Delegated Private Set Intersection On Outsourced Private Datasets

Aydin Kheirbakhsh Abadi

Department of Computer and Information Sciences

University of Strathclyde

A thesis submitted for the degree of

Doctor of Philosophy in Computer Science

Submitted in November, 2016

This work was carried out under the supervision of

Dr Changyu Dong (1^{st} supervisor) and **Dr Sotirios Terzis**.

I would like to dedicate this thesis to my loving parents.

Declaration of Authenticity and Author's Rights

This thesis is the result of the authors original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Aydin Kheirbakhsh Abadi

Signed:

Date:

Acknowledgements

I would like to express my sincere gratitude to Dr Changyu Dong and Dr Sotirios Terzis for their outstanding supervision and valuable guidance throughout my PhD. I would like to express my great appreciation for the love and support of my family, for their help and support.

Publications

- O-PSI: Delegated Private Set Intersection on Outsourced Datasets. Aydin Abadi, Sotirios Terzis, and Changyu Dong. In ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Germany, pages 317, 2015. 46, 128.
- VD-PSI: Verifiable Delegated Private Set Intersection on Outsourced Private Datasets. Aydin Abadi, Sotirios Terzis, and Changyu Dong. In Financial Cryptography and Data Security - 20th International Conference, FC 2016, Barbados, 2016. 107, 130.
- Efficient Delegated Private Set Intersection on Outsourced Private Datasets. Aydin Abadi, Sotirios Terzis, Roberto Metere and Changyu Dong. IEEE Transactions on Dependable and Secure Computing, submitted on July 18, 2016.

Abstract

The significance of cloud computing is increasing and the cloud is receiving growing attention from individuals and companies. The cloud enables ubiquitous and on-demand access to a pool of configurable computing resources that can be scaled up easily. However, the cloud is vulnerable to data security breaches such as exposing confidential data, data tampering, and denial of service. Thus, it cannot be fully trusted and it is crucial for the clients who use the cloud to protect the security of their own data.

In this thesis, we design cryptographic protocols to allow clients to outsource their private data to the cloud and delegate certain computation to the cloud securely. We focus on the computation of set intersection which has a broad range of applications such as privacy-preserving data mining, and homeland security. Traditionally, the goal of Private Set Intersection (PSI) protocols has been to enable two parties to jointly compute the intersection without revealing their own set to the other party. Many such protocols have been designed. But, in the cases where data and computation are outsourced to the cloud, the setting and trust assumptions are considerably changed. The traditional PSI protocols cannot be used directly to solve security problems, without sacrificing the advantages the cloud offers.

The contribution of this thesis is a set of delegated PSI protocols that meet a variety of security and functional requirements in the cloud environment. For most clients, the most critical security concern when outsourcing data and computation to the cloud is data privacy. We start from here and design O-PSI, a novel protocol in which clients encrypt their data before outsourcing it to the cloud. The cloud uses the encrypted data to compute the intersection when requested. The outsourced data remain private against the cloud all the time since the data stored in the cloud is encrypted and the computation process leaks no information. O-PSI ensures that the

computation can be performed only with the clients' consent. The protocol also takes into account several functional requirements in order to take full advantage of the cloud. For example, clients can independently prepare and upload their data to the cloud, and the clients are able to delegate to the cloud the computation an unlimited number of times, without the need to locally re-prepare the data. We then extend O-PSI in several ways to provide additional properties:

* EO-PSI is a more efficient version of O-PSI that does not require public key operations.
* UEO-PSI extends EO-PSI with efficient update operations, making it possible to efficiently handle dynamic data.
* VD-PSI extends O-PSI with verifiability, i.e. the clients can efficiently verify the integrity of the computation result.

For each protocol, we provide a formal simulation-based security analysis. We also compare the protocols against the state of the art. In addition to that, we have implemented the O-PSI and EO-PSI protocols and provide an evaluation of their performance based on our implementation.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Cloud computing offers flexible and cost effective storage and computation resources to clients, and has been attracting the attention of individuals and businesses as a crucial enabling technology [83]. A study by the IBM Institute for Business Value in 2012[1] suggested that cloud computing, as a game-changing technology, is leading business innovation in a number of dimensions. There are many benefits for businesses to adopt the cloud, e.g. cost flexibility, business scalability, and increased collaboration with external partners [15]. Over time, the cloud has become a pool of sensitive data where running computation on the data is prevalent [92, 58]. The data stored in the cloud can be of different kinds such as financial data (e.g. price fixing data from stock markets, revenues of companies), sensitive biological data (e.g. personal genetic data) [42], etc. Moreover, a broad range of computation is performed on the data in the cloud, such as computing intersection or union of sets [23], computing statistics (e.g. averages, variances) [42], etc.

One of the major operations commonly performed in the cloud is computing the intersection of sets [23, 78, 50]. In this case, clients store their data in the cloud and later on ask the cloud to compute the intersection of their sets. There are various real-world applications in which set intersection is needed such as data mining to find common rules [70], authenticated email search [25], range queries over outsourced databases [92], finding the common records or keyword search in outsourced databases [22, 78], verifiable web-content search [54], relationship path discovery in social networks [86],

[1] http://www.ibm.com/cloud-computing/us/en/assets/power-of-cloud-for-bus-model-innovation.pdf

and queries on human genome [10].

However, past incidents [57, 41] and current research [103, 64, 119, 122] suggest that the cloud is vulnerable to information security breaches. Security breaches could result in unauthorized disclosure of confidential information, data tampering, data loss and denial of service. Furthermore, data protection laws (e.g. [97, 17, 24]) obligate individuals and organizations to secure data. Therefore, when outsourcing data and computation to the cloud, it is imperative for the clients to protect their data.

Secure Multi-Party Computation (MPC) protocols are cryptographic protocols that have been designed to allow distributed computation on private data. In particular, these protocols enable two or more parties to jointly run an operation on the sensitive inputs such that each party receives the correct output while the privacy of each party's input is preserved, even in the presence of adversarial behavior. There are generic MPC protocols that can compute any function, however they are often not efficient [68, 34]. Often, for specific computation tasks, special-purpose protocols are more efficient and protocols for electronic auctions, private information retrieval, and private set intersection are examples of that [60].

Private set intersection (PSI) is a vital instance of secure MPC [8, 38, 31]. PSI allows two mutually distrusting parties, each having a private set, to compute the intersection of the sets without revealing any information, about the set elements that are not in the intersection, to the other party [46, 96]. PSI has a wide range of real-world applications. For instance, consider the scenario in which two companies want to compare their respective lists of customers to find customers in common. This may help them to learn about the online shopping behavior of their customers and to make certain offers for joint purchases. As another example, consider the situation where a social welfare organization wants to know whether any of its members receives income from another such organization. In both cases, due to the privacy laws or lack of trust, neither of the parties can reveal its client list to the other party. In such scenarios, PSI can be used to address the problem. Due to its importance and wide applicability, researchers have designed several PSI protocols (see chapter 3 for a survey). However, in the classical MPC and PSI setting where data and computation are not outsourced, data owners use locally stored data and jointly run the computation. When data and computation are outsourced to the cloud, the setting and trust assumptions drastically change. Therefore, these classical protocols cannot directly be used to address security problems in the cloud.

As traditional PSI protocols are not suitable for the cloud setting, where data stor-

age and computation are outsourced to a cloud that is not fully trusted, in this thesis, we investigate *delegated PSI computation on private data stored in the cloud* and we design new protocols that allow clients to fully benefit from the cloud's storage and computation capabilities, while protecting the privacy of data. For example, consider the case in which different online shopping companies keeping their customers' shopping histories on cloud storage (e.g. Amazon S3 [1]) want to find out the customers who purchase from all the companies for joint discount offers. In this case, companies can ask the cloud to run the computation on the data already stored in it. However, due to data protection laws (and the company regulations), the cloud should not learn any information about their customers' shopping behavior and the other companies should not learn more than the computation result. In this scenario, the use of the cloud enables the parties (i.e. companies) to access their outsourced data from any location and remove the need to maintain a local copy of the data. Moreover, in this setting, they can use a delegated PSI protocol to delegate computation of PSI on their private data to the cloud, while the data security is preserved.

1.2 Requirements

In the delegated private set intersection setting, there are primary security and functional requirements that must be satisfied. What follows is a list of the requirements and their significance in such setting.

- Security Requirements:
  - *Protecting Dataset and Result Privacy.*

    This feature guarantees that the cloud cannot figure out the clients' sensitive dataset elements, the dataset intersection, and the intersection cardinality. Furthermore, the result recipient cannot learn anything about the other clients' dataset elements beyond the intersection.
  - *Requiring Computation Authorization.*

    This feature guarantees that only an authorized party can obtain the result of the computation and without all clients' consent no party can carry out the computation. If this property is not satisfied, for example, the cloud (or

[1] https://aws.amazon.com/s3/

any party) can generate a dataset, run the computation, and learn the private data (or the computation result) without their permission.

* *Verifying the Correctness of the Computation Result.*

  The cloud is prone to various misbehaviors, attacks and incidents that can lead to data corruption, data tampering or incorrect computation results. For instance, the cloud may not carry out the entire computation in order to cut some cost [93] and this may yield a false computation result, or the data stored in the cloud may be corrupted as a result of hardware malfunctioning [28], or tampered with as a result of (inside or outside) attacks [7]. Therefore, using a verifiable delegated PSI protocol, clients can *check the integrity* of the result of the delegated computation.

- Functional Requirements:

  * *Supporting Outsourcing of both Dataset Storage and PSI Computation to the Cloud.*

    This property allows clients to fully benefit from the advantages the use of the cloud's storage and computation capabilities offers (e.g. business scalability, increased collaboration with external partners, elasticity, economies of scale, and higher robustness and availability). The clients can outsource their datasets in the cloud once, delete any local copy of them, and ask the cloud to run PSI on the outsourced data. Therefore, the clients do not need to have a large local storage to maintain a copy of the data, and also they do not need to download, re-prepare and upload their outsourced datasets, each time the computation is delegated.

  * *Having a Non-interactive Setup.*

    With this property, clients can prepare and upload their datasets independently at different points in time, without the need to interact with each other, or even know anything about others. This also enables clients to form new collaborations without the need to coordinate with existing clients. As a result the protocol becomes suitable for real-world applications where such flexibility is essential.

* *Efficiency.*

  Efficiency in both computation and communication is desirable in general. Ideally, a secure protocol should have lightweight building blocks that do not impose a high cost. For instance, a protocol that avoids using expensive building blocks and operations (e.g. fully or partially homomorphic encryption or exponentiations) would have lower overall computation cost than those utilizing them. Since designing efficient secure protocols is very challenging, researchers first design secure protocols and then improve their efficiency.

* *Supporting Dynamic Data.*

  There are numerous PSI applications that rely on frequently updated datasets, such as secure file sharing [66], consumer behavior prediction [116] or cancer research on genomic datasets [1]. In these cases, it is vital for the delegated PSI schemes to *efficiently support update* on the outsourced private datasets. Otherwise, they would impose large communication, storage and computation costs, when big data needs to be updated regularly. It should be noted that a trivial secure way to update outsourced private data is to download the entire data, locally update them and re-upload them to the cloud, but this approach is not efficient. Therefore, a delegated PSI protocol that can do so, more efficiently without sacrificing the privacy of the data or the efficiency of the PSI computation, would be desirable.

1.3 Contributions of the Thesis

In general, our contributions center around developing new protocols for delegated private sets intersection on outsourced private datasets. In section 1.2, we listed the requirements for delegated PSI. Ideally, we would like one protocol that satisfies all requirements. However, such a protocol would be dauntingly complex and difficult to design. The resulting protocol is also likely to be inefficient due to too many constraints. Thus, we adopt an incremental design approach in the sense that we start from a small set of core requirements, and then extend it iteratively when designing protocols. Our study starts from delegated PSI that protects data privacy against the cloud (O-PSI). We then improve the protocol's efficiency and develop a new one (EO-PSI)

[1] http://epi.grants.cancer.gov/dac/

and then extend it to a protocol that efficiently supports dynamic datasets (UEO-PSI) too. To achieve better security, we also investigate delegated PSI that protects data privacy and allows verification of the computation result (VD-PSI). More specifically, the contributions that each protocol makes are as follows:

- **O-PSI Protocol.** The first contribution of this thesis includes the design of the first efficient delegated PSI protocol that allows multiple clients to independently store their private data in the cloud. At a later point in time, they can get together and ask the cloud to perform PSI on their private data. The protocol also allows sets to be used an unlimited number of times securely without the need to download and re-prepare the data. In this process, the cloud cannot learn the client set elements, the intersection and the intersection cardinality, and also the result recipient cannot learn anything beyond the intersection. The protocol ensures that intersections can only be computed with the permission of the clients. The protocol uses a single cloud and it does not involve a trusted party (to generate and distribute keys among parties). In this protocol, clients can prepare and outsource their data independently without interacting with each other or having any knowledge of others. The computation and communication costs of O-PSI are linear to the set cardinality. The protocol leverages a public key encryption scheme, a blinding technique and the mathematical properties of polynomials to achieve its goal. The O-PSI protocol, as a full paper, has been published in [1].

- **EO-PSI Protocol.** The next contribution of the thesis includes the development of an *efficient* delegated PSI protocol that does not use any public key encryption or other expensive operations like exponentiations. Moreover, it enables the result recipient to retrieve the computation result faster than it could do in O-PSI. The protocol preserves all O-PSI's advantageous features. The protocol mainly uses the mathematical properties of polynomials, a blinding technique and a hash table to attain its goal. We also implemented both O-PSI and EO-PSI and analyzed their performance. The analysis showed that EO-PSI is 1-2 orders of magnitude faster than O-PSI. The EO-PSI protocol, as a full paper, has been submitted to the IEEE Transactions on Dependable and Secure Computing journal [3].

- **UEO-PSI Protocol.** Another contribution of the thesis comprises the design of UEO-PSI, the first delegated PSI protocol that efficiently supports *dynamic data*.

It allows clients to update outsourced datasets with low computation and communication costs, and without leaking any information about the dataset elements to any party. UEO-PSI preserves the efficient aspects of EO-PSI (i.e. it does not use public key encryption and allows faster result retrieval) and preserves its appealing properties. UEO-PSI is based on a hash table, the mathematical properties of polynomials, a blinding technique, a permutation map, (pseudo)random labels and a (pseudo)random shuffle.

- **VD-PSI Protocol.** The final contribution of the thesis is the design of the first delegated PSI protocol that enables clients to *efficiently verify* the correctness of the computation result. VD-PSI considers the scenario where the cloud is an active adversary (i.e. it may try to arbitrarily deviate from the protocol or tamper with the result) and it also possesses the appealing properties of O-PSI. The main novelty of VD-PSI is a lightweight verification mechanism that allows the client to efficiently verify the correctness of the result without the need to access its outsourced dataset, and have any knowledge of the other client's dataset elements. The verification cost of the protocol is at the most linear to the intersection cardinality. VD-PSI is based on public key encryption, the mathematical properties of polynomials, a blinding technique and a verification mechanism that utilizes a trap-like value to detect the cloud's misbehavior. The VD-PSI protocol, as a full paper, has been published in [2].

  The above contributions are outlined in Fig 1.1.

1.4 Roadmap

Chapter 2 presents notations, definitions and technical tools we use throughout the rest of the thesis. Chapter 3 provides some background on cloud computing, and a survey of protocols designed to address the PSI problem. Chapter 4 presents the O-PSI and EO-PSI protocols. It provides a detailed description of the protocols, their extensions, their security definition and analysis, a comparison to other delegated PSI protocols and a study of their performance. Chapter 5 presents UEO-PSI protocol. The chapter comprises a detailed description of the protocol, its extension, its security definition and analysis, and its communication and computation complexity analysis, and a comparison with the other protocols. Chapter 6 presents our VD-PSI protocol, its

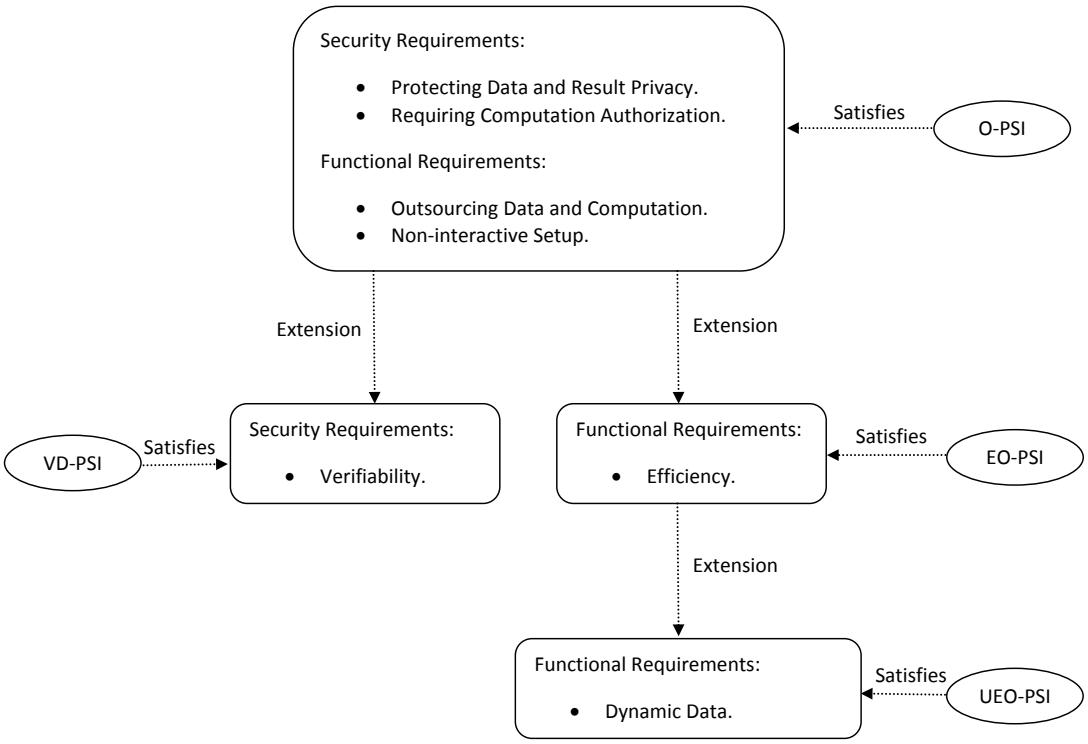

Figure 1.1: An outline of the thesis contributions.

security definition and analysis. The chapter analyses the protocol's complexities and compares it with the other verifiable delegate PSI protocols. The thesis conclusion and directions for future work are provided in chapter 7. Appendices A and B represent the class diagrams of the O-PSI and EO-PSI implementation respectively.

Chapter 2

Preliminaries

2.1 Notation and Definitions

In this thesis, for universe $\mathcal{U}$ of set elements, we assume that there exists a sufficiently large public finite field, $\mathbb{F}_p$, to encode all elements of the universe, where p is a large prime number. Also, we denote the multiplicative inverse and additive inverse of value v_i, by $(v_i)^{-1}$ and $-v_i$ respectively. Also, by $a||b$ we mean a is concatenated with b, by $|c|$ we mean the bit-wise size of c, by $|\vec{\boldsymbol{v}}|$ we mean the size of vector $\vec{\boldsymbol{v}}$ and by $e^{(I)}$ we mean that value e belongs to client I.

