Differentially Private Two-Party Set Operations

Anonymous Author(s)

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 facilitates 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 optimally accurate and does so in optimal communication complexity. For a client set of size m and a server set of size n, where $m < n$, 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.

CCS CONCEPTS

• **Security and privacy** → *Public key encryption*;

KEYWORDS

Private Set Intersection (PSI), Differential Privacy

1 INTRODUCTION

Private set intersection (PSI) [15, 24, 32] can protect sensitive data when the intersection is non-sensitive. It is, for example, used by Google and a partner to compute ad conversions [44]. However, 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 [1]. 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 [10]. 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 [23, 37, 38] 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 [29]. Furthermore, even in the best case for previous protocols, the communication complexity is the sum of the sizes of the two databases [37].

We present the first circuit-based PSI protocol based on (leveled) homomorphic encryption. Our solution is theoretically 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 $O(m)$[1]. Our computation complexity approaches $O(n)$. Our differentially private output is optimally accurate [16] for set intersection cardinality and set intersection. Note that the most recent circuit-based PSI protocols [37, 38] have communication complexity at least $O(m+n)$ and previous PSI protocols based on homomorphic encryption [4, 5] cannot compute arbitrary circuits (as necessary for differential privacy) in addition to having computation complexity $O(nm)$.

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 6 multiplications for 32 bits and hence practically feasible with many homomorphic encryption schemes. Furthermore, we use vectorization of the plaintexts and a different hashing scheme than the theoretically optimal cuckoo hashing. These optimizations increase the theoretic complexity, but also increase the practical performance. 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 [4] or circuit-based [37] – we can still reasonably handle large data sets up to millions of elements, while these protocols [4, 37] do not protect privacy to the intersection.

In particular, our communication cost when comparing $m = 4096$ client elements to $n = 10^6$ server elements is only 232 MByte whereas other circuit-based approaches are much higher in terms of communication cost for like parameters. For example, the latest approach in [37] needs 2.5 GByte in 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 [37]).

Contributions. In summary, this paper contributes new PSI protocols based on leveled homomorphic encryption that

Anonymous Submission to ACM CCS 2019, Due 15 May 2019, London, England

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Actually, our communication complexity is $O(m\ell)$, where ℓ represents the bit length. However, we use the notation of recent related work [4, 5, 37, 38] and concentrate on the parameters n and m.

- 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 defines the operations we compute; including the concept of a differentially private set intersection. Related work on private set intersection and differential privacy is in Section 4. Section 5 contains preliminaries used in our constructions defined in Section 6, which also contains our analysis. The performance results of our implementation are discussed in Section 7, while Section 8 concludes our work.

2 USE CASES

Private set intersection (PSI) is a useful tool with many applications. As mentioned in the introduction Google and a partner organization use it to determine ad conversions [44]. This is the same application as Google and Mastercard are pursuing, but without the use of PSI [1]. However, in many cases not only the non-matching elements are sensitive, but also the elements in the intersection itself. Consider the damage to reputation the news report on [1] has done to Mastercard. The next necessary step to prevent this kind of reputational damage is to protect the set intersection itself.

In this paper, we propose to use a differentially private mechanism on the set intersection (cardinality). Differential privacy as a privacy measure is well suited for this purpose. On the one hand, it guarantees that inferences about any individual element in the database are limited. On the other hand, it ensures that aggregate data is accurate enough to perform meaningful analyses, such as ad conversions. 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.

Using our protocols the use case between Google and its partner, e.g. Mastercard, would then change as follows. Google and Mastercard identify the set of users that they have in common in a certain period of time using unique identifiers, e.g., their phone numbers. The common set would be users that have viewed Google ads from vendor V and that had Mastercard transactions with vendor V. They would then compute the sum of the credit card transactions. However, 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 where each joint element is multiplied by its transaction value – would conceal that information with the privacy parameter ϵ. Using our protocols, nothing but the differentially private sum (and one party's set size) is revealed.

There are many other use cases where our protection is beneficial. Consider the use of genome databases [40] in the use of 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 [30]. 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 his set and the database and the intersection cardinality of his set and all elements in the database that have a specific allele. The quotient determines 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. This use case has also imbalanced set sizes, since the set of queried patients is almost always much smaller than the set of patients in the database and hence highlights further the advantages of our protocols.

We emphasize that our mechanism allows the computation of almost arbitrary predicates over the set intersection. While in the next section we present set intersection and set intersection cardinality as the most common forms, often simple adaptations extend our work to further predicates. For example, the following use case is again a simple extension of our work presented here. Consider two network operations centers that try to determine joint cyber threats. As a first step they would like to determine whether they have any threat indicators [3] in common. This is a predicate over the set intersection cardinality, i.e. whether it is zero or not, which can be implemented as randomizing the computed cardinality by multiplication with a random, non-zero number. If the cardinality is zero, the product remains zero, but if the cardinality is non-zero, the product becomes a random, non-zero number. Note that, we could make this binary result differentially private by using our randomized response mechanism described in Section 3.1.1. The randomized response could even be one-sided, such that only empty, false intersections are introduced, since false non-empty intersections will be detected after the exchange of threat indicators.

In summary, the use of private set intersection is encouraging, but often further protection is needed. In this paper we present differentially private set operations and corresponding cryptographic protocols which can provide this protection with optimal communication and computation complexity as well as optimal accuracy.

3 DIFFERENTIALLY PRIVATE SET OPERATIONS

3.1 Differential Privacy

Definition 1. ((ϵ, δ)-Differential Privacy [12]). A randomized mechanism $S : \mathcal{D} \mapsto \mathcal{F}$ provides (ϵ, δ)-differential privacy ((ϵ, δ)-DP) if for all adjacent inputs $D, D' \in \mathcal{D}$, i.e., differing in one element, and all subsets $F \subseteq \mathcal{F}$

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

where the probability space is S's coin tosses.

The δ parameter indicates a probability of error in the privacy mechanism $S(\cdot)$ providing differential privacy for a element. Although in differential privacy literature, negligible values of δ are acceptable, we are interested in $\delta = 0$, commonly represented by ϵ-Differential Privacy. ϵ-Differential Privacy guarantees that for *every* run of the mechanism $S(\cdot)$, 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. [33] in our security analysis.

Definition 2. Simulation-Based Computational Differential Privacy (SIM-CDP privacy [33]). 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 $\mathrm{negl}(n)(\cdot)$, 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,

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

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

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

3.1.1 Randomized Response. Randomized response is a surveying technique developed in 1965 by Warner [41] for collecting statistics on sensitive topics where survey respondents want their responses to remain confidential. The technique is commonly described through this example. Assume a Yes/No question on a sensitive activity XYZ [13]. The respondent is asked to perform the following steps to answer the question "Have you engaged in XYZ during past week":

(1) Flip a coin, in secret
(2) If heads, then respond truthfully.
(3) If tails, then flip a second coin and respond "Yes" if heads and "No" if tails.

The procedure provides each respondent with very strong deniability for any "Yes" or "No" answer, and provides $(\ln 3, 0)$-differential privacy.

