Artifact Evaluated NDSS
Available
Functional
Reproduced

Recurrent Private Set Intersection for Unbalanced Databases with Cuckoo Hashing and Leveled FHE

Eduardo Chielle
Center for Cyber Security
New York University Abu Dhabi, UAE

Michail Maniatakos
Center for Cyber Security
New York University Abu Dhabi, UAE

***Abstract*—A Private Set Intersection (PSI) protocol is a cryptographic method allowing two parties, each with a private set, to determine the intersection of their sets without revealing any information about their entries except for the intersection itself. While extensive research has focused on PSI protocols, most studies have centered on scenarios where two parties possess sets of similar sizes, assuming a semi-honest threat model. However, when the sizes of the parties' sets differ significantly, a generalized solution tends to underperform compared to a specialized one, as recent research has demonstrated. Additionally, conventional PSI protocols are typically designed for a single execution, requiring the entire protocol to be re-executed for each set intersection. This approach is suboptimal for applications such as URL denylisting and email filtering, which may involve multiple set intersections of small sets against a large set (e.g., one for each email received). In this study, we propose a novel PSI protocol optimized for the recurrent setting where parties have unbalanced set sizes. We implement our protocol using Levelled Fully Homomorphic Encryption and Cuckoo hashing, and introduce several optimizations to ensure real-time performance. By utilizing the Microsoft SEAL library, we demonstrate that our protocol can perform private set intersections in 20 ms and 240 ms on 10 Gbps and 100 Mbps networks, respectively. Compared to existing solutions, our protocol offers significant improvements, reducing set intersection times by one order of magnitude on slower networks and by two orders of magnitude on faster networks.**

I. Introduction

A. Private Set Intersection

Private Set Intersection (PSI) is the ability to compute the intersection of two or more sets held by different parties in a way that reveals the intersection to one or all parties, and nothing more. In other words, it enables parties to determine the common entries in their sets while preserving the privacy of the non-common entries. PSI has applications in data analysis, collaborative filtering, recommendation and personalization systems, market research, and customer profiling. For example, in the context of market research and customer profiling, online advertising can use PSI protocols to privately measure ad conversion rates [42]. In data analysis, companies routinely share and mine shared information, many times using insecure methods of exchanging hashed values [54]. This can be made secure with the use of PSI protocols [35].

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240365
www.ndss-symposium.org

PSI has been actively studied, mainly in the context of two parties with sets of similar sizes [22], [45], [42], [35], [44], [46], [43], [41]. Nevertheless, when there is significant asymmetry in the set sizes, a general solution may underperform a specialized solution in practice, even if it has lower overall complexity. This has been shown in works considering one party with a small set and another with a large set [8], [7], [10], [15], [16], [52], [30]. PSI protocols for unbalanced sets have practical uses in private contact discovery, where a client may want to find which of its contacts also utilize a particular platform (e.g. chat applications) [8]. In this case, the client has a small set (its contact list), while the service provider has a large set (the userbase). Under these circumstances, a PSI protocol optimized for unbalanced sets is preferred.

Existing PSI protocols focus on a single instance of the protocol. For instance, if we want to compute the set intersection $X \cap Y$ and later $X \cap Y'$, where X, Y, and Y' are sets, we have to execute the entire protocol twice, even if one of the sets does not change. Yet, there are scenarios where a party would like to recurrently intersect different small sets with the same large set. Common use cases include fraud detection, supply chain verification, targeted marketing, and URL denylisting. The latter is a security measure employed by organizations to restrict access to specific websites or web pages deemed harmful, inappropriate, or otherwise undesirable [55]. Email security solutions compare URLs in incoming emails against a denylist of known malicious websites to protect users against phishing attacks, thus preventing sensitive information, such as login credentials and financial data, from being compromised by cybercriminals [47]. However, while URL denylisting serves a security purpose, it can also raise privacy concerns, since it typically involves sending the list of URLs to a third party for comparison against its denylist [50].

PSI protocols provide a means to implement these applications while safeguarding the privacy of both the organization's and the service provider's confidential data. In this setting, we anticipate a disparity between the organization's and service provider's set sizes; the former being likely small, and the latter likely large [4]. In URL denylisting, organizations with several users or employees expect a high frequency of incoming emails, while the denylist has relatively seldom updates. This asymmetry makes the re-execution of the entire PSI protocol inefficient as the communication of each instance of the protocol relies on both sets [8]. Our proposal addresses this issue by optimizing redundant computations and communications, thus making the communication on the large set a one-time cost while having a recurrent linear cost on the smaller set.

B. Related Work

Extensive research has been conducted on PSI protocols over the course of many years, most of which focus on two parties in the semi-honest threat model. We follow related work's terminology by denoting the two parties as *sender* and *receiver*, where the receiver is the one that learns the intersection. PSI protocols fall into two main categories: those that are adaptable or readily adjustable to tackle similar tasks that are functions of the intersection $f(X \cap Y)$ (such as the PSI cardinality protocol [43]), termed as *Circuit-based PSI protocols*, and those specifically crafted for private set intersections, necessitating substantial modifications for addressing analogous issues (e.g. Labelled PSI [7], [15]), which we refer as *Exclusive PSI protocols*. We can further categorize Exclusive PSI protocols into two types: those intended for resolving set intersections in a general context, without presumptions about sets sizes, network speed, etc, labeled as *Complexity-based PSI protocols*, and those tailored for particular scenarios (e.g., unbalanced set sizes, low communication bandwidth), referred to as *Specialized PSI protocols*. It is evident that while broader PSI protocols offer greater versatility by addressing a wider range of problems, specialized PSI protocols are more efficient for the specific scenarios they target.

Circuit-based PSI protocols are commonly built using boolean primitives such as Yao's garbled circuits [53], [3], [28]. HEK [32], [31] demonstrated that their Circuit-based PSI protocol crafted using garbled circuits outperforms the blind-RSA protocol of CT [17]. PSZ improved on HEK's protocol by employing features of random Oblivious Transfer (OT) to optimize the performance of multiplexer gates, which comprise the majority of the circuit design [45], [42]. Further advancements were made by PSTY [43] through the incorporation of Oblivious Programmable Pseudo-Random Functions (OPPRF) introduced by KMPRT [36]. PSWW [44] proposes a variant of Cuckoo hashing, where each party contains four tables organized into a 2×2 grid. The receiver maps its entries into a single column of the grid, while the sender maps its entries into a single row of the grid. This approach reduces the number of comparisons per element during set intersection.

Complexity-based PSI protocols prioritize minimizing the computation and communication complexity of computing set intersections. Consequently, such protocols typically demonstrate robust scalability across varying settings. FNP [22] proposed a PSI protocol where the receiver encodes its set as roots of a polynomial, which are evaluated by the sender using an additive homomorphic encryption scheme (Paillier). More recent works typically rely on OT extensions [33] for their construction. DCW [19] combined OT extensions with Bloom filters [5] to obtain an efficient PSI protocol. PSZ [45] improved on that protocol by replacing the Bloom filter with a 2-way Cuckoo hashing with stash, increasing bin utilization and, therefore, reducing communication cost. This work has been significantly improved by PSSZ [42], which is considered the most efficient Complexity-based PSI protocol to date. In their work, the authors used a new 1-out-of-N OT protocol instead of the previously used 1-out-of-2 OT protocol. In addition, they replaced the 2-way Cuckoo hashing with stash by a 3-way Cuckoo hashing with stash, increasing bin utilization further. Moreover, they proposed the use permutation-based hashing to reduce the bit size of data stored in the hash table.

Specialized PSI protocols are tailored for specific scenarios. Common settings are unbalanced set sizes, fast or slow networks, low processing power, or large entry sizes. KKRT [35] proposed an Oblivious Pseudo-Random Function (OPRF) based on the groundbreaking work on Oblivious Transfers of IKNP [33], and applied it to the Private Set Intersection problem in the semi-honest threat model. The proposal removes the dependency on the bit size of set entries, making it relatively faster for large bit sizes (> 128 bits). PRTY [41] presented a protocol tailored for slow networks (10 Mbps). In their paper, they proposed a new variant of oblivious transfer extension called *sparse OT extension* that relies on manipulating high-degree polynomials over large finite fields. In the context of unbalanced sets, CLR [8] proposed an approach using Fully Homomorphic Encryption (FHE) and Cuckoo hashing without stash, optimized for the setting where the receiver's set is much smaller than the sender's set. They showed that their work computes the set intersection faster than PSSZ [42] and KKRT [35] for the targeted scenario.

C. Contributions

Our work addresses a critical gap in the context of Private Set Intersection protocols by introducing the first secure and practical solution tailored for the recurrent intersection of unbalanced sets. While existing work requires parties to recompute the entire PSI protocol for each subsequent set intersection, leading to inefficiencies, our method leverages leveled fully homomorphic encryption to enable efficient computation and communication for the recurring setting. Specifically, our protocol achieves communication linear to the smaller set size for subsequent intersections, which is comparable to the naive, yet insecure solution. Additionally, we propose several optimizations to mitigate computation and communication overheads. In summary, we:

- Develop a PSI protocol based on Leveled FHE, optimized for the recurrent set intersection scenario;
- Propose and integrate various optimizations to significantly reduce computation and communication costs;
- Conduct thorough analysis to determine optimal parameters and avoid resource-intensive FHE operations;
- Demonstrate real-time performance for targeted application, showcasing one to two orders of magnitude faster set intersections compared to prior work;
- Open-source our C++ implementation, built using the Microsoft SEAL library and BFV encryption scheme.

By addressing these challenges, our work contributes a practical solution for efficient and secure PSI in the recurrent setting, with implications for various privacy-preserving applications. We compare our work with CLR [8], the fastest PSI protocol for unbalanced sets, KKRT [35], a fast protocol for large entries in fast networks, and PSSZ [42], the most efficient PSI protocol in the general case.

Paper Roadmap: §II introduces the necessary notations, background information on Leveled FHE, alongside the threat model. In §III, we present a basic PSI protocol compatible with FHE. We propose several optimizations in §IV to significantly improve the basic PSI protocol and make it compatible with Leveled FHE. §V reveals the full protocol, which we evaluate and compare to related work in §VI.

II. PRELIMINARIES

A. Notations

For the remainder of this paper, and unless explicitly stated otherwise, we adhere to the following notations:

- $\mathcal{S}$ denotes *sender* and $\mathcal{R}$ denotes *receiver*;
- σ represents the set entry size in bits;
- $|\circ|$ is the size function. It returns the number of entries in a set or the bit size of an entry;
- X and Y refer to $\mathcal{S}$'s set of size $|X|$ and $\mathcal{R}$'s set of size $|Y|$, respectively;
- n, q, t are terms related to fully homomorphic encryption that represent the degree of the polynomial modulus, ciphertext modulus, and plaintext modulus;
- $\overline{\circ}$ denotes the encryption function under $\mathcal{S}$'s key. It can encrypt an entry or variable $\overline{x}$, or an entire set $\overline{X}$;
- $\widehat{\circ}$ designates the encryption function under $\mathcal{R}$'s key. It can encrypt an entry or variable $\widehat{y}$ or set $\widehat{Y}$;
- κ and λ represent the computational and statistical security parameters, respectively;
- h and k refer to the number of hash functions, and the number of hash tables in Cuckoo hashing;
- η_s and η_r represent $\mathcal{S}$'s and $\mathcal{R}$'s partitioning parameters, respectively.

B. Threat Model

In this work, we design a two-party PSI protocol using Leveled Fully Homomorphic Encryption and prove it to be secure in the semi-honest threat model, wherein parties adhere to the protocol but might attempt to learn information about each other's sets. Following most of the existing PSI protocols, we assume public knowledge of set sizes $|X|$ and $|Y|$, along with the entries' bit size σ. Nevertheless, the entries' values are deemed private. Additionally, $\mathcal{R}$ learns the intersection, while $\mathcal{S}$ learns no information.

C. Leveled Fully Homomorphic Encryption

Homomorphic Encryption (HE) is a special type of encryption that enables meaningful computation to be performed directly in the encrypted domain [2], [13]. The term homomorphism refers to a function between two groups that preserves the group structure in each group [20]. With respect to HE, the groups refer to elements in the plaintext space $\mathcal{P}$ and elements in the ciphertext space $\mathcal{C}$. Suppose we have an operation on plaintexts $\circ$ and its homomorphically equivalent operation on ciphertexts $\circledcirc$. As an example, consider plaintexts $m_a, m_b \in \mathcal{P}$ and $m_c = m_a \circ m_b$. The encryption operation defines a function $enc(\cdot) : \mathcal{P} \mapsto \mathcal{C}$ that maps a plaintext to a ciphertext. Conversely, the decryption function $dec(\cdot) : \mathcal{C} \mapsto \mathcal{P}$ maps a ciphertext to a plaintext. Therefore, we have that $m_a \mapsto enc(m_a) = c_a$, $m_b \mapsto enc(m_b) = c_b$, and $m_c \mapsto enc(m_c) = c_c$. Since $m_c = m_a \circ m_b$, we expect $c_c = c_a \circledcirc c_b$. We formally define the homomorphism property of HE operations as follows:

$$
\begin{aligned}
m_c = m_a \circ m_b \implies enc(m_c) &= enc(m_a) \circledcirc enc(m_b) \\
\implies enc(m_a \circ m_b) &= enc(m_a) \circledcirc enc(m_b) \\
\implies m_a \circ m_b &= dec(c_a \circledcirc c_b)
\end{aligned}
$$

While there are several types of homomorphic encryption, *Fully Homomorphic Encryption* (FHE) possesses at least two orthogonal homomorphic operations allowing any number of arbitrary computations [24], [25]. Without loss of generality, in this work we employ the BFV encryption scheme [21].