In cryptography, we do not require an event to happen with zero probability, or an adversary to always fail. But, we allow the event to happen, (or the adversary to succeed) with some very small non-zero probability. It is denoted by the function of the security parameter: $\epsilon(n)$, and defined as follows, i.e. the following definitions are according to [51].

Definition 1. *(Negligible Function): We call a function $\epsilon : \mathbb{N} \to \mathbb{R}$ negligible if for every positive polynomial $poly(.)$, there exists a $m \in \mathbb{N}$ such that for all $n > m$,*

$$\epsilon(n) < \frac{1}{poly(n)}$$

Several cryptographic primitives and protocols (including ours) rely on the notion of computation indistinguishability. We use this concept extensively in the security analysis of our protocols. Before we formalize the computation indistinguishability concept, we provide two notions used in the definition of the concept.

Definition 2. *(Random Variable): A random variable is a function, $X : \Omega \to \mathbb{R}$ from*

the set of possible outcomes to real numbers.

Random variables are used when we are more interested in the consequence of experiment than the experiment itself (e.g. appearance of a number of heads when a coin is tossed twice). The probability that random variable X takes value $\omega \in \Omega$ is denoted by $Pr[X = \omega]$. In this thesis, we are dealing with only finite spaces, so Ω is a finite set. A random variable X is uniformly distributed over a finite set Ω if it has equal value on every element in Ω, i.e. $\forall \omega \in \Omega, Pr(X = \omega) = \frac{1}{|\Omega|}$.

Definition 3. *(Ensemble): Any* $\boldsymbol{X} = \{X_n\}_{n\in\mathbb{N}}$*, where each* X_n *is a random variable, is an ensemble indexed by* $\mathbb{N}$*.*

In other words, a sequence of (possibly infinite) random variables is called a (probability) ensemble.

Definition 4. *(Computationally Indistinguishable): Two ensembles,* $\boldsymbol{X} = \{X_n\}_{n\in\mathbb{N}}$ *and* $\boldsymbol{Y} = \{Y_n\}_{n\in\mathbb{N}}$*, are computationally indistinguishable, denoted by* $\boldsymbol{X} \overset{c}{\equiv} \boldsymbol{Y}$*, if for every non- uniform probabilistic polynomial time algorithm* B*, there exists a negligible function* $\epsilon(.)$ *such that for every sufficiently large* $n \in \mathbb{N}$*:*

$$\left| Pr[B(X_n, 1^n) = 1] - Pr[B(Y_n, 1^n) = 1] \right| < \frac{1}{\epsilon(n)}$$

2.2 Homomorphic Encryption

The goal of encryption schemes is to ensure confidentiality of data in communication and storage processes. There are two types of encryption schemes: public key schemes (or asymmetric key schemes) and private key schemes (or symmetric key schemes). A public key encryption scheme has three components, the key generation function $Gen()$, the encryption function $E_{pk}()$, and the decryption function $D_{sk}()$. In a public key encryption scheme, the key generation function, generates a key pair (pk, sk), where pk is the public key (and known to public), also sk is the private key (and kept secret). Anyone knowing the public key can encrypt, but only one who has the corresponding private key can decrypt. In a symmetric key scheme, the encryption and decryption keys are the same and secret.

Since in this thesis we use a public key encryption scheme, we first explain the basic security requirement that the scheme should satisfy that is indistinguishability under

chosen-plaintext attack (IND-CPA). Informally, a cryptosystem is considered secure in terms of indistinguishability if no probabilistic polynomial-time (PPT) adversary, given an encryption of a message randomly chosen from a two-element message space (determined by the adversary), can identify the message with probability significantly better than random guessing (i.e. $\frac{1}{2}$). The adversary has access to the public key and can encrypt any messages of its choice. As [71] states, this definition is formally presented as a game (or experiment) $PubK_{\mathcal{A},\Pi}(n)$, for any public-key encryption scheme $\Pi = (Gen, E, D)$, any adversary $\mathcal{A}$, and any value n for the security parameter:

1. $Gen(1^n)$ is run to obtain keys (pk, sk).
2. Adversary $\mathcal{A}$ is given pk. The adversary outputs a pair of messages m_0, m_1 of the same length.
3. A random bit $b \leftarrow \{0, 1\}$ is chosen, and then a ciphertext $c \leftarrow E_{pk}(m_b)$ is computed and given to $\mathcal{A}$.
4. $\mathcal{A}$ outputs a bit b'.
5. The output of the experiment is defined to be 1 if $b = b'$, and 0 otherwise.

Definition 5. *A public key encryption scheme $\Pi = (Gen, E, D)$ has indistinguishable encryptions under a chosen-plaintext attack (or is CPA secure) if for all probabilistic polynomial-time adversary $\mathcal{A}$, there exists a negligible function, $\epsilon(.)$, such that:*

$$Pr[PubK_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \epsilon(n).$$

Indistinguishability under chosen plaintext attack is equivalent to the property of semantic security, and many cryptographic proofs use these term interchangeably [71].

There are situations where operations should be performed on encrypted data (i.e. ciphertext). The encryption schemes that allow such operations on ciphertexts are called Homomorphic Encryption (HE) schemes. They allow certain computations to be carried out on ciphertexts and generate an encrypted result that, when decrypted, equals the result of operations performed directly on the messages (i.e. plaintexts). Due to this essential feature that HE schemes offer, they are extensively used to create secure voting systems, private information retrieval schemes, enable widespread use of the cloud when the confidentiality of processed data is required, and design secure Multi-Party Computation (MPC) protocols (see [121] for more applications of HE schemes).

HE cryptosystems are categorized into two main groups: Partially Homomorphic Encryption and Fully Homomorphic Encryption. The encryption schemes that support only limited operations (e.g. multiplication or addition) on ciphertexts are called Partially Homomorphic Encryption; and those that allow arbitrary operations on ciphertexts are called Fully Homomorphic Encryption. Below, we explain briefly each group.

Fully Homomorphic Encryption (FHE). The notion of encryption schemes that permit arbitrary computation on encrypted data was first proposed by Rivest *et al.* [105]. Since then, there has been intensive work to design a FHE scheme and the problem finally was solved by the breakthrough work of Gentry [48]. Following his work, a set of other FHE schemes have been proposed (e.g. [108, 113]). Since FHE can support any computation on ciphertexts, it has numerous applications such as delegating privacy preserving computation to the cloud against which plaintexts must be protected.

In spite of many attempts made to improve the efficiency of FHE schemes (e.g. [109, 49]), they are still too inefficient to be practical and are not used in the real world [114, 69, 67].

Partially Homomorphic Encryption (PHE). Unlike FHE, PHE supports limited operations on ciphertexts. For instance, (unpadded) RSA [104] and El-Gamal [47] encryptions support homomorphic multiplication, Goldwasser-Micali encryption [53] allows homomorphic XOR and Paillier encryption [91] supports homomorphic addition. PHE schemes are more efficient than FHE [19]. In this thesis, we make use of Paillier encryption scheme due to its efficiency, so we explain the scheme in more details.

Paillier Encryption. The semantic security of Paillier encryption is based on a well-known mathematical problem: *Decisional Composite Residuosity Assumption* (DCRA). In order to explain the assumption, we need the following definition.

Definition 6. *Let N be the product of two large primes. The value $i \in \mathbb{Z}^*_{N^2}$ is said to be an e^{th} residue $\bmod N^2$ if there exists a value j such that $i = j^e \bmod N^2$; where $j \in \mathbb{Z}^*_{N^2}$, $\mathbb{Z}_N = \{g | g \in \mathbb{Z}, 0 \leq g < N\}$ and $\mathbb{Z}^*_{N^2} = \{g | g \in \mathbb{Z}, 0 < g < N^2, gcd(g, N^2) = 1\}$.*

DCRA states that, given N and i, there exists no polynomial time algorithm that distinguishes e^{th} residue from non-e^{th} residues (i.e. it is hard to decide whether i is a

e^{th} residue $\mathrm{mod} N^2$). Now we present each of the three functions (i.e. key generation, encryption, and decryption) of Paillier encryption scheme.

- **Key Generation**: To Generate public and private keys do the following.
  * Choose two random large primes q_1 and q_2 according to a given security parameter and set $N = q_1 \cdot q_2$.
  * Pick a uniformly random value g such that $g \in \mathbb{Z}^*_{N^2}$.
  * Ensure that $s = (L(g^u \bmod N^2))^{-1} \bmod N$ exists where $L(x) = \frac{(x-1)}{N}$, u is the Carmichael value of N, i.e. $u = lcm(q_1 - 1, q_2 - 1)$ and lcm stands for the least common multiple.
  * Set $pk = (N, g)$ as the public key, and $sk = (u, s)$ as the secret key.
- **Encryption**: To encrypt a plaintext $m \in \mathbb{Z}_N$ do the following.
  * Pick a uniformly random value r such that $r \in \mathbb{Z}^*_N$.
  * Compute the ciphertext: $C = E_{pk}(m) = g^m \cdot r^N \bmod N^2$.
- **Decryption**: To decrypt the ciphertext, C, do the following.
  * $D_{sk}(C) = L(C^u mod\ N^2) \cdot s \bmod N = m$.

The scheme has two interesting homomorphic features.

- If we compute the product of two ciphertexts, after decrypting we will have the sum of their corresponding plaintexts:

$$D_{sk}(E_{pk}(a) \cdot E_{pk}(b)) = a + b$$

- If we raise a ciphertext to the power of a constant value, after decrypting we will have the product of constant value and the corresponding plaintext:

$$D_{sk}(E_{pk}(a)^b) = a \cdot b$$

2.3 Pseudorandom Functions

Informally, a pseudorandom function (PRF) is a deterministic function that takes a key and an input; and outputs a value indistinguishable from that of a truly random function with the same input. Pseudorandom functions have many applications in cryptography as they provide an efficient and deterministic way to turn an input into a value that looks random. A PRF is formally defined as follows [71].

Definition 7. *Let* $W : \{0,1\}^n \times \{0,1\}^t \rightarrow \{0,1\}^m$ *be an efficient keyed function. It is said* W *is a pseudorandom function if for all probabilistic polynomial-time distinguishers* B*, there is a negligible function,* $\epsilon(.)$*, such that:*

$$\left| Pr[B^{W_k(.)}(1^n) = 1] - Pr[B^{w(.)}(1^n) = 1] \right| \leq \epsilon(n),$$

where the key, $k \xleftarrow{R} \{0,1\}^n$*, is chosen uniformly at random and* w *is chosen uniformly at random from the set of functions mapping t-bit strings to m-bit strings.*

In practice, the pseudorandom function can be obtained from an efficient block cipher [71]. Also, we require the output of the pseudorandom function to be different with a high probability when two different distinct secret keys are used, i.e. collision-resistant PRF.

Definition 8. *(collision-resistant* PRF*) Let* PRF *be a pseudorandom function defined as above. It is said* PRF *is collision-resistant if:*

$$Pr[\texttt{PRF}(k,i) = \texttt{PRF}(k',i)] \leq \epsilon(n),$$

where $k, k' \xleftarrow{R} \{0,1\}^n$*,* $k \neq k'$ *and* m *(i.e. the output bit-length) is a function of the security parameter,* n*.*

A collision-resistant PRF can be constructed using the schemes proposed in [43].

2.4 Pseudorandom Shuffle

A pseudorandom shuffle (permutation), $\pi(k, \vec{\boldsymbol{v}})$, is a deterministic function that permutes the elements of vector $\vec{\boldsymbol{v}}$, where $|\vec{\boldsymbol{v}}| = m$, pseudorandomly using a secret key k, and the output vector has the same size, m. In general, it is used in cases where a set

of vectors should be permuted in the same way; but given the permuted vectors, a PPT adversary (who does not know the secret key) cannot figure out the original ordering of the vectors. A π is formally defined as follows.

Definition 9. *Let $\pi : \{0,1\}^n \times \vec{\boldsymbol{v}} \rightarrow \vec{\boldsymbol{v}}'$ be an efficient keyed permutation, $|\vec{\boldsymbol{v}}| = |\vec{\boldsymbol{v}}'| = m$. We say that π is a pseudorandom permutation if for all probabilistic polynomial-time distinguisher, B, there exists a negligible function $\epsilon(.)$, such that:*

$$\left| Pr[B^{\pi_k(.)}(1^n) = 1] - Pr[B^{w'(.)}(1^n) = 1] \right| \leq \epsilon(n),$$

where the key, $k \xleftarrow{R} \{0,1\}^n$, is chosen uniformly at random, and w' is chosen uniformly at random from the set of permutations on vectors of size m.

The random shuffle algorithms, in [76] can permute the vector $\vec{\boldsymbol{v}} = [x_1, ..., x_m]$ of m elements in $O(m)$ time. The algorithm works as follows.

1. Set: $j \leftarrow m$.
2. Pick a value, u, uniformly at random from the range $(0, 1)$.
3. Set: $k \leftarrow \lfloor ju \rfloor + 1$.
4. Exchange the values at position k and j: $x_k \leftrightarrow x_j$.
5. Set: $j \leftarrow j - 1$.
6. If $j > 1$ return to step 2; exit otherwise.

In order to turn the above random shuffle protocol into a pseudorandom one, we can use a pseudorandom function to generate the value u (i.e. a pseudorandom value uniformly distributed over the same range). Note, given the permuted vector, one who possesses the secret key can figure out the original ordering of the vector elements in $O(m)$ time.

2.5 Representing Sets by Polynomials

The idea of using a polynomial to represent a set was put forth by Freedman *et al.* in [46]. Since then, the idea has been widely used (e.g. [75, 37, 88, 78]). In this representation, set elements are represented as elements in a finite field $\mathbb{F}_p$ and the set

is represented as a polynomial $\rho(x)$ over the field. For the universe of set elements, $\mathcal{U}$, we define a public field $\mathcal{R} = \mathbb{F}_p$ that is big enough to encode all elements in $\mathcal{U}$. For every $u_i \in \mathcal{U}$, we encode it as:

$$s_i = u_i || \mathtt{G}(u_i), \tag{2.1}$$

where $\mathtt{G}$ is a cryptographic hash function with sufficiently large output size, e.g. $|\mathtt{G}(.)| \geqslant 80$-bit. So given the field's arbitrary element, $s \in \mathbb{F}_p$, and $\mathtt{G}$'s output size, we can parse s into a and b, such that $s = a||b$ and $|b| = |\mathtt{G}(.)|$. Then, we check $b \stackrel{?}{=} \mathtt{G}(a)$. If $b = \mathtt{G}(a)$ then we say s is valid; otherwise, it is invalid (note that if $|s| \leq |\mathtt{G}(.)|$ then the element is invalid and no further check is required). Note that the bit-length of each s_i ranges over $(|\mathtt{G}(.)|, z + |\mathtt{G}(.)|]$, where z is maximum bit-length that an element can have in the set universe.

From now on by set element, we actually mean the encoded form of the element. Shortly, we will explain why this encoding is needed. We can represent a set, $S = \{s_1, ..., s_d\}$, by a polynomial:

$$\rho(x) = \prod_{i=1}^{d} (x - s_i),$$

where $\rho(x) \in \mathcal{R}[x]$, $\mathcal{R}[x]$ is the polynomial ring that consists of all polynomials with coefficients from the field, $s_i \in \mathbb{F}_p$ and $|S| = d$. For two sets $S^{(A)}$ and $S^{(B)}$ represented by polynomials $\rho^{(A)}$ and $\rho^{(B)}$ respectively, polynomial $\rho^{(A)} \cdot \rho^{(B)}$ represents the set union, (i.e. $S^{(A)} \cup S^{(B)}$). Also, $gcd(\rho^{(A)}, \rho^{(B)})$ represents the set intersection (i.e. $S^{(A)} \cap S^{(B)}$), where gcd stands for the greatest common divisor. For two degree d polynomials $\rho^{(A)}$ and $\rho^{(B)}$, and two degree d random polynomials $\gamma^{(A)}$ and $\gamma^{(B)}$ whose coefficients are picked uniformly at random from the field, it is proved in [75, 20] that:

$$\theta = \gamma^{(A)} \cdot \rho^{(A)} + \gamma^{(B)} \cdot \rho^{(B)} = \mu \cdot gcd(\rho^{(A)}, \rho^{(B)}), \tag{2.2}$$

where μ is a uniformly random polynomial. In other words, if $\rho^{(A)}$ and $\rho^{(B)}$ are the polynomials representing sets $S^{(A)}$ and $S^{(B)}$, then polynomial θ contains only information about $S^{(A)} \cap S^{(B)}$ and no information about other elements in $S^{(A)}$ or $S^{(B)}$.

In order to find the intersection, one can extract the roots of polynomial θ, and then consider the set of valid roots as the intersection [1]. Note that the polynomial θ

[1] To find the roots of a polynomial over a field, we can first factorize it to get a set of monic polyno-

contains the roots of both polynomials $gcd(\rho^{(A)}, \rho^{(B)})$ and μ. Polynomial $gcd(\rho^{(A)}, \rho^{(B)})$ represents the two sets intersection and we are interested in its roots, but the roots of μ should be discarded. Therefore, we need a way to distinguish which roots should be retained and which should be discarded. To this end, we use the encoding technique, presented in equation 2.1, to give a structure to each set element. In particular, the probability that the random polynomial, μ, has any root having the above structure is negligibly small (e.g. lower that 2^{-80}) if $\mathtt{G}$'s output size is large enough (e.g. $|\mathtt{G}(.)| \geqslant 80$-bit). Thus, the encoding allows us to effectively eliminate the invalid roots [75]. It should be noted that the encoding increases the bit-length of set elements; therefore, when the set elements are encoded a bigger field size is needed than when the elements are not encoded (e.g. 80-bit bigger).

2.6 Polynomials in Point-value Form

In section 2.5, we showed that a set can be represented as a polynomial and set intersection can be computed by polynomial arithmetic. The previous PSI protocols (e.g. [46, 75, 37]) using polynomial representation of sets, represent a polynomial as a vector of the polynomial's coefficients, i.e. they represent a degree d polynomial $\rho(x) = \sum_{i=0}^{d} a_i x^i$ as a vector $\vec{\boldsymbol{a}} = [a_0, ..., a_d]$. This representation, while it allows the protocols to correctly compute the result, has a major disadvantage. The complexity of multiplying two polynomials of degree d in this form is $O(d^2)$. In PSI protocols, this leads to significant computational overheads. Usually, in these protocols, one polynomial needs to be encrypted and the polynomial multiplication has to be done homomorphically. Homomorphic multiplication operations are computationally expensive as they require exponentiation operations. Thus, PSI protocols using the coefficient-based polynomial representation are not scalable.

