Differentially Private Two-Party Set Operations

Bailey Kacsmar*, Basit Khurram*, Nils Lukas*, Alexander Norton*, Masoumeh Shafieinejad*,
Zhiwei Shang*, Yaser Baseri*, Maryam Sepehri†, Simon Oya*, Florian Kerschbaum*
*Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
*{bkacsmar, mbkhurra, nlukas, ar2norto, masoumeh, z6shang, ybaseri, simon.oya, fkerschb}@uwaterloo.ca
†Dipartimento di Informatica, Universita degli studi di Milano, Milan, Italy
†maryam.sepehri@unimi.it

***Abstract*—Private set intersection (PSI) allows two parties to compute the intersection of their data without revealing the data they possess that is outside of the intersection. However, in many cases of joint data analysis, the intersection is also sensitive. We define differentially private set intersection and we propose new protocols using (leveled) homomorphic encryption where the result is differentially private. Our circuit-based approach has an adaptability that allows us to achieve differential privacy, as well as to compute predicates over the intersection such as cardinality. Furthermore, our protocol produces differentially private output for set intersection and set intersection cardinality that is optimal in terms of communication and computation complexity. For a client set of size m and a server set of size n, where m is smaller than n, our communication complexity is $O(m)$ while previous circuit-based protocols only achieve $O(n+m)$ communication complexity. In addition to our asymptotic optimizations which include new analysis for using nested cuckoo hashing for PSI, we demonstrate the practicality of our protocol through an implementation that shows the feasibility of computing the differentially private intersection for large data sets containing millions of elements.**

***Index Terms*—differential privacy, homomorphic encryption, private set intersection**

1. Introduction

Private set intersection (PSI) [1]–[3] can protect sensitive data when the intersection is non-sensitive. It is, for example, used by Google and a partner to compute ad conversions [4]. In this work, we present protocols designed for cases of joint data analysis where the intersection is also sensitive.

Consider the following case where Google and Mastercard exchanged credit card transaction data without PSI [5]. Google paid Mastercard to access individuals' credit card transactions that they could match to those users' presented ads. One can argue that the fact that a credit card purchase was made is personally identifiable information. User-specific credit card purchases have been used to de-identify anonymized credit card statements [6]. Hence, it is necessary to protect, not only the users outside of the intersection, but those inside it as well. To address the protection needed for users inside the intersection of two data sets, we propose a new variant of PSI and demonstrate that our construction for this variant can be used in practical settings.

In this paper, we define differentially private set operations and we contribute a new private set intersection protocol whose result is differentially private, i.e., the intersection is protected as well. Circuit-based PSI protocols [7]–[9] can perform this function in theory. However, we improve over those protocols in communication complexity and memory consumption. For large circuits the memory consumption is commonly the bottleneck [10]. Furthermore, even in the best case for previous circuit-based protocols, the communication complexity is the sum of the sizes of the two databases [9].

We present the first circuit-based PSI protocol based on (leveled) homomorphic encryption. Our solution is asymptotically optimal in a number of criteria: Let the client have a set of size m and the server a set of size n where $m < n$. Then, our communication complexity is $\mathcal{O}(m)$.[1] Our computation complexity is $\mathcal{O}(n+m)$ (or $\mathcal{O}(n)$ since $m < n$). Our differentially private output is optimally accurate for set intersection cardinality [13]. Note that the most recent circuit-based PSI protocols [8], [9] have communication complexity at least $\mathcal{O}(m+n)$ and previous PSI protocols based on homomorphic encryption [11], [12] cannot compute arbitrary circuits (as is necessary for differential privacy) in addition to having computation complexity $\mathcal{O}(nm)$.

Next to the theoretic optimality, we perform a number of optimizations that make our protocols practical. Let each element have a bit length $\ell \geq \log n$. The multiplicative depth of our circuit is $\log \ell + 1$ which is six multiplications for 32 bits and hence practically feasible with many homomorphic encryption schemes. Furthermore, we use vectorization of the plaintexts, and our implementation uses a hashing technique that achieves better performance than the asymptotically optimal cuckoo hashing. We are the first to show that the secure computation of differentially private set operations – intersection and intersection cardinality – is practically feasible. While our practical performance cannot compete with the most efficient protocols – either using homomorphic encryption [12] or circuit-based [9] – we can reasonably handle large data sets up to millions of elements, and these protocols [9], [12] do not protect privacy in the intersection.

In particular, our communication cost when comparing $m = 4\,096$ client elements to $n = 10^6$ server elements

1. Actually, our communication complexity is $\mathcal{O}(m\ell)$, where ℓ represents the bit length. However, we use the notation of recent related work [8], [9], [11], [12] where ℓ is assumed constant, and concentrate on the parameters n and m.

is only $232\,\text{MB}$ whereas other circuit-based approaches are much higher in terms of communication cost for like parameters. For example, the latest approach in [9] needs $2.5\,\text{GB}$ for similar parameters. The lower communication cost directly translates to lower memory requirements, since the majority of the communication cost in the other approaches is spent on the circuit (96% according to [9]).

Contributions. This paper contributes new PSI protocols based on leveled homomorphic encryption that

- compute a *differentially private* result for both the set intersection or the set intersection cardinality,
- have *optimal* communication and computation complexity as well as accuracy for a given privacy parameter,
- are *practical* for large data sets up to millions of elements.

Organization. The remainder of our paper is organized as follows. Section 2 introduces use cases for our new differentially private PSI protocols. Section 3 contains related work, while Section 4 contains the preliminaries needed to understand our work. In Section 5 we define differentially private set operations, and in Section 6 we explain our constructions. Section 7 contains the complexity, security, and privacy analyses of our algorithms, and the performance results of our implementation are discussed in Section 8. Section 9 concludes our work.

2. Use Cases

Private set intersection (PSI) is a useful tool with many applications. However, in many cases not only are the elements outside the intersection sensitive, but also the elements inside. In this paper, we propose a mechanism to protect the set intersection itself via differential privacy. Our mechanism can also be used to compute differentially private predicates over the intersection. We give an algorithm to compute one of these predicates, namely the set intersection cardinality. Our protocol can be extended to compute other predicates with minimal changes, such as computing whether or not two parties have any elements in common, or computing a weighted sum of the intersection.

We emphasize that providing (leakage-free) cryptographic security in this context is not feasible, since some information needs to be revealed to the other party as the result of the analyses. However, differential privacy is a well suited privacy notion for PSI. Differential privacy guarantees that inferences about any individual element in the database are limited while aggregate data, in many cases, is accurate enough for meaningful analyses. The cost for differential privacy comes in terms of false positives and false negatives in the intersection. However, this cost can be configured to the relevant application as in the following case.

Consider two network operation centers that try to determine joint cyber threats. As a first step, they would like to determine which threat indicators [14] they have in common. By using our scheme, these parties can compute a differentially private intersection that protects each set. Depending on their security and privacy preferences, the operation centers may choose a different false positive and negative trade-off. Security-sensitive centers might want to avoid missing common threats – low false negative rate – but can tolerate high false positives. However, privacy-sensitive centers might want to avoid revealing data unrelated to common threats – low false positive rate – and focus their attention to a few surely common threats – high false negative rate.

For parties interested in computing a differentially private version of the intersection cardinality, consider the use of genome databases [15] in medical research. A genome database can enable researchers to analyze pre-dominant alleles in certain single-nucleotide polymorphisms (SNPs) for a subset of patients – similar to a genome-wide association study [16]. The database allows privacy-preserving queries of the following form: A client assembles a set of patients with a specific trait, e.g., a certain disease. The client then queries the intersection cardinality of their own set and the database and the intersection cardinality of their own set and all elements in the database that have a specific allele. The quotient determines the impact of that allele on this specific illness. Using our differentially private protocols the accuracy can be sufficiently high to allow the researcher to perform meaningful analyses, but prevent a malicious client from inferring genetic information about a specific individual. Note this use case also has imbalanced set sizes, since the set of queried patients is almost always much smaller than the set of patients in the database, and therefore benefits from our design.

Finally, for other predicates over the set intersection, recall the case of Google and Mastercard that we mentioned in the introduction. Assume Google and Mastercard identify the set of users that they have in common in a certain period of time. The result would be users that viewed Google ads from vendor V and that had Mastercard transactions with vendor V. Next, assume the sum of the credit card transactions are computed over the result. If the ad was very specific and only one user viewed it, the sum would reveal part of the purchase history of this user. A differentially private sum – which is only a slight variation of our set intersection cardinality – would conceal that information with the privacy parameter ϵ. Using our protocols, only the differentially private sum (and one party's set size) is revealed.

3. Related Work

Early constructions for private set intersection employed public-key cryptography [2], [3] to some success with respect to communication cost. However, their computation overheads were substantial. Since then, constructions employing oblivious polynomial evaluation (OPE) [1], Oblivious Pseudo-Random Functions (OPRF) [17], [18], Oblivious Transfers (OT) [19]–[24], and Homomorphic Encryption (HE) [11] have been proposed. These proposals offer different trade-offs in terms of their computational cost and communication overhead. OT-based protocols are typically faster than other variants [18], [23], [24], but are outperformed in asymptotic communication complexity by the HE-based approach by Chen et al. [11], whose communication complexity is linear in the smaller set size and logarithmic in the larger set size. Recent work by Pinkas et al. [22] achieves a better balance of computation and communication costs, and has the least monetary cost out of all known protocols.

PSI can be designed with optimizations for the balanced and unbalanced use cases. Balanced PSI refers to the setting where the client and the server posses datasets of comparable sizes. In contrast to this, unbalanced PSI refers to the setting where one of the parties, say the server, has a substantially larger dataset than the other party; such as in the case of private contact discovery [25]. The PSI protocols from Pinkas et al. [21], [22] consider both balanced and unbalanced use cases, but in the unbalanced case are asymptotically outperformed in terms of communication complexity by Chen et al. [11]. Our protocol is surprisingly efficient in the unbalanced PSI setting, since it achieves a communication complexity of $\mathcal{O}(m)$, where m is the size of the smaller set, outperforming Chen et al. [11] in practice. Although our protocol can be expensive in non-asymptotic regimes, its running time can be significantly decreased through parallelization.

