$, $\Pi_{UPSI-AddSum}$, $\Pi_{UPSI-AddCircuit}$ (Figure 4) securely realize the ideal functionalities $\mathcal{F}_{UPSI-Add}$, $\mathcal{F}_{UPSI-AddSum}$, $\mathcal{F}_{UPSI-AddCircuit}$ (Figure 1), respectively, against semi-honest adversaries.

4 UPSI WITH ADDITION AND DELETION

4.1 Definition

The security definition for UPSI with both addition and deletion follows similarly as in Section 3.1. Let $X_{[D]} = \{(x_1^*, \chi_1^-), \ldots, (x_D^*, \chi_D^-)\}$ and $Y_{[D]} = \{(y_1^+, \chi_1^-), \ldots, (y_D^+, \chi_D^-)\}$ be the inputs for $P_0$ and $P_1$ after $D$ days, respectively. Here, $X_d^*$ denotes the elements to be added to $P_0$’s set on day $d$, and $\chi_d^-$ denotes the elements to be deleted from $P_0$’s set on day $d$; similarly, $Y_d^+$ and $\chi_d^-$ denote the elements to be added and deleted, respectively, for $P_1$ on day $d$. The ideal functionalities are defined in Figure 5. Note that for $\mathcal{F}_{UPSI-Delete}$, we achieve a slightly more general functionality than PSI-Sum as defined in [31, 36] (which is the definition used in our addition-only protocol) in that our functionality does not have to reveal the
Subroutine UpdateTree$(\{x_i\}_{i=1}^n, \{p_i\}_{i=1}^n, D, S, F(\cdot), Enc_{jk}(\cdot))$:

1. Let $N$ be the total number of elements (excluding dummy ones) in the tree $D$ and stash $S$ after inserting $\{x_i\}_{i=1}^n$. Extend the tree depth to reach $L = \lceil \log_2 N \rceil$ if needed. Add empty nodes in the new levels of $D$.
2. For each element and payload pair $(x_i, p_i)$ for $i \in [n]$:
   a. Uniformly sample a random leaf node $t_i \sim \{0, 1, \ldots, 2^L - 1\}$ of the tree $D$.
   b. Remove all the elements from the path $P(t_i)$ of the tree $D$. Remove all the elements from the stash $S$. Combine all the removed elements (excluding dummy ones) with $(x_i, p_i)$ to get path$_i$. In the UPSI with addition and deletion protocols, if there are elements with opposite values, namely $(z, p)$ and $(z, -p)$, then remove both from path$_i$.
   c. For $k$ from $L$ down to 0:
      1. Consider the tree node $P(t_i, k)$ at level $k$, remove up to $\sigma$ elements $(z, p)$ from path$_i$, such that $P(t_i, k) = P(F(z)_{[1:L]}, k)$, and add these elements to the node $P(t_i, k)$ of $D$.
   d. Replace the stash $S$ with all the elements left in path$_i$. If there are more than $\rho$ elements left in path$_i$, abort.
   e. Pad every node in the path $P(t_i)$ with dummy elements to reach a size of $\sigma$. Pad the stash $S$ with dummy elements to reach a size of $\rho$.
3. For each $i \in [n]$, gather all the elements in the path $P(t_i)$ and encrypt them to get $\text{updates}_i = \{(Enc_{jk}(x_i), Enc_{jk}(p_i))\}_{j=1}^\sigma$. Encrypt all elements in the stash $S$ to get $\tilde{S} = \{(Enc_{jk}(x_i), Enc_{jk}(p_i))\}_{j=1}^\sigma$. Output $\{(\text{updates}_i, t_i)\}_{i=1}^n, \tilde{S}$.

Figure 2: Subroutine UpdateTree that outputs a succinct update for the tree $D$ that does not reveal the elements being added.

Subroutine GetPath$(\bar{D}, \bar{S}, F(\cdot), x)$:

1. Let $L$ be the height of the tree $D$.
2. Compute the leaf node for the path containing $x$ as $\ell := F(x)_{[1:L]}$.
3. Collect all the elements in the path $P(\ell)$, combine them with the stash $S$ to get path$_\ell = \{(Enc(y_i), Enc(p_i))\}_{i=1}^{L+\rho}$, and output path.

Figure 3: Subroutine GetPath that outputs a collection of potential matching elements with $x$ in the encrypted tree $D$ with stash $\bar{S}$ organized according to the pseudorandom function $F$.

cardinality $C_d$ along with $V_d$. Let $\mathcal{F}_0$ be the output for $P_0$ for all functionalities. Note that we don’t consider the Circuit-PSI functionality in this setting, so $P_1$ has no output in the definition.

Definition 4.1 (One-Sided UPSI with Addition and Deletion). A protocol $\Pi$ is semi-honest secure with respect to ideal functionality $F \in \{F_{\text{UPSI-Add}^\alpha}, F_{\text{UPSI-Add}^\gamma}, F_{\text{UPSI-Add}^\omega}, F_{\text{UPSI-Delete}^\alpha}, F_{\text{UPSI-Delete}^\gamma}, F_{\text{UPSI-Delete}^\omega}\}$ if there exists PPT simulators $\text{Sim}_0^{\Pi} \text{ and Sim}_1^{\Pi}$ such that, for any $D \in \mathbb{N}^+$ and any inputs $(X[D], Y[D])$,

$$
\left(\text{View}_0^{\Pi, D}(X[D], Y[D])\right) \equiv \left(\text{Sim}_0^{\Pi}(\ell^\lambda, X[D], \mathcal{F}_0(X[D], Y[D]))\right),
$$

$$
\left(\text{View}_1^{\Pi, D}(X[D], Y[D]), \text{Out}_0^{\Pi, D}(X,D), Y[D])\right).
$$

4.2 Construction Overview

In this section, we give an overview of our UPSI protocols with both addition and deletion. The oblivious data structure presented in Section 3.2 only supports adding new elements to the tree. We first discuss how to extend the construction to also allow for deletion of elements from the tree. Recall that each element $x$ is associated with a (pseudorandom) designated path known to both parties. When $P_1$ adds an element $x$ to the tree, she will first add (encrypted) $-x$ to the root node of the tree. Then she samples a random root-to-leaf path of the tree and push down elements along that random path, as described in UpdateTree (Figure 2). When deleting an element $x$ from the tree, $P_1$ will follow the exact same approach, except that she will add (encrypted) $-x$ to the root node and then push down. The designated paths for $x$ and $-x$ are the same, so that $P_0$ has a single designated path to compare his elements with. This modified UpdateTree process remains oblivious to $P_0$ because the access pattern for deletion of elements continues to be a random path along with the stash. Note that since additions and deletions of the same element have the same designated path, there is a higher probability of stash overflow if we use the same parameters of maximum node capacity $\sigma$ and maximum stash capacity $\rho$ as in the addition-only setting; hence we need to increase both parameters for our new protocols. We discuss the parameter implications in the security proofs.

To compute PSI-Cardinality, we take a different approach from the addition-only protocols. When $P_0$ queries for an element $x$ in the encrypted tree, he can still identify the designated path and all the candidate encryptions using GetPath (Figure 3). However, there are both $Enc(x)$ and $Enc(-x)$ among the candidates. In case $x$ was added and then deleted from tree, it should be indistinguishable to $P_0$ from the case where $x$ was never added to the tree. We construct a subprotocol $\Pi_{\text{CombinePath}}$ (Figure 7) for the two parties to jointly compute aggregate information about $x$ being matched to the candidate encryptions. In particular, for every candidate encryption, the two parties will learn a secret share of $+1$ if the encryption is $Enc(x)$, $-1$ if the encryption is $Enc(-x)$, and 0 otherwise. The parties can eventually add up these secret shares to learn the PSI-Cardinality.

To compute PSI-Sum, if the two parties compute secure shares of $+\sigma$ or $-\sigma$ in $\Pi_{\text{CombinePath}}$, we present the $\Pi_{\text{CombinePath}}$ protocol in Section 4.3 and our PSI-Cardinality/Sum protocols in Section 4.4. It is worth noting that parties only aggregate their secret shares at the end of the protocol, hence our PSI-Sum protocol does not have to reveal the cardinality of the intersection, which may be useful in certain applications.

Interestingly, achieving plain UPSI with addition and deletion is more challenging than Cardinality and Sum. As briefly discussed in Section 2, the issue comes from the scenario when an element $x$ is added by one party while being deleted by the other party on the same day. In our PSI-Cardinality/Sum protocols, while adding and deleting $x$ from the intersection both occur on the same day, their affect on the output cancels out when their secret shares are combined. However, in plain UPSI, parties need to learn the exact elements to be added or deleted, revealing more information than the ideal functionality. To address this issue, we carefully arranged...
PartDec \text{ updates } y_j to get path_j = \{(Enc_{pk}(x_kj), Enc_{pk}(p_j))\}_{j=1}^{l_0^{\sigma \cdot \log p}}. Then P_0 sends m_2 = \{(\text{path}_i)_{i=1}^{\sigma L_1^{\rho}} to P_1.

(5) Combining candidates. P_1 combines (\text{path}_i)_{i=1}^{\sigma L_1^{\rho}} with (\text{path}_i)_{i=1}^{\sigma L_1^{\rho}} received from P_0, randomly samples a mask \alpha_k \in \mathbb{Z}_q for each element in the combined set, and samples a random permutation \pi over [\Gamma] where \Gamma = \sigma \cdot (n \cdot L_2 + m \cdot L_1) + \rho \cdot (n + m). Compute and send the following to P_0:

m_3 = \pi \{(\text{PartDec}_{sk_i}(\alpha_k \odot Enc_{sk_i}(ak - b_k)), Enc_{pk}(p_k))\}_{k=1}^{\pi(\sigma L_1^{\rho})}.

(6) Output generation. P_0 fully decrypts the first element in each tuple of m_3 to get \alpha_k(a_k - b_k). Let K = \{k | \alpha_k(a_k - b_k) = 0\}.

- In \Pi_{\text{PSI-Addsum}}, P_0 outputs C_d = C_{d-1} + |K|.
- In \Pi_{\text{PSI-Addsum}}, P_0 outputs m_4 = \text{PartDec}_{sk_i}(\alpha_k \odot Enc_{sk_i}(a_k - b_k)) and sends it to P_1. P_1 responds to P_0 with m_4 = \text{PartDec}_{sk_i}(m_4). P_0 fully decrypts it to get V = \text{FullDec}_{sk}(m_4^\pi) and outputs V_d = V_{d-1} + V.

- In \Pi_{\text{PSI-Addsum}}, P_0 samples a random share [\{z_k\}]_0 \in \mathbb{Z}_q for all k \in K, outputs C_d = C_{d-1} + |K| and an updated share set with new random shares [\{\text{z}_k\}]_{k \in K}. Additionally, P_0 computes and sends the following to P_1:

m_4 = \{\text{PartDec}_{sk_i}(\text{Enc}_{pk}(p_k) \odot \text{Enc}_{sk_i}([-\text{z}_k])_{i=1}^{\sigma L_1^{\rho}})\}_{k \in K}.