1) Brakerski/Fan-Vercauteren (BFV) Encryption Scheme: The BFV encryption scheme [21] is an FHE scheme based on the Ring-Learning With Errors (RLWE) problem [37]. It operates on two polynomial rings, one representing the plaintext space, characterized by the polynomial ring $\mathcal{P} = \mathbb{Z}_t[x]/(x^n + 1)$, and the other the ciphertext space, characterized by $\mathcal{C} = \mathbb{Z}_q[x]/(x^n + 1)$, where n denotes the degree of the polynomial modulus, and t, q, and $x^n + 1$ represent the plaintext, ciphertext, and polynomial moduli, respectively. Let $m \in \mathcal{P}$ be a plaintext, $k_p = (k_{p1}, k_{p2} \in \mathcal{C})$ be an encryption key, and $c = (c_1, c_2 \in \mathcal{C})$ be an encryption of m with key k_p; the encryption function $enc(k_p, m) \mapsto c$ defines a map from $\mathcal{P}$ to $\mathcal{C}$. The ciphertext c, consisting of polynomials c_1 and c_2, is computed according to Eqs. 1 and 2, where u is a random polynomial with coefficients from the set $\{-1, 0, 1\}$, e_1 and e_2 are small random polynomials draw from a discrete Gaussian distribution, and $\Delta = \lfloor q/t \rfloor$ serves as a scaling factor.

$$c_1 = k_{p1} \cdot u + e_1 + \Delta m \pmod{q} \tag{1}$$

$$c_2 = k_{p2} \cdot u + e_2 \pmod{q} \tag{2}$$

2) Leveled Fully Homomorphic Encryption: As observed in Eqs. 1 and 2, noise is added to ciphertexts for security. This noise compounds with every homomorphic operation in RLWE-based FHE schemes [6], [21], [9]. It increases linearly with ciphertext additions and exponentially with ciphertext multiplications. Eventually, at a certain computational depth, the noise becomes substantial enough to corrupt the plaintext value within the ciphertext. To mitigate this, bootstrapping can be employed. *Bootstrapping* is a technique that homomorphically decrypts a ciphertext and, thus, resets the ciphertext noise to a lower level [26]. This technique is necessary for enabling FHE in RLWE-based encryption schemes. However, bootstrapping is an extremely costly operation. Therefore, it is avoided whenever possible. FHE without bootstraping is called *Levelled Fully Homomorphic Encryption* (LFHE) [6]. In LFHE, the encryption parameters (n, q, t) are chosen to provide a desired security level κ and sufficient *noise budget* for a predefined arithmetic circuit with known depth δ and message size σ, as formalized by function $setup(\cdot)$ in Eq. 3.

$$n, q, t = setup(\kappa, \sigma, \delta) \tag{3}$$

This strategy significantly enhances the efficiency of homomorphic computation, especially for circuits with shallow depth; hence, LFHE is utilized in this work.

III. Basic Protocol

We present a basic PSI protocol in its naive form in Protocol 1. This protocol is designed for the recurrent setting with unbalanced set sizes. We emphasize that Protocol 1 is inefficient and serves only as a blueprint for our actual PSI protocol (Protocol 2). In Protocol 1, the sender $\mathcal{S}$ has a set X of size $|X|$ and the receiver $\mathcal{R}$ has a set Y of size $|Y|$, where $|Y| \ll |X|$. Both sets contain entries of size σ. In the basic protocol, we assume sets X and Y, and consequently their entries, to be private, while set sizes $|X|$ and $|Y|$, and entry size σ are considered public. Our goal is to employ FHE to privately compute the set intersection $X \cap Y$ and reveal it to $\mathcal{R}$. At the same time, we disclose nothing to $\mathcal{S}$. Furthermore, we would like to enable the recurrent computation of set intersections for different sets $Y^{(1)}, Y^{(2)}, ..., Y^{(m)}$ that may arrive at different moments in time. The goal is to perform this computation without waiting for the last set to arrive, i.e., without merging all sets into a single larger set $Y = \cup_{i=1}^{m} Y^{(i)}$ and then performing the set intersection with X. Additionally, the protocol should avoid rerunning the entire process each time a new set arrives.

Initially, the parties agree on a Fully Homomorphic Encryption scheme and encryption parameters (n, q, t). $\mathcal{S}$ encrypts each entry x_i of its set X using its encryption key, and sends the encrypted set $\overline{X}$ composed of $|X|$ ciphertexts to $\mathcal{R}$. For each entry $y_j \in Y$, $\mathcal{R}$ homomorphically subtracts y_j from each encrypted entry $\overline{x_i} \in \overline{X}$. If $y_j = x_i$, then the subtraction will be zero (encrypted), indicating that $y_j \in X$. Thus, the product of all subtractions in step (3b) will be zero if $y_j \in X$ and non-zero otherwise. To prevent $\mathcal{S}$ from learning the size of the intersection, we add a random value $r_j \in \mathbb{Z}_t$ sampled from a uniform distribution to the result of the product.[1] $\mathcal{R}$ encrypts $r_j \mapsto \widehat{r_j}$ under its own key and sends it to $\mathcal{S}$ together with the result of the computation $\overline{d_j}$, which is under $\mathcal{S}$'s key. $\mathcal{S}$ decrypts $\overline{d_j} \mapsto d_j$ and homomorphically computes $\widehat{r_j} - d_j$. Again, if the result of the subtraction is zero, this time encrypted under $\mathcal{R}$'s key, then $y_j \in X$. We multiply the result of the subtraction by a non-zero random plaintext $\rho_j \in \mathbb{Z}_t \backslash \{0\}$ sampled from a uniform distribution to prevent leaking any information from $\mathcal{S}$'s set X to $\mathcal{R}$ in case $y_j \notin X$. $\mathcal{S}$ sends the resulting ciphertext $\widehat{e_j}$ to $\mathcal{R}$, which $\mathcal{R}$ decrypts and compares to zero. If $e_j = 0$, then $y_j \in X$.

We present the following informal theorem concerning the security and correctness of the basic protocol.

Theorem 1 (Basic Protocol)**.** *Within the semi-honest threat model, Protocol 1 correctly and privately computes the intersection of sets X and Y, contingent upon the fully homomorphic encryption scheme being IND-CPA secure and achieving circuit privacy.*

Proof: We sketch a proof using the parties' views:

$\mathcal{S}$'s view: $\mathcal{S}$ receives ciphertexts $(\overline{d_j}, \widehat{r_j})$. Ciphertext $\widehat{r_j}$ looks pseudorandom to $\mathcal{S}$ since it is under $\mathcal{R}$'s key and the FHE scheme is IND-CPA secure. $\mathcal{S}$ knows the plaintext value of $\overline{d_j}$ because it is under its own key; therefore, it can decrypt it ($\overline{d_j} \mapsto d_j$). Nevertheless, d_j is a uniform random number in $\mathbb{Z}_t$ since it contains the addition of the random number r_j. Thus, $\mathcal{S}$ learns nothing about Y.

$\mathcal{R}$'s view: $\mathcal{R}$ receives $\overline{X}$ from $\mathcal{S}$. Any ciphertext $\overline{x_i} \in \overline{X}$ appears random to $\mathcal{R}$ since they are encrypted under $\mathcal{S}$'s key and the FHE scheme is IND-CPA secure. Furthermore, $\mathcal{R}$ learns the decryption of $\widehat{e_j}$ as it is under its own key ($\widehat{e_j} \mapsto e_j$). Since $\widehat{e_j} = (\widehat{r_j} - d_j) \cdot \rho_j$, where ρ_j is a uniform non-zero random number in $\mathbb{Z}_t \backslash \{0\}$, then $e_j = 0 \iff y_j \in X$ (correctness) and a non-zero random number otherwise. Therefore, $\mathcal{R}$ learns the intersection $X \cap Y$ and nothing more. ■

[1] We can allow $\mathcal{S}$ to learn the size of the intersection while keeping the protocol secure by replacing the addition of a random number with the multiplication of a non-zero random number.

Protocol 1 Basic Protocol

Input: $\mathcal{S}$ inputs set X and $\mathcal{R}$ inputs set Y. Both sets consist of bit strings of length σ. Sizes $|X|$, $|Y|$, and σ are public.
Output: $\mathcal{S}$ outputs $\perp$; $\mathcal{R}$ outputs $X \cap Y$.

1) **Setup**: Parties agree on a Fully Homomorphic Encryption scheme and encryption parameters (n, q, t).
2) **Set encryption**: $\mathcal{S}$ encrypts each element $x_i \in X$ under its own key, and sends the $|X|$ ciphertexts $\overline{X} = (\overline{x_1}, ..., \overline{x_{|X|}})$ to $\mathcal{R}$.
3) **Computing intersection**: For each $y_j \in Y$, $\mathcal{R}$:
   a) samples a random plaintext $r_j \in \mathbb{Z}_t$ from a uniform distribution;
   b) homomorphically computes
   $$\overline{d_j} = r_j + \prod_{i=1}^{|X|} (\overline{x_i} - y_j)$$
   c) encrypts $r_j \mapsto \widehat{r_j}$ under its own key;
   d) sends $\overline{d_j}$ and $\widehat{r_j}$ to $\mathcal{S}$.
4) **Decryption**: For each $(\overline{d_j}, \widehat{r_j})$, $\mathcal{S}$:
   a) decrypts $\overline{d_j} \mapsto d_j$;
   b) samples a non-zero random plaintext value $\rho_j \in \mathbb{Z}_t \backslash \{0\}$ from a uniform distribution;
   c) homomorphically computes
   $$\widehat{e_j} = (\widehat{r_j} - d_j) \cdot \rho_j$$
   d) sends $\widehat{e_j}$ to $\mathcal{R}$.
5) **Result**: $\mathcal{R}$ decrypts $\widehat{e_j} \mapsto e_j$ and outputs
   $$X \cap Y = \cup_{j=1}^{|Y|} \{y_j : e_j = 0\}$$

Now let us now consider several sets $Y^{(1)}, Y^{(2)}, ..., Y^{(m)}$ arriving at different moments in time. We can see that steps (1) and (2) in Protocol 1 are a one-time cost as they do not involve set Y. Avoiding step (2) is particularly relevant since sending $\overline{X}$ to $\mathcal{R}$ has the largest communication cost because $|Y| \ll |X|$. Therefore, for each set $Y^{(i)}$, we have to repeat *only* steps (3) to (5). Yet, step (3) is especially costly as it requires $\mathcal{R}$ to compute $\mathcal{O}(|X| \cdot |Y|)$ expensive homomorphic multiplications. This is compounded by the fact that in FHE, the multiplicative depth of an arithmetic circuit dictates the selection of encryption parameters for a desired security level. Simply put, the larger the multiplicative depth, the larger the encryption parameters n and q, which makes computation and communication using FHE ciphertexts less efficient. In the rest of the paper, we present techniques to transform this inefficient basic protocol into an efficient PSI protocol.

IV. Optimizations

A. Hashing

Hashing is a technique that maps elements into bins of a table using a deterministic function called hash function. The use of hash data structures enable search in $\mathcal{O}(1)$. This is particularly interesting for our problem, since in step (3b) of Protocol 1, we are basically performing a linear search on an unordered list. Instead, if entries in sets X and Y had been previously hashed, we would have to compare only entries within the same bin, greatly reducing the number of homomorphic operations.

1) Simple Hashing: Let us first understand how Protocol 1 would work with the use of simple hashing. $\mathcal{S}$ hashes each entry $x \in X$ with hash function $H(\cdot)$ and inserts x into the bin at position $H(x)$ of the hash table. Clearly, there will be unbalance in the number of entries in each bin, which could leak more information than the intersection to $\mathcal{R}$. To prevent such leakage, we need to pad the bins with dummy values to a maximum size. Finding the maximum load when inserting m elements into n bins is equivalent to a classical problem in probability theory known as *balls into bins problem*. It has been shown that the maximum load for $m = n$ is of order $\mathcal{O}(\frac{\log m}{\log \log m})$ with high probability [48]. This means that the number of homomorphic operations per entry $y \in Y$ performed by $\mathcal{R}$ reduces from $\mathcal{O}(m)$ to $\mathcal{O}(\frac{\log m}{\log \log m})$. On the other hand, $\mathcal{S}$ has to send $\mathcal{O}(\frac{m \log m}{\log \log m})$ ciphertexts to $\mathcal{R}$, which is a significant increase in communication for a large m.

2) Cuckoo Hashing: Ideally, we would like to get the computational benefits from hashing without the extra communication cost. We can do that by combining Cuckoo hashing with Permutation-based hashing. *Cuckoo Hashing* [39] is a method for resolving hashing collisions and build compact hash tables. It maps each one of the $|X|$ entries $x \in X$ into a bin of the hash table using one of the $h \geq 2$ hash functions $H_1, ..., H_h$. To insert entry x, we select a random hash function H_i from the set $\{H_1, ..., H_h\}$ and place (x, i) at location $H_i(x)$. If the bin is occupied, we have a collision since $H_i(x) = H_j(x')$. In such an event, we evict the current entry x' in the bin in favor of the new entry x. Then, we re-hash x' using a different hash function $H_k(x') : k \neq j$ and reinsert it at the new location. This process repeats recursively until there are no more collisions or the algorithm fails. Upon a successful hashing procedure, each bin of the Cuckoo hash table will either contain one entry or remain empty. In order to prevent any leakage, we must pad all empty bins to one entry using dummy values.