In our protocols, we exploit a general form of randomized response where the probability of heads coming up in the first and the second coin flipping are p and q respectively. We show how p and q values define the privacy. Assume a function f with a binary output, b. The *General randomized response* perturbs the output as follows. It tosses the first coin, and preserves the output or alters it based on the result. As the coin is biased, b is preserved with probability p and is changed with probability $1 - p$. If the latter is the case, the algorithm tosses the second coin to decide the value to replace b with. It replaces b with 0 with probability q and with 1 with probability $1 - q$. By calculating the probability of different events for this scenario, we can calculate the privacy provided by the *General randomized response.*

Theorem 3.1. *The* General randomized response *technique satisfies $\epsilon-$differential privacy, where*

$$\epsilon = \ln max\left(1 + \frac{p}{(1-p)(1-q)}, 1 + \frac{p}{q(1-p)}\right)$$

Proof. We show the perturbed function output b by b'. According to Definition 1 the difference in the probability of receiving any value of b' should be limited by ϵ for any change in the input. In other words, the following conditions should hold:

$$\frac{P(b'=1|b=1)}{P(b'=1|b=0)} = \frac{p+q(1-p)}{q(1-p)} \leq e^{\epsilon} \text{ and}$$

$$\frac{P(b'=0|b=0)}{P(b'=0|b=1)} = \frac{p+(1-p)(1-q)}{(1-p)(1-q)} \leq e^{\epsilon}.$$

Hence, we choose ϵ accordingly to satisfy the conditions, i.e. $\epsilon = \ln max(1 + \frac{p}{(1-p)(1-q)}, 1 + \frac{p}{q(1-p)})$. □

3.1.2 Laplace Mechanism. Let f be a numeric query that maps datasets to k real numbers. To define the differential privacy provided to f by Laplace mechanism, we first need to define *Neighboring Datasets* and *Sensitivity.*

Definition 3. Neighboring Datasets. Two datasets D and D' are considered neighbors, shown by $D \sim D'$, if they differ in one random variable; i.e. $D = \{X_1, \cdots, X_{i-1}, X_i, X_{i+1}, \cdots, X_n\}$ and $D' = \{X_1, \cdots, X_{i-1}, X'_i, X_{i+1}, \cdots, X_n\}$.

Definition 4. Sensitivity. The sensitivity of a function $f : \mathbb{N}^{|\mathcal{X}|} \rightarrow \mathbb{R}^k$, is defined as

$$\Delta f = max_{x,y\in\mathbb{N}^{|\mathcal{X}|}, |x-y|=1} = |f(x) - f(y)|.$$

The sensitivity of f is the maximum change in the function output resulted by replacing any individual's data with a different one. In other words, sensitivity is the maximum change in f, if we switch from any input dataset D to any neighbor D' of D.

Definition 5. Laplace Mechanism. Given any function $f : \mathbb{N}^{|\mathcal{X}|} \rightarrow \mathbb{R}^k$, the Laplace mechanism [13] is defined as: $M_L(x, f(.), \epsilon) = f(x) + (Y_1, \cdots, Y_k)$, where:

(1) Y_i are i.i.d. random variables drawn from $Lap(\Delta f/\epsilon)$
(2) $Lap(\Delta f/\epsilon)$ is the Laplace distribution centered with p.d.f $Lap(\Delta f/\epsilon) = Lap(x|\Delta f/\epsilon) = \frac{\epsilon}{2\Delta f} exp(-\frac{\epsilon}{\Delta f}|x|)$

It is proven that the Laplace mechanism described above preserves $(\epsilon, 0)-$ differential privacy [13].

Definition 6. Let M_{RR-SI} be the following differential privacy mechanism employing randomized response in the computation of set intersection: Given two sets $\mathcal{X}$ and $\mathcal{Y}$ where $\mathcal{Z} = \mathcal{X} \cap \mathcal{Y}$, compute the ϵ-randomized response for each element $x_i \in \mathcal{X}$, such that the truthful answer is whether $x_i \in \mathcal{Z}$.

Definition 7. Let M_{LAP-CA} be the following differential privacy mechanism employing Laplace noise in the computation of cardinality: Let $Lap(\Delta f/\epsilon)$ be Laplace noise for privacy parameter ϵ and sensitivity Δf. Given two sets $\mathcal{X}$ and $\mathcal{Y}$, compute $|\mathcal{X} \cap \mathcal{Y}| + Lap(1/\epsilon)$.

3.2 Set Operations

We first define privacy in the semi-honest model. For a deterministic function f, we say that a protocol π privately 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 8. Formally, we define a function f and two-party protocol for computing f, π. The view of the i^{th} 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 theprotocol π on inputs (x, y) is denoted $OUTPUT^{\pi}(x, y)$. The protocol π privately computes f if there exists polynomial-time algorithms, denoted Sim_1 and Sim_2, such that

$$Sim_1(x, f(x, y)) \stackrel{c}{=} VIEW_1^{\pi}(x, y, OUTPUT^{\pi}(x, y)) \text{ and}$$

$$Sim_2(y, f(x, y)) \stackrel{c}{=} VIEW_2^{\pi}(x, y, OUTPUT^{\pi}(x, y)).$$

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 9. *Private Set Intersection (PSI).* For the sets $\mathbb{X}$ and $\mathbb{Y}$ compute $\mathbb{X} \cap \mathbb{Y}$ such that:

- *One-Sided PSI:* The client learns $\mathbb{X} \cap \mathbb{Y}$ while the server learns no additional information and privacy is met as per Definition 8.
- *Two-Sided PSI:* The client learns $\mathbb{X} \cap \mathbb{Y}$, the server learns $\mathbb{X} \cap \mathbb{Y}$, and privacy is met as per Definition 8.

In both the one-sided and two-sided PSI, the client learns nothing additional about the elements belonging to $\mathbb{Y} \setminus (\mathbb{X} \cap \mathbb{Y})$ and the server learns nothing additional about the elements belonging to $\mathbb{X} \setminus (\mathbb{X} \cap \mathbb{Y})$.

Definition 10. *Differentially Private Set Intersection.* For the sets $\mathbb{X}$ and $\mathbb{Y}$ compute $\mathbb{X} \cap \mathbb{Y}$ such that:

- *One-Sided DiPSI:* The client learns the set intersection $\mathbb{X} \cap \mathbb{Y}$ perturbed by a mechanism such as M_{RR-SI} and the conditions for Definition 2 are met.
- *Two-Sided DiPSI:* The client learns the set intersection $\mathbb{X} \cap \mathbb{Y}$ perturbed by a mechanism such as M_{RR-SI}, the server learns $\mathbb{X} \cap \mathbb{Y}$ perturbed by a mechanism such as M_{RR-SI}, and the conditions for Definition 2 are met.

Definition 11. *Private Set Intersection Cardinality (PSI-CA).* For the sets $\mathbb{X}$ and $\mathbb{Y}$ compute $|\mathbb{X} \cap \mathbb{Y}|$ such that:

- *One-Sided PSI-CA:* The client learns $|\mathbb{X} \cap \mathbb{Y}|$ while the server learns no additional information and privacy is met as per Definition 8.
- *Two-Sided PSI-CA:* The client learns $|\mathbb{X} \cap \mathbb{Y}|$, the server learns $|\mathbb{X} \cap \mathbb{Y}|$, and privacy is met as per Definition 8.