P_1 fully decrypts m_4 using sk_i to get its shares [\{\text{z}_k\}]_{k \in K}, and outputs C_d = C_{d-1} + |K| and an updated share set with new random shares [\{\text{z}_k\}]_{k \in K}.

(7) Y_d tree update. P_1 computes m_5 = \{(\text{updates}_i, t_j)\}_{i=1}^{\sigma L_1^{\rho}}, \text{ sends to } P_0, when it then replaces each path \mathcal{P}(t_j) with updates_i in \mathcal{D}_i and replaces \mathcal{S}_i with \mathcal{S}_i'. Both parties update L_1 if needed.

Figure 4: Protocols \Pi_{\text{PSI-Addsum}}, \Pi_{\text{PSI-Addcircuit}}, \Pi_{\text{PSI-Addcircuit}} for one-sided addition-only PSI functionalities \mathcal{F}_{\text{PSI-Addsum}}^\text{Add}, \mathcal{F}_{\text{PSI-Addcircuit}}^\text{Add}, \mathcal{F}_{\text{PSI-Addcircuit}}^\text{Add}, respectively.

4.4 Updatable PSI-Cardinality and PSI-Sum with Addition and Deletion

Our protocols realizing functionalities \mathcal{F}_{\text{PSI-Delete}} and \mathcal{F}_{\text{PSI-Delete}} are described Figure 8.

Computation and Communication Complexity. On each day \delta, let N, M be the total number of additions and deletions of the two parties, respectively. Let the update set sizes be n and m, respectively. Then both the computation and communication complexity are \mathcal{O}(n \cdot (\sigma \cdot \log M + \rho) + m \cdot (\sigma \cdot \log N + \rho)).

Correctness and Security. We state the theorem below and defer its proof to Appendix F.

Theorem 4.2. Assuming \Pi is a secure additively homomorphic encryption scheme, F is a pseudorandom function, the protocols \Pi_{\text{PSI-Delete}} \mathcal{F}_{\text{PSI-Delete}} presented in Figure 8 securely realize the ideal functionalities \mathcal{F}_{\text{PSI-Delete}}^\text{Add}, \mathcal{F}_{\text{PSI-Delete}}^\text{Add}, defined in Figure 5, respectively, against semi-honest adversaries.

4.3 Subprotocols

In this section, we present our subprotocol \Pi_{\text{CombinePath}} in Figure 7 and defer its correctness and security proofs to Appendix D. In the subprotocol \Pi_{\text{CombinePath}}, we need a special secure two-party computation protocol with functionality \mathcal{F}_{\text{lookup}}. We define the ideal functionality in Figure 6 and discuss an efficient realization in Section 5.
We present in Figure 9 our protocol for the plain UPSI functionality.

Then both the computation and communication complexity are concrete efficiency.

In our UPSI protocols and optimizations to further improve the concrete efficiency.

We state the theorem below and defer the proof to Appendix E.

Theorem 4.3. Assuming \( \Pi \) is a secure additively homomorphic encryption scheme, \( F \) is a pseudorandom function, the protocol \( \Pi_{\text{UPSI-Add}} \) presented in Figure 9 securely realizes the ideal functionalities \( F_{\text{UPSI-Add}} \) defined in Figure 5 against semi-honest adversaries.

5 IMPLEMENTATION DETAILS AND OPTIMIZATIONS

In this section, we discuss instantiations of the building blocks in our UPSI protocols and optimizations to further improve the concrete efficiency.

Encryption Schemes. In the addition-only UPSI protocols \( \Pi_{\text{UPSI-Add}_{\alpha}} \) and \( \Pi_{\text{UPSI-Add}_{\text{init}}} \), we instantiate the (2, 2)-threshold additively homomorphic encryption scheme with exponential El Gamal encryption [25] to take advantage of efficient elliptic curve operations. Recall that in this scheme, \( \text{Enc}(m) = (g^f, h^r \cdot g^m) \) where the public key consists of a group generator \( g \) and a random group element \( h = g^t \) with a secret key \( s \). In the (2, 2)-threshold scheme, \( sk_0 \) and \( sk_1 \) form an additive secret share of \( s \). Decryption of exponential El Gamal requires computing the discrete logarithm of a group element \( g^m \), which is possible for a bounded message space. In all our addition-only UPSI protocols presented in Section 4, decryption occurs in Step 6. Observe that \( P_0 \) does not have to fully decrypt the first element in each tuple of \( m_3 \); instead, it is sufficient to check whether the decrypted message is 0 or not. In particular, given a partially decrypted ciphertext \( \hat{c} = (a, b) \), \( P_0 \) can determine if the encrypted message is 0 by checking if \( b = a^k \), without performing discrete logarithm. In \( \Pi_{\text{UPSI-Add}_{\text{init}}} \), \( P_0 \) needs to fully decrypt \( m'_3 \), where the underlying message can be bounded by the maximum sum of associated values.

In \( \Pi_{\text{UPSI-Add}_{\text{init}}} \), while exponential El Gamal can still be used for the first ciphertext in \( m_3 \), the (masked) payload messages are distributed uniformly over the entire plaintext space, hence the payload messages are encrypted using (2, 2)-threshold Paillier encryption [38] instead.

In our protocols with both addition and deletion presented in Section 4, \( \Pi_{\text{UPSI-Add}_{\alpha}}, \Pi_{\text{UPSI-Add}_{\text{init}}} \), and \( \Pi_{\text{UPSI-Add}_{\text{sum}}} \), El Gamal cannot be utilized because all the ciphertexts are encrypting secret shares that are distributed across the message space. Instead, the additively homomorphic encryption scheme is instantiated with Paillier. This has an impact on the computation time, as can be seen in Section 6.

Paillier Modulus Switching. Using Paillier in the deletion protocols introduces an additional technical challenge. Recall that the plaintext space in Paillier encryption is \( Z_n \) for a public key \( n \), which is different for \( P_0 \)'s and \( P_1 \)'s keys. During our deletion protocols,
Initialization:
1. $P_0$ and $P_1$ independently generate key pairs for an additive homomorphic encryption scheme $(pk_0, sk_0)$ $\rightarrow$ KeyGen($1^\lambda$) and $(pk_1, sk_1) \rightarrow$ KeyGen($1^\lambda$) and share the public keys. Both parties agree on a randomly sampled PRF key $k \leftarrow (0, 1)^\lambda$.
2. $P_0$ and $P_1$ generate initial trees with only an empty root and stash: $(D_0, S_0, \vec{D}_1, S_1)$ and $(D_0, S_0, \vec{D}_1, S_1)$, respectively. Initialize $Out_0 = 0$.

Day $d$: $P_0$ and $P_1$ hold $(D_0, S_0, \vec{D}_1, S_1)$ and $(D_0, S_0, \vec{D}_1, S_1)$, respectively. Let $L_0$ and $L_1$ be the heights of $D_0$ and $\vec{D}_1$, and $S_0$ and $\vec{D}_1$, respectively. Both parties update $L_0$ and $L_1$ as they update the trees below. Let $X, Y$ denote the two parties’ sets at the end of the previous day.

$P_0$ and $P_1$ have new input sets $X_d^0, Y_d^0$ which include elements they are adding to their set and $X_d^1, Y_d^1$ of elements they are deleting. Denote $n = |X_d^0 \cup X_d^1|$, \(m = |Y_d^0 \cup Y_d^1|\). In $\Pi_{UPSI-Del3a}$, $P_0$ holds a value $q_i \in \mathbb{Z}_q$ associated with each element $x_i \in X_d^0 \cup X_d^1$.

1. $P_0$ defines a payload for each element $x_i \in X_d^0 \cup X_d^1$ depending on the functionality:

   \[ P_0 = \begin{cases} \phantom{+} (-1)^{(x_i \in X_d^0)} \cdot q_i & \text{for } \Pi_{UPSI-Del3a} \\ \phantom{+} (-1)^{(x_i \in X_d^1)} \cdot q_i & \text{for } \Pi_{UPSI-Del3a} \end{cases} \]

   $P_1$ defines a payload for each element $y_i \in Y_d^0 \cup Y_d^1$: $g_j = (-1)^{(y_i \in X_d^0)}$.

2. $X_d^0 \cup X_d^1$ tree update. $P_0$ sends $\{(\text{updates}_{x_i}, q_i)\}_{i=1}^n$, $S_0'$ $\rightarrow$ UpdateTree($X_d^0 \cup X_d^1$, $(p_i)_{i=1}^n$, $D_0$, $S_0$, $F_i(\cdot)$, Enc($pk_0(\cdot)$)) to $P_1$. $P_1$ replaces each path $\mathcal{P}(t_j)$ with updates $x_i$ in $D_0$ and replaces $S_i$ with $S_i'$.

3. Secret shares for new elements of $X$. For all $x_i \in (X_d^0 \cup X_d^1)$, run $\Pi_{CombinePath}$ with $P_0$ as Initiator inputting $(x_i, P_0, \text{path}_i \leftarrow \text{GetPath}(D_0, S_0, F_i(\cdot), x_i), pk_0)$ and $P_1$ as Responder providing secret shares $sk_i$ corresponding to $p_k$. They receive secret shares $[s_{x_i}]$, respectively.

4. Secret shares for new elements of $Y$. For all $y_j \in (Y_d^0 \cup Y_d^1)$, run $\Pi_{CombinePath}$ with $P_0$ as Responder inputting $sk_j$ corresponding to $p_k$ and $P_1$ as Initiator inputting $(y_j, P_0, \text{path}_j \leftarrow \text{GetPath}(D_0, S_0, F_i(\cdot), y_j), pk_0)$. They receive secret shares $[s_{y_j}]$, respectively.

5. $Y_d^0 \cup Y_d^1$ tree update. $P_1$ sends $\{(\text{updates}_{y_j}, q_j)\}_{j=1}^m$, $S_0'$ $\rightarrow$ UpdateTree($Y_d^0 \cup Y_d^1$, $(q_j)_{j=1}^m$, $D_0$, $S_0$, $F_i(\cdot)$, Enc($pk_1(\cdot)$)) to $P_0$. $P_0$ replaces each path $\mathcal{P}(t_j)$ with updates $y_j$, in $D_0$, and replaces $S_i$ with $S_i'$.

6. Combine all shares. For $b \in \{0, 1\}$, $P_0$ computes $[s_{x_i}] = \sum_{x_i \in B} [s_{x_i}]$, $[s_{y_j}] = \sum_{y_j \in B} [s_{y_j}]$.

7. Output generation: $P_0$ sends $[s_{x_i}]$, $[s_{y_j}]$ to $P_0$, who then computes $Out_d = Out_{d-1} + [s_{x_i}] + [s_{y_j}]$.

$P_0$ outputs $Out_d$ for both $\Pi_{UPSI-Del3a}$ and $\Pi_{UPSI-Del3a}$, respectively.

Figure 8: Protocols $\Pi_{UPSI-Del3a}$ and $\Pi_{UPSI-Del3a}$ for one-side UPSI with both addition and deletion functionalities $\mathcal{F}_{UPSI-Del3a}$ and $\Pi_{UPSI-Del3a}$, respectively.