Some PSI constructions allow for the secure computation of predicates over the set intersection, such as the cardinality, a weighted sum of the intersection, or the noise that is required to provide differentially private predicate outputs. Special constructions exist that compute key agreement over an intersection of credentials [26]. Most constructions that we mentioned above, including [11], cannot be efficiently modified to support the differential private setting we are working in and do not support additional computation over the intersection such as that found in works by Pinkas et al. [8], [9]. Our construction has the adaptability for computation of predicates observed in [8], [9], while optimizing in asymptotic communication complexity over [11].

3.1. Private Set Operations

For private set operations, protocols for intersection, union, and cardinality are defined by Davidson and Cid [27]. Protocols also exist for PSI threshold cardinality, which return whether or not the intersection is greater than a certain threshold [8], [28]. Finally, Ciampi and Orlandi's protocol is designed to compute arbitrary functions on set intersections [29]. Our construction for DiPSI includes functionality for computing the cardinality of the set intersection in a differentially private way, but such computations need not be limited to cardinality and could be potentially extended to include other operations as has been done for PSI protocols.

3.2. Differentially Private Set Computations

Gopi et al. [30] define a differential privacy mechanism for the set union in the central curator model. However, they do not provide a secure computation over several databases as we do.

Private record linkage identifies pairs of records that are similar to one another according to some pre-defined rule. Recent work from He et al. [31], with techniques further improved by Groce et al. [32] for PSI, employs differential privacy and secure computation to solve this problem. The techniques used by He et al. [31] for matching elements are similar to the ones used here. However, in order to fulfill their (slightly weaker) security notion, they require databases to be f-neighbours. f-neighbouring is a restriction on the common neighbouring definition in differential privacy that removes pairs of neighbors that differ in their output of the protocol and thereby partitions the neighbouring graph. This restriction, of course, fails to protect the intersection. In this work we use the *common definition of neighbouring* in differential privacy where any databases that differ in one element are neighbours – even if this element is included in the set intersection – and the impact of the difference on the output of the protocol is limited. This protects the intersection in the stronger simulation-based computational differential privacy model.

4. Preliminaries

4.1. Differential Privacy

Definition 1. (ϵ-Differential Privacy [33]). A randomized mechanism $M : \mathcal{D} \mapsto \mathcal{F}$ provides ϵ-differential privacy (ϵ-DP) iff for all neighbouring inputs $D, D' \in \mathcal{D}$, i.e., differing in one element, and all subsets $F \subseteq \mathcal{F}$,

$$Pr[M(D) \in F] \leq e^{\epsilon} Pr[M(D') \in F] \,, \tag{1}$$

where the probability space is M's coin tosses.

ϵ-Differential Privacy guarantees that for *every* run of the mechanism $M(\cdot)$, the perturbed output is (almost) equally likely to be observed on D and D'.

We use an adaptation of the ϵ-differential privacy definition, simulation-based computational differential privacy, due to Mironov et al. [34] in our security analysis.

Definition 2. Simulation-Based Computational Differential Privacy (SIM-CDP privacy [34]). An ensemble $\{m_k\}_{k\in\mathbb{N}}$ of randomized functions $m_k : \mathcal{D} \mapsto \mathcal{F}$ provides ϵ_k-SIM-CDP if there exists an ensemble $\{M_k\}_{k\in\mathbb{N}}$ of ϵ_k-differentially private mechanisms $M_k : D \mapsto \mathcal{F}_k$ and a negligible function $\mathsf{negl}(n)(\cdot)$, such that for every non-uniform probabilistic polynomial time Turing machine A, every polynomial $p(\cdot)$, every sufficiently large $k \in \mathbb{N}$, every data set $D \in \mathcal{D}$ of size at most $p(k)$, and every advice string z_k of size at most $p(k)$, it holds that,

$$|Pr[A_k(m_k(D)) = 1] - Pr[A_k(M_k(D)) = 1]| \leq \mathsf{negl}(n)(k). \tag{2}$$

That is, $m_k(D)$ and $M_k(D)$ are computationally indistinguishable.

The above requires the existence of an ϵ-differentially private mechanism M (termed a simulator), such that the simulator $M(D)$ and a computed function $m(D)$ are computationally indistinguishable for every set D.

4.2. Private Set Operations

We define privacy in the semi-honest model. For a deterministic function f, we say that a protocol π securely computes f if a participant P's view of the information after π completes could be generated by a simulator given only the input from P and the output of the protocol.

Definition 3. Formally, we define a function f and a two-party protocol for computing f, denoted by π. The view of the ith party during the execution of the protocol π on inputs (x, y) is denoted $VIEW_i^{\pi}(x, y)$ for i representing participant P_1 or participant P_2. The output of the protocol π on inputs (x, y) is denoted $OUTPUT^\pi(x, y)$. The protocol π securely computes f if there exists polynomial-time algorithms, denoted Sim_1 and Sim_2, such that

$$Sim_1(x, f(x, y)) \overset{c}{\equiv} (VIEW_1^\pi(x, y), OUTPUT^\pi(x, y)),$$
$$Sim_2(y, f(x, y)) \overset{c}{\equiv} (VIEW_2^\pi(x, y), OUTPUT^\pi(x, y)).$$

Let $\mathbb{X}$ be a set of size m and let $\mathbb{Y}$ be a set of size n belonging to a client and server respectively and assume that the sizes, m and n, are known to both parties and can be safely revealed to one another during the protocol. We define the following set operations:

Definition 4. *Private Set Intersection (PSI).* For the sets $\mathbb{X}$ and $\mathbb{Y}$, compute $\mathbb{X} \cap \mathbb{Y}$ such that the client learns $\mathbb{X} \cap \mathbb{Y}$ while the *server learns no additional information* and security is met as per Definition 3.

In this definition, the client learns nothing additional about the elements belonging to $\mathbb{Y} \setminus (\mathbb{X} \cap \mathbb{Y})$.

Definition 5. *Private Set Intersection Cardinality (PSI-CA).* For the sets $\mathbb{X}$ and $\mathbb{Y}$, compute $|\mathbb{X} \cap \mathbb{Y}|$ such that the client learns $|\mathbb{X} \cap \mathbb{Y}|$ while the *server learns no additional information* and security is met as per Definition 3.

The client learns nothing additional beyond $|\mathbb{X} \cap \mathbb{Y}|$.

In Section 5, we extend these definitions to the differential privacy setting, so that the computation result observed by the client (i.e., the set intersection $\mathbb{X} \cap \mathbb{Y}$ or the intersection cardinality $|\mathbb{X} \cap \mathbb{Y}|$) is perturbed by noise which prevents the client form learning a significant amount of information about the server data $\mathbb{Y}$.

4.3. Homomorphic Encryption

Homomorphic encryption (HE) schemes are used to perform computations on ciphertexts which then decrypt to the same result as if these computations were performed on plaintexts. HE is especially useful in situations where one party has access to vast computing power while the other party may be modeled with limited computational resources. The secret key to the encryption is only accessible by one party, typically the computationally restricted one, while the other party learns nothing about intermediary or final results of the computations. Following the definition of Yi et al. [35], let (P, C, K, E, D) be an encryption scheme with P, C as the plaintext and ciphertext space, K the key space and E, D as the encryption and decryption algorithms. If $(P, \diamond)$ and $(C, \odot)$ form an algebraic group, then homomorphic encryption using a secret k satisfies the following for all $p, p' \in P, k \in K$:

$$E_k(p) \odot E_k(p') = E_k(p \diamond p')$$

There are three types of HE schemes that are important for our application: Somewhat HE (SWHE), Fully HE (FHE) and Leveled HE (LHE). All schemes are based on the notion of a noise that is embedded into every ciphertext during encryption. This noise grows with each operation and effectively limits the maximal applicable operation depth. Once a threshold is exceeded, the noise overwrites the encoded data and renders it non-decipherable, whereby multiplications typically lead to a significantly larger noise growth than additions. SWHE allow for additions and multiplications, but they do not have a way of resetting the noise term. FHE schemes extend SWHE schemes with a function called bootstrapping, which evaluates the re-encryption of a ciphertext as a function within the HE and thereby resetting the noise in the ciphertext at a considerable computational cost. Lastly, LHE does not rely on bootstrapping but rather allows choosing parameters to accommodate for a fixed operation depth while inducing a "quasi-linear" growth of the computation complexity in the security parameter, as described in [36]. However, the operation depth must be known beforehand, which is the case for our DiPSI protocol. Further efficiency improvements can be obtained through "ciphertext packing", where a vector of plaintexts is encrypted into separate slots of a ciphertext, while data-flow between slots is not possible. This allows for inherent parallelization through single instruction, multiple data (SIMD) operations if the encoded elements can be processed independently from each other.

4.4. PSI via Hashing-to-Bins

A naive scheme for computing the intersection between two sets consists of comparing each element from the first set with each element from the second set. This is highly inefficient in terms of computation. One solution to this problem is to organize the set elements into "bins" within a table, and then perform the comparison only between elements within the same bins. Such solutions are called hashing-to-bins techniques, since they use hash functions to assign elements to bins within a table [21], [24], [37]. In this work, the techniques we use for hashing to bins exclusively consist of employing two hash functions to place inputs in one of two tables or in a secondary storage location called the stash. Each table consists of bins and each bin contains an agreed upon γ number of bin elements. If the client and the server agree upon appropriate hash functions in advance, then it is only necessary to compare inputs that are mapped to the same bins. Note that for both the server and client it is necessary to always send the γ number of bin elements by adding dummy elements as doing otherwise would reveal the number of elements that are mapped to a particular bin and as a result leak information about the inputs [8].

4.5. Collision Handling and Hashing-to-Bins

When using hashing-to-bins techniques it is possible for a *collision* to occur. A collision occurs when the bin an element has been hashed to has already reached the maximum number of elements, γ. In this section, we discuss three of the possible techniques for handling collisions when performing hashing-to-bins.

4.5.1. Cuckoo Hashing. Cuckoo hashing [38], [39] is a technique that handles collisions by ejecting the element currently in a bin, replacing it with the latest element and then hashing the ejected element with a different hash function. Consider a simple instance of cuckoo hashing with two hash functions, $H_1(x)$ and $H_2(x)$. Define T_1 and T_2 as the tables belonging to $H_1(x)$ and $H_2(x)$ respectively. Both T_1 and T_2 are of size $(1 + \varepsilon)n$, where

ε is a chosen constant value, for example $\varepsilon = 0.6$.[2] For any element x, first compute $H_1(x)$. Place x in the bin corresponding to $T_1[H_1(x)]$. For cuckoo hashing, each bin can contain at most one element ($\gamma = 1$). Therefore, if $T_1[H_1(x)]$ is occupied by an element y, remove y and compute a new location, $T_2[H_2(y)]$. After placing y, if the new location is also occupied, the original occupant is again removed and hashed to a different location. This continues until a maximum threshold of attempts has passed, at which point the value is placed in the stash. Any element x hashed using this method can be found in either $T_1[H_1(x)]$, $T_2[H_2(x)]$, or the stash. Therefore, a look-up takes $\mathcal{O}(1)$. See Kirsch et al. [40] for the analysis of using cuckoo hashing with a stash in this manner.