In both the one-sided and two-sided PSI the client and server learn nothing additional beyond $|\mathbb{X} \cap \mathbb{Y}|$.

Definition 12. *Differentially Private Set Intersection Cardinality (DiPSI-CA).* For the sets $\mathbb{X}$ and $\mathbb{Y}$ compute $|\mathbb{X} \cap \mathbb{Y}|$ such that:

- *One-Sided DiPSI-CA:* The client learns the intersection cardinality $|\mathbb{X} \cap \mathbb{Y}|$ perturbed by a mechanism such as M_{LAP-CA}, while the server learns no additional information, and the conditions for Definition 2 are met.
- *Two-Sided DiPSI-CA:* The client learns the intersection cardinality $|\mathbb{X} \cap \mathbb{Y}|$ perturbed by a mechanism such as M_{LAP-CA}, and the server learns the intersection cardinality $|\mathbb{X} \cap \mathbb{Y}|$ perturbed by a mechanism such as M_{LAP-CA}, and the conditions for Definition 2 are met.

4 RELATED WORK

Early constructions for private set intersection employed public-key cryptography [24, 32] to some success with respect to communication cost, however, their computation overheads were substantial. Since then, constructions employing oblivious polynomial evaluation (OPE) [15], Oblivious Pseudo-Random Functions (OPRF) [21], and Oblivious Transfers (OT) [25, 34, 39] have been proposed. The work of Chen et al. [5] is similar to ours in that it employs homomorphic encryption in place of Oblivious Transfer in their polynomial based approach. Our work also employs homomorphic encryption, however, we use a circuit-based approach to facilitate adaptability.

4.1 Private Set Intersection

As defined in Section 3, PSI constructions can be designed for either the one-sided or two sided setting. Additionally, PSI constructions can be designed with optimizations for balanced and unbalanced use cases. Balanced PSI refers to the setting where the participating parties, which we refer to as the client and server, possess datasets of comparable sizes. In contrast to this, unbalanced PSI refers to a 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 [31]. The PSI protocol from Pinkas et al. [39] considers both balanced and unbalanced use cases, but in the unbalanced case is outperformed in terms of communication complexity by Chen et al. [5]. Although performing well on communication complexity, the construction due to Chen et al. [5] is limited in that it is unable to facilitate the computation of predicates. That is, the construction cannot be efficiently modified to support the differential private setting we are working in and does not support additional computation over the intersection such as that found in the 2018 paper from Pinkas et al. [38]. Furthermore, the 2019 paper from Pinkas et al.[2] [37] also provides adaptability in terms of computation over the intersection in addition to optimizing the communication and computation complexity over that of the 2018 paper. Our construction has the adaptability for computation of predicates observed in Pinkas et al. [37, 38], while optimizing communication complexity.

4.2 Private Set Intersection with Computation

For private set operations, protocols for intersection, union, and cardinality are defined by Davidson and Cid [8]. Protocols also exist for PSI threshold cardinality, which returns whether or not the intersection is greater than a certain threshold [9, 38]. Finally, Ciampi and Orlandi's protocol is designed to compute arbitrary functions on set intersections [7]. 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.

4.3 Differentially Private Set Computations

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. [22], with techniques further improved by Groce et

[2]Note that the Pinkas et al. in the 2018 paper is not the same group of authors as in the 2019 Pinkas et al. paper.

al. [19] for PSI, employs differential privacy and secure computation to solve this problem. The techniques used by He et al. [22] 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 differs 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 SIM-CDP model.

5 PRELIMINARIES

5.1 Somewhat 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. [43], 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 [2]. 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. A HE scheme typically implements the following interface:

- `KeyGen`: Generates the secret and public key.
- `Encypt`: Encodes and encrypts plaintexts into ciphertexts.
- `Evaluate`: Evaluates a polynomial function on a ciphertext.
- `Decrypt`: Decrypts a ciphertext back into a plaintext.

5.2 PSI via Hashing-to-Bins

Constructions exist for performing PSI using hash functions to organize the elements into 'bins' within a table [14, 36, 39]. 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 *max* 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 *max* 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 [38]. Note that due to our use of homomorphic encryption the use of hashing is greatly simplified, since only one party needs to reveal its (encrypted) elements in a bin.

5.3 Collision Handling and Hashing-to-Bins

In this section, we discuss two of the possible techniques for handling collisions when performing hashing-to-bins. In Section 6.2 we present a variant of the cuckoo hashing presented below that can be used in our construction for differentially private set operations.

5.3.1 Cuckoo Hashing. Cuckoo hashing [35, 42] 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 and define s as the stash. Both T_1 and T_2 are of size $(1+\epsilon)n$, where ϵ is a chosen constant value, for example $\epsilon = 0.6$[3]. 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. 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 $O(1)$. See Kirsch et al. [27] for the analysis of using cuckoo hashing with a stash in this manner.

5.3.2 Dual Hash Function with Bin Sizes $\geq$ 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

[3]Note that this ϵ is distinct form the ϵ used for differential privacy.

an agreed upon *max* number of bin elements where $max \geq 1$. 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. It is necessary when employing such a hash function for privacy efforts to prevent revealing which bins had elements hashed to them as this leaks information about the hashed data set. After hashing all of the inputs, the client appends dummy elements for each bin in T until all bins hold *max* elements so as not to leak information about the data set in this manner.

5.3.3 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 hash all n elements with $H_2(x)$. When used with our batching strategy presented in Section 6, the resulting process requires $O(2nc)$ comparisons to determine the intersection, where c represents the bin capacity *max*. Note that $c = max = 1$ for conventional cuckoo hashing and $c = max \geq 1$ for the dual hashing variant. Briefly, the intuition for $O(2nc)$ 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.

6 CONSTRUCTIONS FOR DIFFERENTIALLY PRIVATE SET OPERATIONS

In this section, we present constructions for two differentially private set operations. The operations, set intersection and set intersection cardinality, use the same functionality for the encryption and decryption stages, but have their own specific algorithms for computation.

6.1 Overview

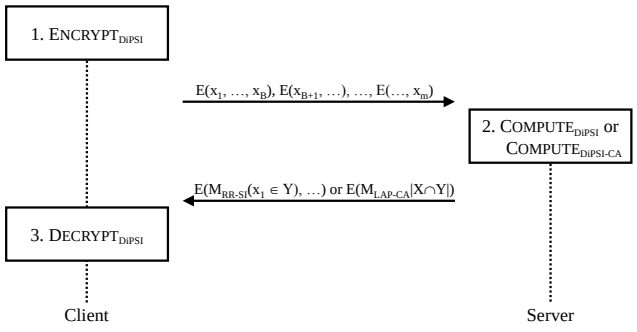

Figure 1: DiPSI Protocol Overview

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 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. [4, 5]. 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. A multi-valued attribute where a 0 encodes a match and any other random number a mismatch, as computed by polynomial-based PSI protocols, cannot be negated with low multiplicative depth circuits and hence limits the predicates that can be computed efficiently. 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. We hash the elements into bins before matching them. This keeps the computation cost at $O(n)$, since the server needs to match its elements only a constant number of times (see Section 6.2 for details and analysis). We use ciphertext vectors, but spread the bits of a ciphertext over multiple ciphertext vectors (see Section 6.3 for details). This avoids rotation of ciphertext elements and saves noise budget for multiplications. Similarly, we optimize the differentially private mechanisms, use dummy elements in the randomized response and spread the set intersection cardinality over the entire vector (see Sections 6.4 and 6.5 for details and analysis).