We can solve this problem by representing polynomials in another well-known form: the point-value form. A degree d polynomial $\rho(x)$ can be represented as a set of m ($m > d$) point-value pairs $\{(x_1, y_1), ..., (x_m, y_m)\}$ such that all x_i are distinct and $y_i = \rho(x_i)$ for all i, $1 \leq i \leq m$. If x_i are fixed, we can omit them and represent polynomials as a vector $\vec{\boldsymbol{y}} = [y_1, ..., y_m]$. As stated in section 2.5, the bit-length of each encoded element s_i ranges over $(|\mathtt{G}(.)|, z + |\mathtt{G}(.)|]$, where z is maximum bit-length of element in the set universe. Therefore, if we pick values x_i (uniformly at random)

mials (see [72] for some algorithms), then find the monic degree-1 polynomials' roots.

from the elements whose bit-length range over $[1, |\mathtt{G}(.)|]$, then x_i would not represent any encoded set element (and x_i would not be a root of polynomial representing the set).

As the theorem of interpolating polynomial states for a set $\{(x_1, y_1), ..., (x_m, y_m)\}$ (where $x_i \neq x_j$), there exists a unique polynomial, $\zeta(x)$, of degree at most $m-1$, such that $\forall i, 1 \leq i \leq m : \zeta(x_i) = y_i$. In other words, a polynomial in point-value form can be converted into coefficient form by polynomial interpolation [4, 16]. To this end, we can use the modified (or improved) Lagrange formula:

$$\zeta(x) = \eta(x) \sum_{i=1}^{m} \frac{\psi_i}{x - x_i} \cdot y_i$$

where $\eta(x) = \prod_{i=1}^{m}(x - x_i)$ and $\psi_i = \frac{1}{\prod_{\substack{i=1 \\ i \neq k}}^{m} (x_i - x_k)}$.

It is well known that when the values ψ_i are precomputed, then the polynomial $\zeta(x)$ can be interpolated with $O(m)$ computation complexity [4, 16]. Also, the computation complexity of evaluating a degree m polynomial at value x_i is $O(m)$ and it involves m multiplication and m addition operations (using Horner's rule) [76].

We can add or multiply two polynomials (represented in point-value form) by adding or multiplying their corresponding y-coordinates; for two degree d polynomials $\rho^{(A)}$ and $\rho^{(B)}$ represented in point-value form by two vectors $\vec{\boldsymbol{y}}^{(A)} = [y_1^{(A)}, y_2^{(A)}, ..., y_m^{(A)}]$ and $\vec{\boldsymbol{y}}^{(B)} = [y_1^{(B)}, y_2^{(B)}, ..., y_m^{(B)}]$, the sum of the two polynomials is computed as:

$$\rho^{(A)} + \rho^{(B)} = (y_{\,1}^{(A)} + y_{\,1}^{(B)}, y_{\,2}^{(A)} + y_{\,2}^{(B)}, ..., y_m^{(A)} + y_m^{(B)}),$$

and the product of the polynomials is calculated as:

$$\rho^{(A)} \cdot \rho^{(B)} = (y_{\,1}^{(A)} \cdot y_{\,1}^{(B)}, y_{\,2}^{(A)} \cdot y_{\,2}^{(B)}, ..., y_m^{(A)} \cdot y_m^{(B)}).$$

It should be noted that the product $\rho^{(A)} \cdot \rho^{(B)}$ is a polynomial of degree $2d$, so each polynomial $\rho^{(A)}$ and $\rho^{(B)}$ must be represented by at least $2d + 1$ points to allow the correct result to be interpolated. The key benefits of point-value representation are that multiplication complexity is reduced to $O(d)$. Furthermore, all the process (e.g. multiplication or addition) on a pair of y-coordinates (e.g. $(y_i^{(A)}, y_i^{(B)})$) can be done in parallel and independent of the other y-coordinates. Such features make our protocols much more scalable.

2.7 Hash Tables

In this thesis, we use a hash table for two reasons; namely, to achieve *efficiency* when (a) computing PSI, and (b) updating data.

In our protocols, a c-element set is represented as a polynomial that needs to be factorized by the result recipient to retrieve the intersection, at the end of the protocols. However, the computation complexity of polynomial factorization is quadratic in the degree of the polynomial being factorized, so it is $O(c^2)$. To improve performance, we can use a hash table to break down a large set into h smaller sized subsets, and represent each subset as a polynomial. Roughly speaking, each client distributes its set elements among the hash table bins, represents each bin as a polynomial, secures them and outsources the entire hash table to the cloud. In this setting, at the end of the protocols, the result recipient factorizes h polynomials of degree $\frac{c}{h}$. Thus, the total cost is reduced from $O(c^2)$ to $O(\frac{c^2}{h})$.

Moreover, we can leverage set partitioning to achieve reduced update cost. In order for a client to insert/delete an element into/from the outsourced dataset (partitioned as above), it only needs to access the bin that should contain the element. Thus, the client accesses and updates only one bin of the hash table. In contrast, when the whole set is encoded as one polynomial, the entire polynomial needs to be downloaded for data update and this costs $O(c)$.

In general, the hash table in the PSI protocols is used as follows. First, public parameters including a random hash function $\mathtt{H}$, the number of bins in the hash table and max bin size are picked. For the parties to compute the sets intersection, each party I maps its set element $s_i^{(I)}$ to the table by computing an address $j = \mathtt{H}(s_i^{(I)})$, using the hash function whose output is modeled as a uniform random number. Then, it inserts $s_i^{(I)}$ into the corresponding bin $\mathtt{HT}_j$. Because the hash function is deterministic, if an element is in the intersection, both parties map it to the same bin. In this case, if the PSI protocol runs on that bin, it will include the element in the result. In our protocol, for the security reasons, the cloud should not learn the original address of a bin, e.g. j. For the client to securely update each bin, it initially tags each bin with a deterministic unique label, l_j, using a secret key at setup phase. The client sends the bins and labels (shuffled) to the cloud. Then, each time the client wants to update a bin, it sends to the cloud the label associated with the bin.

The hash table approach requires pre-determined parameters (e.g. a bin maximum capacity, the maximum number of bins) in order for the PSI to be computed correctly.

Since the hashing process is probabilistic, we need to avoid overloading the bins, i.e. we need that with a high probability each bin receives at most a specific number of elements. Hash table parameters can be determined by analyzing it under the balls into bins model which has been extensively studied in the literature [100, 14]. The aim of algorithms following this paradigm is to assign a set of independent objects (tasks, balls) to a set of resources (servers, bins), and this enables the load to be distributed among them as evenly as possible. The model is used in a variety of areas such as online load balancing protocols (see [14] for a survey and analysis), oblivious ram protocols (e.g. [107]), etc. According to this model, given the maximum number of elements (or balls), c, and the probability that a bin receives more than d elements, we can determine the number of bins as follows.

Theorem 1. ***(Upper Tail in Chernoff Bounds)*** *Let X_i be a random variable defined as $X_i = \sum_{i=1}^{c} Y_i$, where $Pr[Y_i = 1] = p_i$, $Pr[Y_i = 0] = 1 - p_i$, and all Y_i are independent. Let the expectation be $\mu = \mathrm{E}[X_i] = \sum_{i=1}^{h} p_i$. Then:*

$$Pr[X_i > d = (1+\sigma)\cdot\mu] < \Big(\frac{e^{\sigma}}{(1+\sigma)^{(1+\sigma)}}\Big)^{\mu}, \forall\sigma > 0 \tag{2.3}$$

Note that in the balls and bins model, the expectation is $\mu = \frac{c}{h}$, where c is the number of balls and h is the number of bins. Inequality 2.3 provides the probability that bin i gets more than $(1+\sigma)\cdot\mu$ balls. In order to determine such probability for any bin, we need the following lemma.

Lemma 1. ***(The Union Bound)*** *Let $E_1, ..., E_h$ be any collection of events. Then*

$$Pr[E_1 \vee ... \vee E_h] \leq \sum_{i=1}^{h} Pr[E_i]$$

If we combine the Theorem 1 and Lemma 1, then we can calculate the probability that any bin receives more than $d = (1+\sigma)\cdot\frac{c}{h}$ balls (or elements) as below.

$$\begin{aligned} Pr[\exists i, X_i > d] &\leq \sum_{i=1}^{h} Pr[X_i > d] \\ &\leq h \cdot \Big(\frac{e^{\sigma}}{(1+\sigma)^{(1+\sigma)}}\Big)^{\frac{c}{h}} \end{aligned} \tag{2.4}$$

Thus, if we know the number of elements c, we can find the number of bins h and σ such that the probability of any bin exceeding the maximum size $d = (1+\sigma)\cdot\frac{c}{h}$ is negligibly small (e.g. 2^{-40}) using the inequality 2.4.

2.8 Adversary Types

A cryptographic protocol is formalized, and proven secure in the presence of certain type of adversaries such as semi-honest, covert and malicious [52]. In the following, we give the intuition behind semi-honest and malicious adversaries that we consider in this thesis.

Semi-honest Adversaries. They are also known as "passive" or "honest-but-curious" adversaries. In the semi-honest adversarial model, the party corrupted by such adversaries correctly follows the protocol specification. Nonetheless, the adversary obtains the internal state of the corrupted party, including the transcript of all the messages received, and tries to use this to learn information that should remain private. This model covers many typical practical settings such as protection against insider attacks, or cases where the parties essentially trust each other but want to ensure that nothing beyond the protocol output is leaked. Designing and evaluating the performance of protocols in the semi-honest model is the first step towards protocols with stronger security guarantees. Most protocols for practical privacy-preserving applications focus on the semi-honest model as it is more efficient (e.g. [11, 95, 96]).

Malicious Adversaries. They are also called "active adversaries". The malicious adversarial model considers the strongest type of adversaries, which are allowed to arbitrarily deviate from the protocol in order to learn private inputs of the other parties, to influence the outcome of computation, etc. Not surprisingly, protection against such adversaries is more challenging and protocols that achieve this level of security are usually much less efficient [60].

Moreover, typically there are two corruption models: static (or non-adaptive) and dynamic (adaptive). In the static corruption model, the adversaries have a fixed set of parties that they control. In this model, honest and corrupted parties remain the same throughout the protocol. On the other hand, in the dynamic corruption model, adversaries have the capability of corrupting parties during the computation. The choice of whom to corrupt, and when, can be arbitrarily decided by the adversaries and may depend on what they have seen throughout the protocol execution. As highlighted in [61], in general, dynamic adversaries are much harder to protect against and protocols secure against dynamic adversaries are more complex and less efficient. In this thesis, we consider the static model for the sake of simplicity.

2.9 Security Models

The security definition of a PSI protocol follows the standard ideal/real simulation paradigm of secure computation [52]. Accordingly, we follow the same definition and model in this thesis. The model that we consider in this research is that of multi-party computation in the presence of static adversaries. All the definitions in this section are according to [52].

Two-party Computation. A two-party protocol problem is captured by specifying a random process that maps pairs of inputs to pairs of outputs, one for each party. Such process is referred as a functionality, and denoted by $F : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^*$, where $F = (f_1, f_2)$. More specifically, for every pair of inputs (x, y), where $x, y \in \{0,1\}^n$, the output-pair is a random variable $(f_1(x,y), f_2(x,y))$ ranging over pairs of strings. The party, whose input is x, wishes to obtain $f_1(x,y)$, and the other party, whose input is y, wishes to obtain $f_2(x,y)$.

2.9.1 Security in the Presence of Semi-honest Adversaries

Loosely speaking, a protocol is secure, in the semi-honest adversarial model, if whatever can be computed by a party in the protocol can be computed using its input and output only. This statement is formalized according to the simulation paradigm. Intuitively, we require that a party's view in a protocol execution can be simulated given only its input and output. This implies that the parties learn nothing from the protocol execution.

Let F be a probabilistic polynomial-time functionality defined as above and Γ be a two-party protocol for computing F. The party i's view (during an execution of the protocol, Γ) on input tuple (x, y) is denoted by $\mathsf{VIEW}_i^\Gamma(x,y)$ and equals $(w, r^i, m_1^i, ..., m_t^i)$ where $w \in \{x, y\}$ is the input of the i^{th} party, r^i is the outcome of the i^{th} party's internal random coin tosses and m_j^i represents the j^{th} message that the party received. The output of the i^{th} party during an execution of Γ on (x, y) is denoted by $\mathsf{Output}_i^\Gamma(x,y)$ and can be generated from its own view of the execution. The joint output of both parties is denoted by $\mathsf{Output}^\Gamma(x,y) = (\mathsf{Output}_1^\Gamma(x,y), \mathsf{Output}_2^\Gamma(x,y))$.

Definition 10. *(Privacy with Respect to Semi-honest Behavior): Let $F = (f_1, f_2)$ be a functionality. We say that a protocol Γ securely computes F in the presence of static*

semi-honest adversaries if there exists probabilistic polynomial-time algorithms SIM_1 *and* SIM_2 *such that:*

$$\{(\mathsf{SIM}_1(x, f_1(x,y)), F(x,y))\}_{x,y} \stackrel{c}{\equiv} \{(\mathsf{VIEW}_1^{\Gamma}(x,y), \mathsf{Output}^{\Gamma}(x,y))\}_{x,y},$$
$$\{(\mathsf{SIM}_2(y, f_2(x,y)), F(x,y))\}_{x,y} \stackrel{c}{\equiv} \{(\mathsf{VIEW}_2^{\Gamma}(x,y), \mathsf{Output}^{\Gamma}(x,y))\}_{x,y},$$

where $x, y \in \{0,1\}^*$ *and* $|x| = |y|$.

In other words, the view of a party can be simulated by a probabilistic polynomial-time algorithm given access to the party's input and output only. For the case where the functionality F is deterministic, $F(x,y) = \mathsf{Output}^{\Gamma}(x,y)$, it would suffice to show that:

$$\{\mathsf{SIM}_1(x, f_1(x,y))\}_{x,y} \stackrel{c}{\equiv} \{\mathsf{VIEW}_1^{\Gamma}(x,y)\}_{x,y},$$
$$\{\mathsf{SIM}_2(y, f_2(x,y))\}_{x,y} \stackrel{c}{\equiv} \{\mathsf{VIEW}_2^{\Gamma}(x,y)\}_{x,y}.$$

It should be noted that the PSI protocol's functionality is deterministic and we can use the latter definition.

2.9.2 Security in the Presence of Malicious Adversaries

Now we turn our attention to the security definition for the case of malicious adversaries who may use any efficient attack strategy and thus may arbitrarily deviate from the protocol specification. In this case, in contrast to the case of semi-honest adversaries, the adversary may not use the input that is provided. Therefore, beyond the possibility that a corrupted party may learn more than it should, we also require correctness, that means a corrupted party cannot cause the output to be incorrectly distributed. Moreover, we require independence of inputs meaning that a corrupted party cannot make its input depend on the other party's input. In order to capture the threats, the security of a protocol is analyzed by comparing what an adversary can do in the protocol to what it can do in an ideal scenario that is secure by definition. This is formalized by considering an ideal computation involving an incorruptible trusted third party (TTP) to whom the parties send their inputs. The TTP computes the functionality on the inputs and returns to each party its respective output. Intuitively, a protocol is secure if any adversary interacting in the real protocol (where no TTP exists) can do no more harm than if it were involved in the ideal computation (with the involvement of TTP). Here we consider a simpler definition of security in the case of single-output

functionalities (i.e., functionalities in which only one party obtains an output). For the sake of simplicity and without loss of generality, assume that only the first party, P_1 obtains an output (from the functionality F); that is, $F(x, y) = (f_1(x, y), \Lambda)$, where Λ denotes an empty string.

Execution in the Ideal Model. Let P_1, and P_2 be the parties participating in the protocol, $i \in \{1, 2\}$ be the index of the corrupted party, and SIM be a non-uniform probabilistic polynomial-time adversary (or simulator). Also, let x and y be the input of party P_1 and P_2, respectively. And let z be an auxiliary input given to the adversary $\mathcal{A}$. The honest party, P_j, sends its received input to the TTP. The corrupted party, P_i, controlled by the adversary, may either abort (by replacing the input with a special abort message $\perp$), send its received input, or send some other input of the same length to the trusted party. This decision is made by the adversary and may depend on the input value of P_i and the auxiliary input. Upon obtaining an input pair, (x, y), the trusted party replies to the first party with $f_1(x, y)$. If it receives an abort messages from any party, it outputs $\perp$ to the first party. An honest party always outputs the message it has obtained from the trusted party. A malicious party may output an arbitrary (polynomial-time computable) function of its initial input (auxiliary input and its coin tosses) and the message it has obtained from the trusted party. The ideal execution of F on inputs (x, y) and auxiliary input z is denoted by $\mathsf{IDEAL}^{F}_{\mathsf{SIM}_i(z)}(x, y)$ and is defined as the output pair of the honest party and the adversary from the ideal execution.

Execution in the Real Model. In the real model, a real two-party protocol Γ is executed and there exists no TTP. In this setting, the adversary $\mathcal{A}$ sends all messages on behalf of the corrupted party, and may follow an arbitrary polynomial-time strategy. Whereas, the honest party follows the instructions of Γ. Let $i \in \{1, 2\}$ be the index of the corrupted party. The real execution of the protocol, Γ, is denoted by $\mathsf{REAL}^{\Gamma}_{\mathcal{A}_i(z)}(x, y)$, which is the joint output of the parties engaging in the real execution of the protocol, in the presence of the adversary $\mathcal{A}_i$.

Security in the Malicious Model. Having defined the ideal and real models, we can now define the security of protocols. Loosely speaking, the definition asserts that a secure protocol (in the real model) emulates the ideal model. This is formulated by stating that adversaries in the ideal model are able to simulate executions of the real-model protocol.

Definition 11. *Let F be a two-party functionality and let Γ be a two-party protocol that computes F. It is said protocol Γ securely computes F with abort in the presence of static malicious adversaries if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}_i$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary SIM_i for the ideal model, such that for every $i, i \in \{1, 2\}$:*

$$\{\mathsf{IDEAL}^{F}_{\mathsf{SIM}_i(z)}(x, y)\}_{x,y,z} \stackrel{c}{\equiv} \{\mathsf{REAL}^{\Gamma}_{\mathcal{A}_i(z)}(x, y)\}_{x,y,z}$$

where $x, y, z \in \{0, 1\}^*$ and $|x| = |y|$.

Chapter 3

Related Work

In this chapter, we begin with an overview of cloud computing including its deployment models, the services it offers and its vulnerabilities. Then, we provide a survey of protocols designed to address the problem of private set intersection. We categorize the protocols into two groups: (1) *traditional PSI protocols* requiring data owners to interact directly with each other to compute the intersection, (2) *delegated PSI protocols* enabling data owners to take advantage of the cloud capabilities for PSI computation. Our main focus, in this thesis, is on the second group. We briefly describe each of the protocols, their building blocks, the security model(s) they consider, their security flaws (if any) and we review their suitability for the cloud computing setting.

3.1 Cloud Computing Overview

Cloud computing is a specialized distributed computing paradigm. The cloud provides a virtualized pool of computing resources (e.g. storage, processing power, memory, applications, and so on). It can provide a variety of vital services to a wide range of clients. In general, the services can be grouped into three categories [117]: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS).

1. **SaaS** refers to providing applications to clients who can access them through the Internet. Rackspace [1], Google Compute Engine [2], and Salesforce.com [3] are examples of SaaS providers.