4.5.2. Nested Cuckoo Hashing. A modified version of cuckoo hashing with a stash, called nested cuckoo hashing, was proposed by Goodrich et al. [41]. The modified hashing structure employs a primary cuckoo structure with a secondary cuckoo structure in place of the primary structure's stash. For nested cuckoo hashing, after a threshold number of attempts at placing the element in the primary cuckoo structure (with hash functions H_1 and H_2, and tables T_1 and T_2), the secondary cuckoo structure is employed (with hash functions H_3 and H_4, and tables T_3 and T_4). The secondary cuckoo structure works in the same manner as the conventional cuckoo hashing we originally presented, however, each of its tables contain m' elements, where $m' \leq m$ and m is the number of elements in the primary cuckoo structure's tables.

When using cuckoo hashing with a stash it is necessary to account for a failure state. A failure state occurs when it is necessary to place an item in the stash, because the number of insertion attempts has passed the threshold, but the stash is full. When such a failure state is reached, one cannot simply select new hash functions as such an action has the potential to leak information about the data, since the hashes are no longer truly random. Fortunately, the probability of a failure state for nested cuckoo hashing is shown to be negligible given the results found in Theorem 1 and Theorem 2 from Goodrich et al. [41].

4.5.3. Dual Hash Function with Bin Sizes $\gamma \geq 1$. A variant of cuckoo hashing, dual hashing, consists of two hash functions. A table contains m bins and unlike cuckoo hashing each bin contains an agreed upon γ number of bin elements where $\gamma \geq 1$. For any element x, compute $H_1(x)$ and $H_2(x)$. Append the value x to whichever bin, $T[H_1(x)]$ or $T[H_2(x)]$, contains the fewer number of elements. After hashing all of the inputs, the client appends dummy elements for each bin in T until all bins hold γ elements so as not to leak information about the data set in this manner.

4.5.4. Matching for Hashing-to-Bins. When using cuckoo hashing or its variants it can be tempting to have both parties hash all of their elements using the selected cuckoo hashing method and then compare only the elements that were hashed to the same 'bin'. However, such a comparison strategy would miss matches as it is possible for the client to map an element x to $H_1(x)$ while the server mapped x to $H_2(x)$. Therefore, an alternate strategy is required. One technique is to have the client use cuckoo hashing while the server employs regular hashing. That is, the server hashes all n elements with $H_1(x)$ and $H_2(x)$. When used with our batching strategy presented in Section 6, the resulting process requires $\mathcal{O}(n\gamma)$ comparisons to determine the intersection, where γ represents the bin capacity. Note that $\gamma = 1$ for conventional cuckoo hashing and $\gamma \geq 1$ for the dual hashing variant. Briefly, the intuition for $\mathcal{O}(n\gamma)$ comparisons is that for each ciphertext sent by the client to the server, the server does not know whether or not the ciphertext corresponds to T_1 or T_2. Therefore, the server must compare each ciphertext with the appropriate bins in both T_1 and T_2. This results in $\mathcal{O}(2n\gamma) = \mathcal{O}(n\gamma)$ comparisons.

2. Note that this ε is distinct from the ϵ used for differential privacy.

5. Differentially Private Set Operations

In this section, we extend the private set intersection definitions from Section 4.2 to account for the differential privacy setting. As before, let $\mathbb{X}$ be the client set, of size m, and let $\mathbb{Y}$ be the server set, of size n. Assume that m and n are known to both parties.

As explained before, we consider a semi-honest server model, where the server follows the protocol truthfully but might be interested in learning information about $\mathbb{X}$. Our differential privacy definition is one-sided, in the sense that it ensures that the server learns *nothing* about the client's dataset $\mathbb{X}$, and the client learns a *differentially private* version of the intersection or intersection cardinality. This prevents the client from learning significant information about the server dataset $\mathbb{Y}$.

Let $\mathbb{Y} \sim \mathbb{Y}'$ be neighbouring datasets, i.e., they differ on a single element. Let $\mathcal{X}$ be the domain of each element in $\mathbb{X}$ and $\mathbb{Y}$.

Definition 6. *ϵ-Differentially Private Set Operations (ϵ-DiPSI).* The two party computation algorithm $M : \mathcal{X}^m \times \mathcal{X}^n \rightarrow \mathcal{F}$, that takes the client and server data sets and returns a perturbed operation (e.g. intersection or intersection cardinality) to the client, achieves ϵ-DiPSI iff, for all $F \subseteq \mathcal{F}$, all client sets $\mathbb{X}$, and all neighbouring data sets $\mathbb{Y} \sim \mathbb{Y}'$,

$$\Pr(M(\mathbb{X}, \mathbb{Y}) \in F) \leq e^{\epsilon} \Pr(M(\mathbb{X}, \mathbb{Y}') \in F) . \quad (3)$$

We propose two ϵ-DiPSI set operation mechanisms: M_{RR-SI} and M_{LAP-CA}. The former, M_{RR-SI}, computes a differentially private set intersection. It achieves differential privacy by using a secure implementation of the randomized response mechanism [42] over the result of $\mathbb{X} \cap \mathbb{Y}$. In summary, M_{RR-SI}, which is run by the server, takes an encrypted version of the client database and compares it to a plaintext version of the server database, producing an encrypted version of the differentially private set intersection that the client can later decrypt. After decryption, the client learns if $x \in \mathbb{Y}$ with probability p, and gets a random response otherwise. Therefore, the client cannot know for certain whether or not a certain element x is in $\mathbb{Y}$.

M_{LAP-CA} computes a differentially private set intersection cardinality. It uses a secure implementation of the Laplacian mechanism [42] to guarantee differential privacy. In short, after receiving an encrypted copy of

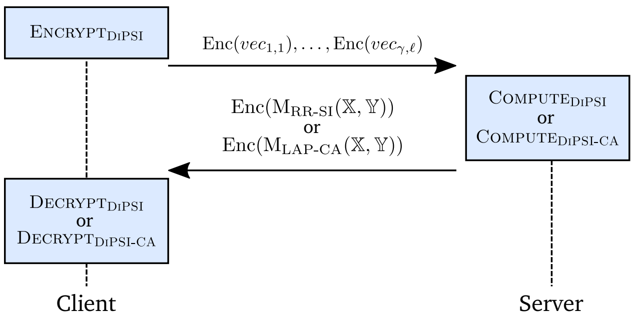

Figure 1. DiPSI Protocol Overview

the client's data, the server performs a comparison with their plaintext data, and returns the encrypted intersection cardinality $|\mathbb{X} \cap \mathbb{Y}|$ perturbed with Laplacian noise to the client. This noise ensures that the client cannot gain significant information about the server dataset. The server does not learn anything in the process.

We implement both mechanisms in ϵ-SIM-CDiPSI private protocols:

Definition 7. A randomized protocol $\pi : \mathcal{X}^m \times \mathcal{X}^n \rightarrow \mathcal{F}$ provides ϵ-SIM-CDiPSI if there exists an ϵ-DiPSI mechanism $M : \mathcal{X}^m \times \mathcal{X}^n \rightarrow \mathcal{F}$ and a negligible function $\mathsf{negl}(n)$, such that for every non-uniform probabilistic polynomial time Turing machine A, all client sets $\mathbb{X}$ and server sets $\mathbb{Y}$, it holds that,

$$\begin{aligned} |Pr[A(VIEW^{\pi}_{client}(\mathbb{X}, \mathbb{Y})) = 1] \\ -Pr[A(Sim_1(M(\mathbb{X}, \mathbb{Y}))) = 1]| \leq \mathsf{negl}(n) \end{aligned} \tag{4}$$

and

$$\begin{aligned} |Pr[A(VIEW^{\pi}_{server}(\mathbb{X}, \mathbb{Y})) = 1] \\ -Pr[A(Sim_2(m)) = 1]| \leq \mathsf{negl}(n)\,. \end{aligned} \tag{5}$$

That is, $\pi(\mathbb{X}, \mathbb{Y})$ and a simulator given $M(\mathbb{X}, \mathbb{Y})$ are computationally indistinguishable.

6. Algorithm Description

In this section, we present our constructions for differentially private set intersection and set intersection cardinality. These mechanisms use the same encryption stage, but have their own specific algorithms for server computation, client decryption, and post-processing.

6.1. Overview

Our DiPSI protocols use (leveled) homomorphic encryption and proceed in three steps. First, the client encrypts its elements and sends them to the server. Second, the server uses the evaluation function of homomorphic encryption to evaluate a circuit that computes the DiPSI functionality (either M_{RR-SI} or M_{LAP-CA}) and returns the result to the client. Third, the client decrypts the result. Figure 1 outlines the steps and the exchanged messages.

The crucial step is the evaluation on the ciphertexts by the server. While in theory the server can evaluate any function using homomorphic encryption, the challenge is to keep the multiplicative depth of the circuit low in order to avoid bootstrapping and keep the protocol practical. Our protocol differs in this step from other PSI protocols based on homomorphic encryption, e.g. [11], [12]. In order to efficiently evaluate predicates over the set intersection, such as a differentially private mechanism or the set intersection cardinality, it is necessary to compute a binary predicate of set inclusion of each client element x_i. One possible way of doing this is using polynomial-based PSI protocols. However, we avoid this approach since polynomial-based PSI protocols return a multi-valued attribute where a zero encodes a match and any other random number a mismatch, which is hard to negate with low multiplicative depth circuits and hence limits the predicates that can be computed efficiently. Instead, we compute a binary-valued predicate using a binary encoding of the set elements to allow for efficient negation.