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. These are mechanisms that are practically relevant in applications, such as analysis of ad conversions. 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 Nested Cuckoo Hashing for DiPSI

A modified version of cuckoo hashing with a stash, called nested cuckoo hashing, was proposed by Goodrich et al. [17]. The modified hashing structure employs a primary cuckoo structure with a secondary cuckoo structure in place of the primary structure's stash. Recall from Section 6.2 that after a threshold number of attempts at placing an element in a table have passed, the element is placed in the stash. For nested cuckoo hashing, after a threshold number of attempts at placing the element in the primary cuckoo structure, the secondary cuckoo structure is employed. 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. [17].

Unlike nested cuckoo hashing, Kirsch et al. [27] showed that, for a stash of constant size c, the probability the stash overflows is at most $O\left(m^{-(s+1)}\right)$ for regular cuckoo hashing. Goodrich and Mitzenmacher [18] showed that the probability of an overflow is negligible when the stash is of size $O(\log m)$. However, since neither of the previous options result in a negligible failure probability with a constant size stash, we propose using a nested hash structure such that $H_1(x)$ and $H_2(x)$ are the hash functions corresponding to the primary cuckoo structure which maps to tables T_1 and T_2 or the stash s_1 to achieve the desired properties for negligibility. The stash s_1 of the primary cuckoo structure is actually the secondary cuckoo structure with hash functions $H_3(x)$ and $H_4(x)$, tables T_3 and T_4, and stash s_2. Tables T_1 and T_2 are each of size $(1+\epsilon)m$ while tables T_3 and T_4 are each of size $(m+\epsilon m)^{2/3}$, where m is the number of client elements.

We propose using nested cuckoo hashing to bin elements for DiPSI to facilitate matching as per hashing-to-bins. We demonstrate that nested cuckoo hashing can be used for DiPSI, with negligible probability of failure, such that we require only $O((4+c)n)$ comparisons, where n is the number of server elements and c is a constant representing the size of the secondary stash.

As with the use of conventional cuckoo hashing (described in Section 5.3.3), while the client performs nested cuckoo hashing, the server must perform simple hashing. Every server element in the set $\mathbb{Y}$ must be hashed using the primary structures two hash functions, $H_1(x)$ and $H_2(x)$, and every server element must be hashed with the secondary hash functions $H_3(x)$ and $H_4(x)$.

Theorem 6.1. *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 $|\mathbb{X}| = m$ and $|\mathbb{Y}| = n$. Define a nested cuckoo hashing structure where $H_1(x)$ and $H_2(x)$ are hash functions mapping to tables T_1 and T_2 of size m, $H_3(x)$ and $H_4(x)$ are hash functions mapping to tables T_3 and T_4 of size $m^{2/3}$, and s is a stash of constant size c. The intersection $\mathbb{X} \cap \mathbb{Y}$ can be computed using $O(n)$ comparisons.*

Proof. The server hashes all n elements using $H_1(x)$ and $H_2(x)$ to tables T_1 and T_2 of size m. The number of server elements n is much larger than the number of client elements. Therefore, we can use the classic balls-into-bins problem [26, 28] such that if $n > m \log m$, the expected maximum load for any particular bin can be calculated as

$$\frac{n}{m} + \sqrt{\frac{n \log m}{m}}.$$

Compare the single element in the i^{th} bin from the client to the $n/m + \sqrt{(n \log m)/m}$ elements in the i^{th} bin from the server for T_1, T_2, T_3, T_4, and s. Then the total number of comparisons is

$$(4+c)m\left(\frac{n}{m} + \sqrt{\frac{n \log m}{m}}\right), \text{ where } c \text{ is the size of the stash } s.$$

Since $n > m \log m$ we have that the required number of comparisons using this method is $O(n)$.

□

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. Additionally, Figure 2 illustrates the client encoding process with respect to bins, bin elements, and batches.

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. For the purpose of the algorithms representing the implementation, assume the hash function used is dual hashing where the hash functions are $H_1(x)$ and $H_2(x)$ and the agreed upon table size is m. The client hashes each element x in its set $\mathbb{X}$ using the process described in Section 5.3.2. After the hashing process, the client has a table T containing m bins that each hold *max* bin elements. Recall that if a bin does not hold *max* 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 *max* elements.

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

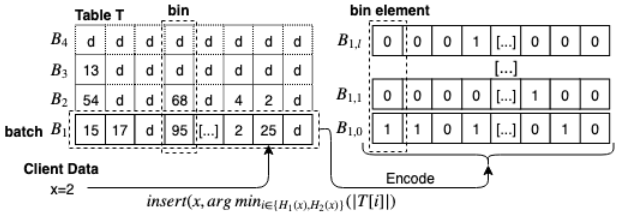

Figure 2: Client Side Plaintext Processing

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 encryption through allowing the processing of multiple plaintext in a batch.

Batching occurs over each table generated while hashing-to-bins. The client generates batches from each of the tables such that the i^{th} batch for a table contains the i^{th} bin element from every bin in that table.

Each batch is encoded before being encrypted. A batch contains m batch elements each of length ℓ. Let $b_1, b_2, \ldots, b_m$ represent the m elements in a batch. These elements are encoded to produce

ℓ vectors such that the i^{th} vector contains the the i^{th} bit from each of the batch elements. For example, let a batch B 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_1 consists of the first bit from each element, producing $vec_1 = 0111$, while the remaining vectors are $vec_2 = 1001$, $vec_3 = 0011$.

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 $B_{1,0}$ represents the most significant bit of the element 15 in the first bin. The second bit in $B_{1,0}$ represents the most significant bit of the element 17, and so on.

Once encoded as vectors, the batched elements are ready to proceed to the encryption stage.

6.3.3 Homomorphic Encryption. We use the Brakerski-Gentry-Vaikuntanathan (BGV) [2] scheme implemented in HELib[4]. Their implemented version supports bootstrapping, but we use it as a leveled homomorphic encryption scheme. Furthermore, we use ciphertext packing and encrypt $s = 4096$ client elements per ciphertext. Furthermore, we use a plaintext modulus of $t = 40961$ 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 [11]. In contrast to the approach taken by Chen et al. [6] who also use HE for DiPSI, our computation uses a binary circuit to allow for arbitrary predicates that can be implemented with a binary circuit to be computed on top of the DiPSI.