[1] http://www.rackspace.co.uk/
[2] https://cloud.google.com/compute/
[3] http://www.salesforce.com/

2. **PaaS** refers to providing a platform (including programming languages, libraries, operating systems, etc) allowing clients to develop, run and manage their own applications. Google App Engine [1] and Microsoft Windows Azure [2] are examples of PaaS providers.

3. **IaaS** refers to providing (virtualized) computing resources (e.g. storage, networks, memory, processors) to clients who can provision them via a user interface. Amazon EC2 [3] and GoGrid [4] are examples of IaaS providers.

Cloud computing is currently offered in four deployment models: private, public, community, and hybrid [117, 65, 112]. A public cloud may be owned, or managed by a business, academic or government organization or a combination of them. The cloud infrastructure, in this model, is on the premises of the cloud provider or a third party. This model targets a wide range of customers, including the general public. Microsoft Windows Azure and Amazon EC2 are examples of public cloud. A private cloud is provided exclusively to one client (e.g. government). In this model, the infrastructure can be hosted on-premises or off-premises at the service provider's side. Examples of private cloud include Amazon Virtual Private Cloud [5] and IBM Private Cloud [6]. A community cloud is used for a specific community of clients (e.g. a set of researchers, or companies) who have common interests and concerns. Google Apps for Government [7] and Microsoft Government Community Cloud [8] are two examples of the community model. A hybrid cloud is a composition of two or more distinct cloud models (private, community, or public). Examples of hybrid cloud include Microsoft hybrid cloud [9] and VMware vCloud Air [10].

The cloud can benefit clients in different ways. For example, clients no longer need to worry about software updates or component upgrades, as all are maintained by the cloud provider, while the resources can be scaled up dynamically on demand. In order for clients to get access to these services, in most cases, they need only Internet access and this makes the services accessible from different geographical locations.

[1] https://cloud.google.com/appengine/
[2] https://azure.microsoft.com/en-gb/
[3] https://aws.amazon.com/ec2/
[4] https://www.datapipe.com/gogrid/
[5] https://aws.amazon.com/vpc/
[6] https://www.ibm.com/cloud-computing/solutions/private-cloud
[7] https://www.google.com/work/apps/government/
[8] https://azure.microsoft.com/en-gb/overview/clouds/government/
[9] https://www.microsoft.com/en-gb/cloud-platform/hybrid-cloud
[10] http://www.vmware.com/cloud-services/infrastructure.html

Thus, instead of buying hardware and software that could be outdated, and depreciated, clients pay only for the services they use. These investments are made by the service providers and this paradigm could be cost effective for both clients and cloud (service providers), due to economies of scale.

Due to the benefits that the cloud can offer to individuals, educational organizations [84] and businesses [83], use of cloud computing is swiftly gaining momentum among them. A study by IDG in 2014 [1] suggested that 69% of enterprises have either applications or infrastructure running in the cloud. Despite the numerous advantages that the cloud offers, some clients avoid adopting such paradigm, due to a series of serious concerns. The concerns mainly stem from the belief that the clients would lose control of their data after the data is outsourced. In other words, in the current cloud setting, the clients have to fully trust the cloud provider, and hope it protects their sensitive data. But, with increasing cloud adoption, the cloud has become an appealing target of (insider or outsider) attacks, e.g. [120, 57, 41, 103, 64, 119, 122, 87], that result in information security breaches, unauthorized disclosure or modification of confidential information, data stealing, denial of service, violation of service level agreement, etc. In this thesis, by the *cloud* we mean the public cloud that can be utilized by the general public, and widely used, but its infrastructure is managed by a service provider who cannot be fully trusted (as we outlined above).

3.2 Traditional PSI Protocols

Computing the intersection of private sets is a special case of the more general secure multi-party computation problem. The techniques and protocols designed for the general case allow parties to jointly run a certain function on private inputs contributed by mutually distrustful parties. Most PSI variations can be realized via this general secure multi-party techniques [68]. However, researchers have realized that it is usually more efficient to design special-purpose protocols, as highlighted in [33, 46, 75, 68]. Therefore, they have proposed protocols specifically designed to address the private set intersection problem, considering the same setting as traditional secure MPC protocols where data owners jointly run the operation.

Private Set Intersection (PSI) protocols were introduced by Freedman *et al.* in [46]. They initially proposed a protocol for semi-honest adversaries that is mainly based on

[1] http://www.idgenterprise.com/resource/research/idg-enterprise-cloud-computing-study-2014/

two primitives: Paillier homomorphic encryption, and polynomial representation of sets. The basic protocol supports two parties: A and B. Party A represents its set, $S^{(A)} = \{s_1^{(A)}, s_2^{(A)}..., s_v^{(A)}\}$, as a polynomial $\rho(x)$ whose roots are the set elements:

$$\rho(x) = (x - s_1^{(A)})(x - s_2^{(A)})...(x - s_v^{(A)}) = \sum_{i=0}^{v} a_i x^i.$$

Then, it encrypts the coefficients of the polynomial $E_{pk}(a_i)$ ($\forall i, 0 \leq i \leq v$) and sends them to party B. Upon receiving the message, party B homomorphically evaluates the encrypted polynomial at each element of its set, $S^{(B)} = \{s_1^{(B)}, s_2^{(B)}..., s_w^{(B)}\}$, as follows.

$$\forall j, 1 \leq j \leq w : v_j = E_{pk}(r_j \cdot \rho(s_j^{(B)}) + s_j^{(B)}),$$

where each r_j is a random value. It sends the encrypted values to client A. If the two sets have any elements in common, they would appear in an encrypted form in the result. Whereas, for other elements not in the intersection, encrypted random values would appear in the result. The protocol has been proven secure in the standard model. In general, protocols security is considered in the standard model when it relies on some well-known computational problems, e.g. the RSA, discrete logarithm, factoring assumptions.

In the above protocol (i.e. [46]), since party B does not know the secret key and the encryption is semantically secure, it cannot learn anything about the other party's set elements. Also, the fresh random values hide party B's elements not in the intersection from party A, so party A learns only the intersection. The authors improved the basic protocol's performance, by using a hash table. Since evaluating a large degree polynomial at a value requires expensive exponentiations, linear to the polynomial degree, at party B side, they use a hash table to break down party A's set into smaller subsets. In this case, the parties first fix the hash table parameters (e.g. hash function, the number of bins). Then, party A can represent each subset as a small degree polynomial, and send the encrypted polynomials to party B who can now evaluate the small degree encrypted polynomials at the corresponding elements of its set, which costs less than the basic scheme. Also, they extended the two-party protocol to a multi-party scheme by sending all the encrypted polynomials along with some random matrix to one of the parties who obliviously combines them and sends them to each party. For the two-party case, the communication complexity of the scheme is $O(v+w)$, where v and w denote the cardinality of party A's and B's sets, respectively. The computation complexity for

party A involves $O(v + w)$ (modular) exponentiation operations, and for B involves $O(w \log \log v)$ exponentiation operations.

The same authors, in [45] provided a complex protocol to accommodate malicious adversaries. In general, to prevent malicious behavior, in protocols the parties must demonstrate that they are well-behaved. To this end, protocols normally use a standard zero-knowledge proof (of knowledge) in the semi-honest protocols, as it is done in [45] (i.e. for proving the correctness of (a) the polynomials sent by party A, (b) the computation performed by party B). Informally, a zero-knowledge proof (ZK) protocol (e.g. [27, 35]) enables one party, called prover, to convince another party, called verifier, that it knows some facts (e.g. a secret value) without revealing to the verifier any information about its knowledge. Protocols designed for secure multi-party computation, can use ZK to enable each party to convince the others that it follows the protocol correctly. The ZK protocols generally involve exponentiation operations that are computationally expensive. The above PSI protocol for malicious adversaries has been proven secure in the non-standard random oracle model. In general, protocols security is considered in the non-standard model when the security relies on some idealized model of computation, e.g. random oracle [12, 71] or ideal cipher model [30]. In the following we briefly explain the (non-standard) random oracle model as it is used more often in PSI protocol design than other non-standard assumptions. In the random oracle model, a hash function is modeled as a publicly accessible random function [12]. A proof in the random oracle model is not fully satisfactory, and considered heuristic, because the proof does not imply that the scheme will remain secure when a concrete hash function is used instead of the random oracle (as a hash function cannot fully perform the role of random oracle); however, it is still used as protocols using this model are often more efficient [71]. In [45] , for the two-party case, the overall communication cost is $O(v + w)$ and the computation involves $O(v + w \log \log v)$ exponentiation operations.

Following the original work, Kissner *et al.* in [75] proposed a number of privacy-preserving protocols for set intersection, cardinality set intersection and over-threshold set union. The protocols allow each party to have a multiset, i.e. a set that may contain many replicas of an element. The protocol for cardinality set intersection, as its name indicates, allows parties to learn only the number of elements common in the sets, while the over-threshold set union protocol enables parties to learn which elements appear in the union of the parties' multisets at least a threshold number of times, and the number of times these elements appeared in the union of the multisets, without

gaining any other information.

The protocols initially consider the semi-honest adversarial model and they are mainly based on threshold Paillier encryption [44], polynomial representation of sets, and polynomial properties. Threshold Paillier encryption is a variant of Paillier encryption that requires a pre-determined number of decryption parties to jointly decrypt a message. Any collusion between fewer than the specified number of decryption parties does not result in a complete decryption and the parties cannot learn any information about the plaintext. This encryption scheme requires parties to jointly share a secret key among them, or a trusted party does that for them before the protocol starts. The protocols in [75] can support multiple parties each of which encrypts the polynomial (representing its set) and broadcasts it to the other parties who can use the encryption's homomorphic features to obliviously perform a certain operation on the received polynomials and send them back to the other parties. At the end of each protocol, the parties combine the resulting polynomials together and jointly decrypt the result.

In the two-party PSI protocol proposed in [75], each party I, $I = \{A, B\}$, encrypts its polynomial $\rho^{(I)}$ and sends $E_{pk}(\rho^{(I)})$ to the other party. Then, upon receiving the other party's encrypted polynomial, the party picks a random polynomial $\gamma^{(I)}$ and multiplies the encrypted polynomial by it, i.e. party A computes $E_{pk}(\rho^{(A)} \cdot \gamma^{(B)})$ and party B computes $E_{pk}(\rho^{(B)} \cdot \gamma^{(A)})$. After that, they homomorphically sum the products to compute the encrypted polynomial whose roots are the set intersection:

$$E_{pk}(\rho^{(A)} \cdot \gamma^{(B)} + \rho^{(B)} \cdot \gamma^{(A)}).$$

Next, they jointly decrypt the result: $\delta = \rho^{(A)} \cdot \gamma^{(B)} + \rho^{(B)} \cdot \gamma^{(A)}$. In order for client I, $I \in \{A, B\}$, to find the intersection, it evaluates $\delta(x)$ at every element, $s_i^{(I)}$, of its set. If the result is zero (i.e. $\delta(s_i^{(I)}) = 0$) , it considers $s_i^{(I)}$ as one of the elements in the intersection. In [75], they proved that polynomial δ contains only information about the intersection and each party cannot decrypt the intermediate result. Due to the threshold property of the encryption, the scheme is secure and leaks nothing beyond the intersection to the other party. This PSI protocol also enables the parties to find the minimum number of times the common elements appear in the sets. This feature is due to the protocol design that allows the resulting polynomial to have the form $\delta(x) = \prod(x - s_i)$. In order for a party to determine the minimum number of times a common element appears in the sets, it computes value a satisfying the following

conditions:

$$(x - e_i^{(I)})^a \mid \delta(x) \wedge (x - e_i^{(I)})^{a+1} \nmid \delta(x),$$

where $\mid$ denotes division. The authors prove the security of the above protocol in the standard model.

The communication complexity of the scheme, for the two-party case, is $O(v+w)$, while the computation complexity is $O(vw)$, for each party. They also utilize zero-knowledge proof to extend the PSI protocol security against a malicious adversary and prove the protocol security in the standard model. In this case, the communication and computation complexity of the protocol is $O(vw)$.

Since then, many PSI protocols (e.g. [33, 32, 38, 96, 95]) have been proposed to improve performance. The PSI protocol in [33], which considers only the semi-honest model, makes use of hash functions and blind RSA signatures to construct an oblivious pseudorandom (OPRF) function. A blind signature scheme enables one to sign a message in a way that its content is blinded before being signed. The resulting signature can be publicly verified against the original blinded message. Knowledge of the blinding factor allows one to remove the blinding of the blind signature and obtain a (message, signature) pair that cannot be correlated to its original (blinded) counterpart [26]. Furthermore, OPRF is a two-party protocol (between a sender and a receiver) that securely computes a pseudorandom function $f_k(.)$ on a key, k, contributed by the sender, and input, x, contributed by the receiver, such that the sender learns nothing from the interaction and the receiver learns only value $f_k(x)$. The PSI protocol based on OPRF works as follows. Party B who holds the secret random key, k, computes $u_j = f_k(s_j^{(B)})$ for each element of its set $s_j^{(B)} \in S^{(B)}$, where $|S^{(B)}| = w$. Then, it sends set $U = \{u_1, ..., u_w\}$ to party A. Next, the parties engage in the OPRF computation of $f_k(s_i^{(A)})$ for each element of party A, $s_i^{(A)} \in S^{(A)}$, where $|S^{(A)}| = v$. In the end, party A learns that $s_i^{(A)} \in \{S^{(A)} \cap S^{(B)}\}$ if $f_k(s_i^{(A)}) \in U$. In this process, party B learns nothing about $S^{(A)}$ (except the size) and party A learns $f_k(s_i^{(A)})$. The protocol has been proven secure in the random oracle model. The computational complexity of this PSI protocol involves $O(v + w)$ and $O(v)$ exponentiations operations for party B and party A, respectively. The communication cost of the protocol is $O(v + w)$. Since the protocol invokes OPRF multiple times and OPRF involves exponentiation operations, the protocol is not very efficient.

Later on, the authors in [32], added zero-knowledge proofs to the protocol to prevent parties from deviating from the protocol, so the protocol becomes secure in the

presence of malicious parties. The authors prove the protocol security in the random oracle model. The communication cost of this PSI scheme is $O(v + w)$, while the computation involves $O(v)$ and $O(v + w)$ exponentiation operations, for party A and B, respectively.

Recently, Dong *et al* in [38] proposed two efficient two-party protocols for the semi-honest and malicious models. They are mainly based on oblivious transfer (OT), garbled Bloom filters and a (XOR) secret sharing technique. The protocols are efficient because they leverage efficient building blocks, and significantly reduce the involvement of (expensive) public key encryption or exponentiation operations. We first briefly explain the building blocks and then outline how the protocols work. The (1-out-of-2) OT [101, 40] is a two-party protocol in which a sender inputs two l-bit strings $\{x_0, x_1\} \in \{0, 1\}^l$ and the receiver inputs a bit b. At the end of the protocol, the receiver obtains x_b but learns no information about x_{1-b}; whereas, the sender learns no information about b. A Bloom filter [18] is a compact data structure for probabilistic efficient set membership testing. The Bloom filter is an array of m bits (initially all set to zero), that can represent a certain number of elements. It is accompanied with a set of k independent hash functions each of which maps elements to index numbers over the range $[0, m - 1]$ uniformly. To insert an element, all the indices (or hash values) of the element are computed and their corresponding bits in the filter are set to 1. To check membership of an element, all its indices are re-computed and checked whether all are set to one in the filter. If all the corresponding bits are one, then the element is probably in the filter; otherwise, it is not in the filter. In this data structure, a false positive is possible, i.e. it is possible that an element is not in the set, but the membership query indicates it is. But, with the right choice of parameters the probability of this error can be made negligible. The secret sharing scheme [106] allows a secret holder to split a secret, d, into n shares such that the secret can be recovered efficiently with any t or more shares. However, given less than t shares, one cannot recover the secret, or learn any partial information about it. In [38], the authors extend the regular Bloom filter to a garbled Bloom filter that uses an array of λ-bit strings, where λ is a security parameter, as opposed to the regular Bloom filter that uses an array of bits. In order to insert an element, b, we first split the element into k λ-bit shares. Then, we hash b using the k hash functions to get k indices. Next, the shares of b are stored in the garbled Bloom filter at the corresponding indices (i.e. one share at each index). In order to check if element b is in the filter, we hash b using the k hash functions. After that, we retrieve all strings at the locations equal to b's hashes, and combine the shares

together. At the end, element b is compared against the result, if they are not equal then b is not in the filter; otherwise, the element is probably in the filter.

Now we explain how the semi-honest secure PSI in [38] works. First, party A encodes its set elements into a Bloom filter and party B computes a garbled Bloom filter that encodes its set elements. Then, they engage in the oblivious transfer protocol and they use their filters as the input; such that, party B plays the sender role and party A acts as the receiver. After the OT ends, for each bit of party A's Bloom filter set to one, the party receives a share. So for those elements in the intersection, it receives all the shares of the elements. But the party does not receive all shares of any element not in the intersection with a high probability (as the authors prove). Therefore, it learns only the intersection. At the end, party A has a garbled Bloom filter encoding the intersection. Party A can check membership of its set elements in the resulting garbled Bloom filter to find out the intersection. The protocol has been proven secure in the random oracle model. The communication cost of this PSI scheme is linear to the set cardinality, $O(w)$. The protocol involves $O(\lambda)$ public key encryption operations, and $O(w)$ efficient private key encryptions, where λ is the security parameter (i.e. $\lambda \in [80, 256]$). Therefore, it involves a small number of public key encryption operations. They extend the semi-honest secure protocol to a protocol secure against malicious adversaries. To this end, they add primitives such as the OT that is secure against malicious adversaries, and private key encryption to the semi-honest secure protocol. This protocol has been proven secure also in the random oracle model. The additional tools used in the malicious variant of the protocol increase the communication and computation complexity of it by a factor of 1.4λ compared to the semi-honest scheme. Both protocols are designed for the two-party case and cannot directly support multiple parties.