We employ a number of further optimization techniques. As we explained in Section 4, we use hashing-to-bins techniques to place elements into bins before matching them. This keeps the computation cost at $\mathcal{O}(n)$, since the server needs to match its elements only a constant number of times (see Section 7 for analysis). Instead of encrypting each client element $x \in \mathbb{X}$ individually, we first encode the client's data by spreading the bits of each client element over multiple ciphertext vectors. This greatly simplifies the computational cost of element comparison on the server side, since it avoids rotation operations that would be mandatory otherwise, saving noise budget for multiplications. Finally, in order to implement our differentially private mechanisms M_{RR-SI} and M_{LAP-CA} we use dummy elements to force non-matches and obfuscate the intersection cardinality by spreading it over a vector and adding Laplacian noise to a single element of this vector.

In combination, this approach ensures that we can efficiently compute over the set intersection. We show that it is practically feasible to compute the differential privacy mechanisms that are appropriate for the specific predicate of set intersection or intersection cardinality. In addition, our protocol is well suited for use cases where the sets are imbalanced, since due to the use of homomorphic encryption our communication complexity is only linear in the size of the smaller set.

6.2. Hashing-to-Bins in DiPSI

Recall that, to reduce the computational cost of the set intersection, both client and server first hash their dataset elements into tables, and then the server performs a comparison between elements within the same bin only. There are many choices of hashing-to-bins algorithms that we could use in our DiPSI constructions. These different choices affect the communication and computational cost of the algorithm. We consider dual hashing (Section 4.5.3) and nested cuckoo hashing (Section 4.5.2).

We use dual hashing for our algorithm description in this section, and for our implementation and evaluation in Section 8, while we use nested cuckoo hashing for our asymptotic complexity analysis in Section 7. The reason for this is that, for our experimentation parameters, dual hashing is more efficient than nested cuckoo hashing. On the other hand, nested cuckoo hashing is asymptotically better than dual hashing (as the client and server data sets grow in size), and thus it allows us to show the asymptotic

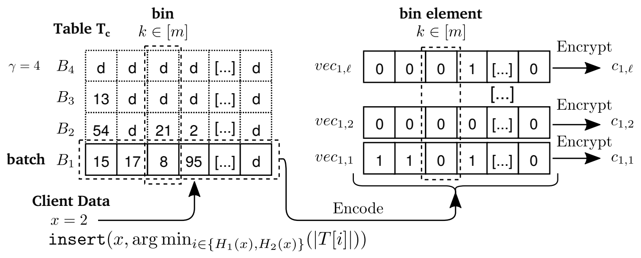

Figure 2. Client Side Encryption Process

Table 1. SUMMARY OF NOTATION FOR ALGORITHM DESCRIPTION (WITH DUAL HASHING TECHNIQUE)

Symbol	Meaning
T_c	Client table containing m bins with γ elements.
T_s	Server table containing m bins with μ elements.
$x \in \mathcal{X}$	Generic element in client's dataset.
$y \in \mathcal{X}$	Generic element in server's dataset.
$x_{i,k}[j]$	The jth bit of the ith element in the kth batch of T_c.
γ	Maximum capacity in any client bin (index $i \in [\gamma]$).
μ	Maximum capacity in any server bin (index $i' \in [\mu]$).
m	Number of bins in T_c and T_s (index $k \in [m]$.
ℓ	Number of bits of an element in $\mathcal{X}$ (index $j \in [\ell]$).
s	Number of client elements per ciphertext ($s = 4\,096$).
t	Plaintext modulus ($t = 40\,961$).

performance that our algorithms can achieve. Later, in Section 6.6, we clarify some specifics of our algorithm when using nested cuckoo hashing.

6.3. Encryption

Encryption proceeds through the following stages: hashing-to-bins, encoding, batching, and batch encryption. The stages are detailed in the following and are presented as Algorithm 1. Figure 2 illustrates the client encryption process, and Table 1 contains notation used in this section.

6.3.1. Hashing-to-Bins. During the encryption stage, a client and a server must first perform an agreed upon process for hashing-to-bins. As mentioned above, for simplicity in this section we assume the hash function used is dual hashing. The client hashes each element x in its set $\mathbb{X}$ using the process described in Section 4.5.3. After the hashing process, the client has a table T_c containing m bins that each hold γ elements. Recall that if a bin does not hold γ elements at the conclusion of the initial hashing, dummy elements are added to that bin, and any other such bin, until all bins do contain γ elements.

Consider the example in Figure 2. Although the number of bins is abstracted, there are visibly $\gamma = 4$ bin elements per bin. The first bin contains the values 15, 54, 13, and d, where d is the client dummy element.

6.3.2. Batching. While the hashing process is used to allow comparisons only between elements mapped to the same bins, batching is a technique used to combine multiple plaintexts into a vector which is then encrypted into a single ciphertext. By combining multiple plaintexts into a single ciphertext, the batching process reduces the concrete overhead of homomorphic element comparisons through allowing the processing of multiple plaintexts in a batch.

Batching occurs over each bin in the client's table, such that the ith batch contains the ith element from every bin ($k \in [m]$) of that table. Each batch is encoded before being encrypted. A batch $i \in [\gamma]$ contains m batch elements $x_{i,1}, x_{i,2}, \ldots, x_{i,m}$, each of length ℓ (bits). These elements are encoded to produce vectors $vec_{i,j} \doteq [x_{i,1}[j], x_{i,2}[j], \ldots, x_{i,m}[j]]$, for all $i \in [\gamma], j \in [\ell]$. For example, let a batch B_i be the set of batch elements $\{2, 4, 5, 7\}$. Written in binary, $\{010, 100, 101, 111\}$, the batch elements are each of at most length $\ell = 3$. The first vector, $vec_{i,1}$ consists of the first bit from each element, producing $vec_{i,1} = 0111$, while the remaining vectors are $vec_{i,2} = 1001$, and $vec_{i,3} = 0011$.

The example shown in Figure 2 contains four batches. The first batch, B_1 is expanded to the right where the first column represents the binary representation of the first element 15. The first bit in $vec_{1,1}$ represents the most significant bit of the element 15 in the first bin. The second bit in $vec_{1,1}$ represents the most significant bit of the element 17, and so on.

Once encoded as vectors $vec_{i,j}$ ($i \in [\gamma], j \in [\ell]$), the batched elements are ready for the encryption stage.

6.3.3. Homomorphic Encryption. We use the Brakerski-Gentry-Vaikuntanathan (BGV) [36] scheme implemented in HELib.[3] Their implemented version supports bootstrapping, but we use it as a leveled homomorphic encryption scheme. Furthermore, we use ciphertext packing and encrypt $s = 4\,096$ client elements per ciphertext. We use a plaintext modulus of $t = 40\,961$ to allow for the cardinality predicate to be computed on top of the DiPSI-CA. If a predicate requires an even larger plaintext modulus, the Chinese Remainder Theorem (CRT) could be used to artificially inflate the plaintext modulus space exponentially in the number of ciphertexts by applying the computations over multiple ciphertexts with co-prime plaintext moduli as described in [43].

Algorithm 1 ENCRYPT$_{\text{DiPSI}}$

```
1:  /* Client Hashing-to-bins (using dual hash) */
2:  for each client element $x \in \mathbb{X}$ do
3:      Compute $H_1(x)$ and $H_2(x)$.
4:      Append $x$ to whichever of $T_c[H_1(x)]$ and $T_c[H_2(x)]$ contains fewest
        elements.
5:  Append dummy elements to fill each bin in $T_c$ to $\gamma$.
6:  /* Client encoding and encryption */
7:  for each batch $i \in [\gamma]$ do
8:      for each bit $j \in [\ell]$ do
9:          $vec_{i,j} = [x_{i,1}[j], x_{i,2}[j], \ldots, x_{i,m}[j]]$
10:         $c_{i,j} = \mathsf{Enc}(vec_{i,j})$
11: Client sends the ciphertexts $c_{i,j}$ for all $i \in [\gamma], j \in [\ell]$ to server.
```

6.4. Server Computation and Decryption for Differentially Private Set Intersection (ϵ-DiPSI)

Details for computing the differentially private set intersection between two sets $\mathbb{X}$ and $\mathbb{Y}$ and the procedure for calculating matches on the server side, along with the client decryption, can be found below.

Recall that in Algorithm 1 the client constructs batches of size m and sends them encrypted to the server (plaintext size is padded to a multiple of s). The server receives

3. https://github.com/homenc/HElib/

these batches and runs Algorithm 2. Conceptually, this algorithm achieves ϵ-DP by running an homomorphic implementation of the randomized response algorithm in two stages. In the first stage, the server flips a coin for each element $k \in [m]$ in each client batch $i \in [\gamma]$, such that if the result is "*heads*" ($coin = 1$ in the algorithm) the algorithm will return whether or not the client element is in the server data set. If, however, the result of the coin flip is "*tails*", the server instead flips a second coin. The second coin flip result determines whether the algorithm returns a match or a non-match for the client element.

The server matching algorithm is described in Algorithm 2 and illustrated in Figure 3. As a preliminary step, the server creates the table T_s by hashing each element $y \in \mathbb{Y}$ using both H_1 and H_2, and appending y to both bins $T_s[H_1(y)]$ and $T_s[H_2(y)]$. Then, the algorithm proceeds independently for each client batch $i \in [\gamma]$, using a fresh copy T_s' of T_s. We describe the matching process for a single client batch below.

Recall that the ciphertexts $c_{i,j}$ (for all $j \in [\ell]$) contain encrypted versions of $x_{i,1}, x_{i,2}, \ldots, x_{i,m}$. For each of these elements, the server flips a biased coin. If the result of the kth coin flip is "*tails*", the server fills the kth bin of their table T_s' with dummy elements and sets a flag $flag[k] = 1$ to remember the result of the coin flip (lines 5-14). After running this process, the server encodes T_s' following the same approach as the client (lines 16-19). This encoding generates the plaintext vectors $vec_{i',j}$ ($i' \in [\mu]$, $j \in [\ell]$), where each vector contains the the jth bit of all the m elements in the i'th server batch. The algorithm then proceeds to the matching phase. First, the server compares the ith client batch with every server batch $i' \in [\mu]$. This comparison is achieved by first computing the vectors $M_{i',i,j} = c_{i,j} \oplus \neg vec_{i',j}$ for all $j \in [\ell]$ and then performing an AND operation over all of them. The result, denoted $M_{i',i}$, is an encryption of a vector that contains ones in those positions k where $x_{i,k}$ was found in the i'th server batch, and zeroes otherwise. Then, the server homomorphically adds $M_{i',i}$ over all the server batches $i' \in [\mu]$, such that the resulting vector M_i contains (encrypted) ones where $x_{i,k}$ was found in the server database (lines 21-26). Note that there will be zeroes in M_i in those positions where the first coin flip was tails (i.e., in those $k \in [m]$ such that $flag[k] = 1$). For each of these forced non-matches $\{k | flag[k] = 1\}$, the server proceeds with the second coin flip and adds a one to the kth position of M_i if the result of the coin flip is heads, and leaves the zero otherwise. Note that all these operations are performed in the encrypted domain, so the server never learns any information about the client database or the intersection $\mathbb{X} \cap \mathbb{Y}$. Finally, the server returns M_i to the client, who then decrypts it to recover an ϵ-DP version of the set intersection for the ith client batch. The process is repeated for all $i \in [\gamma]$. Note that the computations for each client batch are independent, so our algorithm allows parallelization among client batches.