3) Hashing Failures: The Cuckoo hashing algorithm fails when it detects a loop or the recursion depth surpasses a predefined threshold. To tackle hashing failures, two solutions exist: 1) inserting the currently evicted element into a special data structure known as *stash*, typically implemented as a list, or 2) replacing the set of hash functions and restarting the algorithm. For the first approach, it has been shown that for $h = 2$, a hash table with $|T| = 2(1 + \epsilon)m$ bins, and a stash size $s \leq \ln m$, where m is the number of elements, the insertion succeeds with high probability for $\epsilon > 0$ [40], [34]. Essentially, this implies that Cuckoo hashing with stash achieves approximately 50% bin utilization when $h = 2$. Higher utilization is achievable with $h > 2$ [18]. PSSZ [42] empirically determined the stash size in which Cuckoo hashing succeeds with high probability for $h = 3$. Unfortunately, the utilization of a stash data structure severely impacts the efficiency of FHE computation [8], thus it is preferable to avoid using it.

The second option poses an information leakage risk if hashing fails. Since only $\mathcal{S}$ hashes its set into a Cuckoo hash table, it can select hash functions autonomously and share them with $\mathcal{R}$ only upon successful hashing. If hashing fails, $\mathcal{S}$ replaces the hash functions and repeats the process until it succeeds. Consequently, $\mathcal{R}$ deduces that $\mathcal{S}$'s set X originates from the set of sets $X^{(i)}$ for which Cuckoo hashing succeeds. In essence, $\mathcal{R}$ can eliminate sets $X^{(j)}$ that fail with the selected hash functions. Although this leakage is likely weak, quantifying it is challenging. Thus, to prevent any leakage without resorting to a stash, the probability of hash failure must be overwhelmingly low ($\leq 2^{-\lambda}$). CLR [8] estimated the maximum insertion capacity for small tables of sizes $\{8192, 16384\}$ with $h = 3$. In our setting, with a large set and, consequently, a large table, we would like to find the highest load where Cuckoo hashing without stash fails with negligible probability. We determine the maximum load of a Cuckoo hash table experimentally in §VI-A.

4) Permutation-based Hashing: ANS [1] proposed a technique to reduce the memory usage of Cuckoo hashing termed *Permutation-based Hashing*. Let $|x|$ denote the size of an item x, where $x = x_L|x_R$ is the concatenation of its left (x_L) and right (x_R) parts. Additionally, let $\sigma_L = |x_L| = log_2|T|$, where $|T|$ is the number of bins in the hash table. We define a random function $f(\cdot) : \mathbb{Z} \mapsto \mathbb{Z}$ that maps $i \in \mathbb{Z} : 0 \leq i < 2^{\sigma_R}$ to the range $f(i) \in \mathbb{Z} : 0 \leq f(i) < |T|$, where $\sigma_R = |x_R|$. In the original Cuckoo hashing, one has to store $|x|$ bits per item, while with permutation-based hashing, it is necessary to store only $|x_R|$ bits. This reduction is achieved by mapping x to bin $x_L \oplus f(x_R)$. The structure of the mapping function guarantees that if two items x and x' store the same value in the same bin, then $x = x'$. This assertion arises from the fact that the stored items are identical ($x_R = x'_R$), thus $f(x_R) = f(x'_R)$. Moreover, since the items are mapped to the same bin, we have that $x_L \oplus f(x_R) = x'_L \oplus f(x'_R)$, therefore $x_L = x'_L$, and ultimately $x = x'$.

5) Data Compression: When the number of entries in sets X and $Y = \cup_{i=1}^{m} Y^{(i)}$ is much less than the domain size $|X| + |Y| \ll 2^{\sigma_d}$, where σ_d is the number of bits needed to represent an entry, we can use simple hashing to compress entries and reduce their bit size.[2] As a constraint of the PSI protocol, we require the probability of collision when hashing to be $P(\text{collision}) \leq 2^{-\lambda}$. This is equivalent to the *birthday problem* [38]. Assuming that $m = |X| + |Y| \leq \theta$, where $\theta = 2^{\sigma_h}$ is the hashed domain size, we have:

$$P(\text{collision}) = 1 - \prod_{i=1}^{m-1} \left(1 - \frac{i}{\theta}\right) \approx 1 - e^{-\frac{m(m-1)}{2\theta}} \leq 2^{-\lambda}$$

Solving for θ, we find that for the hashed entries to have a $P(\text{collision}) \leq 2^{-\lambda}$, their bit size σ_h should be:

[2]Note that without $\mathcal{R}$ doing any statistical analysis of its sets, we must consider the worst case w.r.t. the combined set size $|Y| = \sum_{i=1}^{m} |Y^{(i)}|$.

$$\theta = \frac{-m(m-1)}{2\ln(1-2^{-\lambda})} \approx m^2 \cdot 2^{\lambda-1} \implies \sigma_h \approx 2\log_2 m + \lambda - 1 \quad (4)$$

Therefore, the number of bits σ necessary to represent each entry is given by:

$$\sigma = \min(\sigma_d, \sigma_h) \quad (5)$$

6) Cuckoo Hashing vs Simple Hashing: Both simple hashing and Cuckoo hashing offer fast search times. For each entry $y \in Y$, $\mathcal{R}$ has to perform $\mathcal{O}(\frac{\log m}{\log\log m})$ homomorphic operations for simple hashing or $\mathcal{O}(h)$ homomorphic operations for Cuckoo hashing, where m represents $\mathcal{S}$'s set size $|X|$ and h denotes the number of hash functions in Cuckoo hashing. For a sufficiently large set X, it holds that $h < \frac{\log m}{\log\log m}$ as h is small ($h \leq 4$). Furthermore, Cuckoo hashing exhibits advantages w.r.t. communication. While simple hashing has a communication complexity of $\mathcal{O}(\frac{m\log m}{\log\log m})$, Cuckoo hashing shows $\mathcal{O}(m)$ complexity. Additionally, in terms of the total size of hash table in bits (excluding encryption), simple hashing scales as $\mathcal{O}(\frac{\sigma_h m \log m}{\log\log m})$, while Cuckoo hashing is of the order $\mathcal{O}(|x_R| \cdot m)$ due to the utilization of permutation-based hashing. Unfortunately, simple hashing cannot leverage permutation-based hashing since it stores more than one element per bin. Limiting the bin capacity of simple hashing to one element is also undesirable, as from Eq. 4, one can see it would require a table of size $|X|^2 \cdot 2^{\lambda-1}$ for a collision probability $\leq 2^{-\lambda}$. This would result in an immense communication cost from $\mathcal{S}$ to $\mathcal{R}$ since empty bins need to be padded with dummy values for security. Therefore, Cuckoo hashing emerges as the clear choice to handle $\mathcal{S}$'s set in our protocol.

B. Quadruple Chinese Remainder Theorem (QCRT)

The *Chinese Remainder Theorem (CRT)* provides a method to solve systems of simultaneous congruences, where each congruence is modulo a different modulus, and the moduli are coprime. In other words, the system of congruences

$$x \equiv a_i \pmod{m_i} \ \forall i \in \mathbb{Z} : 1 \leq i \leq k \quad (6)$$

is solved by

$$x \equiv \sum_{i=1}^{k} a_i \cdot M_i \cdot M_i^{-1} \pmod{M} \quad (7)$$

where k is the number of congruences, $M = \prod_{i=1}^{k} m_i$, $M_i = \frac{M}{m_i}$, and $M_i \cdot M_i^{-1} \equiv 1 \pmod{m_i}$.

In this work, we apply CRT to four different levels.

1) Level 1 CRT: Residue Number System: In FHE, the ciphertext modulus q is a large number with hundreds of bits. This makes computation on encrypted data particularly slow as the coefficient size is not natively supported by modern 64-bit CPU architectures. The *Residue Number System (RNS)* enables us to break the ciphertext modulus q into several smaller unique prime moduli such that $q = \prod_{i=1}^{l} q_i$, where l is the number of limbs or towers. Thus, we can represent a large coefficient with several smaller coefficients following Eq. 6. By setting the size of each modulus $|q_i| < 64$ bits, we can efficiently operate on ciphertexts using the natively supported data sizes. This comes at a slight increase in the ciphertext size $2 \cdot n \cdot |q| \rightarrow 2 \cdot n \cdot W \cdot l$ since $l = \lceil \frac{|q|}{W} \rceil$ and $W = 64$ for a 64-bit architecture. We refer to LPR [29] for a detailed explanation of the RNS used in FHE.

2) Level 2 CRT: Polynomial Multiplications via NTT: Recall that ciphertexts are composed of two large polynomials in $\mathbb{Z}_q[x]/(x^n+1)$. In §IV-B1, we show how RNS breaks the large modulus q into smaller moduli q_i. However, we still need to operate on large polynomials. Polynomial additions and subtractions are efficiently computed in $\mathcal{O}(n)$ since we simply have to add coefficients of the same order. At the same time, naive polynomial multiplications require us to multiply each coefficient of a polynomial with all coefficients of the other polynomial. Thus, the operation has a time complexity of $\mathcal{O}(n^2)$. By using the *Number Theoretic Transform (NTT)*, which is a form of CRT on polynomials, we can convert polynomials from the *coefficient form* to the so-called *evaluation form* [14]. We can multiply (add) two polynomials in the evaluation form in $\mathcal{O}(n)$ by multiplying (adding) coefficients of the same order. This leads to a substantial speed-up of steps (3b) and (4c) of Protocol 1. Converting a polynomial from the coefficient form to the evaluation form or vice-versa has a time complexity of $\mathcal{O}(n\log n)$. Nevertheless, this cost can be amortized since we can perform many operations in the evaluation form. The combination of RNS and NTT is called *Double CRT (DCRT)* by the FHE community [27].

3) Level 3 CRT: Batching: According to the plaintext space $\mathbb{Z}_t[x]/(x^n+1)$, a plaintext must be encoded into a $(n-1)$-degree polynomial, where the n coefficients are modulo t. Encoding a single small plaintext into n coefficients is quite wasteful. *Batching* allows us to encode a vector of n plaintexts into a single $(n-1)$-degree polynomial and operate on them simultaneously in a Single-Instruction Multiple-Data (SIMD) fashion. From Eq. 7, we know it is possible to encode a tuple of messages $(p_1, p_2, p_3) \in \mathbb{Z}_a \times \mathbb{Z}_b \times \mathbb{Z}_c$ into a single value $p \pmod{a \cdot b \cdot c}$. Operations on values in $\mathbb{Z}_{a\cdot b\cdot c}$ are equivalent to SIMD operations on values in $\mathbb{Z}_a$, $\mathbb{Z}_b$, and $\mathbb{Z}_c$ simultaneously. Batching does the same, but on polynomials instead of integers using the inverse NTT operation. This enables us to perform SIMD operations on n plaintexts at the same time [11], [12]. See SV [51] for a complete description of the technique.

4) Level 4 CRT: Packing: The plaintext modulus t let us encode $\log_2 t$ bits of information into a single plaintext. If the amount of information we are encoding per batching slot is significantly less than t, then we are wasting plaintext space. Following the same idea of batching, we can once again apply CRT, but this time in $\mathbb{Z}_t$, i.e., we can divide the plaintext coefficient ring space into several smaller spaces with coprime moduli $\mathbb{Z}_{t_1}, ..., \mathbb{Z}_{t_k}$ such that $t = \prod_{i=1}^{k} t_i$. By doing so, we are able to encode k plaintexts $\in \mathbb{Z}_{t_i}$ per batching slot using Eq. 7. This enables us to pack $k \cdot n$ plaintexts into a ciphertext and operate on them in parallel, resulting in a performance improvement and communication cost reduction of several orders of magnitude since the product $k \cdot n$ is large (in the thousands to tens of thousands). Although several packing techniques for FHE exist, we are unaware of any other work that uses CRT for packing.

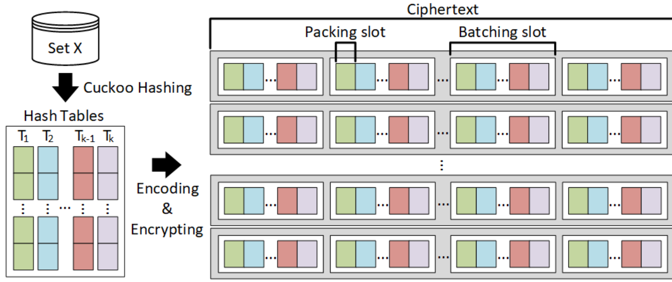

Fig. 1: Data encoding.

5) Data Encoding: With respect to our PSI protocol, the combination of batching and packing allows us to encode n bins of k Cuckoo hash tables into a single ciphertext, where each table corresponds to a particular packing slot index. Thus, all values of a table have the same modulo t_i. The reason for using k tables instead of a single one is to avoid privacy leakage. We defer the discussion of the privacy implications of QCRT to §IV-B6. To distribute the load of $\mathcal{S}$'s set X across k different tables, we define a function $g(\cdot) : \mathbb{Z} \mapsto \mathbb{Z}$ that maps $i \in \mathbb{Z} : 0 \leq i < 2^\sigma$ to the range $g(i) \in \mathbb{Z} : 1 \leq g(i) \leq k$ uniformly. Each entry $x \in X$ is then inserted into a single bin of the Cuckoo hash table $T^{(g(x))}$. With X mapped across k Cuckoo hash tables, we expect the maximum load of a table to exceed that of using a single table $\left(\max \frac{|X^{(i)}|}{|T^{(i)}|} \geq \frac{|X|}{|T|}\right)$. Finding the maximum number of entries associated with a table is equivalent to the *balls into bins problem* for the case where the number of balls m (entries $x \in X$) vastly surpasses the number of bins n (Cuckoo hash tables), i.e., $m \gg n$. When $m > n \log n$, the maximum load is $\frac{m}{n} + \mathcal{O}\left(\sqrt{\frac{m \log n}{n}}\right)$ with high probability [48]. Further details on table sizes for various set sizes and number of tables are provided in §VI-A.