Later on, the performance of the semi-honest secure protocol in [38] was improved by Pinkas *et al.* in [96]. To this end, they benefited from recent performance improvements of the oblivious transfer [6] and combined it with a garbled Bloom filter. The protocol considers only the two-party case under the semi-honest model. They use the idea of an oblivious pseudorandom generator (OPRG), a protocol that takes as inputs bits $b^{(A)}$ and $b^{(B)}$ from party A and B respectively. It generates a random string r, and outputs to party I, $I \in \{A, B\}$, value r if the party's input equals 1 (i.e. $b^{(I)} = 1$). Otherwise, the protocol outputs nothing to the party. However, in this process, one party cannot figure out whether the other party learns r. They construct an efficient OPRG, using the OT protocol. The PSI protocol based on OPRG works as follows. First, each

party I, $I \in \{A, B\}$, having a set $S^{(I)}$, where $|S^{(A)}| = w$, $|S^{(B)}| = v$, constructs the Bloom filter, $BF^{(I)}$. Then, they run OPRG using the bits of $BF^{(I)}$ as inputs. Next, each party inserts the random strings, it received from OPRG, into its garbled Bloom filter, $GB^{(I)}$. After that, party A computes set P whose elements are:

$$p_j^{(A)} = \oplus_{j=1}^{k} GB^{(A)}[h_j(s_i^{(A)})],$$

where $s_i^{(A)} \in S^{(A)}$, $S^{(A)}$ is party A's set, $\oplus$ denotes the XOR operation, k denotes the total number of hash functions used for the (garbled) bloom filters, and $GB^{(A)}[h_j(.)]$ means the value of the filter at position j. Next, party A randomly shuffles the set, so it obtains $P = \{p_j^{(A)}, ..., p_k^{(A)}\}$ (for some j and k, where $1 \leq j, k \leq w$), and sends it to the other party. In order for party B to find the intersection, for every element of its set, $s_i^{(B)} \in S^{(B)}$, it checks:

$$\oplus_{j=1}^{k} GB^{(B)}[h_j(s_i^{(B)})] \stackrel{?}{\in} P.$$

If the check is positive, it considers $s_i^{(B)}$ as an element in the set intersection. The main advantage of the protocol is that all its operations can be parallelized. The protocol's overall computation and communication complexities remain the same as the semi-honest protocol in [38], but the runtime is reduced by up to 55%-60%.

The authors also proposed a new scalable PSI protocol for the semi-honest model that supports two clients. The protocol is based on the most efficient OT extension techniques [90, 6] and a hash table, without the involvement of a (garbled) Bloom filter. They introduce the concept of private equality test (PEQT), a protocol that enables parties A and B to check whether their α-bit elements, $s^{(A)}$ and $s^{(B)}$, are equal by engaging in an efficient 1-out-of-2 OT protocol. In the PEQT protocol, party B uses each bit $b_i^{(B)}$ of its element as input and party A uses two uniformly random strings (r_0^i, r_1^i) as inputs. After each OT run, party B obtains the random string corresponding to its bit (e.g. it obtains r_0^i if $b_i^{(B)} = 0$). Next, party A XORs the random strings corresponding to its bits: $u^{(A)} = \oplus_{i=1}^{\alpha} r_j^i$, where the i^{th} bit of $s^{(A)}$ is $j, j \in \{0, 1\}$. Then, it sends the result to the other party. Party B also XORs the random strings obtained from OT, to get a value, $u^{(B)}$, and then compares it to the value that party A sent: $u^{(B)} \stackrel{?}{=} u^{(A)}$. If the equality holds, the two elements ($s^{(A)}$ and $s^{(B)}$) are equal; otherwise, they are unequal. Note that, the PEQT protocol enables them to fully enjoy the recent OT efficiency. The PEQT is extended to efficiently support PSI computation, i.e. to

check whether any set element of party A equals any set element of party B and this can be done in parallel. In order to further improve the performance of the protocol, they make use of a hash table in the PEQT-based PSI protocol. In this case, first each party maps its set elements to the bins of the hash table, and then the above basic protocol runs on each bin. The authors prove the two protocols security in the random oracle model. The protocol based on PEQT is up to five times faster than the one based on OPRG [96]. The protocol's computation cost is linear in the bit-length of the input elements, and the overall computation complexity is $O(w \log w \log w)$. Moreover, the communication complexity is $O(w \log w)$, where w is set cardinality.

The notable difference between the PEQT-based PSI protocol and the one based on OPRG is the dependence of the performance on the security parameter λ. In the OPRG-based protocol, the number of OT invocations is independent of the bit-length, but it scales linearly with λ. On the other hand, the number of times OT is invoked in the PEQT-based PSI protocol is independent of λ, but linear to the bit-length. Therefore, the performance of the latter protocol would degenerate if the input elements are of large size.

Pinkas *et al.* later on in [95] further improved the performance of their previous efficient protocol proposed in [96], by representing each set element as a short length value in a bin. This protocol also considers the semi-honest model for the two-client case. The main technique used in the new protocol is permutation-based hashing proposed in [5]. This hashing technique works as follows. Let $e = e_1||e_2$ be the bit representation of an element, where $|e_1| = \log_2 h$, h is the total number of bins in a hash table, and for the sake of simplicity let h be a power of two. Also, let $Q(.)$ be a random function whose output range is $[0, h-1]$ (i.e. a hash function). When inserting an element, e, into the hash table, we compute its address as: $j = e_1 \oplus Q(e_2)$, and store e_2 in that address, i.e. in the j^{th} bin. Thus, instead of inserting the entire element, we store a shorter representation of it in the table. To compute the intersection, each party utilizes this hashing technique to map its elements into its hash table, and then run the PEQT-based protocol on each bin. This protocol has also been proven secure in the random oracle model. The protocol has 60%-70% less computational overhead than their previous protocol that directly utilizes PEQT [95]. The protocol's computation and communication complexity is $O(w \log w)$, where w is the set cardinality. The above protocol, which uses a combination of permutation-based hashing and PEQT, is the most efficient traditional PSI protocol to date.

Nonetheless, all the aforementioned PSI protocols, including those based on the

generic MPC that can support PSI, have been designed for scenarios where data owners interact directly with each other to jointly compute the intersection, by using a data (representation) that must be locally available.

3.3 Delegated Private Set Intersection Protocols

Delegated private set intersection protocols are designed for scenarios where clients do not trust the cloud with their data. The privacy and integrity of the data and the correctness of the computation matter to the client. This setting is the main focus of this thesis. Delegated PSI protocols can be categorized into two main groups:

1. The protocols that support only one-off PSI delegation. These protocols allow the clients to send encoded (or secured) data to the cloud and the cloud computes the intersection. The clients need to re-encode their data for each PSI computation thus the delegation is one-off. Because each time the data needs to be re-encoded, the clients either need to keep a copy of the data locally or have to download and re-encode the data every time.

2. The protocols that support repeated PSI delegation. In these protocols, clients outsource their encoded data to the cloud. This is done only once. Later on, the cloud can use the outsourced data to compute the intersection, without requiring the clients to keep a copy of the data locally or to download and re-encode the data.

In the following, we review existing work in both groups.

3.3.1 Protocols Supporting only One-off PSI Delegation

In this section, we briefly describe the protocols designed to support only one-off PSI delegation. In these protocols, clients encode their data and send them to the cloud each time they want to delegate to it the computation of the intersection.

Two variants of delegated PSI protocol were proposed in [73]. They consider the case where the cloud computes the result honestly. One variant of the protocol is based on a keyed hash function and XOR operations. The protocol works as follows. Clients first agree on a secret key, k, and a keyed hashed function $H(.)$ that maps an m bits

string to an $m + l$ bits string. Next, client A computes set:

$$S'^{(A)} = \{H(s_1^{(A)}), ..., H(s_v^{(A)})\},$$

where $s_i^{(A)} \in S^{(A)}$, and $S^{(A)}$ is the client's original set. On the other hand, client B generates the set:

$$S'^{(B)} = \{s_1^{(B)} \oplus H(s_1^{(B)}), ..., s_w^{(B)} \oplus H(s_w^{(B)})\},$$

where $s_i^{(B)} \in S^{(B)}$, and $S^{(B)}$ is the client's set. Then, the clients send the new sets $S'^{(A)}$ and $S'^{(B)}$, to the cloud. After that, the cloud computes the set:

$$\begin{aligned} S^{(C)} = \{&s_1'^{(B)} \oplus s_1'^{(A)}, ..., s_1'^{(B)} \oplus s_v'^{(A)}, ..., s_2'^{(B)} \oplus s_1'^{(A)} \\ &, ..., s_2'^{(B)} \oplus s_v'^{(A)}, ..., s_w'^{(B)} \oplus s_1'^{(A)}, ..., s_w'^{(B)} \oplus s_v'^{(A)}\}, \end{aligned}$$

where $s_1'^{(A)} \in S'^{(A)}$ and $s_1'^{(B)} \in S'^{(B)}$. For each value $s_j^{(C)} \in S^{(C)}$ the cloud checks if the first l bits are all set to 0, if they are then it considers value z_j represented by the remaining m bits as one of the elements in the intersection. The cloud sends the sets intersection in plaintext to the client(s). As it is evident from the protocol's description, the cloud can figure out the set elements that are in the intersection. Accordingly, it can figure out some elements of the sets. The protocol's computation complexity is $O(w + v)$, and its communication complexity is $O(z)$, where z is the intersection cardinality. This variant is efficient as it does not involve any public key encryptions or any other expensive operations. However, the protocol is not fully private and leaks the set elements in the intersection to the cloud. Thus, this protocol does not protect the privacy of the computation's input and output from the cloud.

The other variant of the protocol is based on public key encryption, i.e. RSA encryption [105], and uses the encryption's multiplicative homomorphic feature. The protocol proceeds as follows. The clients first jointly compute an RSA modulus, $N = pq$, along with four values b, f, d and e such that $(b + d)e = f \bmod ((p - 1)(q - 1))$, such that neither party knows q and p. The cloud obtains e, client A obtains b and f, and client B gets d and f. Next, client A and B compute sets:

$$S'^{(A)} = \{(s_1^{(A)})^b, ..., (s_v^{(A)})^b\}, S'^{(B)} = \{(s_1^{(B)})^d, ..., (s_w^{(B)})^d\},$$

respectively. The clients send the sets, $S'^{(A)}$ and $S'^{(B)}$, to the cloud. Given the sets, the

cloud computes a new set:

$$S^{(C)} = \{(s_1'^{(B)} \cdot s_1'^{(A)})^e, ..., (s_1'^{(B)} \cdot s_v'^{(A)})^e, ..., (s_2'^{(B)} \cdot s_1'^{(A)})^e \\ , ..., (s_2'^{(B)} \cdot s_v'^{(A)})^e, ..., (s_w'^{(B)} \cdot s_1'^{(A)})^e, ..., (s_w'^{(B)} \cdot s_v'^{(A)})^e\}.$$

Note that if two elements of the clients' sets, e.g. $s_1^{(A)}$ and $s_1^{(B)}$ are equal, then their corresponding value in $S^{(C)}$ would have the following form: $s_1^{(C)} = (s_1^{(A)})^{e(b+d)}$, where $s_1^{(C)} \in S^{(C)}$. Therefore, in order for client A to find the intersection, it generates another set $S''^{(A)} = \{(s_1^{(A)})^f, ..., (s_v^{(A)})^f\}$ and compares the set elements with the elements of $S^{(C)}$. To do so, client A and the cloud engage in a traditional interactive PSI protocol, using $S''^{(A)}$ and $S^{(C)}$ as the inputs. The other client can generate $S''^{(B)}$ using its original set and value f, and run the interactive PSI protocol with the cloud to learn the result. In the latter protocol, the cloud learns nothing about the computation's input/output. Nevertheless, it is much less efficient than the former one as it involves expensive public key operations. Both protocols support multiple clients and their security proofs are based on the random oracle model. The RSA-based protocol's overall communication and computation complexity is $O(wv)$. It should be noted that in the RSA-based protocol, each time the computation is delegated all clients need to jointly choose the new secret parameters and encode their set elements all over again, otherwise, the process would become deterministic. In this case, if client A and B, and then client A and D engage in the protocol, the cloud would learn whether the sets of client B and D have any elements in common, and it can figure out the number of common elements, without the clients' consent. So, in this protocol the clients cannot outsource their private datasets to the cloud once and ask the cloud to run the computation on their private data securely many times.

The delegated PSI protocol presented in [74] considers semi-honest adversaries and is designed for the two-client case. It utilizes Bloom filters and BGN public key encryption scheme [21] that supports homomorphic addition and only one homomorphic multiplication operation. What follows is a high-level description of the protocol. A trusted party first generates the public and secret parameters and distributes them among the other parties. Then, client A generates a Bloom filter and inserts its set elements into it. Next, it encrypts every bit of the Bloom filter and sends it to the cloud. Client B, constructs w Bloom filters, $BF_j^{(B)}$ ($\forall j, 1 \leq j \leq w$), and inserts each set element $s_j^{(B)}$ into the corresponding Bloom filter, $BF_j^{(B)}$. It encrypts every bit of the Bloom filters, and every bit of each element in its set. After that, it sends all

the encrypted values to the cloud. The cloud uses the homomorphic property of the encryption scheme to perform operations on the encrypted values. Next, it sends the encrypted result to client A, who can decrypt it, and retrieve the result.

The protocol preserves the clients' sets privacy and the cloud cannot learn anything about them. The protocol has been proven secure in the standard model. The protocol's communication complexity is $O(w^2)$, while its computation complexity is also quadratic, as it involves $O(wv)$ public key operations. It should be noted that client B needs to encrypt its set representation under the result recipient's (i.e. client A's) public key. Therefore, if client B wants to engage in the protocol with a different result recipients each time it needs to re-encode (i.e. re-encrypt) its data representation. This feature does not allow them to store their data in the cloud once and delegate PSI computation to the cloud when it is needed. In addition to that, since every bit of the elements is encrypted, and public key encryption is used, the protocol is not computationally efficient.

In the same line of work, Kamara *et al.* proposed a set of PSI protocols in [68] allowing multiple clients to offload some portions of the computation to the cloud in the presence of the semi-honest and malicious adversaries. In general, the protocols are based on pseudorandom permutations (PRP) [71], i.e. informally, an invertible pseudorandom function. The semi-honest protocol works as follows. The clients first agree on a secret key k, and a PRP, $f_k(.)$. Each party, I, having a private set, $S^{(I)}$, generates set:

$$S'^{(I)} = \{f_k(s_1^{(I)}), ..., f_k(s_v^{(I)})\},$$

where $s_i^{(I)} \in S^{(I)}$ and $|S^{(I)}| = v$. Then, each client sends its set representation, $S'^{(I)}$, to the cloud. Since PRP is a deterministic function, and both clients use the same key, the cloud can find the elements common in the outsourced sets, and send them to the clients who can reverse the PRP and retrieve the actual set elements in the intersection. The protocol's communication complexity is $O(z)$ and the computation complexity is $O(v)$, where z is the intersection cardinality.

The authors propose another variant of the protocol that enables the clients to detect if the cloud provides an incorrect result, i.e. the malicious model. The protocol requires the clients to agree on a secret key: k, a security parameter: λ, and three sets: P_0, P_1 and P_2 of random elements, where $|P_u| = t, \forall u, 0 \leq u \leq 2$. Then, client A uses the sets P_0 and P_1 and it encodes each element $p_{i,j}$ of the sets as:

$$\forall z, 1 \leq z \leq \lambda : p_{i,j}||z$$

where $1 \leq i \leq v$ and $0 \leq j \leq 1$. This yields a set, $P'^{(A)}$, containing the encoded elements of the two sets. Client B uses P_1 and P_2, encodes the elements of the sets as above and inserts them in $P'^{(B)}$. After that, each client encodes each element $s_i^{(I)}$ of its original set as $\{s_i^{(I)}||1, ..., s_i^{(I)}||\lambda\}$, then evaluates the PRP on each value. This yields a set $S'^{(I)}$. Finally, each client inserts the elements of $P'^{(I)}$ into $S'^{(I)}$, permutes them and sends them to the cloud. Similar to the above semi-honest secure protocol, the cloud compares the encoded elements, finds the intersection and sends it back to the clients. When the client receives the result it ensures all encoded elements of P_1 are in the result and neither of P_0 and P_2 has any encoded element in the result. After that, it can remove the elements of P_1 from the result, and for every element of its original set it checks whether all λ copies of it are in the result. If all copies exist, then the element is in the intersection. If h copies, where $1 \leq h < \lambda$, exist then the result was computed incorrectly, and if there is no copy of an element, then that element is not in the intersection. The communication and computation cost of the malicious secure protocol is $O(\lambda z)$ and $O(\lambda v)$, respectively. Both protocols are efficient, as they are based on PRP that is efficient.

As PRP is deterministic, and both clients use the same key, in both protocols the cloud can figure out the intersection cardinality. The authors suggest that in order to hide the intersection cardinality, the clients can send their set representations to one of the clients, e.g. client A, who compares the clients set representations and broadcasts the result to the other clients. So, in this case, the burden of computation (in addition to communication and storage) is offloaded to this client. The cloud only re-encodes client A's set representation and sends it back to it, to ensure it does not learn anything beyond the intersection about the other clients' set elements. In other words, in such setting, the clients do not delegate the computation to the cloud. Instead, with its minor assistance, they interactively compute the result.

Note that, the three protocols in [68] require the clients to agree on the new parameters and re-encode their sets each time they delegate the computation to the cloud. Also, in the protocol that hides the intersection size, one client receives all the encoded elements of the other clients. The three protocols have been proven secure in the standard model.

3.3.2 Protocols Supporting Repeated PSI Delegation

Now, we analyze the PSI protocols designed to support repeated PSI delegation. In these protocols, clients outsource their encoded data to the cloud only once. Later on, the cloud can use the outsourced data to compute the intersection. Clients do not need to keep locally a copy of the data or to download and re-encode the data.

The protocol proposed in [81] considers the semi-honest adversarial model, and is based on a hash function: $h(.)$, and public key and secret key encryption schemes. More specifically, each client I, $I \in \{A, B\}$, first generates three vectors:

$$\vec{\boldsymbol{t}}^{(I)} = [P_{sk}(s_1^{(I)}), ..., P_{sk}(s_w^{(I)})], \vec{\boldsymbol{e}}^{(I)} = [h(s_1^{(I)}) + r_1^{(I)}, ..., h(s_w^{(I)}) + r_w^{(I)}],$$
$$\vec{\boldsymbol{d}}^{(I)} = [P_{sk}(r_1^{(I)}), ..., P_{sk}(r_w^{(I)})],$$

using its original set $S^{(I)} = \{s_1^{(I)}, ..., s_w^{(I)}\}$, where the values $r_i^{(I)}$ are picked uniformly at random, $E_{pk}(a)$ and $P_{sk}(a)$ denote the ciphertext of public key and secret key encryption scheme respectively. Each client outsources the three vectors, $\vec{\boldsymbol{t}}^{(I)}$, $\vec{\boldsymbol{e}}^{(I)}$ and $\vec{\boldsymbol{d}}^{(I)}$, to the cloud. When client A gets intersected in the intersection of its set and client B's set, it downloads the entire vector $\vec{\boldsymbol{d}}^{(A)}$, decrypts the vector elements to get the values $r_i^{(A)}$. Then, it picks a single random value, b, and sums it with every $r_i^{(A)}$ as follows. $\forall i, 1 \leq i \leq w : g_i^{(A)} = b + r_i^{(A)}$. It encrypts b under client B's public key: $E_{pk_B}(b)$. Then, it sends vector $\vec{\boldsymbol{g}}^{(A)} = [g_1^{(A)}, ..., g_w^{(A)}]$ and $E_{pk_B}(b)$ to the cloud who sends $\vec{\boldsymbol{d}}^{(B)}$ and $E_{pk_B}(b)$ to client B. Client B decrypts the vector elements and $E_{pk_B}(b)$. Next, it computes a new vector $\vec{\boldsymbol{g}}^{(B)} = [g_1^{(B)}, ..., g_w^{(B)}]$, where $g_i^{(B)} = b + r_i^{(B)}$. It sends $\vec{\boldsymbol{g}}^{(B)}$ to the cloud. Given the two vectors, $\vec{\boldsymbol{g}}^{(A)}$ and $\vec{\boldsymbol{g}}^{(B)}$, the cloud computes two new vectors:

$$\vec{\boldsymbol{g}}^{(C_1)} = [e_1^{(A)} + g_1^{(B)}, ..., e_1^{(A)} + g_w^{(B)}, ..., e_w^{(A)} + g_1^{(B)}, ..., e_w^{(A)} + g_w^{(B)}]$$
$$\vec{\boldsymbol{g}}^{(C_2)} = [e_1^{(B)} + g_1^{(A)}, ..., e_1^{(B)} + g_w^{(A)}, ..., e_w^{(B)} + g_1^{(A)}, ..., e_w^{(B)} + g_w^{(A)}].$$

Then, it computes vector $\vec{\boldsymbol{g}}^{(C_3)}$, where $g_i^{(C_3)} = P_{sk}(s_i^{(A)})$ if and only if $|g_i^{(C_1)} - g_j^{(C_2)}| = 0$, where $i, j \in [1, w]$. It sends vector $\vec{\boldsymbol{g}}^{(C_3)}$ to client A who can decrypt it and find the intersection. The protocol's communication complexity is $O(w)$ and its computation involves $O(w^2)$ modular addition operations.