DECRYPT$_{\text{DiPSI}}$. Upon receiving each M_i, the client decrypts them and removes padding (if it was added to match the plaintext size parameter s). The result is a vector of size m with either zeroes or ones. As mentioned above, a 1 at position k indicates that either $x_{i,k}$ was found in the server database or that it was not found but the

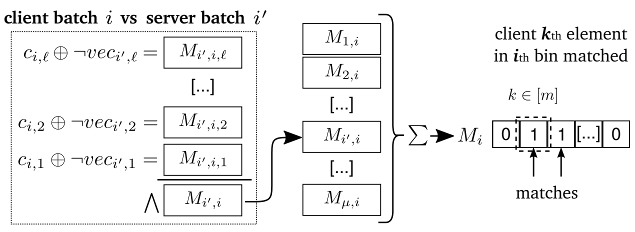
client batch i vs server batch i'
$c_{i,\ell} \oplus \neg vec_{i',\ell} =$
$M_{i',i,\ell}$
[...]
$c_{i,2} \oplus \neg vec_{i',2} =$
$M_{i',i,2}$
$c_{i,1} \oplus \neg vec_{i',1} =$
$M_{i',i,1}$
$\bigwedge$
$M_{i',i}$
$M_{1,i}$
$M_{2,i}$
[...]
$M_{i',i}$
[...]
$M_{\mu,i}$
$\sum$
M_i
client kth element in ith bin matched
$k \in [m]$
0
1
1
[...]
0
matches

Figure 3. Server Side Matching Protocol Computations

randomized response mechanism generated a fake 1 so as to achieve differential privacy. Note that the client can discard the matches for those client elements that were dummies $x_{i,k} = d$.

Algorithm 2 COMPUTE$_{\text{DiPSI}}$

```
1:  /* Server hashing-to-bins */
2:  for each server element $y \in \mathbb{Y}$ do
3:      Append $y$ to the bin $T_s[H_1(y)]$ and to the bin $T_s[H_2(y)]$
4:  for each client batch $i \in [\gamma]$ do
5:      Copy the server table $T_s' = T_s$ to compare with the $i$th client batch.
6:      Initialize flag vector to $m$ zeroes: $flag = [0, 0, \ldots, 0]$.
7:      for each bin element $k \in [m]$ do
8:          $coin \overset{p}{\leftarrow} \$ \{0, 1\}$ //Flip a biased coin with weight $p$
9:          if $coin = 1$ (heads) then
10:             No change to server plaintexts
11:         else
12:             Set all elements in $T_s'[k]$ to dummy elements to force non-matches
13:             Save $flag[k] = 1$ to indicate forced no match
14:     /* Server encoding */
15:     for each server batch $i' \in [\mu]$ do
16:         for each bit $j \in [\ell]$ do
17:             $vec_{i',j} = [y_{i',1}[j], y_{i',2}[j], \ldots, y_{i',m}[j]]$
18:     /* Intersection of the $i$th client batch and $T_s'$ */
19:     for each server batch $i' \in [\mu]$ do
20:         for each bit $j \in [\ell]$ do
21:             $M_{i',i,j} = c_{i,j} \oplus \neg vec_{i',j}$
22:         $M_{i',i} = \bigwedge_{j=1}^{\ell} M_{i',i,j}$
23:     $M_i = \sum_{i'=1}^{\mu} M_{i',i}$ (mod 2).
24:     /* Differential privacy */
25:     Initialize an $m$-sized noise vector: $noise = [0, 0, \ldots, 0]$.
26:     for each batch element $k \in [m]$ do
27:         if $flag[k] = 1$ then
28:             $coin \overset{q}{\leftarrow} \$ \{0, 1\}$
29:             $noise[k] = coin$.
30:     $M_i = M_i + noise$
31: Send the encrypted matching results $M_i$, for all $i \in [\gamma]$, to the client.
```

6.5. Server Computation and Decryption for DP Set Intersection Cardinality (ϵ-DiPSI-CA)

The stages and computations for executing the DiPSI-CA protocol initially proceed in the same way as the DiPSI protocol detailed in the previous section. Two of the key differences between DiPSI and DiPSI-CA, other than the results they return, are first the differential privacy mechanisms employed, and second, DiPSI-CA requires additional post processing on the server side after computing the intersection before the result is returned to the client. Due to these similarities we only discuss the details of the differentially private cardinality steps reflected in Lines 19-23 of Algorithm 3.

After computing the set intersection, following the same procedure as DiPSI, the DiPSI-CA protocol is left with a match vector M_i for each batch provided by the client. The server first sums all of the M_i vectors together to produce a result vector M. This result M is the encryption of a vector whose kth element contains the *number* of matches of the elements in the client's kth bin. Next, the server generates a series of random values, one for each bin, such that the random values sum to zero (mod $t = 40\,961$). The server homomorphically adds these values to M so that the overall cardinality is unchanged. Finally, the server constructs a noise vector where the value in its last bin is sampled from a Laplace distribution. This noise vector is then summed with M, and the result is sent back to the client. Note that all of these operations are performed in the encrypted domain, and the server never learns any information about the intersection cardinality.

DECRYPT$_{\text{DiPSI-CA}}$. The client receives exactly one result ciphertext M, which encodes the set intersection cardinality as the sum over all elements. The client decrypts M to a vector of natural numbers $\{0, .., t-1\}^s$, where t is the chosen plaintext modulus. Finally, the client obtains the differentially private set cardinality by taking the sum over all elements modulo t. Note that the client cannot gain any information about which of their elements $x \in \mathbb{X}$ are actually in $\mathbb{Y}$ as each slot in M appears uniformly random (with the exception that the sum of all of the elements in M is the set intersection plus the Laplacian noise). Also, the Laplacian noise protects the real cardinality value in an ϵ-DP way, which we will prove in the next section.

Algorithm 3 COMPUTE$_{\text{DiPSI-CA}}$

```
1:  /* Server hashing-to-bins */
2:  for each server element $y \in \mathbb{Y}$ do
3:      Append $y$ to the bin $T_s[H_1(y)]$ and to the bin $T_s[H_2(y)]$
4:  /* Server encoding */
5:  for each server batch $i' \in [\mu]$ do
6:      for each bit $j \in [\ell]$ do
7:          $vec_{i',j} = [y_{i',1}[j], y_{i',2}[j], \ldots, y_{i',m}[j]]$
8:  for each client batch $i \in [\gamma]$ do
9:      /* Intersection of the $i$th client batch and $T_s$ */
10:     for each server batch $i' \in [\mu]$ do
11:         for each bit $j \in [\ell]$ do
12:             $M_{i',i,j} = c_{i,j} \oplus \neg vec_{i',j}$
13:         $M_{i',i} = \bigwedge_{j=1}^{\ell} M_{i',i,j}$
14:     $M_i = \sum_{i'=1}^{\mu} M_{i',i}$.
15: $M = \sum_{i=1}^{\gamma} M_i$.        // Total matches in each client bin
16: /* Differential privacy */
17: Add a random value $r_k$ to each element in $M$ such that $\sum_{k=1}^{s} r_k = 0$
    mod $t$.
18: Generate Laplace noise $L \leftarrow\$ \mathrm{Lap}(1/\epsilon)$.
19: Construct a noise vector of length $s$ where the only non-zero value is the
    last element set to $L$: $noise = [0, \ldots, 0, L]$.
20: Compute $M = M + noise$.
21: Return $M$ to client.
```

6.6. Algorithm Modifications When Using Nested Cuckoo Hashing

We have explained how our protocols work using dual hashing as the hashing-to-bins technique. For completeness, below we explain the modifications that would need to be implemented when using nested cuckoo hashing:

Encryption. As explained in Section 4.5.2, the client hashes their elements into four tables T_c^i ($i = 1$ to 4) and places some elements in a stash. The client applies the encoding techniques to each table independently, as well as the stash, and sends the resulting ciphertexts to the server.

Server Computation and Decryption. For each client table $i = 1$ to 4, the server creates another table T_s^i and hashes *all* elements $y \in \mathbb{Y}$ into that table using hash function $H_i(x)$. Then, the ciphertexts that correspond to client encodings of table T_c^i are compared to table T_s^i. Each server element is compared to each stash element.

7. Analysis

In this section, we study the complexity of our algorithms in terms of communication and computation costs, and analyze their privacy, security, and utility.

7.1. Analysis of ϵ-DiPSI

7.1.1. Complexity Analysis. For the following complexity analysis we assume the DiPSI algorithm uses the nested cuckoo hashing described in Section 4.5.2. We first introduce a general result of nested cuckoo hashing:

Theorem 1. *Let the client set $\mathbb{X}$ be a set hashed using nested cuckoo hashing and let the server set $\mathbb{Y}$ be a set hashed using simple hashing, where $|\mathbb{X}| = m$, $|\mathbb{Y}| = n$, and $n > m \log m$. Define a nested cuckoo hashing structure where $H_1(x)$ and $H_2(x)$ are hash functions mapping to tables T_c^1 and T_c^2 of size m, $H_c^3(x)$ and $H_c^4(x)$ are hash functions mapping to tables T_3 and T_4 of size $m^{2/3}$, and the stash has a constant size c. The intersection $\mathbb{X} \cap \mathbb{Y}$ can be computed using $\mathcal{O}(n)$ comparisons.*

Proof. As with the use of conventional cuckoo hashing (described in Section 4.5.4), while the client performs nested cuckoo hashing, the server must perform simple hashing. The server hashes every element in the set $\mathbb{Y}$ into a table T_s^i using hash function $H_i(x)$, for $i = 1, 2, 3, 4$. Note that T_s^1 and T_s^2 have m bins, while T_s^3 and T_s^4 have $m^{2/3}$ bins.