**Realizing $\mathcal{F}_{lookup}$:** While $\mathcal{F}_{lookup}$ can be instantiated with a generic secure two-party computation (2PC) protocol [28, 53], we construct a protocol that achieves better concrete efficiency, leveraging oblivious transfer (OT) and the efficient OT extension [11, 32]. Let $(a, m_0, m_1)$ and $b$ be the inputs to $\mathcal{F}_{lookup}$ where $m_0$ is output when $a = b$ and $m_1$ otherwise. Before comparison, both parties compute a hash function $H : \mathbb{Z}_q \rightarrow \{0, 1\}^b$ on their inputs $a$ and $b$. The parties then run a garbled-circuit based equality testing to compute a binary secret share $[c] \in \{0, 1\}$ of $H(a) \neq H(b)$. Then two two parties run an OT protocol where Sender inputs two messages $(m_1 - [c], m_1 [c])$ and Receiver inputs a choice bit $[c]$. If $a = b$, then $[c] \neq [c]$, in which case Receiver will receive $m_0$, as desired in $\mathcal{F}_{lookup}$. If $a \neq b$, then $[c] = [c]$ with overwhelming probability (see analysis below), and the Responder will receive $m_1$.

In this approach, we need the guarantee that if $a \neq b$, then $H(a) \neq H(b)$ with overwhelming probability, hence $\epsilon_{gc}$ should be sufficiently large. On the other hand, the size of $\text{th}$ equality testing circuit grows with $\epsilon_{gc}$, so we want to choose the smallest $\epsilon_{gc}$ such that the probability of a failure (i.e., that $H(a) = H(b)$ for $a \neq b$) over the entire protocol is less than $2^{-\kappa}$. Since there are at most $2^{33}$ elements held by both parties in our benchmarks, and each element is compared against no more than $2^9$ elements in $\Pi_{CombinePath}$, the total number of $\mathcal{F}_{lookup}$ invocations is bounded by $2^{32}$. Therefore, we set $\epsilon_{gc} = 72$ to ensure that the total failure probability is less than $2^{-\kappa}$ for $\kappa = 40$. 


We implement all of our UPSI protocols in C++ and report their \( L \) and Gamal and Paillier encryptions, Google’s \([52]\) for instantiations of garbled circuits and oblivious tree update. 

1. Deletion:
   - \( X_d^+ \) tree update. \( P_0 \) sends \( (((update_{x_1}, t_1))_{i=1}^{n}, S_0') \) to \( P_1 \). \( P_1 \) replaces each path \( P(t_1) \) with updates in \( D_0 \), and \( S_0 \) with \( S_0' \).
   - Secret shares for \( X_d^+ \cap Y \). For all \( x \in X_d^+ \), run \( \Pi_{CombinePath} \) with \( P_0 \) as Initiator inputting \((x, -1, path_{x}) \leftarrow GetPath(D_0, S_0, F_k(., x)) \) and \( P_1 \) as Responder inputting \( s_k \) corresponding to \( pk_k \). They receive secret shares \([x_{z, k}]_0 \) and \([x_{z, k}]_{i=1}^{n} \), respectively, where \( x_{z, k} = x \) if \( x \in D_0 \cap S_0 \) and 0 otherwise.
   - Secret shares for \( (X \setminus X_d^+) \cap Y \). For all \( y \in Y_d^+ \), run \( \Pi_{CombinePath} \) with \( P_0 \) as Responder inputting \( s_k \) corresponding to \( pk_k \) and \( P_1 \) as Initiator inputting \((y, -1, path_{y}) \leftarrow GetPath(D_0, S_0, F_k(., y)) \). They receive secret shares \([y_{z, k}]_0 \) and \([y_{z, k}]_{i=1}^{n} \), respectively, where \( y_{z, k} = y \) if \( y \in D_0 \cap S_0 \) and 0 otherwise.
   - \( Y_d^+ \) tree update. \( P_1 \) sends \( (((update_{y_1}, t_1))_{i=1}^{n}, S_1') \) to \( P_0 \). \( P_0 \) replaces each path \( P(t_1) \) with updates in \( D_0 \), and \( S_1 \) with \( S_1' \).

2. Addition:
   - \( X_d^+ \) tree update. \( P_0 \) sends \( (((update_{x_1}, t_1))_{i=1}^{n}, S_0') \) to \( P_1 \). \( P_1 \) replaces each path \( P(t_1) \) with updates in \( D_0 \), and \( S_0 \) with \( S_0' \).
   - Secret shares for \( X_d^+ \cap (Y \setminus Y_d^+) \). For all \( x \in X_d^+ \), run \( \Pi_{CombinePath} \) with \( P_0 \) as Initiator inputting \((x, 1, path_{x}) \leftarrow GetPath(D_0, S_0, F_k(., x)) \) and \( P_1 \) as Responder inputting \( s_k \) corresponding to \( pk_k \). They receive secret shares \([x_{z, k}]_0 \) and \([x_{z, k}]_{i=1}^{n} \), respectively, where \( x_{z, k} = x \) if \( x \in D_0 \cap S_0 \) and 0 otherwise.
   - Secret shares for \( (X \setminus X_d^+) \cap (Y \setminus Y_d^+) \). For all \( y \in Y_d^+ \), run \( \Pi_{CombinePath} \) with \( P_0 \) as Responder inputting \( s_k \) corresponding to \( pk_k \) and \( P_1 \) as Initiator inputting \((y, 1, path_{y}) \leftarrow GetPath(D_0, S_0, F_k(., y)) \). They receive secret shares \([y_{z, k}]_0 \) and \([y_{z, k}]_{i=1}^{n} \), respectively, where \( y_{z, k} = y \) if \( y \in D_0 \cap S_0 \) and 0 otherwise.
   - \( Y_d^+ \) tree update. \( P_1 \) sends \( (((update_{y_1}, t_1))_{i=1}^{n}, S_1') \) to \( P_0 \). \( P_0 \) replaces each path \( P(t_1) \) with updates in \( D_0 \), and \( S_1 \) with \( S_1' \).

3. Output Generation:
   - Let \([z_{i}]_{i=1}^{\Gamma} \) be the secrets received by \( P_0 \) and \( P_1 \) above, where \( \Gamma = n^- + m^- + n^+ + m^+ \). \( P_0 \) sends \( Enc_{pk_{z}}([z_{i}]_{i=1}^{\Gamma}) \) to \( P_1 \).
   - \( P_1 \) samples a random permutation \( \pi \) over \([\Gamma]\). \( P_1 \) samples a random mask \( a_{i} \sim Z_q \) for each \( i \in [\Gamma] \) and homomorphically adds them to the encryptions received from \( P_0 \). \( P_1 \) sends the following to \( P_0 \): \( \pi \left((Enc_{pk_{z}}([z_{i}]_{i=1}^{\Gamma}) \oplus Enc_{pk_{z}}(a_{i})), ([z_{i}]_{i=1}^{\Gamma}) - a_{i}\right)_{i=1}^{\Gamma} \).
   - \( P_0 \) decrypts the first element in each pair using \( s_k \), and adds up each pair of shares to learn the shuffled set \([z_{i}]_{i=1}^{\Gamma} \).

Output \( \Pi_{Del_{ps}} = \Pi_{Del_{ps}} \setminus \{z_{i} | z_{i} < 0\} \).
N. The Circuit-PSI protocols can also be used to compute PSI-Cardinality or PSI-Sum. We don’t compare with the protocols specifically designed for PSI-Cardinality or PSI-Sum [26, 31] because these protocols are outperformed by [16, 46]. A more recent work [15] improves PSI and Circuit-PSI communication by 12% compared to [46], but we don’t compare with it for three reasons: (1) their construction is built on the Silver codes [22], which turn out to be insecure [47], (2) their source code is not available online, and (3) even if their construction is instantiated with secure codes, from our comparison with the other works, we expect our protocols to perform better in certain settings as well. Note that [46] is also instantiated with the insecure Silver codes, but their open-sourced library [45] supports instantiating the construction with secure codes, which is what we compare with. In all of our comparison tables, cells in green indicate the state-of-the-art performance, and those in blue indicate that our protocols perform better.

In the setting with both addition and deletion, standalone PSI protocols need only compute over elements that remain in the input sets. In the extreme case where the same set of elements are added and deleted every day, the input sets remain small and so the standalone PSI protocols would likely be optimal. Alternatively, if the input sets are growing at a steady rate, then our constructions may be best. Benchmarks for UPSI with addition and deletion are provided, but these caveats should be understood and application-specific context would play a role in choosing a solution.

Concrete Parameters. We set the computational security parameter $\lambda = 128$ and the statistical security parameter $\kappa = 40$. Following the analysis in [50], we set the maximum tree node capacity $\sigma = 4$ and the maximum stash capacity $\rho = 89$ to achieve failure probability of $2^{-80}$ for inserting a single element into the tree. Even with our largest set size of $2^{22}$, the combined failure probability is bounded well below $2^{-8}$. In protocols with addition and deletion, we allow parties to add and delete each element at most once (i.e., an element cannot be readded once it has been deleted), and so we double both our node size (to $\sigma = 8$) and stash size (to $\rho = 178$) following Lemma E.1.

### 6.2 Addition-Only UPSI

We compare our addition-only UPSI performance for extended functions (PSI-Cardinality, PSI-Sum, and Circuit-PSI) with [46] (RR22) and [16] (CGS22) in Table 2 with total set sizes ranging from $2^{18}$ to $2^{22}$ and update sizes from $2^{6}$ to $2^{10}$. Our computation and communication complexity grows logarithmically with the total size and linearly with the update size $N_d$, so our protocols are more competitive in larger input sizes and smaller update sizes. Note that [16] (CGS22) presents two constructions (C-PSI1 and C-PSI2) with different trade offs between computation and communication, but for all the parameters we choose, C-PSI2 outperforms C-PSI1 in all aspects. We were unable to run CGS22 with input size of $2^{22}$, so we use the communication cost and running time under LAN reported in their paper [16], and estimate the running time in the WAN settings.

**Communication Comparison.** Since our communication grows linearly with the update size $N_d$ and only logarithmically with the total set size $N$, our protocols have a communication advantage in settings where $N_d \ll N$. For $N = 2^{18}$, our communication has an improvement of $2.2 - 13\times$ when $N_d = 2^6$ in all functionalities, and when $N_d = 2^8$, $\Pi_{\text{UPSI-Add}_{\text{sum}}}$ and $\Pi_{\text{UPSI-Add}_{\text{sum}}}$ have an advantage $1.8 - 3.4\times$. For $N = 2^{20}$, our protocols outperform RR22 by $2.2 - 50\times$ depending on the functionality and update size, with only $\Pi_{\text{UPSI-Add}_{\text{sum}}}$ at $N_d = 2^{10}$ not showing an improvement. When $N = 2^{22}$, that improvement extends to all settings and increases to a factor of $2.2 - 200\times$.