Fig. 1 illustrates the process of encoding k Cuckoo hash tables into packing slots, batching slots, and ciphertexts, where each ciphertext contains n batching slots, and each batching slot contains k packing slots. Formally, we define an encoding function $\gamma(\cdot) : \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ that maps the table index $i \in \mathbb{Z} : 1 \leq i \leq k$ and bin index $j \in \mathbb{Z}_{|T^{(i)}|}$ at table $T^{(i)}$ to ciphertext index $\left\lfloor \frac{j}{n} \right\rfloor \in \mathbb{Z}_{\left\lceil \frac{|T^{(i)}|}{n} \right\rceil}$, batching slot index $j \pmod n \in \mathbb{Z}_n$, and packing slot index $i - 1 \in \mathbb{Z}_k$. The complete mapping function from the set of Cuckoo hash tables $T^{(i)}$ to the set of ciphertexts $C^{(\cdot)}$ is given by

$$T_j^{(i)} \mapsto C_{j \bmod n,\ i-1}^{\left(\left\lfloor \frac{j}{n} \right\rfloor\right)} \tag{8}$$

6) Privacy Implications of QCRT: Given that RNS and NTT manipulate ciphertexts, they do not raise any privacy concerns for our PSI protocol. However, special attention is required for batching and packing as they involve plaintext data encoding. When performing the equivalent of step (3b) of Protocol 1 on $\mathcal{S}$'s Cuckoo hash table, adding the same random value r_j to all batching slots could enable $\mathcal{S}$ to learn which bins the ciphertext contains, potentially exposing $\mathcal{R}$'s entry y. This can be mitigated by adding an independent random value to each batching slot. Therefore, instead of adding the tuple $(r_j, ..., r_j)$, we add $(r_{j1}, ..., r_{jn})$ slot-wise. The same principle applies to ρ_j at step (4c). Nevertheless, another privacy concern arises at step (4c). If $y \in X$, one slot of ciphertext $\overline{e_j}$ will decrypt to zero. By discerning the position of the zero value, $\mathcal{R}$ could deduce which of the h hash functions $\mathcal{S}$ used to hash item $x = y$. This can be prevented by homomorphically rotating ciphertext $\overline{e_j}$ by a random constant $\omega \in \mathbb{Z}_n$ before returning it to $\mathcal{R}$.

Packing encounters the same two privacy issues as batching since it involves multiple packed plaintexts within a batching slot. Nevertheless, addressing the first issue of batching also resolves the corresponding issue for packing, as a random number modulo t is equivalent to a set of random numbers modulo t_i $\forall i \in \mathbb{Z} : 1 \leq i \leq k$ (Eq. 7). Attention is needed regarding ρ_j at step (3b) since none of the set of random numbers should be zero, implying that the random number modulo t cannot be a multiple of any modulo t_i $\forall i \in \mathbb{Z} : 1 \leq i \leq k$. With respect to the second issue however, homomorphic rotation does not solve it for packing since each packed value remains under its respective modulo t_i. This problem is addressable with the use of k Cuckoo hash tables, each corresponding to a modulo t_i. This ensures that any value x is consistently mapped under the same modulo, regardless of the hash function used. Since each table T_i operates with its unique modulo t_i, $\mathcal{R}$ cannot discern which hash function $\mathcal{S}$ used to hash entry $x = y$.

7) Constraints on Parameter Selection: In §IV-B1, we establish q as a product of primes. To ensure efficient RNS in 64-bit architectures, we limit the size of each prime q_i to < 64 bits. Furthermore, we constrain the polynomial modulus degree n to be a power of two for optimal NTT (§IV-B2). Additionally, we set each $q_i = 2\beta n+1$ for some $\beta \in \mathbb{Z} : \beta > 1$ to enable the use of negative wrapped convolution. Moreover, batching (§IV-B3) requires the plaintext modulus t to be congruent to $t \equiv 1 \pmod{2n}$. Thus, for packing (§IV-B4) to be compatible with batching, it follows that $\prod_{i=1}^{k} t_i \equiv 1 \pmod{2n}$, and hence $t_i \equiv 1 \pmod{2n}$ $\forall i \in \mathbb{Z} : 1 \leq i \leq k$.

C. Other FHE Considerations

1) Partitioning: It is a technique that permits reducing the number of homomorphic operations and multiplicative depth in exchange for extra communication. Consider, for instance, the product in step (3b) of Protocol 1. When employing Cuckoo hashing, the number of homomorphic multiplications decreases from $\mathcal{O}(|X|)$ to $\mathcal{O}(h)$. Without partitioning, one would need to execute $h - 1$ homomorphic multiplications and transmit one ciphertext $\overline{d_j}$ to $\mathcal{S}$. However, with partitioning, it becomes possible to compute η fewer multiplications at the cost of sending η additional ciphertexts to $\mathcal{S}$. Partitioning operates along a spectrum: on one end lies the conventional approach of performing $h-1$ homomorphic multiplications and sending one ciphertext to $\mathcal{S}$, while on the opposite extreme, no homomorphic multiplication is performed, but h ciphertexts are sent to $\mathcal{S}$. Through the use of partitioning, we can reduce the number of homomorphic operations and multiplicative depth, albeit necessitating to transmit more ciphertexts. Nevertheless, due to the reduced multiplicative depth, it becomes possible to utilize more efficient encryption parameters, thereby resulting in smaller ciphertext sizes.

2) Modulus Switching: It is a method used to decrease the size of ciphertexts, which involves replacing the ciphertext

modulus q with a smaller modulus $q' < q$ [6]. This reduction in size is beneficial as it lowers the communication overhead. Nevertheless, modulus switching can only be applied to response ciphertexts, i.e., those ciphertexts that will not undergo further homomorphic operations since it reduces the noise budget of ciphertexts. In the context of Protocol 1, modulus switching can optimize the communication cost when $\mathcal{R}$ sends $\overline{d_j}$ to $\mathcal{S}$ in step (3d) and when $\mathcal{S}$ sends $\widehat{e_j}$ to $\mathcal{R}$ in step (4c). We highlight that modulus switching is a public operation and poses no security issues.

V. Full Protocol

Protocol 2 presents the full protocol, where we apply the optimizations proposed in §IV to Protocol 1. Similarly to Protocol 1, in Protocol 2, $\mathcal{S}$ inputs set X of size $|X|$, while $\mathcal{R}$ inputs set Y of size $|Y|$, where $|Y| \ll |X|$, with each set comprising entries of size σ_d. Initially, in step (1), parties agree on various parameters encompassing security, encryption, data compression, Cuckoo hashing, and permutation-based hashing. Subsequently, in step (2), $\mathcal{S}$ compresses each entry $x \in X$, maps it to a Cuckoo hash table using function $g(\cdot)$, and inserts the right part of the compressed entry into the designated table $T^{(g(\cdot))}$. Then, $\mathcal{S}$ encodes and encrypts the k Cuckoo hash tables in step (3) adhering to the data encoding discussed in §IV-B5, and sends the ciphertexts $\overline{C}$ to $\mathcal{R}$. Steps (1-3) encompass one-time cost operations and, therefore, do not require re-execution on subsequent set intersections with new sets $Y^{(i)}$. This is especially relevant to step (3) since it includes the most costly operations of the protocol.

The recurrent part starts in step (4), where $\mathcal{R}$ compresses each entry $y \in Y$ and locates the h potential slots for the entry. From each potential slot, $\mathcal{R}$ subtracts the right part of the compressed entry from the encrypted table. If $y \in X$, the slot becomes zero at that location (encrypted). The resulting ciphertexts $\overline{S^{(j)}}$ are multiplied in a depth-optimized way, respecting the partitioning parameter η_s, generating $\eta_s + 1$ ciphertexts $\overline{M}^{(i)}$ $\forall i \in \mathbb{Z}_{\eta+1}$. This is followed by the addition of random values $R^{(i)}$ to each slot $\overline{D}^{(i)} = \overline{M}^{(i)} + R^{(i)}$. Finally, $\mathcal{R}$ encrypts the random values $R^{(i)} \mapsto \widehat{R}^{(i)}$ with its key, modulus switches $\overline{D}^{(i)}$, and sends $(\overline{D}^{(i)}, \widehat{R}^{(i)})$ $\forall i \in \mathbb{Z}_{\eta_s+1}$ to $\mathcal{S}$. Note that $\mathcal{S}$'s ciphertexts contain n_s batching slots, while $\mathcal{R}$'s encompass n_r, where $n_s \geq n_r$. If $n_s > n_r$, $\widehat{R}^{(i)}$ comprises > 1 ciphertexts, implying that the rotation operation also shuffles the ciphertexts. We highlight that having $n_s < n_r$ is forbidden because it requires extra mitigation against leakage, which leads to a false positive rate $> 2^{-\lambda}$ for commonly used encryption parameters. In step (5), $\mathcal{S}$ decrypts the ciphertexts $\overline{D}^{(i)} \mapsto D^{(i)}$ $\forall i \in \mathbb{Z}_{\eta_s+1}$ and subtracts it from $\widehat{R}^{(i)}$. The resulting ciphertexts $\widehat{M}^{(i)}$ are multiplied in a depth-optimized way, respecting the partitioning parameter η_r, generating $\eta_r + 1$ ciphertexts $\widehat{N}^{(j)}$ $\forall j \in \mathbb{Z}_{\eta_r+1}$. If $y \in X$, then one packing slot of a single ciphertext $\widehat{N}^{(j)}$ contains the encryption of zero. To prevent any leakage from $\mathcal{S}$'s set X, $\mathcal{S}$ multiplies each batching slot by a random number non-multiple of any packing modulus, which is equivalent to multiplying each packing slot by a non-zero random number in the respective ring. In addition, $\mathcal{S}$ rotates the batching slots by a random amount, following the discussion of §IV-B6. Finally, $\mathcal{S}$ modulus switches and transmits the outcome to $\mathcal{R}$, who decrypts it $\widehat{E}^{(j)} \mapsto E^{(j)}$ $\forall j \in \mathbb{Z}_{\eta_r+1}$, verifying the presence of zero. If detected, $y \in X$.

We present the following theorem concerning the security and correctness of the full protocol.

Theorem 2 (Full Protocol)**.** *Within the semi-honest threat model, Protocol 2 correctly and privately computes the intersection of sets X and Y, contingent upon the (leveled) fully homomorphic encryption scheme being IND-CPA secure and achieving circuit privacy.*

Proof: We demonstrate the security and correctness of the protocol using the parties' views:

$\mathcal{S}$'s view: $\mathcal{S}$ receives ciphertexts $(\overline{D}^{(i)}, \widehat{R}^{(i)})$ $\forall i \in \mathbb{Z}_{\eta_s+1}$. $\mathcal{S}$ would like to recover the value of $\widehat{M}^{(i)} = D^{(i)} - \widehat{R}^{(i)}$ and find which slots in $M^{(i)}$ are zero. However, ciphertext $\widehat{R}^{(i)}$ looks pseudorandom to $\mathcal{S}$ since it is under $\mathcal{R}$'s key and the FHE scheme is IND-CPA secure. $\mathcal{S}$ knows the plaintext values encoded in $\overline{D}^{(i)}$ because it is under its own key; therefore, it can decrypt it $\overline{D}^{(i)} \mapsto D^{(i)}$. Nevertheless, from $\mathcal{S}$'s view, all entries in $D^{(i)}$ are independent and identically distributed random values coming from a uniform distribution in $\mathbb{Z}_t$. The fact that $D^{(i)} = M^{(i)} + R^{(i)}$ provides no statistical information (e.g. looking for smaller values) since each batching slot in $R^{(i)}$ is random in $\mathbb{Z}_t$; thus, values wrap around modulo t. Therefore, $\mathcal{S}$ learns nothing about Y. The extension of this proof for the recurrent setting is trivial since each new $(\overline{D}^{(i)}, \widehat{R}^{(i)})$ received by $\mathcal{S}$ contains i.i.d. random values from a uniform distribution, which gives no further information.