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

1: /* $H_1(x)$ and $H_2(x)$ are hash functions */
2: /* T_c is a table of size m for storing the hashed elements */
3: /* Let $max = \frac{\log m}{\log \log m}$ is the maximum capacity of any bin in the table */
4: /* Hashing-to-bins using dual hash as per the implementation */
5: **for each** Client element $x \in \mathbb{X}$ **do**
6: Compute $H_1(x)$ and $H_2(x)$
7: Append x to whichever of $T_c[H_1(x)]$ and $T_c[H_2(x)]$ contains the fewest elements
8: Append dummy elements to fill each bin in T_c to max
9: /* Batch hashed values */
10: **for each** batch $B_i \in T_c$ **do**
11: **for** $j = 1$ to $\ell = 32$ **do** // 32 bits in a batch element
12: /* $b_k[j]$, the j^{th} bit of the k^{th} batch element */
13: $vec_j = b_1[j]b_2[j] \ldots b_m[j]$
14: $c_j = \text{Enc}(vec_j)$ // See Section 6.3.3
15: Client sends set containing ciphertexts c_j to server

6.4 Differentially Private Set Intersection (DiPSI)

Details for computing the differentially private set intersection between two sets $\mathbb{X}$ and $\mathbb{Y}$ are presented in this section. The procedure for calculating matches on the server side, along with the proofs for communication complexity, computation complexity, and the security of *DiPSI* can be found below.

[4]https://github.com/homenc/HElib/

6.4.1 Compute for DiPSI. Our first protocol, *DiPSI* computes the private set intersection between two sets $\mathbb{X}$ and $\mathbb{Y}$ such that the result is differentially private. For the purpose of this section and for Section 7 we evaluate the protocol for the case where the hashing function being employed is dual hashing. After agreeing on two hash functions, H_1 and H_2, the client executes Algorithm 1 over their dataset of size m. The next stage, where the server hashes its elements to bins, is straightforward, with the exception of the requirement to force non-matches based on a coin flip.

Recall that in Algorithm 1 the client constructed batches of size m that were sent to the server in the form of vectors. Conceptually, the differentially private mechanism employed in Algorithm 2 consists of two stages. In the first stage we want to flip a coin for each client element such that if the result is "*heads*" we return whether or not a server element and a client element match. If, however, the result of the coin flip is "*tails*", we instead flip a second coin. If the result of the second coin is "*heads*" then return that the element does not have a match, else return that it does have a match. Our optimizations for communication complexity, in the form of batching, means that forcing a match as per the first coin flip requires the server to create an instance of its own batch where every element in the corresponding bin is set to a dummy element. So, for every element sent by the client, the server generates a parallel version of its data based on a coin flip and keeps track of the results of the coin flip for the second stage of the differential privacy mechanism.

Upon completion of the first coin flip stage, represented by lines 3-12 in Algorithm 2, the matching phase begins. The stages of match computation are illustrated in Figure 3, but we describe the stages further here. The procedure we describe applies to each batch sent by the client.

The matching process applies to each batch sent from the client. We illustrate the process for an individual batch in Figure 4 and describe the process below. Matches are computed first at the bit level, then at an element level (across all bits), and finally summed to merge the results of comparing different potential server elements.

Define a server batch that contains ℓ vectors as $vec_{j,i}$, where a vector $vec_{j,i}$ belongs to the j^{th} server batch and contains the i^{th} bit from each bin. Each client batch must be compared with every server batch, as illustrated by the boxes in Figure 3 where the first box exclusively contains vectors $vec_{j,1}$ from batch one and the last box exclusively contains $vec_{j,\mu}$ from batch μ. For any of the hash methods we have discussed, where the server employs simple hashing, the expected number of server elements mapped to a particular bin can be computed as $\mu = \theta(\frac{n}{m} + \sqrt{\frac{n \log m}{m}})$ (See the proof of Theorem 6.1's use of the balls-to-bins problem). In the case of dual hashing, for example, we can expect 2μ server elements per client bin as each server element must be hashed with each hash function and each hash function has an expected bin load of μ given n server elements and m bins in a table. The server therefore constructs 2μ batches.

A particular client batch consists of s ciphertexts, where s represents the batch size.[5] Each ciphertext c_j, belonging to a batch, contains m elements. The j^{th} ciphertext is a vector that contains m

[5]Note that unless otherwise stated we use a batch size of $s = m$.

elements where the i^{th} element is j^{th} bit from the i^{th} client bin in the batch (as illustrated in Figure 2). To compute the match at a bit level, XOR each client ciphertext c_j with the negation of a server vector vec_j, 1 to produce a bit match vector $M_{1,1}$. Next, AND every bit match vector $M_{j,k}$ for $j = 1$ to ℓ. This process, illustrated in each of the dotted boxes in Figure 3, is repeated with every client ciphertext c_j and every server vector vec_j, k, to produce resulting resulting vectors M_k, where $k = 1$ to μ. Once M_k has been computed for $k = 1$ to μ, the remaining step is to compute the final match vector $M = \sum_{k=1}^{\mu} M_k$.

The decryption process on the client side proceeds as described in Section 6.6.

6.4.2 Complexity. For the following complexity analysis we assume the *DiPSI* algorithm uses the nested cuckoo hashing described in Section 6.2.

Theorem 6.2. *The communication complexity of DiPSI where the hashing-to-bins method is nested cuckoo hashing is $O(m)$ for m client elements and n server elements.*

Proof. Trivially, T_1, T_2, T_3, T_4, and the stash s must be sent encoded from the client to the server. T_1 and T_2 are each of size m, T_3 and T_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 $O(m)$. □

Theorem 6.3. *The computation complexity of DiPSI where the hashing-to-bins method is nested cuckoo hashing is $O(n)$ for m client elements and n server elements.*

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 6.1, computing the set intersection when the client uses nested cuckoo hashing while the server employs simple hashing requires $O(n)$ comparisons. The remaining m decryptions do not significantly contribute to the computation costs such that the total computation costs can be computed as

$$m + 4n + O(n) + m.$$

Therefore, the computation cost for DiPSI when using nested cuckoo hashing is $O(n)$. □

6.4.3 Security Analysis.

Theorem 6.4. *Algorithm 1 with server computation Algorithm 2 satisfies ϵ-SIM-CDP for the mechanism M_{RR-SI}.*