Although the protocol is efficient (as it mainly uses a hash function and secret key encryption) and can support multiple clients, it suffers from a set of major problems. First, each time the computation is delegated, every client needs to download the encrypted vector, $\vec{\boldsymbol{d}}^{(I)}$, whose size is equal to the client's set size. Also, each element in

the vector has the same size as the elements of the outsourced set. This is equivalent to the case where every client first downloads its outsourced set, prepares and uploads it before the cloud computes the result.

Second, in the protocol, value b is used as a one-time pad and according to its definition it must be used only once [71, 52]. But, the same pad is used multiple times, in the protocol, to blind the elements of the vectors $\vec{\boldsymbol{g}}^{(A)}$ and $\vec{\boldsymbol{g}}^{(B)}$. This approach is not secure and leaks information about the values $r_i^{(I)}$ supposed to be protected from the cloud. Knowledge of the values $r_i^{(I)}$ allows the cloud to figure out the hash values of each client's set elements, $h(s_i^{(I)})$, that should be protected by $r_i^{(I)}$. Third, since the encoding scheme used in the protocol is deterministic, the cloud learns (from vectors $\vec{\boldsymbol{g}}^{(C_1)}$ and $\vec{\boldsymbol{g}}^{(C_2)}$) the intersection cardinality, and whether the two sets have any elements in common. Fourth, due to the deterministic characteristics of the scheme, if client A and B and then client A and C delegate the computation to the cloud, the cloud can learn whether the sets of client B and C have any elements in common, and how many elements are in common, without the clients' consent. The authors have not provided the protocol's security proof.

Another delegated PSI protocol is proposed in [123] that supports multiple clients who can verify the integrity of the computation result. The protocol utilizes a cryptographic accumulator [36, 89] scheme, and the idea of proxy re-encryption [9]. The cryptographic accumulator scheme allows accumulation of a set of elements into a short value called accumulator, such that each of the elements has a short witness that can be used to verify its membership in the set. However, it is computationally infeasible to find a witness for any non-accumulated value. Moreover, a proxy re-encryption scheme is a public key encryption scheme that allows client A to encrypt a message under its public key, and send it to an untrusted proxy server. Later on, client A can provide a key (other than its private key) to the proxy server who can use the key and ciphertext to generate another ciphertext that can be decrypted by client B using its own secret key. But, the proxy server learns nothing about the plaintext in this process. There is a major difference between the standard (semantically secure) proxy re-encryption scheme and the technique used in this protocol. In particular, the re-encrypted ciphertext generated in the protocol contains no random value, so the scheme is not semantically secure.

In the following, we provide a high-level description of the protocol. First, a trusted third party generates the secret and public parameters and distributes them among the clients. Next, given the parameters and set $S^{(I)}$, each client I independently encrypts

its set elements as follows. $\forall i, 1 \leq i \leq w : t_i^{(I)} = E_{pk_I}(s_i^{(I)})$, where $s_i^{(I)} \in S^{(I)}$. Moreover, it generates an accumulator $\alpha^{(I)}$ and witnesses for its set, and then sends the encrypted values $T^{(I)} = \{t_1^{(I)}, ..., t_w^{(I)}\}$ and the witnesses to the cloud. It keeps locally the accumulator, $\alpha^{(I)}$. When the clients want to delegate PSI computation to the cloud, each client computes a token $q^{(I)}$ that allows the cloud to re-encrypt its outsourced ciphertexts, $t_i^{(I)}$ to different ciphertexts, $t_i'^{(I)}$. So, if there is an element in common in the sets, its re-encrypted ciphertexts would be the same in both sets, i.e. if $s_i^{(A)} = s_j^{(B)}$ then $t_i'^{(A)} = t_j'^{(B)}$. Given $q^{(I)}$ and the clients' outsourced datasets, the cloud performs the re-encryption and then sends to client A those outsourced ciphertexts whose re-encrypted ciphertexts are equal. Also, it uses the clients' witnesses to generate a proof of the computation correctness. The client who knows the secret key decrypts the result, and then uses the accumulator and the proof to verify the result integrity. The protocol's communication complexity is $O(z)$, where z is the intersection cardinality, and involves $O(w)$ public key encryption operations, where w is the set cardinality. The protocol suffers from the same security issues as [81] mentioned above, due to its deterministic nature. Besides, as we mentioned earlier, the re-encryption scheme is not semantically secure. Hence, the cloud after computing $t_i'^{(I)}$, can perform a "plaintext guessing" attack to determine the corresponding set element (or plaintext), $s_i^{(I)}$. Such attack is feasible (i.e. can be carried out in polynomial time) when the message universe is small. The authors used the standard model to prove the protocol's security.

Although the protocol proposed by Qiu *et al.* in [99] is very similar to the protocol in [123] stated above, for the sake of completeness we briefly present it. The protocol considers the semi-honest adversarial model, supports multiple clients, and uses also a trusted third party to generate and distribute a set of public and private parameters among clients. This protocol uses the same idea as in [123] that each set element is encrypted separately by its owner and uploaded to the cloud. Later on, when the clients want to delegate the computation, they provide a token and the cloud can re-encrypt the cihpertexts and compares them and then return the result to the clients. However, in this protocol, each client includes its ID when it generates the ciphertexts and the token. The authors analyzed the protocol's security in the random oracle model. The communication complexity of the protocol is $O(z)$, and its computation involves $O(w)$ public key encryption operations. In this scheme, in order for the ciphertext to be secure, the set universe $\mathcal{U}$ must be sufficiently large (e.g. $|\mathcal{U}| > poly(\lambda)$, where λ is the security parameter). Otherwise, the cloud can carry out a brute-force attack on the encrypted set elements and figure them out. But, this requirement cannot be met in

numerous real-world applications (e.g. short digits in the stock market, or names, etc). In this protocol, similar to the one in [123], the cloud learns the intersection cardinality and can figure out whether two clients have any elements in common even though they did not together delegate the computation to the cloud.

The only protocol that can securely support delegation of both storage and computation to an untrusted cloud is the one proposed by Lopez-Alt *et al.* in [82] which is designed for generic multi-party computation. Nonetheless, the protocol is very inefficient due to its computationally expensive building blocks. The protocol considers both the semi-honest and malicious adversarial models. The authors introduce a variant of fully homomorphic encryption (FHE), called multi-key FHE and use it to construct a generic delegated multi-party computation protocol. Multi-key FHE enables different ciphertexts encrypted under different public keys to be homomorphically combined together, even though the public keys have been generated independently. In other words, given the ciphertexts, the cloud can perform any arbitrary computation on them without learning the computation inputs and outputs. What follows is an overview of the protocol that considers the semi-honest model. First, each client generates its key pair independently without interacting with the other clients or having a prior knowledge about them. After that, the client encrypts its inputs, and sends the ciphertexts to the cloud who homomorphically operates on the client ciphertexts and computes the result. Then, it broadcasts the result to the clients who engage in a multi-party computation and use their secret keys and the ciphertexts to obtain the result. The scheme's overall communication complexity is $O(w)$, and its computation requires $O(w)$ FHE operations. As it is highlighted in [98, 94, 85], the scheme is far from being practical and the main reason is that it leverages a FHE scheme that is computationally very expensive.

The authors also use generic zero-knowledge proofs to turn the semi-honest secure protocol to the protocols secure in the presence of malicious parties. In one variant, at the end of the protocol, the cloud broadcasts all the encrypted inputs to all clients and uses zero-knowledge proofs to prove that the result has been computed correctly. In another variant, each client after encrypting its inputs, computes a hash value of each ciphertext using a keyed hash function. Then, it sends a copy of the hash values (along with the ciphertexts) to the cloud and keeps a copy of the hash values locally. Then, at the end of the protocol, the cloud broadcasts all the hash values and the computation result to all clients and proves that it computed the result correctly. So, this scheme also imposes high communication and storage costs to the participants especially when

the number of inputs is high. The overall computation and communication complexity of the malicious protocols is also $O(w)$; nevertheless, they involve expensive generic proofs systems (in addition to FHE operations) which make the verification procedure very inefficient. Due to the nature of FHE, in this protocol, the outsourced data can be updated securely. Both protocols in [82] have been proven secure in the standard model.

3.4 Concluding Remarks

The cloud is receiving considerable attention from individuals and companies due to the benefits and services it offers. Clients can outsource their data to the cloud, and ask it to run computation on the outsourced data. However, the cloud is vulnerable to data security breaches and is not fully trusted. PSI is a vital cryptographic protocol that has many real world applications, but existing PSI protocols cannot be used securely in a setting where the computation and storage are outsourced to the cloud. In particular, traditional PSI protocols allow clients to jointly run the computation. These protocols preserve client data privacy, and can be very efficient. However, they require clients to have their data locally. On the other hand, there are also some protocols allowing clients to take advantage of the cloud's computation power, however they support only one-off PSI delegation and the clients have to re-encode the data every time PSI computation is delegated to preserve their data privacy. In these protocols, the clients have to either keep locally a copy of the data or download it every time the computation is delegated. In contrast, those protocols that support repeated PSI delegation can support outsourcing of both data storage and the computation. However, most of these protocols cannot fully protect the privacy of data from the cloud. The only protocol that allows secure delegation of data storage and PSI computation uses FHE and is very inefficient.

Chapter 4

Delegated PSI on Outsourced Private Datasets

4.1 Introduction

As we have shown in section 3, existing PSI protocols cannot be used securely on outsourced private data. In particular, the traditional PSI protocols require parties to have their sets locally and mutually compute the result. Moreover, the protocols proposed to take advantage of the cloud's computation capabilities support only one-off delegation and clients need to re-prepare the data every time PSI is delegated. Also, the protocols that support repeated PSI delegation and allow outsourcing of both data storage and the computation are not fully private. Although the protocol based on fully homomorphic encryption can securely support repeated PSI delegation, it is computationally expensive.

In this chapter, we first propose O-PSI, a protocol that efficiently supports secure outsourcing of data storage and the computation. The protocol does not require FHE, instead it uses an additive homomorphic encryption scheme that is more efficient. However, additive homomorphic operations are still costly and have a major impact on the protocol's performance. To mitigate this problem, we propose a more efficient protocol, EO-PSI, that preserves all O-PSI's desirable characteristics, while requires no public key encryption or exponentiation operations. The material about O-PSI protocol comes from the published paper titled "*O-PSI: Delegated Private Set Intersection On Outsourced Datasets*" [1] and that about EO-PSI protocol comes from the paper titled "*Efficient Delegated Private Set Intersection on Outsourced Private Datasets*"

[3].

This chapter is organized as follows. Section 4.2 starts with a high-level overview of O-PSI followed by a detailed description of the two-client O-PSI, an outline of the multi-client case, the security definition for the protocol and a formal security analysis of O-PSI. Section 4.3 comprises a high-level description of EO-PSI, a detailed explanation of the two-client and multi-client case, and a formal security analysis of the protocol. In section 4.4, we compare our protocols with the outsourced PSI protocols presented in chapter 3. In section 4.5, we explain how the right parameters for EO-PSI can be chosen, we provide an overview of O-PSI and EO-PSI code design, and we compare their performance. Finally, we conclude the chapter in section 4.6.

4.2 O-PSI: Delegated Private Set Intersection on Outsourced Private Datasets

In this section, we provide our first delegated PSI protocol that supports secure repeated PSI delegation. It allows two clients to delegate both data storage and PSI computation to the cloud. In particular, the protocol enables clients to independently outsource their private datasets to the cloud. Once the clients outsource their data, they can delegate PSI computation on the outsourced data to the cloud an unlimited number of times while the privacy of the data and the computation result is protected in the cloud. The clients can delete any local copy of their datasets once they upload them. To achieve our goal, we use partially homomorphic encryption (i.e. Paillier encryption) that is more efficient than fully homomorphic encryption, and widely used in secure multi-party computation protocols (e.g. [46, 79, 39, 62]). The protocol, in addition to the encryption scheme, is based on point-value set representation and a blinding technique.

4.2.1 An Overview of O-PSI

The interaction between the parties in the protocol is depicted in Fig 4.1. At a high level, O-PSI works as follows. First, each client represents its set as a point-value polynomial representation to get a vector of y-coordinates. Then, it independently blinds the vector elements and sends the vector of blinded values to the cloud. As the vector elements have been blinded the cloud cannot figure out the set elements. When client B wants the intersection of its outsourced set and client A's outsourced set, it

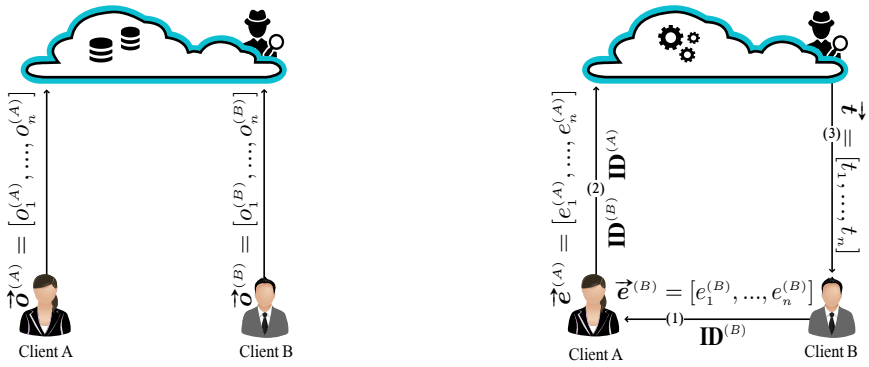

Figure 4.1: The left-hand side figure: party interactions at data outsourcing phase in O-PSI; the right-hand side figure: party interactions at the computation delegation phase in O-PSI.

first sends an encrypted message to client A to obtain its permission. If client A agrees, it performs some (homomorphic) operation on the encrypted message and sends the result to the cloud. Upon receiving the message, the cloud uses it to switch client A's dataset blinding factors such that after the switching the blinding factors of client A's outsourced data would be the same as that of client B's. After the switching process, client A's data would be in an encrypted form. Then, the cloud uses the homomorphic feature of the encryption scheme to combine client A's switched data with client B's outsourced data. Next, it sends the result to client B. Given the cloud's response, client B decrypts it to get a vector of y-coordinates. Then, it uses the vector to interpolate a polynomial and considers the polynomial's roots as the sets intersection. In this process, the cloud learns nothing about the clients' sets and client B learns only the sets intersection.

4.2.2 O-PSI Protocol

In the following, we first present the two-client O-PSI protocol in detail, where the cloud C, and clients A and B engage in the protocol. Then, we explain the rationale behind the protocol design. We use $E_{pk_I}(h_i)$ and $D_{sk_I}(h_i)$ to say that value h_i is encrypted using client I's public key, and decrypted using his secret key, respectively.

a. **Cloud-Side Setup.** The cloud constructs a field $\mathbb{F}_p$, where p is a large prime number. Then, it sets c as an upper bound of the set cardinality. The cloud constructs a vector, $\vec{x}$, of $n = 2c + 1$ distinct elements chosen uniformly at random from the field. Moreover, it picks a pseudorandom function, PRF, which

takes an *l*-bit key and *b*-bit message, and maps the message to an element in the field pseudorandomly: $\texttt{PRF} : \{0,1\}^b \times \{0,1\}^l \rightarrow \mathbb{F}_p$, where $|p| = l'$ and l, l' are security parameters. The cloud publishes the description of the field along with vector $\vec{\boldsymbol{x}}$ such that all the clients can access them.

b. **Client-Side Setup and Data Outsourcing.** This step is the same for both clients. Let $S^{(I)}$ be client I's set, where $I \in \{A, B\}$ and $|S^{(I)}| \leq c$. Client I performs as follows.

1. Generates a Paillier key pair (pk_I, sk_I) and publishes the public key. It also chooses a random private key $k^{(I)}$ for the pseudorandom function, PRF. All keys are generated according to a given security parameter. It makes sure values $x_i \in \vec{\boldsymbol{x}}$ are not equal to its set elements; otherwise, it aborts.
2. Constructs a polynomial $\tau^{(I)}(x)$ that represents its set $S^{(I)}$.

$$\tau^{(I)}(x) = \prod_{m=1}^{|S^{(I)}|} (x - s_m^{(I)}).$$

3. Generates a set of y-coordinates associated with $\tau^{(I)}(x)$. To do so, it evaluates $\tau^{(I)}(x)$ at each element of vector $\vec{\boldsymbol{x}}$ (already published by the cloud). This yields n values $\tau^{(I)}(x_i)$, where $1 \leq i \leq n$.
4. Computes a set of n pseudorandom values, given the key $k^{(I)}$.

$$\forall i, 1 \leq i \leq n : z_i^{(I)} = \texttt{PRF}(k^{(I)}, i).$$

5. Blinds the y-coordinates, by using the pseudorandom values:

$$\forall i, 1 \leq i \leq n : o_i^{(I)} = z_i^{(I)} \cdot \tau^{(I)}(x_i).$$

6. Sends the vector containing the blinded values, $\vec{\boldsymbol{o}}^{(I)} = [o_1^{(I)}, ..., o_n^{(I)}]$, to the cloud.

c. **Set Intersection: Computation Delegation.** This phase starts when client B becomes interested in the intersection of its set and client A's set.