$\mathcal{R}$'s view: $\mathcal{R}$ receives $\overline{C}$ and $\widehat{E}^{(j)}$ $\forall j \in \mathbb{Z}_{\eta_r+1}$ from $\mathcal{S}$. Any value encoded and encrypted in ciphertext $\overline{C}^{(i)} \in \overline{C}$ appears random to $\mathcal{R}$ since they are encrypted under $\mathcal{S}$'s key and the FHE scheme is IND-CPA secure. Nevertheless, $\mathcal{R}$ learns the decryption of $\widehat{E}^{(j)} \mapsto E^{(j)}$ as it is under its own key. If and only if $y \in X$, there is a zero in a single packing slot of a single batching slot of a single polynomial $E^{(j)}$ $\forall j \in \mathbb{Z}_{\eta_r}$. Otherwise, if $y \notin X$, all values are non-zero (correctness). Each entry $e_a \in E^{(j)}$ is given by $e_a = n_b \cdot \rho_b$, where $n_b \in N^{(j)}$, $\rho_b \in P^{(j)}$, and a and b are slot indices. Due to the rotation operation in step (5d), it is not possible to infer b from a. Furthermore, the correlation of several indices $a_1, a_2, \ldots, a_n$ cannot recover any index b since each n_b is multiplied by a non-zero random value $\rho_b \in \mathbb{Z}_{t_i} \backslash \{0\}$ and each modulo t_i $\forall i \in \mathbb{Z} : 1 \leq i \leq k$ is a unique odd prime, meaning that the result of the multiplication wraps around each t_i and the parity of the multiplication is broken. Therefore, $\mathcal{R}$ can infer no information about $N^{(i)}$, and consequently $\mathcal{S}$'s set X, other than the intersection. Similarly to $\mathcal{S}$, the extension of $\mathcal{R}$'s proof to the recurrent setting is also trivial since each value in a new $\widehat{E}^{(j)}$ received by $\mathcal{R}$ is i.i.d. and uniformly distributed $\in \mathbb{Z}_{t_i} \backslash \{0\}$ for $y \notin X$, which means that $\mathcal{R}$ learns the intersection and nothing more. ■

With respect to malicious adversaries, our protocol encounters several challenges. Although $\mathcal{R}$'s set Y is protected against $\mathcal{S}$, a malicious $\mathcal{S}$ could potentially tamper with the intersection's accuracy. In other words, it has the ability to make the intersection falsely appear as any subset of Y without gaining knowledge of Y itself. Effectively thwarting such attempts poses significant challenges [8]. In the scenario of a malicious

Protocol 2 Full Protocol

Input: $\mathcal{S}$ inputs set X of size $|X|$. $\mathcal{R}$ inputs set Y of size $|Y|$. Both sets consist of bit strings of length σ_d. $|X|$, $|Y|$, and σ_d are public.

Output: $\mathcal{S}$ outputs $\perp$; $\mathcal{R}$ outputs $X \cap Y$.

1) **Setup**: In this step, parties select all parameters to be used in the protocol.
   a) **Security parameters**: Parties agree on computational and statistical security parameters (κ and λ, respectively).
   b) **Data compression**: From Eqs. 4 and 5 (§IV-A5), we have that $\sigma = \min(\sigma_d, \sigma_h)$. Therefore, if $\sigma_h < \sigma_d$, parties agree on a hash function $\mathcal{Z}(\cdot) : \mathbb{Z}_{2^{\sigma_d}} \mapsto \mathbb{Z}_{2^{\sigma_h}}$ for data compression. Otherwise, $\mathcal{Z}(x) = x$.
   c) **Cuckoo hashing and Permutation-based hashing**:
      i) Parties agree on the number of hash functions h and insertion depth limit d for Cuckoo hashing.
      ii) They select the number of Cuckoo hash tables k ($\equiv$ number of packing slots), and a function $g(\cdot)$ that maps a σ-bit entry x to Cuckoo hash table $T^{(g(x))}$ (§IV-B5).
      iii) Following §IV-A3 and §VI-A, $\mathcal{S}$ chooses the Cuckoo hash table size $|T^{(i)}|$ for a failure rate $\leq 2^{-\lambda}$.
      iv) By knowing $|T^{(i)}|$, one can calculate the left $\sigma_L = \lfloor \log_2 |T^{(i)}| \rfloor$ and right $\sigma_R = \sigma - \sigma_L$ sizes for Permutation-based hashing (§IV-A4).
      v) $\mathcal{S}$ selects the h different hash functions $H : \mathbb{Z}_{2^\sigma} \mapsto \mathbb{Z}_{|T^{(i)}|}$, where $H_j(z_L, z_R) = z_L \oplus f_j(z_R)$ and $f_j(z_R) : \mathbb{Z}_{2^{\sigma_R}} \mapsto \mathbb{Z}_{|T^{(i)}|}$ (§IV-A4).
   d) **Encryption parameters**:
      i) Parties agree on partitioning parameters η_s and η_r (§IV-C1) and plaintext modulus $t = \prod_{i=1}^{k} t_i : t_i \geq 2^{\sigma_R} + 2$ (§IV-B4 and §IV-B7).
      ii) Parties select encryption parameters $(n_s, q_s, t), (n_r, q_r, t) : n_s \geq n_r$ following Eq. 3 and §IV-B7. Encryption parameters are public.
      iii) Each party generates decryption and encryption keys, which are kept private.
2) **Cuckoo hashing**: For each entry $x \in X$, $\mathcal{S}$ performs $z = z_L | z_R = \mathcal{Z}(x) : |z_L| = \lfloor \log_2 |T^{(i)}| \rfloor$ and inserts z_R at $T^{(g(z))}_{H_j(z_L, z_R)} : 1 \leq j \leq h$. A dummy value is assigned to empty bins after inserting entries $\in X$ into tables $T^{(i)}$ (§IV-A2).
3) **Set encryption**: $\mathcal{S}$ encodes all Cuckoo hash tables and bins into polynomials following Eq. 8 (§IV-B5), encrypts them into ciphertexts under its own key, and sends the $\lceil \frac{|T^{(i)}|}{n} \rceil$ ciphertexts $\overline{C}$ to $\mathcal{R}$.
4) **Computing intersection**: For each $y \in Y$, $\mathcal{R}$:
   a) Computes $z = z_L | z_R = \mathcal{Z}(y) : |z_L| = \lfloor \log_2 |T^{(i)}| \rfloor$, and for each Cuckoo hash function $H_j(\cdot)\ \forall j \in \mathbb{Z} : 1 \leq j \leq h$, $\mathcal{R}$:
      i) Creates a polynomial $P \in Z_t[x]/(x^{n_s} + 1)$ s.t. $P_{u,v} = z_R$ for $(u, v) = (H_j(z_L, z_R) \bmod n_s,\ g(z) - 1)$ and a dummy value otherwise.
      ii) Homomorphically computes the difference $\overline{S}^{(j)} = \overline{C}^{\left(\lfloor \frac{H_j(z_L, z_R)}{n_s} \rfloor\right)} - P$.
   b) Performs $h - \eta_s - 1 : \eta_s \in \mathbb{Z}_h$ homomorphic multiplications between $\overline{S}^{(j)} : j \in \mathbb{Z}_{h+1} \backslash \{0\}$, resulting in $\eta_s + 1$ ciphertexts $\overline{M}^{(i)}\ \forall i \in \mathbb{Z}_{\eta_s + 1}$.
   c) Samples $\eta_s + 1$ tuples of random values $(r_1, ..., r_{n_s} \in \mathbb{Z}_t)$ from a uniform distribution, where each tuple is encoded as a polynomial $R^{(i)} \in \mathbb{Z}_t[x]/(x^{n_s} + 1)$.
   d) Computes $\overline{D}^{(i)} = \overline{M}^{(i)} + R^{(i)}\ \forall i \in \mathbb{Z}_{\eta_s + 1}$, modulus switches it, encrypts $R^{(i)} \mapsto \widehat{R}^{(i)}$ under its key, and sends the $(\overline{D}^{(i)}, \widehat{R}^{(i)})$ pairs to $\mathcal{S}$. Note that each $\widehat{R}^{(i)}$ will be several ciphertexts if $n_s > n_r$.
5) **Decryption**: For all pairs $(\overline{D}^{(i)}, \widehat{R}^{(i)})\ \forall i \in \mathbb{Z}_{\eta_s + 1}$, $\mathcal{S}$:
   a) Decrypts each $\overline{D}^{(i)} \mapsto D^{(i)}$, and homomorphically computes $\widehat{M}^{(i)} = \widehat{R}^{(i)} - D^{(i)}\ \forall i \in \mathbb{Z}_{\eta_s + 1}$.
   b) Performs $h - \eta_s - 1 - \eta_r : \eta_r \in \mathbb{Z}_{\eta_s + 1}$ homomorphic multiplications between $\widehat{M}^{(i)} : i \in \mathbb{Z}_{\eta_s + 1}$, resulting in $\eta_r + 1$ ciphertexts $\widehat{N}^{(j)}\ \forall j \in \mathbb{Z}_{\eta_r + 1}$.
   c) Samples $\eta_r + 1$ tuples of random values $(\rho_1, ..., \rho_{n_r} \in \mathbb{Z}_t \backslash \{\alpha \cdot t_i\}\ \forall \alpha, i \in \mathbb{Z} : 1 \leq i \leq k)$ such that no random value is a multiple of any packing modulus t_i, and encode each tuple as a polynomial $P^{(j)} \in \mathbb{Z}_t/(x^{n_r} + 1)$.
   d) Homomorphically computes $\widehat{E}^{(j)} = rotate(\widehat{N}^{(j)} \cdot P^{(j)}, \omega_j)\ \forall j \in \mathbb{Z}_{\eta_r + 1}$, where the function $rotate(\cdot)$, rotates the batching slots by a random number of positions $\omega_j \in \mathbb{Z}_{n_r}$, modulus switches and sends each $\widehat{E}^{(j)}$ to $\mathcal{R}$.
6) **Result**: $\mathcal{R}$ decrypts each $\widehat{E}^{(j)} \mapsto E^{(j)}$ and outputs $X \cap Y = \cup_{j=1}^{|Y|} \{y_j \iff 0 \in \cup_{j=0}^{\eta_r} E^{(j)}\}$.

$\mathcal{R}$, it should feasibly discern with non-negligible probability the hash function employed by $\mathcal{S}$ for an entry $y \in X$, if it pertains to the intersection. While this leakage might be weak, it surpasses mere knowledge of the intersection. Nevertheless, we believe that Protocol 2 offers adequate protection against malicious parties. It already safeguards $\mathcal{R}$'s set Y from a malicious $\mathcal{S}$. Additionally, it is highly unlikely for a malicious $\mathcal{R}$ to determine x' simply by knowing the hash function $H_i(\cdot)$ used for $x = y \in X \cap Y$. A more comprehensive analysis of malicious scenarios and potential countermeasures is reserved for future investigation.

VI. Experimental Results

A. Analysis of Hashing Failures

We must select parameters for Cuckoo hashing that ensure a successful hashing procedure with overwhelming probability for a given set X. There are three parameters we want to estimate: 1) the recursion depth limit d for entry insertion, which impacts the maximum load of the hash table; 2) the maximum load $\mathscr{L}$ for a given $(|T|, d)$, so that hashing has a negligible failure rate ($\leq 2^{-\lambda}$); and 3) the maximum load when using k hash tables. In our experiments, we aim for a set size of approximately $|X| \approx 2^{20}$, which is in line with typical sizes of commercial denylist databases [4].

1) Recursion Depth Limit: Before estimating the maximum load of a Cuckoo hash table, it is essential to establish a recursion depth limit for entry insertion since it is a factor influencing the hashing success rate, and consequently, the maximum load. To determine this limit, we conduct a screening experiment comprising 2^{10} runs for each configuration. Within each setup, we specify the number of hash functions h, table size $|T|$, and recursion depth limit d. For every run, we continuously insert $\sigma = 32$-bit unique random values into the Cuckoo hash table T until insertion fails, documenting the load $\mathscr{L}$ it attains before failure and the runtime. We report $h = \{2, 3, 4\}$, $|T| = 2^i \ \forall i \in \{16, 18, 20, 22, 24\}$, and $d = 2^j \ \forall j \in \mathbb{Z} : 0 \leq j \leq 16$. We constrain $d \leq 2^{16}$ as the hashing procedure with a large d becomes slow for large sets. We observe in Fig. 2 that a high load factor is achieved for $h = \{3, 4\}$ when $d \geq 2^9$. The maximum load for $h = 2$ caps around 50%, which is in accordance with previous work [45]. Furthermore, we notice that inserting large sets into large hash tables becomes computationally intensive when $d \geq 2^{11}$ without providing much increase in the load factor. This leaves us with two options $d = \{2^9, 2^{10}\}$, of which we select $d = 2^{10}$ as the slight increase in the hashing time is more than compensated by the reduction in the number of ciphertext encryptions since a higher load factor allows us to use a smaller hash table.

2) Single-Table Analysis: As discussed in §IV-A3, hashing failures can potentially leak a small amount of information from $\mathcal{S}$ to $\mathcal{R}$ beyond the set intersection. Although there are theoretical results on the failure rates of Cuckoo hashing [23], [18], determining the hidden constants remains challenging [8]. Consequently, related work [42], [8] relied on empirical methods to find the maximum load that a hash table can support for successful hashing with high probability. Our study adopts a similar empirical approach to examine the failure probability of Cuckoo hashing, with some important differences. PSSZ [42] analyzed the load factor for Cuckoo hashing with stash and $h = 2$, while CLR [8] investigated Cuckoo hashing without stash and $h = 3$ for small table sizes of $|T| = \{2^{13}, 2^{14}\}$. In contrast, we focus on estimating the load factor for Cuckoo hashing without stash with $h = \{3, 4\}$ hash functions and larger table sizes.