**Computation Comparison.** Our computational complexity is similar to communication, growing linearly with the update size and logarithmically with the input size. Despite this, our computation times do not reflect this asymptotic improvement as clearly, which stems from our usage of costly public key operations. As a result, we show better performance only when $N$ is sufficiently large. In the LAN setting with $N = 2^{20}$, $N_d = 2^8$, our $\Pi_{\text{UPSI-Add}_{\text{sum}}}$ and $\Pi_{\text{UPSI-Add}_{\text{sum}}}$ are faster by $3.2\times$ and $2.1\times$, respectively. By $N = 2^{22}$, $N_d = 2^6 - 2^9$, our $\Pi_{\text{UPSI-Add}_{\text{sum}}}$ protocols outperform CGS22 by $1.4 - 15\times$.

**End to End Comparison.** Given these communication and computation trade offs, our protocols perform best with more realistic network configurations with lower network bandwidth. At
$N = 2^{15}$, we begin to have competitive runtimes for $\Pi_{\text{UPSI-Add}_{\text{psi}}}$ and $\Pi_{\text{UPSI-Add}_{\text{sum}}}$ in the smaller update size $N_d = 2^6$. By $N = 2^{20}$ our protocols show improvements of up to 4.7x at 200 Mbps, 6.1x at 50 Mbps, and 20x at 5 Mbps. At $N = 2^{22}$ and $N_d = 2^8$, our protocols outperform in all network settings by 15 – 76x for $\Pi_{\text{UPSI-Add}_{\text{psi}}}$, 11 – 46x for $\Pi_{\text{UPSI-Add}_{\text{sum}}}$, and 1.8 – 9.4x for $\Pi_{\text{UPSI-Add}_{\text{circulant}}}$.

### 6.3 UPSI with Addition and Deletion

Our performance for $\Pi_{\text{UPSI-Del}_{\text{psi}}}$ and $\Pi_{\text{UPSI-Del}_{\text{sum}}}$ in comparison with [16, 46] is presented in Table 3. Since the two protocols are implemented in the same way except that $P_0$’s inputting payloads are different, they have close experimental results. We combine them in the table. This protocol is more expensive than the addition-only ones, so we set smaller update sizes of $N_d = 2^4, 2^5, 2^6$ to demonstrate the turning point where our protocols start to perform better. Our experiments for input size $N = 2^{22}$ are run on a Google Cloud c2-standard-38 virtual machine with 120 GB of RAM as we run out of 64 GB memory.

### Communication Comparison

Similarly as in our addition-only protocols, our communication is $O(N_d \cdot \log N)$. Our communication improvements are not as stark though, for two reasons: (1) the increased stash and node sizes required, and (2) in addition to exchanging ciphertexts, the parties also perform OT and garbled circuits. Despite this, our protocol still achieves lower communication overhead in most settings. At $N = 2^{20}$, our communication has an improvement of 1.3 – 2.5x when $N_d \leq 2^5$. By $N = 2^{22}$, our communication has an improvement of 2.5 – 9.9x for all update sizes.

### Computation Comparison

Our performance under LAN is again dominated by public key operations, but, unlike in the addition-only protocols, does not benefit from the efficient El Gamal instantiations. Our computation has the same growth rate as communication, and so we expect our performance to eventually beat CGS22 when $N$ is sufficiently large.

### End to End Comparison

As shown in Table 3, the end to end running time of our protocol begins to outperform RR22 and CGS22 at 5 Mbps when $N = 2^{20}, N_d = 2^4$ by 1.4x. By $N = 2^{22}$, we show an improvement of 1.3 – 5.1x at 5 Mbps for all update sizes, and an improvement of 1.5x at 50 Mbps for $N_d = 2^4$.

### 6.4 UPSI for Plain PSI

We compare our plain UPSI protocols $\Pi_{\text{UPSI-Add}_{\text{psi}}}$ and $\Pi_{\text{UPSI-Del}_{\text{psi}}}$ with [46] (RR22) in Table 4. We evaluate two constructions in RR22 with different encoding sizes of $1.28n$ and $1.23n$, which have different trade offs in computation and communication. In Table 4, we use fast and small to denote them respectively. Note that $\Pi_{\text{UPSI-Add}_{\text{psi}}}$ (Figure 4) contains only one encrypted tree, unlike our other addition-only protocols that contain two encrypted trees — one on each party’s side — hence it is more efficient. To best demonstrate the turning point of our protocols, we use $N_d = 2^4, 2^5, 2^6$ for $\Pi_{\text{UPSI-Add}_{\text{psi}}}$ and $N_d = 2^4, 2^5, 2^6$ for $\Pi_{\text{UPSI-Del}_{\text{psi}}}$.

### Table 3: Communication cost (in MB) and running time (in seconds)

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N_d$</th>
<th>Protocol</th>
<th>Comm. (MB)</th>
<th>Total Running Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{20}$</td>
<td></td>
<td>RR22</td>
<td>149</td>
<td>31.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CGS22 (C-PSI)</td>
<td>2190</td>
<td>31.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{psi}}}$</td>
<td>1408</td>
<td>24.3</td>
</tr>
<tr>
<td>$2^4$</td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{sum}}}$</td>
<td>58.5</td>
<td>96.1</td>
</tr>
<tr>
<td>$2^5$</td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{psi}}}$</td>
<td>116</td>
<td>190</td>
</tr>
<tr>
<td>$2^6$</td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{sum}}}$</td>
<td>231</td>
<td>364</td>
</tr>
<tr>
<td>$2^{22}$</td>
<td></td>
<td>RR22</td>
<td>606</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CGS22 (C-PSI)</td>
<td>666.7</td>
<td>93.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{psi}}}$</td>
<td>443.5</td>
<td>77.9</td>
</tr>
<tr>
<td>$2^4$</td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{sum}}}$</td>
<td>61.4</td>
<td>103</td>
</tr>
<tr>
<td>$2^5$</td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{psi}}}$</td>
<td>122</td>
<td>203</td>
</tr>
<tr>
<td>$2^6$</td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{sum}}}$</td>
<td>243</td>
<td>385</td>
</tr>
</tbody>
</table>

### Table 4: Communication cost (in MB) and running time (in seconds)

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N_d$</th>
<th>Protocol</th>
<th>Comm. (MB)</th>
<th>Total Running Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{20}$</td>
<td></td>
<td>RR22 (fast)</td>
<td>34.3</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RR22 (small)</td>
<td>32.1</td>
<td>1.00</td>
</tr>
<tr>
<td>$2^4$</td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{psi}}}$</td>
<td>1.95</td>
<td>3.54</td>
</tr>
<tr>
<td>$2^5$</td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{psi}}}$</td>
<td>7.57</td>
<td>21.6</td>
</tr>
<tr>
<td>$2^6$</td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{psi}}}$</td>
<td>29.6</td>
<td>84.9</td>
</tr>
<tr>
<td>$2^{22}$</td>
<td></td>
<td>RR22 (fast)</td>
<td>58.7</td>
<td>98.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RR22 (small)</td>
<td>117</td>
<td>195</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Pi_{\text{UPSI-Del}_{\text{psi}}}$</td>
<td>231</td>
<td>370</td>
</tr>
</tbody>
</table>

### REFERENCES

Extended Private Set Intersection Revisited: Enhanced Functionalities, Deletion, and Worst-Case Complexity


A CRYPTOGRAPHIC SCHEMES

Additively Homomorphic Encryption. An additively homomorphic encryption scheme is a public-key encryption scheme that consists of a tuple of PPT algorithms (KeyGen, Enc, Dec) over message space $M$ with correctness, chosen-plaintext attack (CPA) security, and linear homomorphism.

- $(pk, sk) \leftarrow \text{KeyGen}(1^n)$: On input of the security parameter, output a public key pk and a secret key sk.

- $c \leftarrow \text{Enc}_{pk}(m)$: On input of a public key pk and a message $m \in M$, output a ciphertext c.

- $m_1 \oplus m_2 \leftarrow \text{Dec}_{sk}(c)$: On input of a secret key sk and a ciphertext c, output a plaintext message $m_1$, $m_2 \in M$.

- $\text{Enc}_{pk}(m_0 \cdot m_1) \leftarrow \text{Enc}_{pk}(m_0) \oplus \text{Enc}_{pk}(m_1)$: On input two ciphertexts of $m_0$, $m_1$ encrypted under pk, output a ciphertext for their sum.

- $\text{Enc}_{pk}(m_0 \cdot m_1) \leftarrow m_0 \oplus \text{Enc}_{pk}(m_1)$: On input a plaintext message $m_0$ and a ciphertext of $m_1$ encrypted under pk, output a ciphertext for their product.
Threshold Additively Homomorphic Encryption. A (2, 2)-threshold additively homomorphic encryption scheme consists of a tuple of PPT algorithms (KeyGen, Enc, PartDec, FullDec) over message space $\mathcal{M}$.

- $(pk, sk_0, sk_1) \leftarrow \text{KeyGen}(1^3)$: On input of the security parameter, output a public key $pk$ and a pair of secret key shares $sk_0$ and $sk_1$.
- $c \leftarrow \text{Enc}_{pk}(m)$: On input of a public key $pk$ and a message $m \in \mathcal{M}$, output a ciphertext $c$.
- $\hat{c} \leftarrow \text{PartDec}_{sk_b}(c)$: On input a secret key share $sk_b$ (for $b \in \{0, 1\}$) and a ciphertext $c$, output a partially decrypted ciphertext $\hat{c}$.
- $m/\perp \leftarrow \text{FullDec}_{sk_b}(\hat{c})$: On input a secret key share $sk_b$ (for $b \in \{0, 1\}$) and a partially decrypted ciphertext $\hat{c}$ by the other secret key $sk_{1-b}$, output a plaintext $m$ or the symbol $\perp$.

The scheme satisfies correctness and CPA security even given a secret key share $sk_b$ for $b \in \{0, 1\}$. It also supports linear homomorphic operations $\oplus$ and $\odot$.

Re-randomization. We implicitly assume that each homomorphic operation is followed by a re-randomization process, where the resulting ciphertext is added with an independently generated encryption of zero. This is required in our protocols to ensure that the randomness of the final ciphertext is independent of the randomness used in the original ciphertexts. For the popular (threshold) additively homomorphic encryption schemes such as exponential El Gamal encryption [25] and Paillier encryption [38], a homomorphically evaluated ciphertext can be made statistically identical to a fresh ciphertext. We refer to [25, 38] for formal definitions of correctness and CPA security.

**B ADDITION-ONLY PLAIN UPSI**

One-sided addition-only plain UPSI with worst-case complexity can be achieved with slight modification of the protocol in [13] and we present a protocol $\Pi_{\text{UPI-AddPSI}}$ in Figure 10. See Section 2 for an overview of the previous construction. By replacing the tree updates and querying with UpdateTree and GetPath, the worst case communication and computation can be reduced to $O(N_d \log N)$ from $O(N)$. Benchmarks for this protocol can be found in Table 4.