Since the number of server elements n is much larger than the number of client elements m, we can compute the number of bins in each server table using the classic balls-into-bins problem [45], [46] such that if $n > m \log m$, the expected maximum load for any particular bin in T_s^1 or T_s^2 can be calculated as

$$\frac{n}{m} + \sqrt{\frac{n \log m}{m}} \, . \tag{6}$$

For the secondary tables T_s^3 and T_s^4, replace m for $m^{2/3}$ in (6). The server compares every single element in the kth client bin to the number elements in the kth server bin given by (6). The total number of comparisons with the table T_s^1, since there are m bins, is

$$m \left(\frac{n}{m} + \sqrt{\frac{n \log m}{m}} \right) = n + \sqrt{nm \log m} = \mathcal{O}(n) \ , \tag{7}$$

since $m \log m < n$. The total number of comparisons with T_s^2 is the same. Likewise, we can show that total number of comparisons with the secondary table is $\mathcal{O}(n)$ (replace all m's above for $m^{2/3}$). Finally, the server compares every stash element (a constant number c) with T_s^1, which also results in communication cost of $\mathcal{O}(n)$. This means that the total required number of comparisons using nested cuckoo hashing is $\mathcal{O}(n)$. □

Now, we are ready to state the communication and computation complexity theorems of DiPSI:

Theorem 2. *The communication complexity of DiPSI where the hashing-to-bins method is nested cuckoo hashing is $\mathcal{O}(m)$ for m client elements and n server elements.*

Proof. Trivially, T_c^1, T_c^2, T_c^3, T_c^4, and the stash must be sent encoded from the client to the server. T_c^1 and T_c^2 are each of size m, T_c^3 and T_c^4 are each of size $m^{2/3}$ and the stash is of size c, where c is some constant. Each element in a bin has ℓ bits. Therefore, the communication from client to server and server to client is $2(2m+2m^{2/3}+c)\ell$, or $\mathcal{O}(m)$.[4] □

Note that a communication cost of $\mathcal{O}(m)$ is optimal, since the client always needs to send at least all the elements $x \in \mathbb{X}$ to the server.

Theorem 3. *The computation complexity of DiPSI where the hashing-to-bins method is nested cuckoo hashing is $\mathcal{O}(n+m)$ for m client elements and n server elements with $n > m \log m$.*

Proof. Both the server and the client have to perform the necessary hashing steps. The client hashes m elements, while the server hashes n elements four times (once for each hash function the client uses).

The main computation occurs during the comparison stage required to compute the set intersection. As stated in Theorem 1, computing the set intersection when the client uses nested cuckoo hashing while the server employs simple hashing requires $\mathcal{O}(n)$ comparisons. The remaining $4m+c$ decryptions do not significantly contribute to the computation costs such that the total computation costs can be computed as

$$m + 4n + \mathcal{O}(n) + 4m + c.$$

Therefore, the computation cost for DiPSI when using nested cuckoo hashing is $\mathcal{O}(n+m)$. □

Note that the computation cost can be written as $\mathcal{O}(n)$, since we are assuming $m \log m < n$. This is asymptotically optimal, since every server element needs to be compared with at least one client element.

7.1.2. Privacy Analysis.

Theorem 4. *Mechanism M_{RR-SI} satisfies ϵ-DiPSI as per Definition 6, with*

$$\epsilon = \log\left[\left(1+\frac{p}{(1-p)q}\right)\left(1+\frac{p}{(1-p)(1-q)}\right)\right]. \tag{8}$$

Proof. Let V be the random binary vector recovered by the client in DiPSI after decryption. Let x_k be the kth client element in $\mathbb{X}$. If $x_k \in \mathbb{Y}$, then the kth bit in $V[k]$ will be one if the first coin flip shows up heads, or if it is tails and the second coin flip shows up heads, i.e.,

$$\Pr(V[k]=1|x_k \in \mathbb{Y}) = p + (1-p)q\,. \tag{9}$$

Likewise, we can compute

$$\Pr(V[k]=0|x_k \in \mathbb{Y}) = (1-p)(1-q) \tag{10}$$
$$\Pr(V[k]=1|x_k \notin \mathbb{Y}) = (1-p)q \tag{11}$$
$$\Pr(V[k]=0|x_k \notin \mathbb{Y}) = p + (1-p)(1-q)\,. \tag{12}$$

Since M_{RR-SI} performs the comparison of each element independently, we can write

$$\Pr(M_{RR-SI}(\mathbb{X},\mathbb{Y})=v) = \prod_{k=1}^{m} \Pr(V[k]=v[k]|x_k \in \mathbb{Y})\,. \tag{13}$$

Recall that ϵ can be defined as an upper bound for the ratio

$$\mathcal{L} \doteq \frac{\Pr(M_{RR-SI}(\mathbb{X},\mathbb{Y})=v)}{\Pr(M_{RR-SI}(\mathbb{X},\mathbb{Y}')=v)} \le e^{\epsilon}\,. \tag{14}$$

We need to find which combination of client data $\mathbb{X}$, neighbouring datasets $\mathbb{Y} \sim \mathbb{Y}'$ and output vector v maximize this ratio. We can plug (13) into (14) and get

$$\mathcal{L} = \prod_{k=1}^{m} \frac{\Pr(V[k]=v[k]|x_k \in \mathbb{Y})}{\Pr(V[k]=v[k]|x_k \in \mathbb{Y}')} \tag{15}$$

Let $y \in \mathbb{Y}$ and $y' \in \mathbb{Y}'$ be the element in which these data sets differ (i.e., $y \notin \mathbb{Y}'$ and $y' \notin \mathbb{Y}$). Let's consider different cases for y and y'.

1) If $y, y' \notin \mathbb{X}$, then all terms in the product in (15) are one, since $\mathbb{X} \cap \mathbb{Y} = \mathbb{X} \cap \mathbb{Y}'$. Therefore, in this case $\mathcal{L} = 1$.
2) If $y \notin \mathbb{X}$ but $y' \in \mathbb{X}$ then there will be a single non-one element in the product in (15), corresponding to the index k such that $x_k = y'$. The same happens if $y \in \mathbb{X}$ but $y' \notin \mathbb{X}$.
3) If both $y, y' \in \mathbb{X}$, however, there will be two terms in (15) that can be larger than one, so this is the worst case that we need to analyze.

Therefore, let's consider $y, y' \in \mathbb{X}$, and use indices k and k' such that $x_k = y$ and $x_{k'} = y'$. We have

$$\mathcal{L} \le \frac{\Pr(V[k]=v[k]|x_k \in \mathbb{Y})}{\Pr(V[k]=v[k]|x_k \notin \mathbb{Y}')} \cdot \frac{\Pr(V[k']=v[k']|x_{k'} \notin \mathbb{Y})}{\Pr(V[k']=v[k']|x_{k'} \in \mathbb{Y}')}\,. \tag{16}$$

By looking at the probabilities in (9) to (12) above, we can see that this is maximized when $v[k]=1$ and $v[k']=0$. In that case,

$$\mathcal{L} \le \frac{p+(1-p)q}{(1-p)q} \cdot \frac{p+(1-p)(1-q)}{(1-p)(1-q)} = e^{\epsilon}\,, \tag{17}$$

which concludes our proof. □

7.1.3. Security Analysis.

Theorem 5. *Algorithm 1 with server computation Algorithm 2 satisfies ϵ-SIM-CDiPSI as per Definition 7 for the mechanism M_{RR-SI}.*

Proof. We prove Theorem 5 in two steps. First, we prove that Algorithm 2 correctly produces the output of mechanism M_{RR-SI}. Second, we prove that we can simulate

4. Note that, in order to be comparable with previous works, we are assuming that ℓ is a constant and thus does not affect our asymptotic analysis.

(additional) output using only the party's input and the mechanism's output.

Correct output of M_{RR-SI}. The server flips a coin for randomized response for each element $x \in \mathbb{X}$. The set intersection is computed truthfully on "heads" by comparing to all elements that hashed to the same bin. On tails, the element is matched with a dummy element that necessarily results in a mismatch. Then another coin is flipped and if it is "heads", the mismatch result is negated by homomorphically adding a 1. Hence, Algorithm 2 returns the (encrypted) randomized response for each element x_i.

Simulation of (additional) output: Dummy elements from the client. The client includes several dummy elements to even the number of elements in the bins. The server cannot distinguish these dummy elements from real elements in the set and must also match these elements. This may result in "fake" matches for dummy elements, if the first coin shows up "tails" and the second one "heads". However, the client can simulate the number of "fake" matches from its set size and the protocol parameters, i.e., the number of dummy elements. Given the probabilities of the coin flips, which are deducible from ϵ, the client can simulate the corresponding number of random events. Since each random event is independent of the server's input, this leaks no information about the server's input.

Note that the server's view can be simulated by a number of semantically secure ciphertexts linear in the client's set size. □

7.1.4. Utility. Mechanism M_{RR-SI} returns the set intersection with both false negatives and false positives. Following (11) and (10), the probability of false positives is $(1-p)q$ while the probability of false negatives is $(1-p)(1-q)$. Note that, if we configure p and q to make any of these probabilities zero, then $\epsilon \to \infty$ according to (8). This proves that it is not possible to achieve a differentially private set intersection without both types of error.

Achieving low values of ϵ in DiPSI requires high false positive and false negatives rates, which are not reasonable in some use cases. We note that this issue is not unique to DiPSI, but happens with all mechanisms that reveal a differentially private intersection (our proof in Theorem 4 applies to any mechanism that hides the intersection with false positives and false negatives).

7.2. Analysis of ϵ-DiPSI-CA

7.2.1. Complexity. For the following complexity analysis we assume the DiPSI-CA algorithm uses the nested cuckoo hashing described in Section 4.5.2.