We start by fixing the Cuckoo hash table size $|T|$ and find how many items $|X|$ we can insert before failure. Following related work, we run each data point 2^{30} times. Related work [42], [8] has shown that λ is linearly related to the ratio $\frac{|T|}{|X|}$. Thus, we employ linear regression to estimate the load factor $\mathscr{L} = \frac{|X|}{|T|}$ for $\lambda = 40$. We notice that the load factor increases monotonically with $|T|$ for a fixed (h, d), implying that any load factor enabling successful hashing with overwhelming probability for a table of size $|T|$ also does so for any table of size $\geq |T|$. However, this increase becomes small for table sizes $|T| > 2^{16}$. Hence, we adopt $|T| = 2^{16}$ as the basis for our comprehensive analysis. Our observations indicate that for $h = 3$, the load factor is $\mathscr{L} \approx 0.73$, while for $h = 4$, the load factor increases to $\mathscr{L} \approx 0.87$, considering a failure rate $\leq 2^{-40}$. Related work [8] found $\mathscr{L} \approx 0.68$ for $(h, |T|) = (3, 2^{14})$. We attribute the increase in the load factor for $h = 3$ to larger parameters $(|T|, d)$. Table I provides the number of entries for several table sizes.

3) Multi-Table Analysis: From §IV-B5 and §IV-B6, we identify the need for a separate Cuckoo hash table for each packing modulus $t_i \ \forall i \in \mathbb{Z} : 1 \leq i \leq k$. To distribute entries $x \in X$ evenly across the k hash tables, we introduce a function $g(\cdot)$ that uniformly maps each entry x to one of the k hash tables. Thus, we insert a subset of unique entries $X^{(i)} \subset X$ into table $T^{(i)}$, where $X = \cup_{i=1}^{k} X^{(i)}$ and $X^{(i)} \cap X^{(j)} = \emptyset : i \neq j$. There may be variations in the number of entries across the different tables purely by chance. Consequently, we must account for that when defining the maximum load of a table to ensure successful Cuckoo hashing with high probability. Determining the maximum size of a subset is akin to the *balls into bins* problem. Hence, we want the probability of any subset exceeding a given threshold τ to be negligible:

$$P(\max_{i \in \{1,\dots,k\}} |X^{(i)}| > \tau) \leq 2^{-\lambda} \quad (9)$$

We can model this problem as a multinomial distribution where all subsets have the same probability of receiving an entry. Thus, the probability that the highest load exceeds a threshold τ is given by the sum of all probabilities from $\max_{i \in \{1,\dots,k\}} |X^{(i)}| = \tau + 1$ to $\max_{i \in \{1,\dots,k\}} |X^{(i)}| = |X|$. Eq. 10 formalizes it, which we employ to determine an appropriate threshold τ for the maximum size of a subset. Additionally, as discussed in §VI-A2, the load can be calculated as $\mathscr{L} = |X| \cdot |T|^{-1}$. Consequently, we need to adjust the new load to $\mathscr{L} = |X|^2 \cdot (\tau \cdot k^2 \cdot |T^{(i)}|)^{-1}$. Table I presents the Cuckoo hash table size for various settings.

$$P(\max_{i \in \{1,\dots,k\}} |X^{(i)}| > \tau) = \frac{1}{k^{|X|-1}} \sum_{i=\tau+1}^{|X|} \binom{|X|}{i} (k-1)^{|X|-i} \quad (10)$$

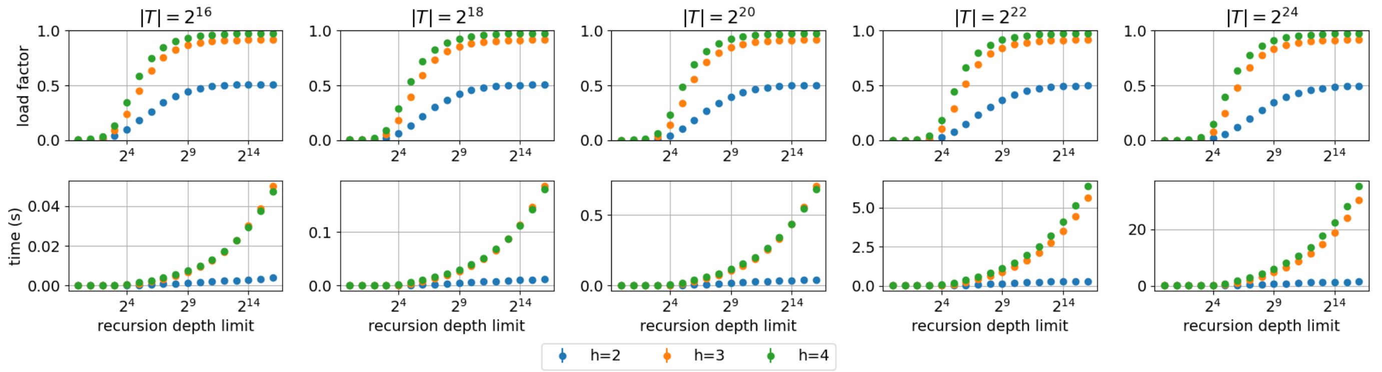

Fig. 2: Analysis of recursion depth limit for entry insertion into Cuckoo hash tables of different sizes.

$\|T\|$	h	$k=1$	$k=2$	$k=3$	$k=4$
2^{16}	3	48,094	46,576	45,951	45,476
	4	56,981	55,324	54,639	54,120
2^{17}	3	96,188	94,020	93,120	92,436
	4	113,963	111,600	110,619	109,864
2^{18}	3	192,376	189,292	188,001	187,012
	4	227,926	224,566	223,158	222,076
2^{19}	3	384,752	380,372	378,528	377,112
	4	455,852	451,078	449,070	447,524
2^{20}	3	769,414	763,288	760,668	758,644
	4	911,705	904,934	902,076	899,872
2^{21}	3	1,538,828	1,530,198	1,526,472	1,523,596
	4	1,823,411	1,813,814	1,809,756	1,806,620
2^{22}	3	3,077,657	3,065,540	3,060,255	3,056,164
	4	3,646,823	3,633,232	3,627,474	3,623,020
2^{23}	3	6,155,314	6,138,370	6,130,875	6,125,076
	4	7,293,646	7,274,404	7,266,246	7,259,928
2^{24}	3	12,310,629	12,287,074	12,276,450	12,268,236
	4	14,587,293	14,560,060	14,548,491	14,381,016

TABLE I: Maximum number of entries for several Cuckoo hash table sizes $|T| = \sum_{i=1}^{k} |T^{(i)}|$, number of hash functions h, and number of hash tables k for $\lambda = 40$.

B. Performance Evaluation

1) Experimental Setup: We evaluate our PSI protocol with two configurations, named *Fast Setup* and *Fast Intersection*, designed for fast setup and efficient set intersection, respectively. Our experiments target a $\mathcal{S}$'s set X of size $|X| \approx 2^{20}$ and $\mathcal{R}$'s set Y with sizes $|Y| = \{4, 16, 64\}$. In both configurations, we employ the following parameters: For encryption, we set the degree of the polynomial modulus to $n_s = n_r = 2^{12}$ and the ciphertext modulus to $\log_2 q = 109$ divided into four towers of $\{27, 27, 27, 28\}$ bits. This provides us with a sufficient noise budget for homomorphic multiplications and ensures at least 128 bits of computational security ($\kappa \geq 128$). Following related work [42], [8], we set the statistical security parameter to $\lambda = 40$. Based on the analysis regarding the maximum load of Cuckoo hashing (VI-A) we opt to use $h = 4$ hash functions. All experiments are implemented using Microsoft SEAL library [49] with the BFV encryption scheme [21] and conducted using 4 threads of an AMD Ryzen 7 5800h processor with 16 GB of memory, running Ubuntu 22.04 LTS.

With respect to the Fast Setup configuration, we use the following parameters: $k = 2$ Cuckoo hash tables, each with $|T^{(1)}| = |T^{(2)}| = 2^{19}$ bins, totaling $|T| = 2^{20}$ bins. This allows the insertion of $|X| = 904{,}934$ entries according to Table I for $k = 2$. We set the plaintext modulus to $t = 40961 \cdot 65537$, where $t_1 = 40961$ and $t_2 = 65537$. Moreover, we choose the partitioning parameters to be $\eta_s = \eta_r = 1$, as these are the smallest parameters that enable the protocol to run for the selected encryption parameters (n, q, t). This means that we double the number of ciphertexts transferred during the online phase, while reducing computation time. Note that it is still more efficient to send two ciphertexts with $(n, \log_2 q) = (2^{12}, 109)$ than increasing n to 2^{13} to provide more noise budget and compress all results into a single ciphertext, because for $n = 2^{13}$ we would require $\log_2 q = 218$ for a computational security parameter $\kappa \geq 128$, meaning we would quadruple the ciphertext size. Regarding the Fast Intersection configuration, we use the following parameters: $k = 1$ Cuckoo hash table with $|T| = 2^{20}$ bins, allowing $|X| = 911{,}705$ entries to be inserted following the results presented in Table I for $k = 1$. We set the plaintext modulus to $t = 40961$ and the partitioning parameters to $\eta_s = \eta_r = 0$, as with the selected t we have enough noise budget for the entire multiplicative depth required by our protocol.

2) Discussion: Tables II and III display the computation time (in seconds) and communication cost (in MiB) for $\mathcal{S}$ and $\mathcal{R}$ when executing the PSI protocol described in Protocol 2. These computations use the parameters discussed in §VI-B1 for fast setup and fast intersection. In addition to presenting our results, we include the performance of CLR [8], currently the fastest PSI protocol in related work for unbalanced set sizes, KKRT [35], a well-known PSI protocol for large entries in fast networks, and PSSZ [42], the most efficient PSI protocol to-date in terms of complexity. $\mathcal{S}$'s computation time is divided into pre-computation and online phases, while $\mathcal{R}$ only has an online phase. The size of $\mathcal{S}$'s set X is determined by $\mathscr{L} \cdot 2^{20}$, as outlined in Table I. Here the load factor takes values $\mathscr{L} = \{\approx 0.86, \approx 0.87, 1.00, 1.00, 1.00\}$ for each configuration, respectively. It is worth noting that in related work, the load factor is $\mathscr{L} = 1.00$. This is because, unlike our approach, CLR and PSSZ employ Cuckoo hashing not in $\mathcal{S}$'s set, but in $\mathcal{R}$'s set instead. The size of $\mathcal{R}$'s set Y is set to $|Y| = \{4, 16, 64\}$. For each $|Y|$, we evaluate $m = \{1, 4, 16, 64\}$ set intersections, meaning we intersect m sets $Y^{(1)}, \ldots, Y^{(m)}$ with set X. We present amortized results based on the number of set intersections m, as the latency per set intersection is more relevant than the sum of latencies of several set intersections.

$\|X\|$	$\|Y\|$	m	Fast Setup			Fast Intersection			CLR [8]			KKRT [35]		PSSZ [42]	
			$\mathcal{S}$ pre	$\mathcal{S}$ online	$\mathcal{R}$ online	$\mathcal{S}$ pre	$\mathcal{S}$ online	$\mathcal{R}$ online	$\mathcal{S}$ pre	$\mathcal{S}$ online	$\mathcal{R}$ online	$\mathcal{S}$ pre	$\mathcal{S}$+$\mathcal{R}$ online	$\mathcal{S}$ pre	$\mathcal{S}$+$\mathcal{R}$ online
$\mathscr{L} \cdot 2^{20}$	4	1	0.09	0.01	0.01	0.30	0.01	0.02	1.21	1.42	0.79	NA	1.62	NA	0.86
		4	0.02	0.01	0.01	0.07	0.01	0.02	0.30	1.42	0.79	NA	1.62	NA	0.86
		16	<0.01	0.01	0.01	0.02	0.01	0.02	0.08	1.42	0.79	NA	1.62	NA	0.86
		64	<0.01	0.01	0.01	<0.01	0.01	0.02	0.02	1.42	0.79	NA	1.62	NA	0.86
	16	1	0.09	0.05	0.06	0.30	0.03	0.07	1.21	1.42	0.79	NA	1.62	NA	0.86
		4	0.02	0.05	0.06	0.07	0.03	0.07	0.30	1.42	0.79	NA	1.62	NA	0.86
		16	<0.01	0.05	0.06	0.02	0.03	0.07	0.08	1.42	0.79	NA	1.62	NA	0.86
		64	<0.01	0.05	0.06	<0.01	0.03	0.07	0.02	1.42	0.79	NA	1.62	NA	0.86
	64	1	0.09	0.20	0.23	0.30	0.10	0.27	1.21	1.42	0.79	NA	1.62	NA	0.71
		4	0.02	0.20	0.23	0.07	0.10	0.27	0.30	1.42	0.79	NA	1.62	NA	0.71
		16	<0.01	0.20	0.23	0.02	0.10	0.27	0.08	1.42	0.79	NA	1.62	NA	0.71
		64	<0.01	0.20	0.23	<0.01	0.10	0.27	0.02	1.42	0.79	NA	1.62	NA	0.71

TABLE II: Computation time (s) for two configurations of our PSI protocol, *Fast Setup* and *Fast Intersection*, and related work, considering a target $\mathcal{S}$'s set size $|X|$, several sizes of $\mathcal{R}$'s set $|Y|$, and various numbers of recurrent set intersections m. KKRT and PSSZ do not have one-time costs. Moreover, their implementations do not report computation time per party, only total time.