**C PROOF OF THEOREM 3.2**

Correctness. We first show correctness of our protocols by induction. On day 0, all sets are initialized as empty sets so correctness trivially holds. Next, on any day $d$, our goal is to compute $I_d = (X \cup X_d) \cap (Y \cup Y_d)$. First, from the correctness of the subroutines GetPath and UpdateTree as well as the threshold additive homomorphic encryption scheme, one can observe that in the output generation, in Step 6, the output computed is indeed using the candidates from Step 2 and Step 3. In particular, we can expand $I_d$ as

$$I_d = (X \cup X_d) \cap (Y \cup Y_d)$$

$$= ((X \cup X_d) \cap Y) \cup ((X \cup X_d) \cap Y_d)$$

$$= (X \cap Y) \cup (X_d \cap Y) \cup ((X \cup X_d) \cap Y_d)$$

**Initialization:**

1. $P_0$ and $P_1$ independently generate key pairs for an additive homomorphic encryption scheme $(pk_b, sk_b) \leftarrow \text{KeyGen}(1^3)$ and $(pk, sk_1) \leftarrow \text{KeyGen}(1^3)$ and share the public keys. $P_0$ and $P_1$ sample $k_0, k_1 \leftarrow \mathcal{Z}_q$, respectively. Both parties agree on a randomly sampled PRF key $k \leftarrow \{0, 1\}^3$.

2. $P_0$ and $P_1$ generate initial trees with only an empty root and stash: $(\mathcal{D}, \mathcal{S})$, and $(\mathcal{D}, \mathcal{S})$, respectively. Initialize $I_0 = \emptyset$.

**Day $d$:** $P_0$ and $P_1$ hold $(\mathcal{D}, \mathcal{S})$ and $(\mathcal{D}, \mathcal{S})$, respectively. Let $L$ be the tree height of $\mathcal{D}$ and $\mathcal{D}$. Both parties update $L$ as they update the trees below. $P_0$ also holds $H_X$, let $Y$, $Y$ denote the two parties’ sets at the end of the previous day, respectively. $P_0$ holds a new input set $X_d$ and $P_1$ holds a new input set $Y_d$. Let $n = |X_d|$ and $m = |Y_d|$.

1. $P_0$ learns $X \cap Y_d$.
   a. $P_0$ computes $H(Y_d)$ and sends to $P_0$.
   b. $P_0$ raises each element by $k_0$ to obtain $H(Y_d)$ and compares against $H_X$ (which equals $H(X \setminus I_{d-1})$) to learn $I_Y = X \cap Y_d$.

2. $Y$ tree update. $P_1$ sends $\{(\text{UpdateTree}(Y, \perp, \mathcal{D}, \mathcal{S}, F_k(\cdot), \text{Enc}_{pk}(\cdot))\}$ to $P_0$. $P_0$ replaces each path $P(j)$ with updates in $\mathcal{D}$, and replaces $\mathcal{S}$ with $\mathcal{S}'$.

3. $P_0$ learns $X_d \cap (Y \cup Y_d)$. $P_0$ samples a random mask $\alpha \leftarrow \mathcal{Z}_q^*$ and sends $\text{Enc}_{sk_b}(\alpha)$ to $P_1$, then for every element $x_i \in X_d$:
   a. $P_0$ obtains
   $$(\text{Enc}_{sk_b}(y_{i,j}), \alpha_{\perp}^{L_0}) \leftarrow \text{GetPath}(\mathcal{D}, \mathcal{S}, F_k(\cdot), x_i),$$
   b. $P_0$ raises each element by $k_1$ and homomorphically computes path $= (\beta_i \oplus \text{Enc}_{sk_b}(y_{i,j}) \oplus \text{Enc}_{sk_b}(\alpha))^{\perp}_{j=1}$ and sends path to $P_1$.

4. $Y$ Output Generation. $P_0$ outputs $I_y = I_Y \cup Y_d$.

5. $H_X$ updates.
   a. $P_0$ creates a set $X'_d = (X_d \setminus I_X)$ and then pads it with dummy elements until $|X'_d| = |X_d|$. Then samples $\epsilon \leftarrow \mathcal{Z}_q$ and sends $H(X'_d)^{\epsilon_{sk_b}}$ to $P_1$.
   b. $P_1$ raises each element in $H(X'_d)^{\epsilon_{sk_b}}$ to the power $k_1$ to obtain $H(X'_d)^{\epsilon_{sk_b}k_1}$ which she sends back to $P_0$.
   c. $P_0$ raises each element in $H(X'_d)^{\epsilon_{sk_b}k_1}$ to the power of $e^{-1}$ to obtain $H(X'_d)^{\epsilon_{sk_b}k_1e^{-1}}$ from which he derives $H(X_d)^{\epsilon_{sk_b}k_1}$.
   d. $P_0$ updates $H_X = (H_X \setminus H(X'_d)^{\epsilon_{sk_b}k_1}) \cup H(X_d)^{\epsilon_{sk_b}k_1}$.

**Figure 10:** Protocol $\Pi_{\text{UPI-AddPSI}}$ for addition-only plain UPSI $\mathcal{T}_{\text{UPI-AddPSI}}$.

Recall that $X_d \cap X = \emptyset$ and $Y_d \cap Y = \emptyset$. Hence the new elements to be added to the intersection consists of two disjoint sets, $(X_d \cap Y)$ obtained in Step 2 and $(X \cup X_d) \cap Y_d)$ obtained in Step 3.

**Imported Lemma C.1** (Theorem 1 from [50]). Let $\{x_i\}_{i=1}^{N_d}$ be any sequence of elements being added to a binary tree via UpdateTree. Let tree node size $\sigma = 5$, tree height $L = [\log(N)]$, and stash size $\rho$. If the function $F(\cdot)$ is a random function, namely $F(x_i)$ outputs a
random leaf node, then the probability of failure (abort in Step 2d) after a sequence of UpdateTree operations corresponding to \( \{x_i\}_{i=1}^N \) can be bounded by \( \Pr[|S| > \rho] < 14 \cdot (0.6002)^\rho \).

**Security Against Corrupted \( P_0 \).** The simulator \( \text{Sim}_0 \) can be constructed that simulates \( P_0 \)'s view as follows. On input \((1^\lambda, \{X_i[D], \mathcal{T}_0(X_i[D], Y_i[D])\})\), \( \text{Sim}_0 \) runs the honest \( P_0 \) to generate its view and behaves on behalf of an honest \( P_1 \) with the following exceptions on each day \( d \in [D] \):

- In Step 5, \( \text{Sim}_0 \) will change \( m_3 \) depending on the functionality. Let \( \Gamma_d \) be the number of elements in \( m_3 \) for an honest \( P_1 \) on day \( d \), which is derived from the public parameters.
  - In \( \mathcal{T}_0(X_i[D], Y_i[D]) = \{C_1, C_2, \ldots, C_D\} \) where \( C_d \) is the number of elements in the intersection after day \( d \). Let \( C := C_d - C_{d-1} \). \( \text{Sim}_0 \) samples \( a_i \sim \mathbb{Z}_q \) for all \( i \in [\Gamma_d - C] \) and a random permutation \( \pi \) over \( [\Gamma_d] \), and sends the message:

\[
m_3 = \pi \left( \left\{ \text{PartDec}_k(\text{Enc}_{pk}(a_i)) \right\}_{i=1}^{\Gamma_d - C} \cup \left\{ \text{PartDec}_k(\text{Enc}_{pk}(0)) \right\}_{j=\Gamma_d}^C \right).
\]

- In \( \mathcal{T}_0(X_i[D], Y_i[D]) = \{C_1, Z_1), (C_2, Z_2), \ldots, (C_D, Z_D)\} \) where \( C_d \) is the number of elements in the intersection after day \( d \). Let \( C := C_d - C_{d-1} \). \( \text{Sim}_0 \) samples \( a_i, \beta_i \sim \mathbb{Z}_q \) for all \( i \in [\Gamma_d - C] \) and \( y_j \sim \mathbb{Z}_q \) for all \( j \in [C] \). It also samples a random permutation \( \pi \) over \( [\Gamma_d] \) and sends the message:

\[
m_3 = \pi \left( \left\{ \text{PartDec}_k(\text{Enc}_{pk}(a_i)), \text{Enc}_{pk}(\beta_i) \right\}_{i=1}^{\Gamma_d - C} \cup \left\{ \text{PartDec}_k(\text{Enc}_{pk}(0)), \text{Enc}_{pk}(y_j) \right\}_{j=\Gamma_d}^C \right).
\]

- In \( \mathcal{T}_0(X_i[D], Y_i[D]) = \{C_1, Z_1), (C_2, Z_2), \ldots, (C_D, Z_D)\} \) where \( C_d \) is the number of elements in the intersection after day \( d \). Let \( C := C_d - C_{d-1} \). \( \text{Sim}_0 \) samples \( a_i, \beta_i \sim \mathbb{Z}_q \) for all \( i \in [\Gamma_d - C] \) and \( y_j \sim \mathbb{Z}_q \) for all \( j \in [C] \). It also samples a random permutation \( \pi \) over \( [\Gamma_d] \) and sends the message:

\[
m_3 = \pi \left( \left\{ \text{PartDec}_k(\text{Enc}_{pk}(a_i)), \text{Enc}_{pk}(\beta_i) \right\}_{i=1}^{\Gamma_d - C} \cup \left\{ \text{PartDec}_k(\text{Enc}_{pk}(0)), \text{Enc}_{pk}(y_j) \right\}_{j=\Gamma_d}^C \right).
\]

- In Step 6, an honest \( P_1 \) sends a message \( \pi' \) in \( \Pi_{\text{UPSI-Add}} \), where \( \text{Sim}_0 \) will instead send \( \pi' = \text{PartDec}_{sk_i}(\text{Enc}_{pk}(V)) \) where \( V := V_d - V_{d-1} \) on day \( d \).
- In Step 7, \( \text{Sim}_0 \) samples a random set \( Y'_d \sim \mathbb{Z}_q^{[X_d]} \) and sends \( m_5 = \left( \{ \text{update}_d(j, \xi)(\cdot) \}_{j=1}^N, \mathcal{S}_1 \right) \leftarrow \text{UpdateTree}(Y'_d, \bot, D_1, S_1, F_k(\cdot), \text{Enc}_{pk}(\cdot)) \).

Finally, \( \text{Sim}_0 \) outputs \( P_0 \)’s view. Using a hybrid argument, we can prove for any \( D \in \mathbb{N} \) and any inputs \((X_i[D], Y_i[D])\),

\[
\left( \text{View}_0^{\Pi D}(X_i[D], Y_i[D]), \text{Out}_1^{\Pi D}(X_i[D], Y_i[D]) \right)
\leq \left( \text{Sim}_0(1^\lambda, X_i[D], \mathcal{T}_0(X_i[D], Y_i[D]), \mathcal{T}_1(X_i[D], Y_i[D]) \right).
\]

**Hyb_0:** \( P_0 \)’s view together with \( P_1 \)’s output in the real protocol.

**Hyb_1:** Same as \( \text{Hyb}_1 \) except that \( P_1 \)’s output is replaced with \( \mathcal{T}_1(X_i[D], Y_i[D]) \). This follows from the correctness of the protocol.

**Hyb_2:** Same as \( \text{Hyb}_1 \) except that in \( \Pi_{\text{UPSI-Add}} \), \( m_5' \) is replaced with \( \text{PartDec}_{sk_i}(\text{Enc}_{pk}(V)) \) for \( V = V_d - V_{d-1} \) on each day \( d \in [D] \). This hybrid is statistically indistinguishable from \( \text{Hyb}_1 \) by the re-randomization of the encryption scheme.