Proof. The server flips a coin for randomized response for each element x_i. The set intersection is computed truthfully on "tails" by comparing to all elements that hashed to the same bin. On heads, 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 ComputeDiPSI

```
/* H_1(x) and H_2(x) are hash functions */
/* T_s is a table of size m for storing the hashed elements */
for each Server element y ∈ 𝕐 do
    Append y to the bin T_s[H_1(y)] and to the bin T_s[H_2(y)]
for each client ciphertext c_i do
    for each vector element e_j in c_i do
        p ←$ {0, 1}                    //Flip a coin
        if p = 0 then
            No change to server plaintexts
        else
            Set all elements in server bin T_s[j] to dummy elements (no matches)
            Save flag of p_{i,j} = 1 to indicate forced no match
/*Batch hashed values */
for each batch B_i ∈ T_s do
    for j = 1 to ℓ = 32 do    //32 bits in a batch element
        /* b_k[j], the j^th bit of the k^th batch element */
        vec_{j,i} = b_1[j]b_2[j] . . . b_m[j]
/*Compute intersection for each B̂ containing ciphertexts c_1, c_2, . . . , c_ℓ */
for each batch B̂ from client do
    /*Iterate over each batch for the server */
    for each server vector vec as server batch i do
        for j = 1 to ℓ do          //ℓ bits in a batch element
            M_{j,i} = c_j ⊕ ¬vec_{j,i} //C_j is the j^th ciphertext from the current batch
        M_j = ∧_{b=1}^{ℓ} M_{j,b}
    M = ∑ M_j                      //Determine if at least one server element matched
/*Differential privacy */
for each M computed do             //There is one for each batch
    for j = 1 to m do
        if P_{i,j} = 1 then        //Forced match
            q ←$ {0, 1}            //Flip a fair coin
            Construct a vector v of 0's of length m where the j^th element is q
            M = M + v              //Force result as per coin flip
```

Algorithm 2 returns the (encrypted) randomized response for each element x_i.

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 both coins show up "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. □

6.5 Differentially Private Set Intersection Cardinality (DiPSI-CA)

We now present details on the predicate specific computation for computing private set intersection cardinality. This section includes details on variations between *DiPSI* and *DiPSI-CA* as well as proofs for communication complexity, computation complexity, and security.

6.5.1 Computation for 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

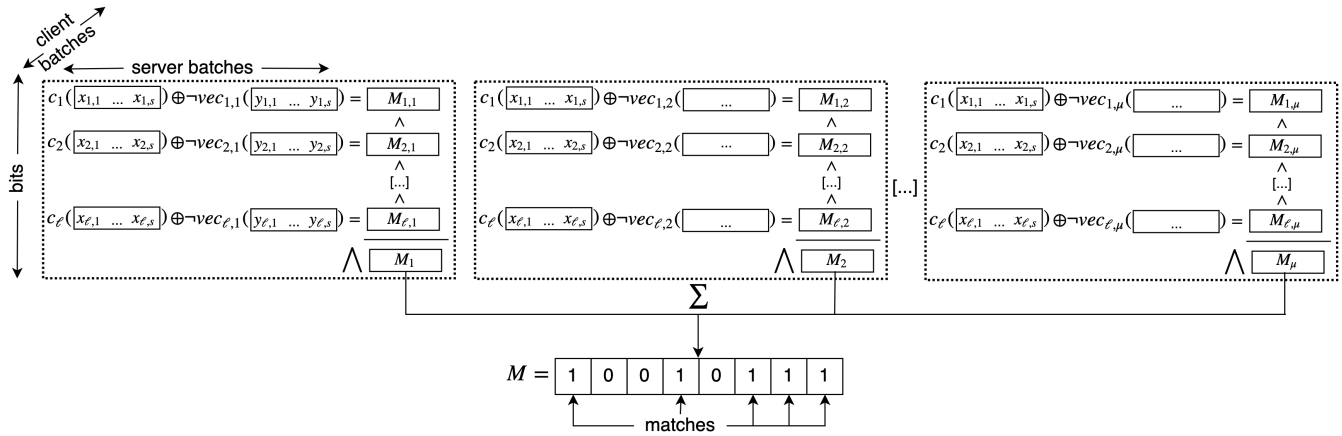

Figure 3: Server Side Matching Protocol Computations

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 25-31 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 $\hat{B}_i$ provided by the client. The server first sums all of the M_i vectors together to produce a result vector R, where the length of R corresponds to the number of bins in a batch. Next, the server generates a series of random values, one for each bin, such that the random values sum to zero. This way when we add one of the random values to each bin, the overall cardinality is unchanged. Finally, the server constructs an additional vector where the value in a random bin within it is assigned a newly generated Laplace noise value $\mathcal{L}$. This additional noise is then summed with the result vector to produce the final vector which is then sent back to the client.

Decryption follows as described in Section 6.6.

6.5.2 Complexity. For the following complexity analysis we assume the *DiPSI-CA* algorithm uses the nested cuckoo hashing described in Section 6.2.

Theorem 6.5. *The communication complexity of DiPSI-CA where the hashing-to-bins method is nested cuckoo hashing is $O(m)$ 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 form Theorem 6.2 as $O(m)$. □

Theorem 6.6. *The computation complexity of DiPSI-CA where the hashing-to-bins method is nested cuckoo hashing is $O(n)$ for m client elements and n server elements.*

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 6.3 and is $O(n)$. □

Algorithm 3 ComputeDiPSI-CA

1: /* $H_1(x)$ and $H_2(x)$ are hash functions */
2: /* T_s is a table of size m for storing the hashed elements */
3: /* Server hash-to-bins */
4: **for each** Server element $y \in \mathbb{Y}$ **do**
5: Append y to the bin $T_s[H_1(y)]$ and to the bin $T_s[H_2(y)]$
6: /* Batch hashed values */
7: **for each** batch $B_i \in T_s$ **do**
8: **for** $j = 1$ to $\ell = 32$ **do** // 32 bits in a batch element
9: /* $b_k[j]$, the jth bit of the kth batch element */
10: $vec_{j,i} = b_1[j]b_2[j]\ldots b_m[j]$
11: /* Compute intersection for each $\hat{B}$ containing ciphertexts $c_1, c_2, \ldots, c_\ell$ */
12: **for each** batch $\hat{B}$ from client **do**
13: /* Iterate over each batch for the server */
14: **for each** server vec as server batch i **do**
15: **for** $j = 1$ to ℓ **do** // ℓ bits in a batch element
16: $M_{j,i} = c_j \oplus \neg vec_{j,i}$ // C_j is the j^{th} ciphertext from the current batch
17: $M_j = \wedge_{b=1}^{\ell} M_{j,b}$
18: $M = \sum M_j$ // Determine if at least one server element matched
19: /* Compute cardinality */
20: Sum all match vectors M together as result vector R
21: /* Add differential privacy */
22: /* t =40961 is the plaintext modulus for the homomorphic encryption */
23: Add a random value $\hat{r}$ to each element in R such that $\sum_{i=1}^{i=m} \hat{r}_i = 0 \mod 40961$
24: Generate Laplace noise $\mathcal{L} = Lap(1/\epsilon)$
25: Construct a vector v of length m where the only non-zero value is the m^{th} element set to the value for $\mathcal{L}$
26: $R = R + v$ // Add Laplace noise
27: Return result vector R to client

6.5.3 Security Analysis.

Theorem 6.7. *Algorithm 1 with server computation Algorithm 3 satisfies ϵ-SIM-CDP for the mechanism M_{LAP-CA}.*

Proof. 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.

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. □

6.6 Decryption

For DiPSI the client receives one result ciphertext per l ciphertexts that were sent to the server. The client decrypts each result ciphertext to a plaintext vector $\{0,1\}^s$, where $s = 4096$ is the batchsize. Subsequently, a 1 at position i indicates that the client element i of that batch probably[6] matched with some server element. The client may discard those results for dummy elements which were used to fill up the bins, as the randomized response used as part of the differential privacy protects each match separately.