$\|X\|$	$\|Y\|$	m	Fast Setup		Fast Intersection		CLR [8]		KKRT [35]		PSSZ [42]	
			$\mathcal{S}\rightarrow\mathcal{R}$	$\mathcal{R}\rightarrow\mathcal{S}$	$\mathcal{S}\rightarrow\mathcal{R}$	$\mathcal{R}\rightarrow\mathcal{S}$	$\mathcal{S}\rightarrow\mathcal{R}$	$\mathcal{R}\rightarrow\mathcal{S}$	$\mathcal{S}\rightarrow\mathcal{R}$	$\mathcal{R}\rightarrow\mathcal{S}$	$\mathcal{S}\rightarrow\mathcal{R}$	$\mathcal{R}\rightarrow\mathcal{S}$
$\mathscr{L} \cdot 2^{20}$	4	1	13.16	1.32	25.39	0.53	2.20	3.40	57.15	0.02	24.02	0.03
		4	3.69	1.32	6.45	0.53	2.20	3.40	57.15	0.02	24.02	0.03
		16	1.32	1.32	1.71	0.53	2.20	3.40	57.15	0.02	24.02	0.03
		64	0.73	1.32	0.53	0.53	2.20	3.40	57.15	0.02	24.02	0.03
	16	1	14.74	5.27	25.79	2.11	2.20	3.40	57.15	0.02	24.02	0.03
		4	5.27	5.27	6.85	2.11	2.20	3.40	57.15	0.02	24.02	0.03
		16	2.90	5.27	2.11	2.11	2.20	3.40	57.15	0.02	24.02	0.03
		64	2.31	5.27	0.93	2.11	2.20	3.40	57.15	0.02	24.02	0.03
	64	1	21.09	21.09	27.40	8.45	2.20	3.40	57.15	0.02	27.02	0.09
		4	11.62	21.09	8.46	8.45	2.20	3.40	57.15	0.02	27.02	0.09
		16	9.25	21.09	3.72	8.45	2.20	3.40	57.15	0.02	27.02	0.09
		64	8.50	21.09	2.54	8.45	2.20	3.40	57.15	0.02	27.02	0.09

$\|X\|$	$\|Y\|$	m	Fast Setup		Fast Intersection		CLR [8]		KKRT [35]		PSSZ [42]	
			10 Gbps	100 Mbps	10 Gbps	100 Mbps	10 Gbps	100 Mbps	10 Gbps	100 Mbps	10 Gbps	100 Mbps
$\mathscr{L} \cdot 2^{20}$	4	1	0.01	1.32	0.02	2.23	<0.01	0.60	0.05	4.84	0.02	2.08
		4	<0.01	0.44	<0.01	0.60	<0.01	0.60	0.05	4.84	0.02	2.08
		16	<0.01	0.22	<0.01	0.19	<0.01	0.60	0.05	4.84	0.02	2.08
		64	<0.01	0.17	<0.01	0.09	<0.01	0.60	0.05	4.84	0.02	2.08
	16	1	0.02	1.76	0.02	2.39	<0.01	0.60	0.05	4.84	0.02	2.08
		4	<0.01	0.88	<0.01	0.76	<0.01	0.60	0.05	4.84	0.02	2.08
		16	<0.01	0.66	<0.01	0.35	<0.01	0.60	0.05	4.84	0.02	2.08
		64	<0.01	0.61	<0.01	0.25	<0.01	0.60	0.05	4.84	0.02	2.08
	64	1	0.03	3.53	0.03	3.03	<0.01	0.60	0.05	4.84	0.02	2.34
		4	0.03	2.66	0.01	1.39	<0.01	0.60	0.05	4.84	0.02	2.34
		16	0.02	2.44	0.01	0.98	<0.01	0.60	0.05	4.84	0.02	2.34
		64	0.02	2.38	<0.01	0.88	<0.01	0.60	0.05	4.84	0.02	2.34

TABLE III: Communication cost (MiB) for two configurations of our PSI protocol, *Fast Setup* and *Fast Intersection*, and related work, considering a target $\mathcal{S}$'s set size $|X|$ and several sizes of $\mathcal{R}$'s set $|Y|$, where we vary the number of recurrent set intersections m. We assume a Round-Trip Time (RTT) delay of 0.2ms and 80ms for 10 Gbps and 100 Mbps network speeds, respectively.

Table IV presents the total time (in seconds) for performing the one-time phase and the recurrent phase at two different network speeds (10 Gbps and 100 Mbps). By comparing our protocol's two settings, we observe that the Fast Setup incurs a one-time communication cost that is half of what the Fast Intersection requires. Additionally, the computation time is reduced as the Fast Setup employs two smaller hash tables instead of a single larger one. As depicted in Fig. 2, insertion time grows superlinearly with table size. Consequently, for fast networks (10 Gbps), Fast Setup incurs a one-time cost of one-third that of Fast Intersection, and for slower networks (100 Mbps), it is half. However, the scenario changes when considering recurrent costs. While the computation time remains comparable, the communication cost of Fast Setup is nearly triple that of Fast Intersection. On fast networks, both configurations are comparable, but on slower networks, the Fast Intersection configuration is $37-114\%$ faster. Moreover, for small set sizes $|Y|$ and few recurrences m, Fast Setup is faster, while for larger $|Y|$ and more recurrences m, Fast Intersection is faster. Ultimately, the choice between these two configurations depends on the expected set size $|Y|$, number of recurrent set intersections per setup, and network speed.

			$\|X\| = \mathscr{L} \cdot 2^{20}$
One-time	Computation	Fast Setup	0.09
		Fast Intersection	0.30
		CLR [8]	1.21
		KKRT [35]	NA
		PSSZ [42]	NA
	Communication	Fast Setup	12.63
		Fast Intersection	25.25
		CLR [8]	0.00
		KKRT [35]	NA
		PSSZ [42]	NA
	Total time (10 Gbps)	Fast Setup	**0.10**
		Fast Intersection	0.32
		CLR [8]	1.21
		KKRT [35]	NA
		PSSZ [42]	NA
	Total time (100 Mbps)	Fast Setup	1.26
		Fast Intersection	2.48
		CLR [8]	**1.21**
		KKRT [35]	NA
		PSSZ [42]	NA

			$\|Y\|$ 4	16	64
Recurrent	Computation	Fast Setup	0.02	0.11	0.43
		Fast Intersection	0.03	0.10	0.37
		CLR [8]	2.21	2.21	2.21
		KKRT[35]	1.62	1.62	1.62
		PSSZ [42]	0.86	0.86	0.71
	Communication	Fast Setup	1.85	7.39	29.55
		Fast Intersection	0.66	2.65	10.59
		CLR [8]	5.60	5.60	5.60
		KKRT [35]	57.17	57.17	57.17
		PSSZ [42]	24.05	24.05	27.11
	Total time (10 Gbps)	Fast Setup	**0.02**	0.12	0.45
		Fast Intersection	0.03	**0.10**	**0.38**
		CLR [8]	2.21	2.21	2.21
		KKRT [35]	1.67	1.67	1.67
		PSSZ [42]	0.88	0.88	0.73
	Total time (100 Mbps)	Fast Setup	0.33	0.86	2.95
		Fast Intersection	**0.24**	**0.47**	**1.38**
		CLR [8]	2.81	2.81	2.81
		KKRT [35]	6.46	6.46	6.46
		PSSZ [42]	2.94	2.94	3.05

TABLE IV: Total time (in seconds) for two configurations of our PSI protocol, *Fast Setup* and *Fast Intersection*, and related work, considering a target $\mathcal{S}$'s set size $|X| = \mathscr{L} \cdot 2^{20}$ and several sizes of $\mathcal{R}$'s set $|Y|$, where we evaluate the one-time costs and recurrent costs of performing set intersections. KKRT and PSSZ do not have one-time costs.

3) Comparison to Related Work: Compared to related work, our proposal demonstrates a substantial improvement in set intersection latency. In terms of computation, our protocol outperforms CLR during the setup phase by a factor of $4 - 12\times$. Notably, KKRT and PSSZ lack a setup phase.[3] During the recurrent phase, our protocol provides a speed-up of approximately two orders of magnitude against all related work PSI protocols. With respect to communication, the proposed protocol exhibits varying costs depending on the size $|Y|$ of $\mathcal{R}$'s set Y, and the number of recurrent set intersections m per setup. For larger $|Y|$ and smaller m, communication costs are higher, while smaller $|Y|$ and larger m prove more efficient. Compared to the state of the art, our protocol reduces the communication cost during the recurrent phase by as much as one order of magnitude compared to CLR, and reaching two orders of magnitude in relation to KKRT and PSSZ, when the set size $|Y|$ is small. These improvements are particularly suitable for the URL denylisting application described in §I-A, as we expect a small number of URLs in incoming emails. In terms of total time, our protocol's runtime during setup is comparable to CLR on slower networks (100 Mbps), while being around one order of magnitude faster on fast networks (10 Gbps). With regards to the recurrent phase, which is clearly more time-sensitive for the target scenario, our protocol is *approximately one order of magnitude faster than related work on slower networks* (100 Mbps) when $\mathcal{R}$'s set Y is small ($|Y| \leq 16$), and *up to two orders of magnitude faster than existing work on fast networks* (10 Gbps).

[3]KKRT splits its execution into offline and online phases. The offline phase, which computes in 0.18s, is used to obtain an OT extension matrix unrelated to the sets' entries using IKNP OT extensions [33]. However, a new extension matrix is needed for each subsequent set intersection. We could apply the same offline label to generating and exchanging random values $R^{(i)}$ and $P^{(j)}$, and ω in our PSI protocol. Regardless, these are recurring costs.

4) Scalability: The proposed PSI protocol is optimized for $\mathcal{R}$ with a small set ($|Y| \ll |X|$), making it inefficient when $|Y|$ is large. The size $|X|$ of $\mathcal{S}$'s set X impacts the setup phase, as a larger set requires more data to be hashed, encrypted, and transmitted. However, $|X|$ has no effect on the recurring costs, indicating that the values reported in Table IV for the recurrent phase remain constant for different sizes of X. Our protocol performs well in comparison to related work, especially for large set sizes $|X|$, as demonstrated by additional experiments presented in Appendix A.

VII. Conclusions

This work presents an optimized Private Set Intersection (PSI) protocol tailored for the recurrent setting with unbalanced sets, where the sender possesses the larger set and the receiver holds the smaller set. We have proposed several optimizations, leveraging fully homomorphic encryption and hashing techniques, and introduced novel encoding techniques to enhance a basic PSI protocol. Through extensive analysis of failure probabilities, we have ensured that our protocol performs reliably with overwhelming probability. Using the Microsoft SEAL library and the BFV encryption scheme, we implemented our protocol and demonstrated its real-time performance with two different instantiations. Our results show a significant reduction in set intersection times compared to existing approaches. Specifically, for 100 Mbps networks, our protocol achieves a one-order-of-magnitude reduction, while for 10 Gbps networks, the reduction is by two orders of magnitude. Overall, our optimized PSI protocol offers efficient real-time performance, making significant advancements in the field of private set intersection protocols.

Resources

A C++ implementation of our PSI protocol is available at https://github.com/momalab/psi-ndss2025/.

REFERENCES

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

[2] F. Armknecht, C. Boyd, C. Carr, K. Gjøsteen, A. Jäschke, C. A. Reuter, and M. Strand, "A guide to fully homomorphic encryption," *Cryptology ePrint Archive*, 2015.

[3] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, ser. STOC '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 503–513. [Online]. Available: https://doi.org/10.1145/100216.100287

[4] S. Bell and P. Komisarczuk, "An analysis of phishing blacklists: Google safe browsing, openphish, and phishtank," in *Proceedings of the Australasian Computer Science Week Multiconference*, 2020, pp. 1–11.

[5] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[6] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[7] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled psi from fully homomorphic encryption with malicious security," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1223–1237. [Online]. Available: https://doi.org/10.1145/3243734.3243836

[8] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1243–1255. [Online]. Available: https://doi.org/10.1145/3133956.3134061

[9] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *ASIACRYPT*, 2017.

[10] E. Chielle, H. Gamil, and M. Maniatakos, "Real-time private membership test using homomorphic encryption," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1282–1287.

[11] E. Chielle, O. Mazonka, H. Gamil, and M. Maniatakos, "Accelerating fully homomorphic encryption by bridging modular and bit-level arithmetic," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[12] ——, "Coupling bit and modular arithmetic for efficient general-purpose fully homomorphic encryption," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 4, pp. 1–28, 2024.

[13] E. Chielle, O. Mazonka, H. Gamil, N. G. Tsoutsos, and M. Maniatakos, "E3: A framework for compiling c++ programs with encrypted operands," Cryptology ePrint Archive, Paper 2018/1013, 2018. [Online]. Available: https://eprint.iacr.org/2018/1013

[14] E. Chielle, O. Mazonka, and M. Maniatakos, "Optimizing ciphertext management for faster fully homomorphic encryption computation," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[15] K. Cong, R. C. Moreno, M. B. da Gama, W. Dai, I. Iliashenko, K. Laine, and M. Rosenberg, "Labeled psi from homomorphic encryption with reduced computation and communication," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1135–1150. [Online]. Available: https://doi.org/10.1145/3460120.3484760

[16] A. C. Davi Resende and D. de Freitas Aranha, "Faster unbalanced private set intersection in the semi-honest setting," *Journal of Cryptographic Engineering*, vol. 11, no. 1, pp. 21–38, 2021.

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

[18] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink, "Tight thresholds for cuckoo hashing via xorsat," in *Automata, Languages and Programming: 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I 37*. Springer, 2010, pp. 213–225.

[19] C. Dong, L. Chen, and Z. Wen, "When private set intersection meets big data: an efficient and scalable protocol," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 789–800.

[20] D. S. Dummit and R. M. Foote, *Abstract algebra*. Wiley Hoboken, 2004, vol. 3.