1. Client B re-generates the blinding factors, $z_i^{(B)}$, it used (in step b.4) to protect its outsourced data. Then, the client encrypts each of the blinding

factors.

$$\forall i, 1 \leq i \leq n : e_i^{(B)} = E_{pk_B}(z_i^{(B)}).$$

2. Client B sends its id, $\mathsf{ID}^{(B)}$ and the vector comprising the encrypted blinding factors, $\vec{\boldsymbol{e}}^{(B)} = [e_1^{(B)}, ..., e_n^{(B)}]$, to client A.
3. When client A receives the other client's message, it first re-generates the blinding factors, $z_i^{(A)}$, it used (in step b.4) to secure its outsourced data. Next, it creates new encrypted values, given client B's message:

$$\forall i, 1 \leq i \leq n : e_i^{(A)} = (e_i^{(B)})^{(z_i^{(A)})^{-1}} = E_{pk_B}(z_i^{(B)} \cdot (z_i^{(A)})^{-1}).$$

4. Client A sends $\vec{\boldsymbol{e}}^{(A)} = [e_1^{(A)}, ..., e_n^{(A)}]$, $\mathsf{ID}^{(A)}$, $\mathsf{ID}^{(B)}$ and a request message: **Compute** to the cloud.

d. **Set Intersection: Cloud-Side Result Computation.**

1. After receiving client A's message, the cloud picks two degree c random polynomials $\omega^{(A)}(x)$ and $\omega^{(B)}(x)$ (one for each client), where each polynomial coefficients are chosen uniformly at random from the field, $\mathbb{F}_p$. After that, it evaluates each of the random polynomials $\omega^{(I)}(x)$ at every element $x_i \in \vec{\boldsymbol{x}}$. This results in the y-coordinates, $\omega^{(I)}(x_i)$, where $I \in \{A, B\}$.
2. The cloud fetches client A's outsourced dataset $\vec{\boldsymbol{o}}^{(A)}$ and then (given vector $\vec{\boldsymbol{e}}^{(A)}$) it switches client A's blinding factors to client B's, and multiplies the result by the y-coordinates of the random polynomial, $\omega^{(A)}(x_i)$.
   $\forall i, 1 \leq i \leq n :$

$$(e_i^{(A)})^{o_i^{(A)} \cdot \omega^{(A)}(x_i)} = E_{pk_B}(z_i^{(B)} \cdot (z_i^{(A)})^{-1} \cdot z_i^{(A)} \cdot \tau^{(A)}(x_i) \cdot \omega^{(A)}(x_i)).$$

3. The cloud fetches client B's outsourced dataset, multiplies its elements by the y-coordinates of the corresponding random polynomial, $\omega^{(B)}(x_i)$, and then encrypts each value.

$$\forall i, 1 \leq i \leq n : E_{pk_B}(o_i^{(B)} \cdot \omega^{(B)}(x_i)) = E_{pk_B}(z_i^{(B)} \cdot \tau^{(B)}(x_i) \cdot \omega^{(B)}(x_i)).$$

4. The cloud homomorphically sums the values generated in step d.2 with those computed in step d.3.

$\forall i, 1 \leq i \leq n$:

$$t_i = E_{pk_B}(z_i^{(B)} \cdot (z_i^{(A)})^{-1} \cdot z_i^{(A)} \cdot \tau^{(A)}(x_i) \cdot \omega^{(A)}(x_i)) \cdot E_{pk_B}(z_i^{(B)} \cdot \tau^{(B)}(x_i) \cdot \omega^{(B)}(x_i))$$
$$= E_{pk_B}(z_i^{(B)} \cdot (z_i^{(A)})^{-1} \cdot z_i^{(A)} \cdot \tau^{(A)}(x_i) \cdot \omega^{(A)}(x_i) + z_i^{(B)} \cdot \tau^{(B)}(x_i) \cdot \omega^{(B)}(x_i)).$$

5. The cloud sends the vector containing the encrypted elements, $\vec{\boldsymbol{t}} = [t_1, ..., t_n]$, computed in the previous step to client B.

e. **Set Intersection: Client-Side Result Retrieval**

1. Client B decrypts every element in $\vec{\boldsymbol{t}}$ and removes the blinding factors: $\forall i, 1 \leq i \leq n$:

$$g_i = D_{sk_B}(t_i) \cdot (z_i^{(B)})^{-1} = \tau^{(A)}(x_i) \cdot \omega^{(A)}(x_i) + \tau^{(B)}(x_i) \cdot \omega^{(B)}(x_i).$$

2. Given n pairs of (x_i, g_i), it interpolates a polynomial, $\phi(x)$, and considers the (valid) roots of $\phi(x)$ as the elements of the set intersection.

Remark 1: In step a, the cloud publishes vector $\vec{\boldsymbol{x}}$ that contains $2c + 1$ elements, because the polynomial $\phi(x)$, in step e.2, is of degree $2c$ and at least $2c + 1$ points are needed to interpolate it. The elements in $\vec{\boldsymbol{x}}$ are picked uniformly at random from a sub-set of $\mathbb{F}_p$ that is disjoint from the (field sub-) set containing the encoded element; therefore, x_i will not be a root of a client's polynomial (as explained in section 2.6).

Remark 2: In step b.5, if the client does not blind the y-coordinates and stores $\tau^{(I)}(x_i)$ directly without protecting them in the cloud, then the cloud could use n pairs of $(x_i, \tau^{(I)}(x_i))$ to interpolate the client's polynomial (recall $\vec{\boldsymbol{x}} = [x_1, ..., x_n]$ is public and known to all parties). As a result, the client's set would be revealed to the cloud. Whereas, when they are blinded the cloud cannot learn anything about the client's set unless it knows the pseudorandom function key used by the client. The client blinds the values by multiplication. Multiplication cannot blind $\tau^{(I)}(x_i) = 0$. This is why we require that $x_i \in \vec{\boldsymbol{x}}$ to be unequal to a root of a client's polynomial (or one of its set elements). Recall, in section 2.6 we showed how x_i can be picked such that it does not equal any set elements.

Remark 3: The data stored in the cloud are independently blinded by its owner. In particular, each client I, where $I \in \{A, B\}$, independently picks the pseudorandom key, $k^{(I)}$, so different clients have different keys. Consequently, they would have a different set of pseudorandom values with a high probability (i.e. $\texttt{PRF}(k^{(A)}, i) \neq \texttt{PRF}(k^{(B)}, i)$). On the other hand, in order to correctly compute the result, the datasets must have the same blinding factors ($z_i^{(I)}$ in the protocol) before they are combined with each other. Moreover, the factors must be eliminated at the end of the protocol by the result recipient. Hence, to ensure result correctness, during the computation delegation, the clients compute vector $\vec{\boldsymbol{e}}^{(A)}$ that enables the cloud to obliviously "switch" client A's blinding factors to client B's. In step d.2, the cloud uses the vector to insert the values that can eliminate client A's blinding factors, $z_i^{(A)}$, from its dataset, and insert client B's blinding factors, $z_i^{(B)}$, into it. After that, it combines the datasets together and sends the result to client B. In step e.1, client B can eliminate all the blinding factors. Note that since the values in $\vec{\boldsymbol{e}}^{(A)}$ are encrypted and only client B knows the secret key, the cloud learns nothing in this process.

Remark 4: The clients' original blinded datasets remain unchanged in the cloud. In fact in steps d.2 and d.3, the cloud multiplies *a copy* of the client's blinded dataset by the vector of $\omega^{(I)}(x_i)$.

Remark 5: The only information that the cloud learns about the clients' datasets is the upper bound on the dataset cardinality (i.e. value c) that was initially set by it. Thus, the cloud learns nothing about: (a) the sets' elements, (b) the exact number of set elements, (c) the intersection, and (d) the intersection cardinality.

Remark 6: Unlike the schemes that use polynomials in coefficient form to represent a set, e.g. [46, 75], we represent the polynomials in point-value form. The point-value representation brings two advantages in our protocol. First, all the steps between step c and e.1 (inclusive) can be done in parallel. In particular, an operation on each element of a vector (i.e. y-coordinate) can be done independently without the involvement of the rest of the vector's elements. This property relieves client A from needing to have a local storage linear to the set cardinality. Instead, it can receive the elements in streaming mode (from client B), and compute each y-coordinate separately. Furthermore, the cloud who has numerous computation cores can work on each y-coordinate in parallel resulting in a speed up of the computation. Second, the point-value representation

reduces the overall computation cost as the computation complexity of multiplying two polynomials of degree m in point-value from is $O(m)$, whereas it is $O(m^2)$ in coefficient form. However, the communication cost in both point-value and coefficient representations is $O(m)$.

4.2.3 Extensions

4.2.3.1 Multi-client O-PSI

Before we show how the two-client O-PSI can be turned into multi-client O-PSI, we briefly outline why multi-client O-PSI, or in general multi-client PSI, matters. At first glance, in order to compute the intersection of multiple clients' sets, any two-client PSI protocol (regardless whether it is classical PSI or delegated one) can be directly used multiple times. For example, client B who has set $S^{(B)} = \{a, b, c\}$ uses two-client PSI, and engages in the protocol with client A whose set is $S^{(A)} = \{a, b\}$, and then it extracts the intersection which is $S^{(B)} \cap S^{(A)} = \{a, b\}$. Next, client B uses the intersection, as its input and participates in the protocol with client C whose input is $S^{(C)} = \{a, d\}$. At the end, client B finds $S^{(B)} \cap S^{(A)} \cap S^{(C)} = \{a\}$. Nonetheless, in this process, client B learns more information than the intersection of the sets. For instance, it learns $S^{(A)}$ has element b whereas $S^{(C)}$ does not have this element. This leakage may discourage some clients to participate. In contrast, if the protocol could support multiple clients and a client would not engage if only two of them participate (i.e. no two-client PSI is allowed), then client B would only figure out the final result, i.e. $S^{(B)} \cap S^{(A)} \cap S^{(C)} = \{a\}$. In addition to that, the (communication and computation) cost of running multiple clients PSI protocol once may be less than the cost of running two-client PSI multiple times.

In the following, we explain how two-client O-PSI can be adjusted to support m-client O-PSI, where $m > 2$. In the multi-client case, all the clients prepare their data in the same way as in the two-client case. Also, the client interested in the intersection, client B, sends the same request (step c.2) to all other clients, A_q, where $1 \leq q \leq y$ and $y = m - 1$. The protocol for each client A_q remains unchanged. However, the computation at the cloud side slightly changes. Specifically, the cloud separately switches the blinding factors of the dataset of each client A_q, and multiplies each dataset by a random polynomial, $\omega^{(A_q)}(x)$, using vector $e_i^{(A_q)}$ provided by the client (step d.2). Finally, the cloud combines the values, generated in the previous step, with the other clients' datasets (step d.4).

$\forall i, 1 \leq i \leq n$:

$$t_i = E_{pk_B}(\omega^{(B)}(x_i) \cdot o_i^{(B)}) \cdot \prod_{q=1}^{y} (e_i^{(Aq)})^{o_i^{(Aq)} \cdot \omega^{(Aq)}(x_i)}$$

$$= E_{pk_B}(z_i^{(B)} \cdot (\omega^{(B)}(x_i) \cdot \tau^{(B)}(x_i) + \sum_{q=1}^{y} \omega^{(Aq)}(x_i) \cdot \tau^{(Aq)}(x_i))).$$

The rest of the steps remain the same. Note that in the multiple clients setting, even if client B colludes with $y-1$ clients, it could not infer the set elements of the non-colluding client, as the random polynomials $\omega^{(Aq)}$ and $\omega^{(B)}$ are picked by the cloud, and are unknown to the clients.

4.2.3.2 How to Avoid Client-to-client Interaction in O-PSI

The protocol can be made more flexible by avoiding the direct communication between the clients. In this case, in step c.2, client B can send vector $\vec{e}^{(B)}$ and $\mathsf{ID}^{(B)}$ to the cloud (instead of sending them to client A). Then, it can go offline. When client A comes online it downloads the vector from the cloud and (similar to the original protocol) it locally performs the operation on the vector and sends $\vec{e}^{(A)}$ to the cloud. This approach is also secure as vector $\vec{e}^{(B)}$ contains the encrypted elements and the cloud cannot figure out the original message. Moreover, the overall computation remains unchanged and although the overall communication increases the communication complexity does not change.

4.2.4 Security Definition

In this chapter, for both O-PSI and EO-PSI, we consider a setting where static semi-honest adversaries, who corrupt one party at a time, are present [52]. We assume the cloud does not collude with the clients. This is reasonable as it is often a well established IT company and such collusion will jeopardize its reputation, as there is always a risk that the client (or any party) may expose its collusion with the cloud and this potentially will have a negative impact on the cloud's revenue. The non-collusion assumption is widely used in the literature [111, 102, 68, 67, 20, 118, 77].

For the sake of simplicity, we assume there are three parties, the cloud C and clients A and B, engaging in the protocol where client A authorizes the computation and client B is interested in the result. We need a mechanism that allows client A to

identify a legitimate client, so it can verify its true identity. Such mechanism can help the client avoid engaging in the protocol with those clients it does not want to, e.g. its rival or the cloud. To address this issue, similar to [29, 55], we can assume that there exists an infrastructure, like public key infrastructure PKI (or Kerberos, etc), such that every party has a unique public key/ID and digitally signed certificate. Given one's public key/ID and certificate, a party can verify its identity. Let $\texttt{Authen}(.)$ be such an authentication function that authenticates a user u_i, and defined as:

$$\texttt{Authen}(u_i) = \begin{cases} 1 & \text{if authentication succeeds} \\ 0 & \text{otherwise} \end{cases}$$

That means client A can always authenticate client B via the infrastructure. We also assume client A has an authorization policy: $\texttt{Policy}(.)$, defined by client A that can be viewed as an internal subroutine of client A. The policy function, similar to authentication function, returns a binary decision on any request received by client A. The function returns 1 if the requesting client, $u_i \in U$, meets client A's policy and returns 0 otherwise. It totally depends on client A how to define the policy function. For instance, the client can maintain a set, U', of clients with which it is not interested in engaging in the protocol and the function returns 0 if the requesting client (i.e. client B) is $u_i \in U'$. As another example, client A can maintain a set of clients who have already engaged in the protocol and the function returns 0 if the requesting client is one of them. Client A authorizes the computation and the protocol proceeds, if (a) the other client's identity is approved, i.e. authentication succeeds, and (b) the client meets client A's policy; otherwise, client A rejects the computation and the protocol aborts. The parties would output message *authorization failed* if client A rejects the computation. In the simulation, the simulator only needs to simulate the view up to the point where the above message (i.e. authorization failed) is observed.

Definition 12. *Let* $\texttt{Policy}(.)$ *and* $\texttt{Authen}(.)$ *respectively be policy and authentication functions defined as above. We say client* A *authorizes the computation with client* u_i *iff:*

$$\texttt{Authen}(u_i) \wedge \texttt{Policy}(u_i) = 1$$

otherwise client A *rejects it.*

A three-party delegated PSI protocol computes a function that maps the inputs to

some outputs. We define this function as $F : \Lambda \times 2^{\mathfrak{U}} \times 2^{\mathfrak{U}} \rightarrow \Lambda \times \Lambda \times f_{\cap}$, where Λ denotes the empty string, $2^{\mathfrak{U}}$ denotes the powerset of the set universe and $f_{\cap}$ denotes the set intersection function. For every tuple of inputs Λ, $S^{(A)}$ and $S^{(B)}$ belonging to C the cloud, client A and client B respectively, the function outputs nothing to C and A, and $f_{\cap}(S^{(A)}, S^{(B)}) = S^{(A)} \cap S^{(B)}$ to B. Following the standard model (presented in section 2.9), the protocol is secure in the semi-honest adversarial model if whatever can be computed by a party in the protocol can be obtained from its input and output only. This is formalized by the simulation paradigm such that a party's *view* in a protocol execution should be simulatable given only its input and output. The party I's view on input tuple (x, y, z) is denoted by $\mathsf{VIEW}_I(x, y, z)$, where $w \in \{x, y, z\}$ is the input of party I and $I \in \{A, B, C\}$.

Definition 13. *Let F be a deterministic function defined above. We say that the protocol securely computes F in the presence of static semi-honest adversaries if there exists a probabilistic polynomial-time algorithm $\mathsf{SIM}_I, I \in \{A, B, C\}$, such that given the input and output of a party, it can simulate a view that is computationally indistinguishable from the party's view in the protocol:*

$$\{\mathsf{SIM}_B(S^{(B)}, f_{\cap}(S^{(A)}, S^{(B)}))\}_{S^{(A)},S^{(B)}} \stackrel{c}{\equiv} \{\mathsf{VIEW}_B(\Lambda, S^{(A)}, S^{(B)})\}_{S^{(A)},S^{(B)}}$$
$$\{\mathsf{SIM}_A(S^{(A)}, \Lambda)\}_{S^{(A)},S^{(B)}} \stackrel{c}{\equiv} \{\mathsf{VIEW}_A(\Lambda, S^{(A)}, S^{(B)})\}_{S^{(A)},S^{(B)}}$$
$$\{\mathsf{SIM}_C(\Lambda, \Lambda)\}_{S^{(A)},S^{(B)}} \stackrel{c}{\equiv} \{\mathsf{VIEW}_C(\Lambda, S^{(A)}, S^{(B)})\}_{S^{(A)},S^{(B)}}$$

4.2.5 O-PSI Security Proof

Now we sketch the proof of the O-PSI security in the presence of static semi-honest adversaries. We conduct the security analysis for the three cases where one of the parties is corrupted. We use the security model presented in 4.2.4 for the analysis.

Theorem 2. *If the homomorphic encryption scheme is semantically secure and* `PRF` *is a collision-resistant pseudorandom function, then the O-PSI protocol is secure in the presence of static semi-honest adversaries.*

Proof. We will prove the theorem by considering in turn the case where each of the parties has been corrupted. In each case, we invoke a simulator with the corresponding party's input and output. Our focus is on the case where party A wants to engage in the computation of the intersection, i.e. it authorizes the computation. If party A does not want to proceed with the protocol, the views can be simulated in the same way up to the point where the execution stops.

Case 1: Corrupted Cloud. In this case, we show that we can construct a simulator, SIM_C, that can produce a view computationally indistinguishable from an adversary's view in the real model, given its input and output. In the real execution, the cloud's view, is:

$$\mathsf{VIEW}_C(\Lambda, S^{(A)}, S^{(B)}) = \{\Lambda, r_C, \vec{\boldsymbol{o}}^{(A)}, \vec{\boldsymbol{o}}^{(B)}, \vec{\boldsymbol{e}}^{(A)}, \mathbf{ID}^{(A)}, \mathbf{ID}^{(B)}, \mathbf{Compute}, \Lambda\}.$$

In the above view, r_C is the outcome of internal random coins of the cloud, $\vec{\boldsymbol{o}}^{(A)}$ and $\vec{\boldsymbol{o}}^{(B)}$ are the clients' outsourced datasets, and $\vec{\boldsymbol{e}}^{(A)}$ is the vector of encrypted elements.

To simulate this view, SIM_C does the following:

1. Creates an empty view and appends to it Λ and uniformly at random chosen coins r'_C.
2. Constructs two vectors, $\vec{\boldsymbol{o}}'^{(A)}$ and $\vec{\boldsymbol{o}}'^{(B)}$ where each of them contains n values picked uniformly at random from the field, $\mathbb{F}_p$. Then, it appends $\vec{\boldsymbol{o}}'^{(A)}$ and $\vec{\boldsymbol{o}}'^{(B)}$ to the view.
3. Generates a vector, $\vec{\boldsymbol{e}}'^{(A)}$, comprising n encrypted random values. Afterwards, it inserts the vector to the view.
4. Constructs also four strings, $\mathbf{ID}^{(A)}$, $\mathbf{ID}^{(B)}$, **Compute** and Λ. It appends them to the view and outputs the view.