Theorem 6. *The communication complexity of DiPSI-CA where the hashing-to-bins method is nested cuckoo hashing is $\mathcal{O}(m)$ for m client elements and n server elements.*

Proof. Computing the cardinality does not require any additional communication beyond that of DiPSI and therefore the result follows from Theorem 2 as $\mathcal{O}(m)$. □

Theorem 7. *The computation complexity of DiPSI-CA where the hashing-to-bins method is nested cuckoo hashing is $\mathcal{O}(n+m)$ for m client elements and n server elements with $n > m \log m$.*

Proof. The additional computation required to perform DiPSI-CA is limited to the computation of the Laplace noise and the generation of m random values. Therefore, the computation complexity still follows from Theorem 3 and is $\mathcal{O}(n+m)$. □

7.2.2. Privacy Analysis.

Theorem 8. *Mechanism M_{LAP-CA} satisfies ϵ-DiPSI as per Definition 6.*

Proof. Let $S \doteq \sum_{k=1}^{m} V[k] = |\mathbb{X} \cap \mathbb{Y}| + L$, where $L \leftarrow_{\$} \text{Lap}(1/\epsilon)$, and therefore

$$\Pr(M_{LAP-CA}(\mathbb{X},\mathbb{Y}) = v) = \Pr(S = s|\, |\mathbb{X} \cap \mathbb{Y}|) \quad (18)$$

From the Laplacian distribution, we can write

$$\Pr(S = s|\, |\mathbb{X} \cap \mathbb{Y}|) = \frac{\epsilon}{2} \cdot e^{-\epsilon |s - |\mathbb{X}\cap\mathbb{Y}||}\,. \quad (19)$$

Therefore, we can prove that

$$\begin{aligned}\mathcal{L} &= \frac{\Pr(M_{LAP-CA}(\mathbb{X},\mathbb{Y}) = v)}{\Pr(M_{LAP-CA}(\mathbb{X},\mathbb{Y}') = v)} && (20)\\ &= \frac{\Pr(S = s|\, |\mathbb{X}\cap\mathbb{Y}|)}{\Pr(S = s|\, |\mathbb{X}\cap\mathbb{Y}'|)} && (21)\\ &= \frac{\frac{\epsilon}{2}\cdot e^{-\epsilon|s-|\mathbb{X}\cap\mathbb{Y}||}}{\frac{\epsilon}{2}\cdot e^{-\epsilon|s-|\mathbb{X}\cap\mathbb{Y}'||}} && (22)\\ &= e^{\epsilon\left(|s-|\mathbb{X}\cap\mathbb{Y}'||-|s-|\mathbb{X}\cap\mathbb{Y}||\right)} \le e^{\epsilon}\,, && (23)\end{aligned}$$

where the last step comes from the fact that, since $\mathbb{Y}$ and $\mathbb{Y}'$ are neighbouring, they can only differ in at most one element. This concludes our proof.[5] □

7.2.3. Security Analysis.

Theorem 9. *Algorithm 1 with server computation Algorithm 3 satisfies ϵ-SIM-CDiPSI as per Definition 7 for the mechanism M_{LAP-CA}.*

Proof. We prove Theorem 9 in the same two steps as Theorem 5.

Correct output of M_{LAP-CA}. Algorithm 3 computes the set intersection cardinality for each element in a batch. Hence, the sum of all elements in a batch is $|\mathcal{X} \cap \mathcal{Y}|$. It now adds Laplace noise $Lap(1/\epsilon)$ to the last element. To hide the sum in each element of a batch it adds a uniform random number from $\mathbb{Z}_t$ such that the sum over all elements is 0.

Simulation of output. Consequently, the (encrypted) message from the server to the client can be simulated as follows: Let s be the batch size. Choose $s-1$ uniform random elements from $\mathbb{Z}_t$ and let their sum be S. Set the last element in the batch to be $|\mathcal{X} \cap \mathcal{Y}| + Lap(1/\epsilon) - S$.

The view of the server can again be simulated using semantically secure ciphertexts. □

5. Note that the Laplace noise added in Algorithm 3 is a discrete (mod t) version of the Laplace noise. This does not affect our proof, since the fact that our mechanism is ϵ-DP for the continuous Laplacian (19) implies that it is also ϵ-DP when the observation is quantized.

7.2.4. Utility. Since the mechanism M_{LAP-CA} reports an aggregate value plus some added noise, it can provide high privacy levels without sacrificing utility. Formally, the noise added by M_{LAP-CA} is below $-\epsilon^{-1} \cdot \log(1-P)$, for a specified probability P. For example, using a high privacy level of $\epsilon = 0.1$, the noise is below 40 with a probability of 0.98. This amount of noise is negligible when the true set cardinalities are in the tens or hundreds of thousands elements, such as may be the case in our examples in Section 2.

8. Performance Evaluation

In the following, we evaluate DiPSI implemented with dual hashing.[6] We describe our parameter choices, how DiPSI compares with other approaches, and the computation and communication overhead.

8.1. Setup

We implement DiPSI using the beta version of HElib 1.0.0 by Halevi et al. [47], which uses the BGV [36] encryption scheme with bootstrapping. We benchmark the matching time, the total communication cost, and the maximum number of server elements in any bin. Our experiments are for unbalanced PSI where $m < n$ with client set size m and server set size n. We use $\ell = \log_2 n$ so every server element is representable with ℓ bits and a seemingly large plaintext modulus of $t = 40\,961$ to ensure the intersection cardinality fits within the modulus. We set a security of $k = 128$ bits, and a modulus chain length of $L = 300$ when $\ell \leq 16$, or $L = 450$ when $16 < \ell \leq 32$. These values of L support the required multiplicative depth, which is $\mathcal{O}(\log \ell)$, without bootstrapping. We set the cyclotomic field $c = 8\,192$ which gives us a slot count of $s = 4\,096$ slots. The cyclotomic field was chosen heuristically to work well for our evaluation; other choices may lead to better performance. We confirm the security of our simulation parameters with the HELib function `securityLevel()`, which returns a security level of 101.5 and 112.9 bits for $L = 300$ and $L = 450$, respectively. Our experiments were executed on a single-threaded process on an Intel E5-2650 clocked at 2.00 GHz with access to 256 GB of main memory.

8.2. Computation

Table 2 shows the runtime in seconds of DiPSI and DiPSI-CA (average and standard deviation of 10 runs), as well as previous approaches [8], [11], for $m = 2^{12} = 4\,096$ client elements (we chose $m' = 5\,535$ for [11] since that's the setting closest to ours). We show the microbenchmarks of our protocols for their initialization, the overall matching time (i.e., comparing all client batches with all server batches), the average time it takes to compare a single client and server batch, and the differential privacy operations of the algorithm. Our algorithms are considerably slower than previous work, since we do not perform the matching in parallel (HElib does not allow for easy parallelization). In practice, parallelizing could allow to reduce the matching time to a single per batch comparison, which would substantially reduce the running time of our algorithm. We show an estimation of the time it would take to run the algorithm while parallelizing the batch comparisons in parenthesis under the total running time of each algorithm.

Table 2. Runtime in Seconds of Protocols for $m = 2^{12}$

Protocol	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$
CLR [11]	-	1.7	8.7
PSWW [8]	1.2	8.5	120.7
DiPSI			
Init	7.7 ± 0.4	8.6 ± 0.4	11.3 ± 0.4
Matching	84.2 ± 7.2	521.0 ± 6.7	$11\,680.3 \pm 23.3$
(per batch)	(3.4 ± 0.07)	(4.07 ± 0.05)	(9.27 ± 0.02)
DP	0.08 ± 0.00	0.08 ± 0.01	0.08 ± 0.01
Total	92.2 ± 7.1	531.62 ± 6.6	$11\,719.4 \pm 23.2$
	(4.6 ± 0.4)	(6.6 ± 0.4)	(29.9 ± 0.6)
DiPSI-CA			
Init	7.8 ± 0.5	8.5 ± 0.4	11.2 ± 0.35
Matching	81.4 ± 1.0	520.2 ± 5.8	$11\,712.2 \pm 73.8$
(per batch)	(3.39 ± 0.04)	(4.06 ± 0.05)	(9.30 ± 0.06)
DP-CA	0.05 ± 0.02	0.06 ± 0.02	0.14 ± 0.02
Total	89.5 ± 1.3	530.7 ± 5.7	$11\,751.9 \pm 74.5$
	(4.7 ± 0.4)	(6.5 ± 0.4)	(30.4 ± 1.0)

Table 2 also shows that the running times of DiPSI and DiPSI-CA are almost identical. This is because the computational cost of our algorithms is largely dominated by performing the intersection (lines 19-22 in Alg. 2 and lines 10-13 in Alg. 3). The additional cost incurred by the coin flipping that achieves differential privacy in Alg. 2 or the cardinality computation and noise addition in Alg. 3 is negligible. This suggests that computing more complex predicates on top of the intersection will incur very little additional cost.

Figure 4 shows the computation time of Algorithm 2 for varying numbers of client and server elements $2^{10} \leq m \leq 2^{16}$ and $2^{12} \leq n \leq 2^{21}$. Note that the number of server elements is plotted on a logarithmic axis. As expected, the computation time grows linearly with the number of server elements. We also observe that the matching time is very large for $m = 2^{10}$, reaches its minimum around $m = 2^{12}$, and then increases with m. This is because increasing m decreases the number of elements in any bin on the server (see Figure 5), which means that overall fewer comparisons have to be computed. This phenomenon in combination with the chosen minimal batchsize of $s = 4\,096$ turns into a disadvantage for small $m = 2^{10}$ as the runtime is significantly higher than for $m = 2^{12}$.

Although these running times may be too long for some applications when no parallelization is implemented, there are use cases where they are sufficient (see Section 2).