[21] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.

[22] M. J. Freedman, K. Nissim, and B. Pinkas, "Efficient private matching and set intersection," in *Advances in Cryptology - EUROCRYPT 2004*, C. Cachin and J. L. Camenisch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–19.

[23] A. Frieze, P. Melsted, and M. Mitzenmacher, "An analysis of random-walk cuckoo hashing," in *International Workshop on Approximation Algorithms for Combinatorial Optimization*. Springer, 2009, pp. 490–503.

[24] C. Gentry, *A fully homomorphic encryption scheme*. Stanford University, 2009.

[25] ——, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[26] C. Gentry, S. Halevi, and N. P. Smart, "Better bootstrapping in fully homomorphic encryption," in *International Workshop on Public Key Cryptography*. Springer, 2012, pp. 1–16.

[27] ——, "Homomorphic evaluation of the aes circuit," in *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 850–867.

[28] O. Goldreich, "Cryptography and cryptographic protocols," *Distributed Computing*, vol. 16, pp. 177–199, 2003.

[29] S. Halevi, Y. Polyakov, and V. Shoup, "An improved rns variant of the bfv homomorphic encryption scheme," in *Topics in Cryptology – CT-RSA 2019*, M. Matsui, Ed. Cham: Springer International Publishing, 2019, pp. 83–105.

[30] M. Hao, W. Liu, L. Peng, H. Li, C. Zhang, H. Chen, and T. Zhang, "Unbalanced {Circuit-PSI} from oblivious {Key-Value} retrieval," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 6435–6451.

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

[32] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure {Two-Party} computation using garbled circuits," in *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[33] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently," in *Advances in Cryptology - CRYPTO 2003*, D. Boneh, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 145–161.

[34] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1543–1561, 2010.

[35] V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu, "Efficient batched oblivious prf with applications to private set intersection," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 818–829.

[36] V. Kolesnikov, N. Matania, B. Pinkas, M. Rosulek, and N. Trieu, "Practical multi-party private set intersection from symmetric-key techniques," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1257–1272.

[37] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *J. ACM*, vol. 60, no. 6, 2013.

[38] F. H. Mathis, "A generalized birthday problem," *SIAM review*, vol. 33, no. 2, pp. 265–270, 1991.

[39] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *European Symposium on Algorithms*. Springer, 2001, pp. 121–133.

[40] ——, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[41] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai, “Spot-light: Lightweight private set intersection from sparse ot extension,” in *Advances in Cryptology – CRYPTO 2019*, A. Boldyreva and D. Micciancio, Eds. Cham: Springer International Publishing, 2019, pp. 401–431.

[42] B. Pinkas, T. Schneider, G. Segev, and M. Zohner, “Phasing: Private set intersection using permutation-based hashing,” in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 515–530. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/pinkas

[43] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai, “Efficient circuit-based psi with linear communication,” in *Advances in Cryptology – EUROCRYPT 2019*, Y. Ishai and V. Rijmen, Eds. Cham: Springer International Publishing, 2019, pp. 122–153.

[44] B. Pinkas, T. Schneider, C. Weinert, and U. Wieder, “Efficient circuit-based psi via cuckoo hashing,” in *Advances in Cryptology – EUROCRYPT 2018*, J. B. Nielsen and V. Rijmen, Eds. Cham: Springer International Publishing, 2018, pp. 125–157.

[45] B. Pinkas, T. Schneider, and M. Zohner, “Faster private set intersection based on OT extension,” in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 797–812. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/pinkas

[46] ——, “Scalable private set intersection based on ot extension,” *ACM Trans. Priv. Secur.*, vol. 21, no. 2, jan 2018. [Online]. Available: https://doi.org/10.1145/3154794

[47] P. Prakash, M. Kumar, R. R. Kompella, and M. Gupta, “Phishnet: Predictive blacklisting to detect phishing attacks,” in *2010 Proceedings IEEE INFOCOM*, 2010, pp. 1–5.

[48] M. Raab and A. Steger, ““balls into bins” — a simple and tight analysis,” in *Randomization and Approximation Techniques in Computer Science*, M. Luby, J. D. P. Rolim, and M. Serna, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 159–170.

[49] “Microsoft SEAL (release 4.1),” https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[50] I. Shaik, N. Emmadi, H. Tupsamudre, H. Narumanchi, and R. M. A. Bhattachar, “Privacy preserving machine learning for malicious url detection,” in *Database and Expert Systems Applications-DEXA 2021 Workshops: BIOKDD, IWCFS, MLKgraphs, AI-CARES, ProTime, AISys 2021, Virtual Event, September 27–30, 2021, Proceedings 32*. Springer, 2021, pp. 31–41.

[51] N. P. Smart and F. Vercauteren, “Fully homomorphic SIMD operations,” *Des. Codes Cryptogr.*, vol. 71, no. 1, pp. 57–81, 2014. [Online]. Available: https://doi.org/10.1007/s10623-012-9720-4

[52] Y. Son and J. Jeong, “Psi with computation or circuit-psi for unbalanced sets from homomorphic encryption,” in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023, pp. 342–356.

[53] A. C.-C. Yao, “How to generate and exchange secrets,” in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, 1986, pp. 162–167.

[54] M. Yung, “From mental poker to core business: Why and how to deploy secure computation protocols?” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–2.

[55] Z. Zhou, T. Song, and Y. Jia, “A high-performance url lookup engine for url filtering systems,” in *2010 IEEE International Conference on Communications*, 2010, pp. 1–5.

APPENDIX

ADDITIONAL EXPERIMENTAL RESULTS

			$\|X\| = \mathscr{L} \cdot 2^{16}$
One-time	Computation	Fast Setup	<0.01
		Fast Intersection	0.01
		CLR [8]	0.19
		KKRT [35]	NA
		PSSZ [42]	NA
	Communication	Fast Setup	0.79
		Fast Intersection	1.58
		CLR [8]	0.00
		KKRT [35]	NA
		PSSZ [42]	NA
	Total time (10 Gbps)	Fast Setup	<0.01
		Fast Intersection	0.01
		CLR [8]	0.19
		KKRT [35]	NA
		PSSZ [42]	NA
	Total time (100 Mbps)	Fast Setup	0.15
		Fast Intersection	0.22
		CLR [8]	0.19
		KKRT [35]	NA
		PSSZ [42]	NA

			$\|Y\|$		
			4	16	64
Recurrent	Computation	Fast Setup	0.02	0.11	0.43
		Fast Intersection	0.03	0.10	0.37
		CLR [8]	0.88	0.88	0.88
		KKRT[35]	0.09	0.09	0.09
		PSSZ [42]	0.24	0.24	0.23
	Communication	Fast Setup	1.85	7.39	29.55
		Fast Intersection	0.66	2.65	10.59
		CLR [8]	3.50	3.50	3.50
		KKRT [35]	3.46	3.46	3.46
		PSSZ [42]	1.55	1.55	1.61
	Total time (10 Gbps)	Fast Setup	0.02	0.12	0.45
		Fast Intersection	0.03	0.10	0.38
		CLR [8]	0.88	0.88	0.88
		KKRT [35]	0.09	0.09	0.09
		PSSZ [42]	0.24	0.24	0.23
	Total time (100 Mbps)	Fast Setup	0.33	0.86	2.95
		Fast Intersection	0.24	0.47	1.38
		CLR [8]	1.25	1.25	1.25
		KKRT [35]	0.46	0.46	0.46
		PSSZ [42]	0.45	0.45	0.45

TABLE V: Total time (in seconds) for two configurations of our PSI protocol, *Fast Setup* and *Fast Intersection*, and related work, considering a $\mathcal{S}$'s set size $|X| = \mathscr{L} \cdot 2^{16}$ and several sizes of $\mathcal{R}$'s set $|Y|$, where we evaluate the one-time costs and recurrent costs of performing set intersections. KKRT and PSSZ do not have one-time costs.

			$\|X\| = \mathscr{L} \cdot 2^{24}$
One-time	Computation	Fast Setup	1.79
		Fast Intersection	7.74
		CLR [8]	21.54
		KKRT [35]	NA
		PSSZ [42]	NA
	Communication	Fast Setup	202.04
		Fast Intersection	404.08
		CLR [8]	0.00
		KKRT [35]	NA
		PSSZ [42]	NA
	Total time (10 Gbps)	Fast Setup	1.96
		Fast Intersection	8.08
		CLR [8]	21.54
		KKRT [35]	NA
		PSSZ [42]	NA
	Total time (100 Mbps)	Fast Setup	18.68
		Fast Intersection	41.44
		CLR [8]	21.54
		KKRT [35]	NA
		PSSZ [42]	NA

			$\|Y\|$		
			4	16	64
Recurrent	Computation	Fast Setup	0.02	0.11	0.43
		Fast Intersection	0.03	0.10	0.37
		CLR [8]	23.22	23.22	23.22
		KKRT[35]	30.42	40.42	30.42
		PSSZ [42]	11.11	11.11	8.54
	Communication	Fast Setup	1.85	7.39	29.55
		Fast Intersection	0.66	2.65	10.59
		CLR [8]	11.00	11.00	11.00
		KKRT [35]	946.75	946.75	946.75
		PSSZ [42]	432.05	432.05	432.11
	Total time (10 Gbps)	Fast Setup	0.02	0.12	0.45
		Fast Intersection	0.03	0.10	0.38
		CLR [8]	23.22	23.22	23.22
		KKRT [35]	31.21	31.21	31.21
		PSSZ [42]	11.47	11.47	8.90
	Total time (100 Mbps)	Fast Setup	0.33	0.86	2.95
		Fast Intersection	0.24	0.47	1.38
		CLR [8]	23.59	23.59	23.59
		KKRT [35]	109.27	109.27	109.27
		PSSZ [42]	47.14	47.14	44.57

TABLE VI: Total time (in seconds) for two configurations of our PSI protocol, *Fast Setup* and *Fast Intersection*, and related work, considering a $\mathcal{S}$'s set size $|X| = \mathscr{L} \cdot 2^{24}$ and several sizes of $\mathcal{R}$'s set $|Y|$, where we evaluate the one-time costs and recurrent costs of performing set intersections. KKRT and PSSZ do not have one-time costs.

Artifact Appendix

A. Description & Requirements

1) How to access: The archival version of the artifact is hosted on Zenodo at https://doi.org/10.5281/zenodo.14249467, and the maintained version is available on GitHub at https://github.com/momalab/psi-ndss2025.

2) Hardware dependencies: A commodity computer.

3) Software dependencies: Linux operating system, Git, CMake (version 3.13 or higher), GNU C++ compiler (version 7.3 or higher), GMP library, Python 3, TeX Live (texlive, texlive-latex-base, texlive-latex-extra).

4) Benchmarks: None.

B. Artifact Installation & Configuration

[5 human-minutes + 10 compute-minutes]

To set up the artifact, follow these steps:

1) Install all necessary dependencies;
2) Download and unzip the archived version from Zenodo, or clone the repository from GitHub;
3) Install the Microsoft SEAL v4.1 library locally by executing the bash script `install_seal.sh` located in the `3p` directory;

C. Major Claims

- (C1): Our private set intersection protocol achieves computation time, communication cost, and round-trip total set intersection time in accordance with the results present in Tables II, III, IV, V, and VI. For example, the total recurrent time for the *Fast Intersection* mode with $|X| = \mathscr{L} \cdot 2^{20}$ and $|Y| = 4$ is 0.02 seconds on a 10 Gbps network and 0.24 seconds on a 100 Mbps network, as shown in Table IV.

D. Evaluation

[12 human-minutes + 7 compute-minutes]

Here, we show how to reproduce the results presented in Tables II, III, IV, V, and VI for our protocol. We provide a script that runs all experiments and generates a PDF file with the five tables from the paper. Navigate to the `src/main` directory and execute `python3 reproduce.py`.

The `reproduce.py` script:

1) Compiles the protocol;
2) Runs the protocol with a set of parameters;
3) Collects results;
4) Cleans temporary files;
5) Repeats steps 2-4 with a new set of parameters;

The script adjusts the four following parameters:

- Mode: Defines if the protocol will run using the *Fast Setup* (0) or *Fast Intersection* (1) configuration;
- Size of set X: $|X| = \mathscr{L} \cdot \{2^{16}, 2^{20}, 2^{24}\}$;
- Size of set Y: $|Y| = \{4, 16, 64\}$;
- Number of recurrences $m = \{1, 4, 16, 64\}$.

After running the script, there will be a file named `artifact-evaluation.pdf` in the `src/main` directory. This file includes five tables, Tables II through VI, with results matching those in the paper. The communication results should closely align with the paper, while computation results may vary slightly due to differences in processing capacity.

E. Notes

We do not include the reproduction of Fig. 2 and Table I in the artifact for two reasons:

1) These results focus on parameter selection and have minimal impact on protocol execution time;
2) Generating these results is extremely time-consuming, taking a few months to complete.

Given their limited impact and excessive computational cost, we decided it was impractical to include them. Nonetheless, the experimental methodology is described in the paper and aligns with related work. To verify the limited impact of parameter changes on execution time, one can modify the number of hash functions (`src/main/protocol.cpp`, line 72) from 4 to 3, and adjust the load factor (line 76) from `load_factor = mode ? 0.86 : 0.87` to `load_factor = mode ? 0.72 : 0.73`, then rerun the `reproduce.py` script.

This artifact provides steps to replicate our protocol results. For related work results, refer to their implementations.