For DiPSI-CA, the client receives exactly one result ciphertext which encodes the set intersection cardinality as the sum over all elements. The client decrypts the result to a vector of natural numbers $\{0,..,t-1\}^s$ where t is the chosen plaintext modulus. Recall that the client is unable to associate the matching cardinality for any subset of client inputs because the differentially private mechanism added a random noise to each slot individually that adds up to zero modulo the plaintext modulus, hence not affecting the resulting set cardinality. Thereafter, the Laplacian noise is added to any random slot which protects the cardinality. Finally, the client obtains the differentially private set cardinality simply by taking the sum over all elements modulo the plaintext modulus.

7 PERFORMANCE EVALUATION

Our results obtained from running *DiPSI*, where *DiPSI* is implemented with dual hashing are presented in the following. We describe our parameter choices, how *DiPSI* compares with other approaches and how large the overhead imposed by the HE framework is in practice. Furthermore, we show hwo are approach is able to scale better then previous appraoches.

7.1 Setup

In our evaluation, we used the beta version of HElib 1.0.0 by Halevi et al. [20], which implements the BGV [2] scheme with additional bootstrapping functionality. For the *DiPSI* protocol, we benchmarked the matching time, total communication cost, and the length of the maximum number of server elements in any bin. The goal is to compare our approach to other circuit-based approaches in practice. We chose a plaintext modulus of $t = 40961$, a security of $k = 128$ bits, a modulus chain length of $L = 300$ for a bit length of $\ell \leq 16$, and $L = 450$ for $16 < \ell \leq 32$. We set the cyclotomic field to $m = 4096$ which gives us a batchsize of $s = 4096$ slots. We chose these parameters for HELib in all experiments, but optimized the parameters of the *DiPSI* protocol, like the bit-length, for each experiment. Our experiments were executed within a single-threaded process on an Intel E5-2650 clocked at 2.00GHz with access to 256GB of main memory.

7.2 Computation

For computation, the results show that our approach does not compete with previous approaches [5, 38]. To compare to the work of Chen et al. [5], we chose the setting closest with reported evaluation results, which is $m' = 5535$ client elements using a single thread, while we use $m = 4096$ client elements. The results are summarized in Table 1, where the last two rows denote the improvement factor of our version over two previous protocols. For $n = 2^{20}$, we experience a drop relative to [38] in the improvement factor that most likely occurs because the modulus chain length L has to be increased from $L = 300$ to $L = 450$ to account for the additional multiplicative depth of the computation. The large gap in computation costs may be attributed to high constants in the HE framework. Furthermore, it should be noted that the *DiPSI* protocol is trivially parallelizable which could reduce the response latency significantly if necessary. Our multiplicative depth is only $O(\log \ell)$, where ℓ is the maximal bit-length; indicating that our scheme is extensible to extremely large datasets. Also, our performance is inhibited by the fact that each comparison has to be computed over every bit, which is not the case anymore once the intersection result has been computed. Thus, we presume that for complex predicates on top of the intersection our approach might be feasible from a runtime cost perspective due to its scalability.

The computation time of Algorithm 2 for varying numbers of client and server elements $2^{10} \leq m \leq 2^{16}$ and $2^{10} \leq n \leq 2^{21}$ is depicted in Figure 4. 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. From the graph it can be seen that our approach scales well for different number of client elements, which can be attributed to our hashing approach. Increasing m also decreases the number of server elements in any bin on the server, which means that overall fewer comparisons have to be computed. This phenomenon in combination with the chosen minimal batchsize of $s = 4096$ turns into a disadvantage for small $m = 2^{10}$ as the runtime is significantly higher than for $m = 2^{12}$. Computing *DiPSI-CA* on top of the *DiPSI* has virtually no impact on the runtime as it is entails essentially just $O(\log m)$ more additions. In that case, the communication cost is reduced as only $O(1)$ ciphertexts have to be returned by the server instead of $O(m)$. Moreover, note that the plaintext modulus can not chosen to be binary, because the cardinality predicate on top of the *DiPSI-CA* produces as a result a natural number possibly greater than 1.

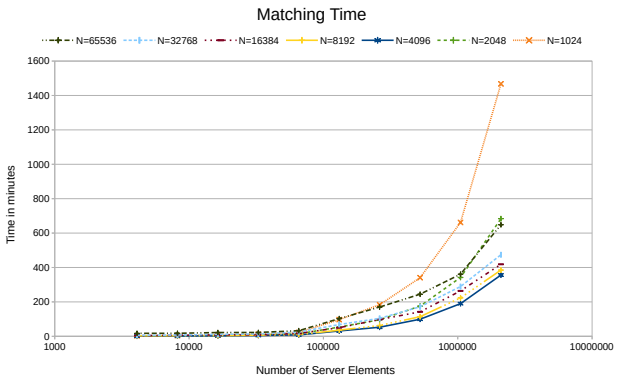

Figure 4: Total Computation Cost for *DiPSI*

[6]The condition of "probably" comes from the differential privacy and not from any imprecision in the protocol.

Runtime	Arbitrary bit-length			Binary
Protocol n=	2^{12}	2^{16}	2^{20}	Circuit
CLR [5]	-	1.7	8.7	✗
PTWW [38]	1.2	8.49	120.73	✓
DiPSI	73.48	526.91	11394.03	✓
DiPSI, CLR	-	-309.95x	-1309.66x	✗
DiPSI, PTWW	-61.23x	-62.06x	-94.38x	✓

Table 1: Protocols without Stash, Runtime in Seconds

Communication	Arbitrary bit-length			Binary
Protocol n=	2^{12}	2^{16}	2^{20}	Circuit
CLR [5]	-	3.5	5.6	✗
PTWW [38]	9	149	2540	✓
DiPSI	119.44	133.03	231.56	✓
DiPSI, CLR	-	-38.01x	-41.35x	✗
DiPSI, PTWW	-13.22x	1.12x	10.97x	✓

Table 2: Protocols without Stash, Communication in MB

The maximum number of server elements in any bin on the server is illustrated in Figure 6. Note that the graph is very similar to Figure 4 which depicts the matching time. For our chosen parameter configuration we measured the best results, however future improvement for guiding parameter selection in HELib may lead to even better performance for our use-case.

7.3 Communication

The communication complexity is significantly lower when the larger set exceeds a certain threshold dependent on the size of the public key. The experiments show that the public key alone consumes a baseline of about $60MB$ for $L = 300$ and about $79MB$ for $L = 450$. The results are summarized in Table 2, where the last two rows depict the improvement factor. For communication cost, *DiPSI* scales with the smaller set size and for $m = 4096$ client elements and $N = 2^{20}$ server elements, our protocol requires only $231.56MB$ of total communication costs as opposed to $2.54GB$ as in the approach from Pinkas et al. [38].In comparison with Chen et al. [5], their communication complexity does not include the size of the public key, but instead only the ciphertexts, which partly explains why their communication is significantly lower than ours. The communication cost for the *DiPSI* protocol is plotted in Figure 5 relative to the number of server elements. The total communication happens in two rounds. First, the public key including all bit-wise encrypted client element vectors are sent to the server. As a response for the intersection the client receives equally many ciphertexts as were sent to the server. A steady increase can be observed for a larger bit length as more ciphertexts are transmitted. For a bit-length of $\ell > 16$, the maximal multiplicative depth of the circuit increments which has to be counteracted by increasing the modulus chain length from $L = 300$ to $L = 450$.