8.3. Communication

Table 3 shows the communication cost in MB of DiPSI ($m = 4\,096$) and the approaches in [8], [11]. Our experiments also reveal that the public key alone consumes a baseline of about 60 MB for $L = 300$ and about 79 MB for $L = 450$. For $n = 2^{20}$ server elements our protocol requires only 232 MB of total communication costs as opposed to 2.54 GB as in the approach from Pinkas et al. [8]. The communication complexity of Chen

6. Code available at https://github.com/cryspuwaterloo/DiPSI.

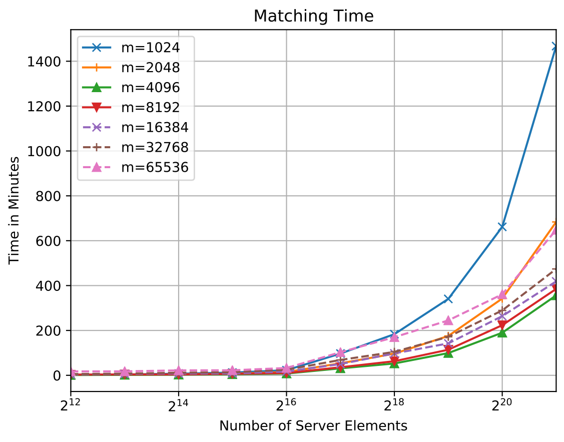

Figure 4. Total Computation Cost for DiPSI

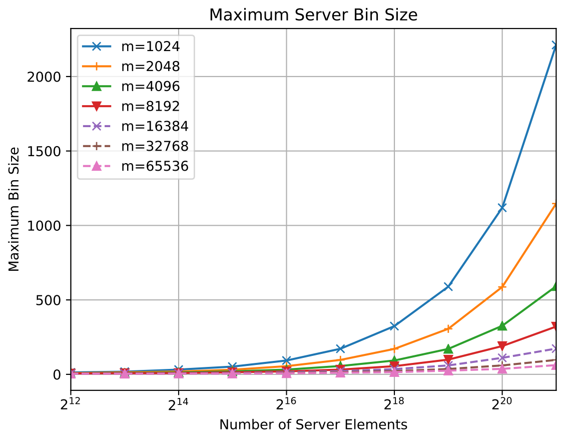

Figure 5. Maximal Number of Elements per Server Bin, μ

et al. [11] does not include the size of the public key, but instead only the ciphertexts, which partly explains why their cost is significantly lower than ours. Also, the approach in [11] does not allow to compute predicates on top of the intersection. We do not show the communication cost of DiPSI-CA, but it is smaller since the server only returns $\mathcal{O}(1)$ ciphertexts to the client instead of $\mathcal{O}(m)$.

Figure 6 shows the communication cost of DiPSI when varying both the number of client and server elements. As we explain in Section 8.1, in our evaluation we set $\ell = \log_2 n$. Therefore, as we increase n, the number of bits that we use to represent an element increases, and so does the communication cost (logarithmically with n, which shows linearly in the plot). Additionally, when $\ell > 16$ we increase the modulus chain length from $L = 300$ to 450 to accommodate the multiplicative depth of the circuit. This increase in L causes the sudden bandwidth increase in Figure 6. We note that, as predicted in Section 7.1.1, the communication cost only depends linearly on m, and not n, if ℓ is kept fixed.

In summary, in the balanced PSI setting ($n = m$), DiPSI cannot compete with other protocols that also allow the computation of predicates over the intersection [8].

Table 3. COMMUNICATION IN MB OF PROTOCOLS FOR $m = 2^{12}$

Protocol	$n = 2^{12}$	$n = 2^{16}$	$n = 2^{20}$	Binary Circuit
CLR [11]	-	≈ 3	≈ 6	✗
PSWW [8]	9	149	2540	✓
DiPSI	119	133	232	✓

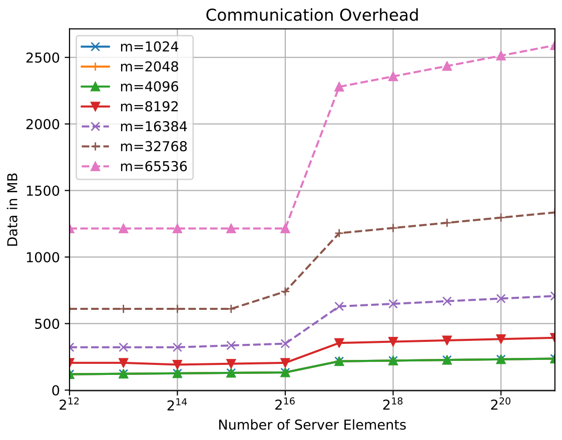

Figure 6. Total Communication Cost for DiPSI

However, DiPSI achieves optimal complexity for unbalanced sets, as shown in our experiments that increase the size of the larger set (n).

9. Conclusion

In this work, we present a new variant of private set intersection and present our new protocols for PSI. Our protocols go beyond protecting the elements outside of the intersection and compute differentially private results, providing protection for use cases where the intersection is also sensitive. We identify a number of real-world problem settings of interest where the stakeholders benefit from accepting a margin of variance in inferences based on the aggregate data in trade for protecting the set intersection. Such use cases include instances where the set intersection to be computed is unbalanced; such as in the case of medical research and genome databases. Our protocols are optimized to suit such settings where one dataset is much larger than the other while still achieving optimal computational and communication complexity. Additionally, our protocols have applicability beyond our initial predicates as the adaptability that follows from the circuit-based design enables us to support computation of arbitrary predicates over the intersection.

Acknowledgment

We gratefully acknowledge the support of NSERC for grants RGPIN-05849, RGPAS-507908, CRDPJ-531191, and DGDND-00085, the Royal Bank of Canada and European Union's Horizon 2020 Research and Innovation Programme under grant agreement n. 830927 (CONCORDIA) for funding this research.

References

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

[2] Bernardo A Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. *EC*, 99:78–86, 1999.

[3] Catherine Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*, pages 134–134. IEEE, 1986.

[4] Moti Yung. From mental poker to core business: Why and how to deploy secure computation protocols? In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 1–2, 2015.

[5] Mark Bergen and Jennifer Surane. Google and mastercard cut a secret ad deal to track retail sales, 2018. https://www.bloomberg.com/news/articles/2018-08-30/google-and-mastercard-cut-a-secret-ad-deal-to-track-retail-sales.

[6] Yves-Alexandre De Montjoye, Laura Radaelli, Vivek Kumar Singh, and Alex Pentland. Unique in the shopping mall: On the reidentifiability of credit card metadata. *Science*, 347(6221):536–539, 2015.

[7] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

[8] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based psi via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 125–157, Cham, 2018. Springer International Publishing.

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

[10] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21th USENIX Security Symposium*, pages 285–300, 2012.

[11] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1243–1255, New York, NY, USA, 2017. ACM.

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

[13] Arpita Ghosh, Tim Roughgarden, and Mukund Sundararajan. Universally utility-maximizing privacy mechanisms. *SIAM J. Comput.*, 41(6):1673–1693, 2012.

[14] Eric W. Burger, Michael D. Goodman, Panos Kampanakis, and Kevin A. Zhu. Taxonomy model for cyber threat intelligence information exchange technologies. In *Proceedings of the 2014 ACM Workshop on Information Sharing & Collaborative Security*, pages 51–60, 2014.

[15] Daniel Rigden and Xose Fernandez. The 2018 nucleic acids research database issue and the online molecular biology database collection. *Nucleic Acids Research*, 46(D1):D1–D7, 2018.

[16] Teri A. Manolio. Genomewide association studies and assessment of the risk of disease. *New England Journal of Medicine*, 363(2):166–176, 2010.

[17] Carmit Hazay and Kobbi Nissim. Efficient set operations in the presence of malicious adversaries. *Journal of cryptology*, 25(3):383–433, 2012.

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

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

[20] Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-n ot extension with application to private set intersection. In *Cryptographers' Track at the RSA Conference*, pages 381–396. Springer, 2017.

[21] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.*, 21(2):7:1–7:35, January 2018.

[22] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse ot extension. In *Annual International Cryptology Conference*, pages 401–431. Springer, 2019.

[23] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on {OT} extension. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 797–812, 2014.

[24] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 515–530, 2015.

[25] Moxie Marlinspike. The difficulty of private contact discovery. A company sponsored blog post, 2014. https://whispersystems.org/blog/contact-discovery/.

[26] Mark Manulis, Benny Pinkas, and Bertram Poettering. Privacy-preserving group discovery with linear complexity. In *Proceedsings of the 8th International Conference on Applied Cryptography and Network Security*, pages 420–437. Springer Berlin Heidelberg, 2010.

[27] Alex Davidson and Carlos Cid. An efficient toolkit for computing private set operations. In Josef Pieprzyk and Suriadi Suriadi, editors, *Information Security and Privacy*, pages 261–278, Cham, 2017. Springer International Publishing.

[28] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis, editors, *Cryptology and Network Security*, pages 218–231, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[29] Michele Ciampi and Claudio Orlandi. Combining private set-intersection with secure two-party computation. In Dario Catalano and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 464–482, Cham, 2018. Springer International Publishing.

[30] Sivakanth Gopi, Pankaj Gulhane, Janardhan Kulkarni, Judy Han-wen Shen, Milad Shokouhi, and Sergey Yekhanin. Differentially private set union. *arXiv preprint arXiv:2002.09745*, 2020.

[31] Xi He, Ashwin Machanavajjhala, Cheryl Flynn, and Divesh Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1389–1406. ACM, 2017.

[32] Adam Groce, Peter Rindal, and Mike Rosulek. Cheaper private set intersection via differentially private leakage. Cryptology ePrint Archive, Report 2019/239, 2019. https://eprint.iacr.org/2019/239.

[33] Cynthia Dwork. Differential privacy. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, pages 1–12, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[34] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. Computational differential privacy. In *Annual International Cryptology Conference*, pages 126–142. Springer, 2009.

[35] Xun Yi, Russell Paulet, and Elisa Bertino. *Homomorphic encryption and applications*, volume 3. Springer, 2014.

[36] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[37] Michael J Freedman, Carmit Hazay, Kobbi Nissim, and Benny Pinkas. Efficient set intersection with simulation-based security. *Journal of Cryptology*, 29(1):115–155, 2016.

[38] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, *Algorithms — ESA 2001*, pages 121–133, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[39] Udi Weider. Hashing, load balancing and multiple choice, 2016. https://udiwieder.files.wordpress.com/2014/10/hashbook.pdf.

[40] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009.

[41] Michael T Goodrich, Daniel S Hirschberg, Michael Mitzenmacher, and Justin Thaler. Fully de-amortized cuckoo hashing for cache-oblivious dictionaries and multimaps. *arXiv preprint arXiv:1107.4378*, 2011.

[42] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[43] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Manual for using homomorphic encryption for bioinformatics. *Proceedings of the IEEE*, 105(3):552–567, 2017.

[44] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1255. ACM, 2017.

[45] Valentin Fedorovich Kolchin, Boris Aleksandrovich Sevastyanov, and Vladimir Pavlovich Chistyakov. Random allocations. 1978.

[46] Norman Lloyd Johnson and Samuel Kotz. Urn models and their application; an approach to modern discrete probability theory. 1977.

[47] Shai Halevi and Victor Shoup. Helib-an implementation of homomorphic encryption. *Cryptology ePrint Archive, Report 2014/039*, 2014.