**Hyb_3:** Same as \( \text{Hyb}_1 \) except that \( P_0 \)'s view as follows. On input \((1^\lambda, X_i[D], \mathcal{T}_0(X_i[D], Y_i[D])\)), \( \text{Sim}_0 \) runs the honest \( P_0 \) to generate its view and behaves on behalf of an honest \( P_0 \) with the following exceptions on each day \( d \in [D] \):

- In Step 2, \( \text{Sim}_0 \) samples a random set \( X'_d \sim \mathbb{Z}_q^{[X_d]} \) and random associated values \( P'_d \sim \mathbb{Z}_q^{[X_d]} \) and sends \( m_1 = \left( \{ \text{update}_d(j, \xi)(\cdot) \}_{j=1}^N, \mathcal{S}_0 \right) \leftarrow \text{UpdateTree}(X'_d, P'_d, D_0, S_0, F_k(\cdot), \text{Enc}_{pk}(\cdot)) \).

**Security Against Corrupted \( P_1 \).** \( \text{Sim}_1 \) can be constructed that simulates \( P_1 \)'s view as follows. On input \((1^\lambda, X_i[D], \mathcal{T}_0(X_i[D], Y_i[D])\)), \( \text{Sim}_1 \) runs the honest \( P_0 \) to generate its view and behaves on behalf of an honest \( P_0 \) with the following exceptions on each day \( d \in [D] \):

- In Step 2, \( \text{Sim}_1 \) samples a random set \( X'_d \sim \mathbb{Z}_q^{[X_d]} \) and random associated values \( P'_d \sim \mathbb{Z}_q^{[X_d]} \) and sends \( m_1 = \left( \{ \text{update}_d(j, \xi)(\cdot) \}_{j=1}^N, \mathcal{S}_0 \right) \leftarrow \text{UpdateTree}(X'_d, P'_d, D_0, S_0, F_k(\cdot), \text{Enc}_{pk}(\cdot)) \).
In Step 3, $Sim_1$ will sample random values $a_i, b_i \overset{\$}{\leftarrow} \mathbb{Z}_q$ for all $i \in \{A_i\}$ where $A_i = \{m_2\}$ is the number of elements in $m_2$ for an honest $P_b$, and sends the message $m_2 = \{\{\text{Enc}_{pk}(a_i), \text{Enc}_{sk}(b_i)\}\}_{i=1}^{A_i}$.

In Step 6, in $\Pi_{\text{UniP-Add}}$, $Sim_1$ samples a random value $y \leftarrow \mathbb{Z}_q$ and sends $m_4 = \text{Enc}_{pk}(y)$. In $\Pi_{\text{UniP-Add}}^\text{out}$, $F_0(X[D], Y[D]) = \{(C_1, Z_1), (C_2, Z_2), \ldots, (C_d, Z_d)\}$ where $C_d$ is the number of elements in the intersection, $Z_d = \{(z_{i1}), (z_{i2}), \ldots, (z_{ic_d}, c_{d-1})\}$ is a set of $P_b$’s secret shares for elements added to the intersection on day $d$. $Sim_1$ sends the message

$$m_4 = \{\text{PartDec}_{sk_i}(\text{Enc}_{pk}(\{z_{i1}\}))\}_{i=1}^{C_d-c_{d-1}}.$$  

Finally, $Sim_1$ outputs $P_1$’s view. Using a hybrid argument, we can prove for any $D \in \mathbb{N}$ and any inputs $(X[D], Y[D])$.

$$\text{View}^{\Pi_{\text{ID}}} = \{\text{Out}^{\Pi_{\text{ID}}} = \text{Enc}_{pk}(y)\}. \subseteq \text{Sim}_1(1^d, X[D], Y[D]), F_0(X[D], Y[D]), F_0(X[D], Y[D]) \cdot$$

$Hyb_1$; $P_1$’s view together with $P_0$’s output in the real protocol.

$Hyb_2$; Same as $Hyb_2$ except that $P_0$’s output in $\Pi_{\text{UniP-Add}}$ is replaced with $F_0(X[D], Y[D])$. This follows from the correctness of the protocol.

$Hyb_3$; Same as $Hyb_3$ except that in $\Pi_{\text{UniP-Add}}$, $Sim_1$ replaces $m_4$ with a partial decryption of a fresh ciphertext $p_k - [[z]]\|_0$. This is actually a series of hybrids where the $i$-th element is replaced in $Hyb_3$. These hybrids are statistically indistinguishable by the re-randomization of the encryption scheme. Note that $p_k - [[z]]\|_0 = [[z]]\|_0$ so this can also be seen as a fresh encryption of $[[z]]\|_0$. Let $Hyb_2$ be the last hybrid in this series of hybrids.

$Hyb_4$; Same as $Hyb_4$ except that in $\Pi_{\text{UniP-Add}}^\text{add}$, $m_4$ is replaced by a fresh encryption of $\text{Enc}_{pk}(\Sigma_{k \in K} p_k)$. This is statistically indistinguishable by the re-randomization of the encryption scheme.

$Hyb_5$; Same as $Hyb_5$ except that in $\Pi_{\text{UniP-Add}}^\text{sum}$, $Sim_1$ samples $y \leftarrow \mathbb{Z}_q$ and replaces $m_4$ with $\text{Enc}_{sk}(y)$. This is computationally indistinguishable by the CPA security of the encryption scheme.

$Hyb_6$; Same as $Hyb_6$ except that each tuple in $m_2$ is replaced with a tuple of fresh encryptions: $(\text{Enc}(y_i, j - x_i), \text{Enc}_{sk}(p_i))$. This is a series of hybrids where the $b$-th element of the $i$-th tuple is replaced in $Hyb_6$. These are statistically indistinguishable by the re-randomization of the encryption scheme. Let $Hyb_6$ be the last hybrid in this series of hybrids.

$Hyb_7$; Same as $Hyb_7$ except that each tuple in $m_2$ is replaced with a tuple of fresh encryptions of random values. This is a series of hybrids where the $b$-th element of the $i$-th tuple is replaced in $Hyb_7$. These are computationally indistinguishable by the CPA security of the encryption scheme. Let $Hyb_7$ be the last hybrid in this series of hybrids.

$Hyb_8$; Same as $Hyb_8$ except that $P_0$ never aborts in Step 2d of $\text{UpdateTree}$. This hybrid computationally indistinguishable from $Hyb_8$ because of the pseudorandomness of $F_k(\cdot)$ and Imported Lemma C.1.

$Hyb_9$; Same as $Hyb_9$ except that $X_d$ is replaced with $X_d'$ and $P_d$ is replaced with $P_d'$ in $\text{UpdateTree}$ for $m_1$. By the construction of $\text{UpdateTree}$ and CPA security of the encryption scheme, this hybrid is computationally indistinguishable. $P_b$’s view in this hybrid is exactly $Sim_1$’s output, concluding the proof.

## D PROOF FOR SUBPROTOCOL $\Pi_{\text{CombinePath}}$

In this section, we prove correctness and security of the subprotocol $\Pi_{\text{CombinePath}}$ presented in Section 4.3.

### Correctness

Given the correctness of the additionally homomorphic encryption scheme and $\mathcal{F}_{\text{lookup}}$, we prove that $\Pi_{\text{CombinePath}}$ correctly outputs shares $\|\Sigma_{x=y}(p \cdot q_i)\|$ over $\mathbb{Z}_q$ where $(x, p)$ are inputs to the subroutine and $(y_i, q_i)$ are input as encrypted ciphertexts in path. Consider first $t$ such that $x \neq y_i$. Per the homomorphic operations done in Step 1, $\gamma_i = y_i - x + \alpha_i$ over $\mathbb{Z}_q$ and so $\mathcal{F}_{\text{lookup}}$ will receive $a = y_i - x + \alpha_i$ and $b = \alpha_i$. Because $x \neq y_i, \alpha_i \neq y_i - x + \alpha_i$ over $\mathbb{Z}_q$ and so Responder will receive $m_1$ from $\mathcal{F}_{\text{lookup}}$. Per Step 4, $\|\Sigma_{r_i} = \text{Dec}_{sk}(m_1) = -b_i$ and $\|\Sigma_{r_i} = \beta_i$ and so $r_i = 2 \|\Sigma_{r_i} + \|\Sigma_{r_i} = -\beta - \beta = 0$. In the case where $x = y_i, \alpha_i = y_i - x + \alpha_i$, the Responder receives $m_1$ and $\|\Sigma_{r_i} = \text{Dec}_{sk}(m_1) = p - q_i$. Therefore $r_i = p - q_i$. In Step 5, each party will output $\|\Sigma_{r_i} = \|\Sigma_{r_i}$. Because $r_i = p - q_i$ if and only if $y_i = x$ and 0 otherwise, this correctly results in shares of $\|\Sigma_{x=y}(p \cdot q_i)\|$.

### Security

For the security of our addition and deletion protocols, it suffices to show that the $\Pi_{\text{CombinePath}}$ subroutine can be simulated against an adversarial Responder. Let $\text{View}_R^\text{CP}(x, p, path, sk, pk)$ be the Responder’s view, and we can construct a simulator $Sim_{\text{CP}}$ like so:

- **In Step 1**, $Sim_{\text{CP}}$ samples $\gamma_i \leftarrow \mathbb{Z}_q$ for all $i \in \{k\}$ and sets $req_i = \text{Enc}_{pk}(\gamma_i)$.
- **In Step 4**, $Sim_{\text{CP}}$ samples $\delta_i \leftarrow \mathbb{Z}_q$ for all $i \in \{k\}$ such that $\sum_{i} \delta_i = 1$ and sends $m_i = \text{Enc}_{pk}(\delta_i)$ to Responder on behalf of $\mathcal{F}_{\text{lookup}}$.

Given that, we present the following lemma:

**Lemma D.1.** In the $\mathcal{F}_{\text{lookup}}$-hybrid model, there exists a PPT simulator $Sim_{\text{CP}}$ such that, for any inputs $x, p, path, sk, pk$, and Responder output $\|\Sigma_{\gamma} = 1\|$, $\text{View}_R^\text{CP}(x, p, path, sk, pk) \subseteq \text{Sim}_{\text{CP}}(1^d, \|\Sigma_{\gamma} = 1\|, pk, k)$.

**Proof.** We prove the lemma with a hybrid argument:

$Hyb_0$; $P_0$’s real view — i.e., $\text{View}_R^\text{CP}(x, p, path, sk, pk)$.

$Hyb_1$; This is a series of hybrids where, in $Hyb_{1, t}, m_{1,t} \overset{\$}{\leftarrow} \mathbb{Z}_q$ is replaced with a fresh encryption of $(p \cdot q_i - \beta_i)$. This is indistinguishable by the re-randomization property of the encryption scheme.