In practice, our *DiPSI* approach is computationally worse than the state-of-the-art approach presented by Pinkas et al. [38]. However, our communication complexity achieves better optimality for

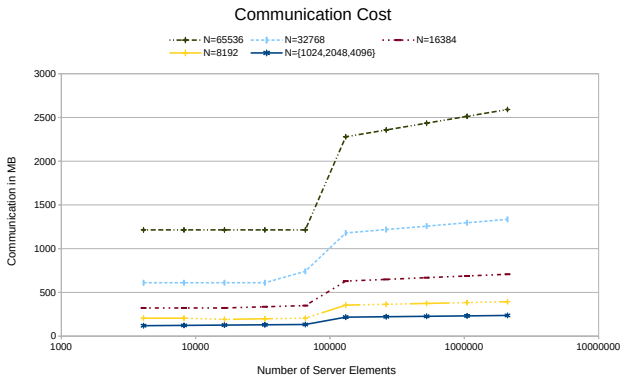

Figure 5: Total Communication Cost for DiPSI

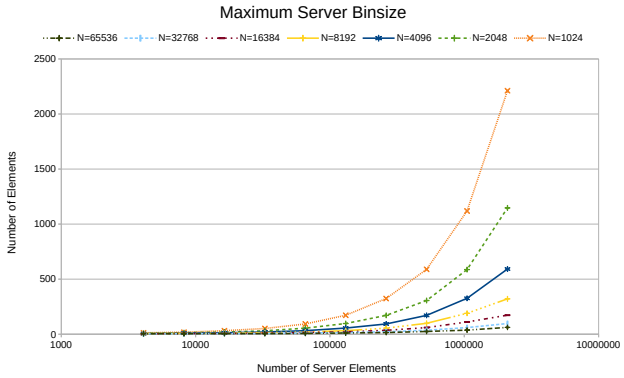

Figure 6: The Maximal Number of Elements per Server Bin

unbalanced sets in both complexity and in practice. Any improvements in HELib in with respect to ciphertext size and computation efficiency will immediately translate to our approach. For now, considerably higher computation cost can presumably be ameliorated in terms of latency by implementing our approach on multiple processors.

8 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 as, *DiPSI* and *DiPSI-CA*, 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 optimum complexity and accuracy. 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 as long as they can be represented by a circuit.

REFERENCES

[1] Mark Bergen and Jennifer Surane. 2018. Google and Mastercard Cut a Secret Ad Deal to Track Retail Sales. https://www.bloomberg.com/news/articles/2018-08-30/google-and-mastercard-cut-a-secret-ad-deal-to-track-retail-sales.

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

[3] Eric W. Burger, Michael D. Goodman, Panos Kampanakis, and Kevin A. Zhu. 2014. Taxonomy Model for Cyber Threat Intelligence Information Exchange Technologies. In *Proceedings of the 2014 ACM Workshop on Information Sharing & Collaborative Security*. 51–60.

[4] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 2018. Labeled PSI from Fully Homomorphic Encryption with Malicious Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1223–1237.

[5] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 1243–1255. https://doi.org/10.1145/3133956.3134061

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

[7] Michele Ciampi and Claudio Orlandi. 2018. Combining Private Set-Intersection with Secure Two-Party Computation. In *Security and Cryptography for Networks*, Dario Catalano and Roberto De Prisco (Eds.). Springer International Publishing, Cham, 464–482.

[8] Alex Davidson and Carlos Cid. 2017. An Efficient Toolkit for Computing Private Set Operations. In *Information Security and Privacy*, Josef Pieprzyk and Suriadi Suriadi (Eds.). Springer International Publishing, Cham, 261–278.

[9] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. 2012. Fast and Private Computation of Cardinality of Set Intersection and Union. In *Cryptology and Network Security*, Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 218–231.

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

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

[12] Cynthia Dwork. 2006. Differential Privacy. In *Automata, Languages and Programming*, Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12.

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

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

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

[16] Arpita Ghosh, Tim Roughgarden, and Mukund Sundararajan. 2012. Universally Utility-maximizing Privacy Mechanisms. *SIAM J. Comput.* 41, 6 (2012), 1673–1693.

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

[18] Michael T Goodrich and Michael Mitzenmacher. 2011. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages, and Programming*. Springer, 576–587.

[19] Adam Groce, Peter Rindal, and Mike Rosulek. 2019. Cheaper Private Set Intersection via Differentially Private Leakage. Cryptology ePrint Archive, Report 2019/239. https://eprint.iacr.org/2019/239.

[20] Shai Halevi and Victor Shoup. 2014. HElib-An Implementation of homomorphic encryption. *Cryptology ePrint Archive, Report 2014/039* (2014).

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

[22] Xi He, Ashwin Machanavajjhala, Cheryl Flynn, and Divesh Srivastava. 2017. 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*. ACM, 1389–1406.

[23] Yan Huang, David Evans, and Jonathan Katz. 2012. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*.

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

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

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

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

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

[29] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. 2012. Billion-Gate Secure Computation with Malicious Adversaries. In *Proceedings of the 21th USENIX Security Symposium*. 285–300.

[30] Teri A. Manolio. 2010. Genomewide Association Studies and Assessment of the Risk of Disease. *New England Journal of Medicine* 363, 2 (2010), 166–176.

[31] Moxie Marlinspike. 2014. The Difficulty of Private Contact Discovery. A company sponsored blog post. https://whispersystems.org/blog/contact-discovery/.

[32] Catherine Meadows. 1986. 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*. IEEE, 134–134.

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

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

[35] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo Hashing. In *Algorithms — ESA 2001*, Friedhelm Meyer auf der Heide (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 121–133.

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

[37] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. 2019. Efficient Circuit-based PSI with Linear Communication. Cryptology ePrint Archive, Report 2019/241. https://eprint.iacr.org/2019/241.

[38] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. 2018. Efficient Circuit-Based PSI via Cuckoo Hashing. In *Advances in Cryptology – EUROCRYPT 2018*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 125–157.

[39] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2018. Scalable Private Set Intersection Based on OT Extension. *ACM Trans. Priv. Secur.* 21, 2, Article 7 (Jan. 2018), 35 pages. https://doi.org/10.1145/3154794

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

[41] Stanley L. Warner. 1965. Randomized Response: A Survey Technique for Eliminating Evasive Answer Bias. *J. Amer. Statist. Assoc.* 60, 309 (1965), 63–69. http://www.jstor.org/stable/2283137

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

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

[44] Moti Yung. 2015. 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*. 1–2.