$Hyb_{2,t}$; This is a series of hybrids where in $Hyb_{2,t}$ $Sim_{\text{CP}}$ samples $\delta_i \leftarrow \mathbb{Z}_q$ such that $\sum_{i=1}^{k} \delta_i = 1$ and sends $\text{Enc}_{pk}(\delta_i)$ to the Responder as the output of $\mathcal{F}_{\text{lookup}}$. In the case where $m_i = m_{1,t}$, this is statistically indistinguishable because $p \cdot q_i - \beta_i$ has a uniform distribution over $\mathbb{Z}_q$ because $\beta_i$ is sampled uniformly from it. In the case where $m_i = m_{1,t}$, this is indistinguishable because $-\beta_i$ has a uniform distribution over $\mathbb{Z}_q$ for the same reason.
Hyb_{d,i}: This is a series of hybrids where, in Hyb_{d,i}, SimCP samples \( y_1 \leftarrow Z_q \) and sends req_{i} = Enc_{pk}(y_1) in Step 2. This is statistically indistinguishable because \( y_1 - x + \alpha_i \) has a uniform distribution over \( Z_q \) because \( \alpha_i \) is sampled uniformly from it.

The Responder’s view in this hybrid is exactly SimCP’s output, concluding the proof.

\[ \square \]

**E  PROOF OF THEOREM 4.3**

**Correctness.** As before, we prove correctness by induction. On day 0, all sets are initialized as null sets so correctness trivially holds. Now, on any day \( d \), our goal is to compute \( I_d = (X \cup X^+_d \setminus X^-_d) \cap (Y \cup Y^+_d \setminus Y^-_d) \). First, similar to the addition only protocols in Section 3, from the correctness of the subroutines UpdateTree, GetPath and subprotocol \( \Pi_{CombinePath} \), the additive homomorphic encryption scheme and the secret sharing scheme, one can observe that in the output generation, in Step 3c, the output computed is indeed using the reconstruction of the intersections listed in Steps 1b, 1c of the deletion phase and steps 2b, 2c of the addition phase. That is, in those 4 steps, after the reconstruction, we compute

\[
I_1 = X^-_d \cap Y, \\
I_2 = (X \setminus X^-_d) \cap Y^-_d, \\
I_3 = X^+_d \cap (Y \setminus Y^+_d), \\
I_4 = (X \setminus X^+_d) \cap Y^+_d.
\]

Elements in \( I_1 \) and \( I_2 \) are removed from \( I_d \) (since the reconstructed share is negative) and elements in \( I_3 \) and \( I_4 \) are added to \( I_d \) (since the reconstructed share is positive). That is, our protocol computes

\[
I_d = ([I_{d-1} \cup I_2] \setminus I_1) \setminus I_2, \text{ where } I_{d-1} = X \cap Y.
\]

Recall that we assume no element can be added and deleted on the same day. That is, \( X^+_d \cap X^-_d = Y^+_d \cap Y^-_d = \emptyset \). Now, let’s expand \( I_d \) as follows:

\[
I_d = (X \cup X^+_d \setminus X^-_d) \cap (Y \cup Y^+_d \setminus Y^-_d)
\]

\[
= \left( (X \cup X^+_d \setminus X^-_d) \cap (Y \setminus Y^+_d) \right) \cup \left( (X \cup X^+_d \setminus X^-_d) \cap (Y^+_d) \right)
\]

\[
= \left( (X \setminus X^-_d) \cap (Y \setminus Y^+_d) \cup I_4 \right) \cup I_4 \quad \text{(by definition)}
\]

\[
= \left( (X \setminus X^-_d) \cap (Y \setminus Y^+_d) \cup (X^+_d \setminus X^-_d) \cap (Y \setminus Y^+_d) \right) \cup I_4
\]

\[
= \left( (X \setminus X^-_d) \cap (Y \setminus Y^+_d) \right) \cup I_3 \cup I_4 \quad \text{(by definition since } X^+_d \cap X^-_d = \emptyset)\]

Now, let’s rewrite things a bit more. Observe that:

\[
X \cap Y = \left( (X \setminus X^-_d) \cap (Y \setminus Y^+_d) \right) \cup \left( (X \setminus X^-_d) \cap Y \right) \cup \left( (X \setminus X^-_d) \cap Y^+_d \right)
\]

\[
I_{d-1} = \left( (X \setminus X^-_d) \cap (Y \setminus Y^+_d) \right) \cup I_1 \cup I_2
\]

In other words, \( (X \setminus X^-_d) \cap (Y \setminus Y^+_d) = (I_{d-1} \setminus I_1) \setminus I_2 \). Putting this back into the first equation above, we get:

\[
I_d = ((I_{d-1} \setminus I_1) \setminus I_2) \cup I_3 \cup I_4
\]

Since \( I_1 \cap I_2 = I_1 \cap I_4 = I_4 \cap I_2 = \emptyset \) (by definition and by the assumption that \( X^+_d \cap X^-_d = Y^+_d \cap Y^-_d = \emptyset \)). This concludes the proof of correctness.

**Lemma E.1** (Corollary of Imported Lemma C.1). Let \( \{x_i\}_{i=1}^N \) be a sequence of elements being added or removed to a binary tree via UpdateTree where any single element is added and removed at most \( t \) times. By increasing the node size \( \sigma \) and stash size \( \rho \) by a factor of \( \min(2t, \log N) \), the probability of abort is negligible.

**Proof.** By Imported Lemma C.1, abort is negligible after adding \( N \) elements with node size \( \sigma \) and stash size \( \rho \). Let us first consider the case where we allow arbitrary additions or deletions. Note that because UpdateTree removes duplicates before placing elements into the tree, only a single additions or deletions for \( x_i \) will appear in any node. Additionally, UpdateTree keeps the invariant that additions and deletions for any element \( x_i \) in the tree will appear either in the root to leaf path \( P(F_k(x_i)) \) or in \( S \). Therefore, for any \( x_i \), the maximum number of additions and deletions that can appear in the tree or stash is \( \log N + 1 \) — one for each node in \( P(F_k(x_i)) \) and one for \( S \). In the worse case, every element appears \( \log N \) times in the tree and once in the stash, so they can be added to a tree with node size \( \log N \cdot \sigma \) and stash size \( \log N \cdot \rho \) without abort. In the case where elements can be added and removed at most \( t \) times and \( 2t < \log N \), in the worst case any element \( x_i \) will be duplicated in the tree \( 2t \) times. Therefore, a tree with node size \( 2t \cdot \sigma \) and stash size \( 2t \cdot \rho \) will suffice.

**Security against Corrupted \( P_0 \), Sim0** can be constructed that simulates \( P_0 \)’s view as follows. Let \( F_0(X_{d1}, Y_{d1}) = (Z_1, Z_2, \ldots, Z_D) \) where \( Z_i \) is the set of elements added to the intersection on day \( d \). On input \((1^t, X_{d1}, Y_{d1}, F_0(X_{d1}, Y_{d1}))\), Sim0 runs the honest \( P_0 \) to generate its view and behaves on behalf of an honest \( P_1 \) with the following exceptions on each day \( d \in [D] \):

- In the deletion phase Step 1c, Sim0 runs SimCP\((1^t, \{\overline{z_{ij}}\}_0, p_{k_0}, k_d)\) to simulate \( P_0 \)’s view of \( \Pi_{CombinePath} \) for all \( j \in [m^*] \) where \( k_d = \sigma \cdot L_0 + \rho \).
- In the deletion phase Step 1d, Sim0 samples a random set \( Y^+_{d^*} \leftarrow Z_q \) and sends \((\{\text{updates}_j, f_j\})_{j=1}^{m^*}, f_0'\) — UpdateTree\((Y^+_{d^*}, \{y_{ij}' : y_{ij}' \in Y^+_{d^*}\}_{j=1}^{m^*}, D_1, S_1, p_{k_1}, F_0(\cdot))\).
- In the addition phase Step 2c, Sim0 runs SimCP\((1^t, \{\overline{z_{ij}}\}_0, p_{k_0}, k_d)\) to simulate \( P_0 \)’s view of \( \Pi_{CombinePath} \) for all \( j \in [m^*] \) where \( k_d = \sigma \cdot L_0 + \rho \).
- In the addition phase Step 2d, Sim0 samples a random set \( Y^+_{d^*} \leftarrow Z_q \) and sends \((\{\text{updates}_j, f_j\})_{j=1}^{m^*}, S_1'\) — UpdateTree\((Y^+_{d^*}, \{y_{ij}' : y_{ij}' \in Y^+_{d^*}\}_{j=1}^{m^*}, D_1, S_1, p_{k_1}, F_0(\cdot))\).
- In the output generation phase Step 3, Sim0 does the following:
  - In Step 3b, for \( 1 \leq i \leq |Z_d| \), Sim0 sets \( \{\overline{z_i}\}_0 = 0 \) and \( \{z_i\}_1 = z_i \) where \( z_i \in Z_d \). For \( |Z_d| \leq t \leq \Gamma \) Sim0 samples shares of zero uniformly \( \{\overline{z_i}\}_0 \leftarrow Z_q \) and \( \{z_i\}_1 = -\{\overline{z_i}\}_0 \).
  - In Step 3c, Sim0 encrypts the all \( \{\overline{z_i}\}_0 \) set in Step 3b and uses that in their response.

Finally, Sim0 outputs \( P_0 \)’s view. Using the below hybrid argument, we show that the real and ideal worlds are indistinguishable.
Hyb\textsubscript{0}: This is the real world.

Hyb\textsubscript{1}: This is same as Hyb\textsubscript{0} except that the message in Step 3b is computed using the shares set by Sim\textsubscript{0}. Since \{\pi\}_{i\in [\mathcal{N}]} are randomly sampled and \pi is a random permutation, the set of shares that \text{P}_0 learns are identically distributed in both hybrids. Hence, they are statistically indistinguishable.

Hyb\textsubscript{2}: Same as Hyb\textsubscript{1} except that \text{P}_1 never aborts in Step 2d of UpdateTree. This hybrid computationally indistinguishable from Hyb\textsubscript{0} because of the pseudorandomness of \text{F}_k(\cdot). If one can distinguish between Hyb\textsubscript{1} and Hyb\textsubscript{2}, then it means the abort probability in Hyb\textsubscript{2} is non-negligible. By Lemma E.1, if the function \text{F}(\cdot) used in UpdateTree is a random function, then the probability of abort is negligible. Hence we can use the abort probability to distinguish between a pseudorandom function \text{F}_k(\cdot) and a random function \text{F}(\cdot).

Hyb\textsubscript{3}: Same as Hyb\textsubscript{2} except in the addition phase Step 2d, \text{Y}_d^{' +} is replaced with \text{Y}_d^{+} in UpdateTree. By the construction of UpdateTree and CPA security of the encryption scheme, this hybrid is computationally indistinguishable.

Hyb\textsubscript{4,i}: This is a series of hybrids where, in Hyb\textsubscript{4,i}, \text{P}_0’s view of the ith \text{P}_{\text{CombinePath}} in Step 2c is simulated with \text{Sim}\textsubscript{CP}(1^i, \{z_{y,ij}\}_0, \text{pk}_0, \text{kd}_0). This is computationally indistinguishable by Lemma D.1. Let Hyb\textsubscript{4} be the last hybrid in this series of hybrids.

Hyb\textsubscript{5,i}: Same as Hyb\textsubscript{4} except that \text{P}_1 never aborts in Step 2d of UpdateTree. This hybrid computationally indistinguishable from Hyb\textsubscript{4} because of the pseudorandomness of \text{F}_k(\cdot) and Lemma E.1.

Hyb\textsubscript{4}: Same as Hyb\textsubscript{4,i} except in the addition phase Step 1d, \text{Y}_d^{−} is replaced with \text{Y}_d^{' −} in UpdateTree. By the construction of UpdateTree and CPA security of the encryption scheme, this hybrid is computationally indistinguishable.

Hyb\textsubscript{5}: Same as Hyb\textsubscript{4} except that \text{P}_1 never aborts in Step 2d of UpdateTree. This hybrid computationally indistinguishable from Hyb\textsubscript{5} because of the pseudorandomness of \text{F}_k(\cdot) and Lemma E.1.

Security Against Corrupted \text{P}_1. \text{Sim}_1 can be constructed that simulates \text{P}_1’s view as follows. On input \{1^i, \text{Y}_{1[D]}\}, \text{Sim}_1 runs the honest \text{P}_1 to generate its view and behaves on behalf of an honest \text{P}_0 with the following exceptions on each day \text{d} \in [D]:

- In the deletion phase Step 1a, \text{Sim}_1 samples a random set \text{X}_d^{' +} \stackrel{\$}{\leftarrow} Z_q[X_d] and sends ((\text{update}_d, f_i))_{i=1}^{n}, \text{S}_d' \leftarrow \text{UpdateTree}(\text{X}_d^{' +}, \text{X}_d^{' −} \cup \text{X}_d), \text{S}_d, \text{pk}_0, \text{kd}_0(\cdot)).
- In the deletion phase Step 1b, \text{Sim}_1 runs \text{Sim}\textsubscript{CP}(1^i, \{z_{y,ij}\}_0, \text{pk}_1, \text{kd}_1) to simulate \text{P}_1’s view of \text{P}_{\text{CombinePath}} for all \text{d} \in [n] where \text{kd}_d = \sigma \cdot L_1 + \rho.
- In the addition phase Step 2a, \text{Sim}_1 samples a random set \text{X}_d^{' −} \stackrel{\$}{\leftarrow} Z_q[X_d] and sends \text{S}_d' \leftarrow \text{UpdateTree}(\text{X}_d^{' +}, \text{X}_d^{' −} \cup \text{X}_d), \text{S}_d, \text{pk}_0, \text{kd}_0(\cdot)).
- In the addition phase Step 2b, \text{Sim}_1 runs \text{Sim}\textsubscript{CP}(1^i, \{z_{y,ij}\}_0, \text{pk}_1, \text{kd}_1) to simulate \text{P}_1’s view of \text{P}_{\text{CombinePath}} for all \text{d} \in [n] where \text{kd}_d = \sigma \cdot L_1 + \rho.
- In the output generation phase Step 3a, \text{Sim}_1 samples random shares \{z_l\}_0 for \text{l} \in [\Gamma] and sends \text{Enc}_{\text{pk}_0}(\{z_l\}_0)^\Gamma to \text{P}_1.

Finally, \text{Sim}_1 outputs \text{P}_1’s view. Using the below hybrid argument, we show that the real and ideal worlds are indistinguishable.

Hyb\textsubscript{6}: This is the real world.

Hyb\textsubscript{7}: Same as Hyb\textsubscript{6} except that in the output generation phase Step 3a, the ciphertexts \text{Enc}_{\text{pk}_0}(\{z_l\}_0) is a tuple of fresh encryptions of random values. This is actually a series of sub-hybrids where each ciphertext is replaced in each sub-hybrid. These are computationally indistinguishable by the CPA security of the encryption scheme.

Hyb\textsubscript{6,i}: This is a series of hybrids where, in Hyb\textsubscript{6,i}, \text{P}_1’s view of the ith \text{P}_{\text{CombinePath}} in Step 2b is simulated with \text{Sim}\textsubscript{CP}(1^i, \{z_{y,ij}\}_0, \text{pk}_1, \text{kd}_1). This is computationally indistinguishable by Lemma D.1. Let Hyb\textsubscript{6} be the last hybrid in this series of hybrids.

Hyb\textsubscript{7}: Same as Hyb\textsubscript{2} except that \text{P}_1 never aborts in Step 2d of UpdateTree. This hybrid computationally indistinguishable from Hyb\textsubscript{6} because of the pseudorandomness of \text{F}_k(\cdot) and Lemma E.1.

Hyb\textsubscript{6,i}: This is a series of hybrids where, in Hyb\textsubscript{6,i}, \text{P}_1’s view of the ith \text{P}_{\text{CombinePath}} in Step 1b is simulated with \text{Sim}\textsubscript{CP}(1^i, \{z_{y,ij}\}_0, \text{pk}_1, \text{kd}_1). This is computationally indistinguishable by Lemma D.1. Let Hyb\textsubscript{6} be the last hybrid in this series of hybrids.

Hyb\textsubscript{7}: Same as Hyb\textsubscript{4} except that \text{P}_1 never aborts in Step 2d of UpdateTree. This hybrid computationally indistinguishable from Hyb\textsubscript{6} because of the pseudorandomness of \text{F}_k(\cdot) and Lemma E.1.

Hyb\textsubscript{6}: Same as Hyb\textsubscript{2} except in the addition phase Step 1a, \text{X}_d^{' +} is replaced with \text{X}_d^{' −} in UpdateTree. By the construction of UpdateTree and CPA security of the encryption scheme, this hybrid is computationally indistinguishable.

Hyb\textsubscript{5}: Same as Hyb\textsubscript{4} except in the addition phase Step 1a, \text{X}_d^{' +} is replaced with \text{X}_d^{' −} in UpdateTree. By the construction of UpdateTree and CPA security of the encryption scheme, this hybrid is computationally indistinguishable.

Hyb\textsubscript{6,i}: This is the last hybrid in this series of hybrids.\text{P}_0’s view in this hybrid is exactly Sim\textsubscript{0}’s output, concluding the proof.

F PROOF OF THEOREM 4.2

Correctness. As before, we prove correctness by induction. On day 0, all sets are initialized as null sets so correctness trivially holds. Now, on any day \text{d}, let’s define a function \text{f}(\text{x}_i, \text{y}_j) = \text{p}_i \cdot \text{q}_j where for any \text{x}_i \in (\mathcal{X} \cup \mathcal{X}_d \cup \mathcal{X}_d^{' +}) \text{y}_j \in (\mathcal{Y} \cup \mathcal{Y}_d^{' +} \cup \mathcal{Y}_d^{' −}) (\text{p}_i, \text{q}_j) are defined as in the protocol. Then, as in the correctness of the previous protocol, by definition, observe that, for both functionalities:

\[
\text{Out}_d = \sum_{\text{x}_i \in (\mathcal{X} \cup \mathcal{X}_d \cup \mathcal{X}_d^{' +})} \sum_{\text{y}_j \in (\mathcal{Y} \cup \mathcal{Y}_d^{' +} \cup \mathcal{Y}_d^{' −})} \text{f}(\text{x}_i, \text{y}_j)
\]

We can observe the following in the protocol from the correctness of the underlying primitives:
• In Step 3, the outputs are secret shares of \( \sum_{x_i \in \{X_d \cup X_d'\}, \; f(x_i, y_j)} \).
• In Step 4, the outputs are secret shares of \( \sum_{x_i \in \{X \cup X_d'\}, \; f(x_i, y_j)} \).

and by induction, \( \text{Out}_{d-1} = \sum_{y_j \in Y} f(x_i, y_j) \). Then, by the correctness of the reconstruction of the secret sharing scheme, \( \text{Out}_d \) is correctly computed and this completes the proof.

**Security Against Corrupted**\( P_0 \). Sim_0 can be constructed that simulates \( P_0 \)'s view as follows. On input \((1^k, X_d), \) Sim_0 runs the honest \( P_0 \) to generate its view and behaves on behalf of an honest \( P_1 \) with the following exceptions on each day \( d \in [D] \):

- In Step 4, Sim_0 runs Sim_{CP}(1^k, \{[z_{ij}], 0, pk_0, k_d\}) to simulate \( P_0 \)'s view of \( \Pi_{\text{CombinePath}} \) for all \( j \in [m] \) where \( k_2 = \sigma \cdot L_0 + \rho \).
- In Step 5, Sim_0 samples a random set \( (Y_d' \cup Y_d') \) \( \overset{\$}{\leftarrow} \mathbb{Z}^{|Y_d'|+|Y_d'|} \) and sends \(( ((\text{updates}, t_i))_{i=1}^{|Y_d'|}, S_i) \) \( \overset{\$}{\leftarrow} \Pi_{\text{UpdateTree}}(Y_d' \cup Y_d') \).

- In the output generation phase Step 7, Sim_0 does the following.
  - First, let \( \mathcal{F}_0(X_d, Y_d) = \{\text{Out}_2, \text{Out}_3, \ldots, \text{Out}_D\} \) where \( \text{Out}_d \) is different for \( \mathcal{F}_{\text{UPSI-Del}} \) and \( \mathcal{F}_{\text{UPSI-DelSum}} \) respectively.
  - Since Sim_0 knows \( P_0 \)'s input (and randomness), it can compute the value \([z_{ij}]_0\) that \( P_0 \) would have computed in Step 6.
  - Sim_0 computes and sends \([z_{ij}]_1 = \text{Out}_d - \text{Out}_{d-1} - [z_{ij}]_0\).

Finally, Sim_0 outputs \( P_0 \)'s view. Using the below hybrid argument, we show that the real and ideal worlds are indistinguishable.

**Hyb_0**: This is the real world.

**Hyb_1**: This is same as Hyb_0 except that the the output generation phase Step 7 happens as in the ideal world. That is, \([z_{ij}]_1\) is set as \((\text{Out}_d - \text{Out}_{d-1} - [z_{ij}]_0) \) where \([z_{ij}]_0\) is \( P_0 \)'s share computed in Step 6.

From the correctness of \( \Pi_{\text{CombinePath}} \) and security of the secret sharing scheme, the share \([z_{ij}]_1\) that \( P_0 \) learns is identically distributed in both hybrids and so are statistically indistinguishable.

**Hyb_2**: Same as Hyb_1 except that \( P_1 \) never aborts in Step 2d of \( \Pi_{\text{UpdateTree}} \). This hybrid computationally indistinguishable from Hyb_0 because of the pseudorandomness of \( F_k(\cdot) \) and Lemma D.1.

**Hyb_3**: Same as Hyb_2 except in Step 5, \( (Y_d^+ \cup Y_d^-) \) is replaced with \((Y_d^+ \cup Y_d^-)^* \) in \( \Pi_{\text{UpdateTree}} \). By the construction of \( \Pi_{\text{UpdateTree}} \) and CPA security of the encryption scheme, this hybrid is computationally indistinguishable.

**Hyb_4**: This is a series of hybrids where, in Hyb_{4,i}, \( P_0 \)'s view of the \( i^{th} \) \( \Pi_{\text{CombinePath}} \) in Step 4 is simulated with Sim_{CP}(1^k, \{[z_{ij}], 0, pk_0, k_d\}) which is computationally indistinguishable by Lemma D.1. \( P_0 \)'s view in the last hybrid of this series is exactly Sim_0's output, concluding the proof.