Now we argue that the simulated view is computationally indistinguishable from the real view. In both views, the input parts are identical (i.e. both are Λ), the random coins are both uniformly random, and so they are indistinguishable. Client I, (for all I, $I \in \{A, B\}$), blinds the elements of $\vec{\boldsymbol{o}}^{(I)}$ with the outputs of the pseudorandom function, PRF. On the other hand, each vector $\vec{\boldsymbol{o}}'^{(I)}$ contains n elements picked uniformly at random from the same field. Since the output of the pseudorandom function is indistinguishable from a random value, vectors $\vec{\boldsymbol{o}}^{(I)}$ and $\vec{\boldsymbol{o}}'^{(I)}$ are computationally indistinguishable. Furthermore, as the encryption scheme is semantically secure, the encrypted elements of vectors $\vec{\boldsymbol{e}}^{(I)}$ and $\vec{\boldsymbol{e}}'^{(I)}$ are also computationally indistinguishable. Moreover, values x_i are not equal to any set elements (as shown in section 2.6). Finally, the strings $\mathbf{ID}^{(A)}$, $\mathbf{ID}^{(B)}$, **Compute** and Λ are identical in both views. Thus, we conclude that the two views are computationally indistinguishable.

Case 2: Corrupted Client A. In the real execution, client A's view is:

$$\mathsf{VIEW}_A(\Lambda, S^{(A)}, S^{(B)}) = \{S^{(A)}, r_A, \vec{\boldsymbol{e}}^{(A)}, \mathsf{ID}^{(B)}, \Lambda\}.$$

The simulator, SIM_A, who receives client A's input, $S^{(A)}$, and output, Λ, does the following.

1. Creates an empty view and appends the party's input to it.
2. Inserts coins r'_A, uniformly at random chosen, to the view.
3. Constructs vector $\vec{\boldsymbol{e}}'^{(A)}$, containing n encrypted random values, and inserts it into the view. Finally, it inserts the strings $\mathsf{ID}^{(B)}$ and Λ into the view and outputs the view.

We now explain why the two views are computationally indistinguishable. In both views, the client's input, $S^{(A)}$, is identical. Moreover, both r_A and r'_A are picked uniformly at random so they are indistinguishable, too. Vectors $\vec{\boldsymbol{e}}^{(B)}$ and $\vec{\boldsymbol{e}}'^{(B)}$ containing the encrypted elements are computationally indistinguishable, as the encryption scheme is semantically secure. Also, the strings $\mathsf{ID}^{(B)}$ and Λ are identical in both views. Hence, the views are computationally indistinguishable.

Case 3: Corrupted Client B. In the real execution, client B's view is:

$$\mathsf{VIEW}_B(\Lambda, S^{(A)}, S^{(B)}) = \{S^{(B)}, r_B, \vec{\boldsymbol{g}}, f_\cap(S^{(A)}, S^{(B)})\}.$$

The simulator SIM_B receives the party's input, $S^{(B)}$, and output, $f_\cap(S^{(A)}, S^{(B)})$, and performs as follows.

1. Creates an empty view and appends the client's input, $S^{(B)}$, and uniformly at random chosen coins, r'_B, to the view.
2. Chooses two sets, $S'^{(A)}$ and $S'^{(B)}$, such that $S'^{(A)} \cap S'^{(B)} = f_\cap(S^{(A)}, S^{(B)})$ and $|S'^{(A)}|, |S'^{(B)}| \leq c$.
3. Represents each set $S'^{(I)}$, (for all I, $I \in \{A, B\}$), as a polynomial:

$$\tau'^{(I)}(x) = \prod_{m=1}^{|S'^{(I)}|} (x - s'^{(I)}_m),$$

   where $s'^{(I)}_m \in S'^{(I)}$.
4. Picks two random polynomials $\omega'^{(A)}(x)$ and $\omega'^{(B)}(x)$ of degree c.

5. Multiplies each polynomial, $\tau'^{(I)}(x)$, by the random polynomial, $\omega'^{(I)}(x)$, and then sums up the products:

$$\phi'(x) = \tau'^{(A)}(x) \cdot \omega'^{(A)}(x) + \tau'^{(B)}(x) \cdot \omega'^{(B)}(x).$$

6. Evaluates polynomial $\phi'(x)$ at every element of vector $\vec{\boldsymbol{x}}$. This results:

$$\forall i, 1 \leq i \leq n : g'_i = \phi'(x_i).$$

7. Inserts $\vec{\boldsymbol{g}}' = [g'_1, ..., g'_n]$ along with the client's output, $f_\cap(S^{(A)}, S^{(B)})$, into the view and outputs the view.

In the following, we discuss why the two views are computationally indistinguishable. In both models $S^{(B)}$ is identical, also r_B and r'_B are chosen uniformly at random, therefore they are indistinguishable. The polynomial $\phi'(x)$, interpolated from n pairs (g'_i, x_i), has the form $\tau'^{(A)}(x) \cdot \omega'^{(A)}(x) + \tau'^{(B)}(x) \cdot \omega'^{(B)}(x) = \mu' \cdot gcd(\tau'^{(A)}(x), \tau'^{(B)}(x))$. On the other hand, in the real view, polynomial $\phi(x)$ interpolated from n pairs (g_i, x_i) has the form $\tau^{(A)}(x) \cdot \omega^{(A)}(x) + \tau^{(B)}(x) \cdot \omega^{(B)}(x) = \mu \cdot gcd(\tau^{(A)}(x), \tau^{(B)}(x))$. As it has been proven in [75, 20], μ and μ' are uniformly random polynomials and indistinguishable, and the probability that their roots represent set elements is negligibly small. Moreover, polynomials $gcd(\tau'^{(A)}(x), \tau'^{(B)}(x))$ and $gcd(\tau^{(A)}(x), \tau^{(B)}(x))$ represent the intersection: $f_\cap(S^{(A)}, S^{(B)})$. Therefore, $\vec{\boldsymbol{g}}'$ and $\vec{\boldsymbol{g}}$ are computationally indistinguishable. Finally, the output part, $f_\cap(S^{(A)}, S^{(B)})$, in both views is identical. Thus, the two views are computationally indistinguishable. □

4.3 EO-PSI: Efficient Delegated Private Set Intersection on Outsourced Private Datasets

In this section, we introduce EO-PSI that preserves all O-PSI's desirable properties and is more efficient. EO-PSI improves O-PSI from two perspectives. First, unlike O-PSI, EO-PSI does not use any public key encryption that is not computationally very efficient. In O-PSI, the public key encryption is mainly used to prevent the cloud from eventually learning any information about the blinding factors (and set elements) during the cloud-side switching of the blinding factors, especially when the computation is delegated multiple times. Recall that in O-PSI, given vector $\vec{\boldsymbol{e}}$, the cloud can switch

one client's blinding factors to another's. In contrast, in EO-PSI no such switching is required. Therefore, no public key encryption is needed. In order to achieve this, we slightly change the way each client blinds its polynomial. In EO-PSI, instead of multiplying value $\tau(x_i)$ by a pseudorandom value, the client adds a pseudorandom value to it. Moreover, the interaction between the clients is changed, in the sense that client A sends a message to both the cloud and client B when it authorizes the computation. Second, EO-PSI allows each client to break down its original polynomial into smaller degree polynomials. This allows the result recipient to factorize a set of smaller degree polynomials rather than one of very large degree. As a result, (as we discussed in section 2.7) it can find the roots of the polynomials (i.e. the set intersection) faster than it could in O-PSI. To achieve this, the protocol gets each client to insert its elements into the bins of a (fixed-size) hash table.

4.3.1 An Overview of EO-PSI

The interaction between parties in EO-PSI is depicted in Fig. 4.2. In order for each client to prepare its set, first it constructs a hash table whose parameters are published by the cloud. Then, it inserts its set elements into the hash table bins, represents the set of elements in each bin as a point-value polynomial representation to get a vector of y-coordinates using x_i values provided by the cloud. It blinds the y-coordinates by adding pseudorandom values to them and then it sends them to the cloud. When client B becomes interested in the intersection of its own set and client A's set, it obtains client A's permission by sending a message to it. Also client B sends a message to the cloud. If client A agrees, it generates a set of vectors and sends them to client B. The vectors will help client B to unblind the cloud's response. Client A also sends a key for a pseudorandom function to the cloud. The key is generated on the fly and can be discarded when the protocol ends. The cloud uses the key to multiply each client's hash table's bin by a pseudorandom polynomial and then it combines them together. It sends a set of bins containing the result to client B. Given the cloud's response and client A's message, client B unblinds them, interpolates a set of polynomials and considers the union of the polynomials' (valid) roots as the sets intersection.

4.3.2 EO-PSI Protocol

Similarly, here we first consider the two-client case, where clients A and B, and the cloud engage in the protocol.

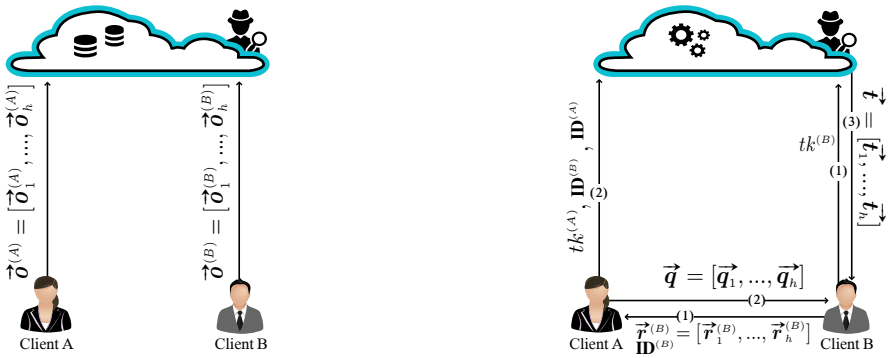

Figure 4.2: The left-hand side figure: party interactions at data outsourcing phase in EO-PSI; the right-hand side figure: party interactions at the computation delegation phase in EO-PSI.

a. **Cloud-Side Setup.** The cloud sets c as an upper bound of the set cardinality. Given c, it uses equation 2.4 to set parameters for a hash table. In particular, it sets d as the maximum load that a bin in the hash table can have, and h as the hash table length (or a total number of bins). Moreover, it chooses a cryptographic hash function, $\mathtt{H}$. The cloud constructs a field $\mathbb{F}_p$, where p is a large prime number. It also generates a vector, $\vec{\boldsymbol{x}}$, containing $n = 2d + 1$ distinct non-zero x_i values chosen uniformly at random from $\mathbb{F}_p$. Furthermore, it picks a pseudorandom function $\mathtt{PRF}$ defined as in O-PSI protocol. The cloud publishes the parameters of the hash table, the description of the field, the value n, vector $\vec{\boldsymbol{x}}$, the pseudorandom function $\mathtt{PRF}$, and the hash function $\mathtt{H}$.

b. **Client-Side Setup and Data Outsourcing.** This step is the same for both clients. Let client I have a set $S^{(I)}$, where $I \in \{A, B\}$ and $|S^{(I)}| \leq c$. Each client I performs as follows.

   1. It makes sure values $x_i \in \vec{\boldsymbol{x}}$ are not equal to its set elements; otherwise, it aborts. Then, given the hash table parameters, generates a hash table and inserts its set elements into it.

$$\forall s_i^{(I)} \in S^{(I)} : \mathtt{H}(s_i^{(I)}) = j,\ s_i^{(I)} \rightarrow \mathtt{HT}_j^{(I)},$$

      where $1 \leq j \leq h$.
   2. Assigns a key (for the pseudorandom function) to each bin in the hash table by picking a master key $mk^{(I)}$, and generating h pseudorandom values (or

keys):

$$\forall j, 1 \leq j \leq h : k_j^{(I)} = \mathtt{PRF}(mk^{(I)}, j).$$

3. For every bin $\mathtt{HT}_j^{(I)}$, if it has less than d set elements, pads it with random elements. So each bin would have d elements in total. Then, it encodes the bin elements as below.
   (a) Constructs a polynomial representing the elements in the bin.

$$\tau_j^{(I)}(x) = \prod_{m=1}^{d}(x - e_m^{(I)}),$$

   where $e_m^{(I)} \in \mathtt{HT}_j^{(I)}$and $e_m^{(I)}$ is either a set element or a random value.
   (b) Generate a set of y-coordinates associated with each $\tau_j^{(I)}(x)$. To do that, it evaluates each $\tau_j^{(I)}(x)$ at every elements $x_i \in \vec{\boldsymbol{x}}$. This yields n values $\tau_j^{(I)}(x_i)$ for each bin, where $1 \leq i \leq n$.
   (c) Blinds every value $\tau_j^{(I)}(x_i)$. To do so, it first generates a pseudorandom value $z_{j,i}^{(I)} = \mathtt{PRF}(k_j^{(I)}, i)$, where key $k_j^{(I)}$ was generated in step b.2. After that, it sums the pseudorandom value with the corresponding y-coordinate.

$$\forall i, 1 \leq i \leq n : o_{j,i}^{(I)} = z_{j,i}^{(I)} + \tau_j^{(I)}(x_i).$$

   By the end of this step, the elements in each bin, $\mathtt{HT}_j$, are represented as a vector, $\vec{\boldsymbol{o}}_j^{(I)} = [o_{j,1}^{(I)}, ..., o_{j,n}^{(I)}]$.
4. Sends $\vec{\boldsymbol{o}}^{(I)} = [\vec{\boldsymbol{o}}_1^{(I)}, ..., \vec{\boldsymbol{o}}_h^{(I)}]$ to the cloud.

c. **Set Intersection: Computation Delegation.** This phase starts when client B wants the intersection of its set and client A's set. The phase proceeds as follows.

1. Client B picks a temporary key $tk^{(B)}$. Also, it regenerates $z_{j,i}^{(B)}$, the blinding factors it used to blind each element in bin $\mathtt{HT}_\mathtt{j}$ in step b.3c. Then, it computes vectors $\vec{\boldsymbol{r}}_j^{(B)}$ whose elements are computed as below.

$$\forall j, 1 \leq j \leq h, \forall i, 1 \leq i \leq n : r_{j,i}^{(B)} = z_{j,i}^{(B)} + \mathtt{PRF}(tk_j^{(B)}, i),$$

   where $tk_j^{(B)} = \mathtt{PRF}(tk^{(B)}, j)$.

2. Client B sends $\vec{\boldsymbol{r}}^{(B)} = [\vec{\boldsymbol{r}}_1^{(B)}, ..., \vec{\boldsymbol{r}}_h^{(B)}]$ and its id, $\mathsf{ID}^{(B)}$, to client A, and $tk^{(B)}$ to the cloud.
3. Client A upon receiving the other client's request, regenerates keys $k_j^{(A)}$ (see step b.2), where $1 \leq j \leq h$.
4. Client A assigns three fresh keys to each bin $\mathtt{HT}_j$. To do that, first it picks a temporary key, $tk^{(A)}$, and then carries out the following (steps c.4a-c.6 are depicted in Fig 4.3).
   (a) It uses the key, $tk^{(A)}$, to generate three pseudorandom values k_t.

$$\forall t, 1 \leq t \leq 3 : k_t = \mathtt{PRF}(tk^{(A)}, t).$$

   (b) It uses each k_t to compute h pseudorandom values.

$$\forall j, 1 \leq j \leq h : k_{1,j} = \mathtt{PRF}(k_1, j), k_{2,j} = \mathtt{PRF}(k_2, j), k_{3,j} = \mathtt{PRF}(k_3, j).$$

5. For each bin, $\mathtt{HT}_j$, client A uses key $k_{1,j}$ to generate a set of pseudorandom values $a_{j,i}$.

$$\forall i, 1 \leq i \leq n : a_{j,i} = \mathtt{PRF}(k_{1,j}, i).$$

6. Client A uses keys $k_{2,j}$ and $k_{3,j}$ to generate two degree d pseudorandom polynomials $\omega_j^{(A)}(x)$ and $\omega_j^{(B)}(x)$ for each bin, $\mathtt{HT}_j$.

$$\forall i, 0 \leq m \leq d : b_{j,m}^{(A)} = \mathtt{PRF}(k_{2,j}, m), b_{j,m}^{(B)} = \mathtt{PRF}(k_{3,j}, m),$$

   where

$$\omega_j^{(A)}(x) = b_{j,0}^{(A)} \cdot x^0 + ... + b_{j,d}^{(A)} \cdot x^d,$$
$$\omega_j^{(B)}(x) = b_{j,0}^{(B)} \cdot x^0 + ... + b_{j,d}^{(B)} \cdot x^d.$$

7. Client A, for each bin $\mathtt{HT}_j$, regenerates the pseudorandom values $z_{j,i}^{(A)}$ using the keys it derived in step b.2. Then, it computes vectors $\vec{\boldsymbol{q}}_j$ as follows.

$$\forall j, 1 \leq j \leq h,\ \forall i, 1 \leq i \leq n : q_{j,i} = \omega_j^{(A)}(x_i) \cdot z_{j,i}^{(A)} + \omega_j^{(B)}(x_i) \cdot r_{j,i}^{(B)} + a_{j,i},$$

   where values $r_{j,i}^{(B)} \in \vec{\boldsymbol{r}}_j^{(B)}$ were sent to client A in step c.2. Note that vectors $\vec{\boldsymbol{q}}_j$ allow client B to remove the blinding factors from the cloud's response without learning the pseudorandom polynomials.

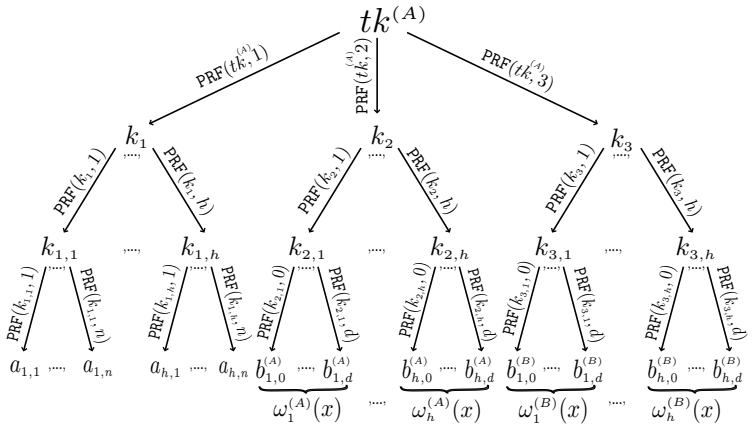

Figure 4.3: The Key Tree virtually constructed in steps c.4a-c.6

8. Client A sends $\vec{\boldsymbol{q}} = [\vec{\boldsymbol{q}}_1, ..., \vec{\boldsymbol{q}}_h]$ to client B. Also, client A sends key $tk^{(A)}$ (generated in step c.4), $\mathsf{ID}^{(A)}$, $\mathsf{ID}^{(B)}$, and **Compute** message to the cloud.

d. **Set Intersection: Cloud-Side Result Computation.** This phase starts when the cloud receives the messages computed in the previous step.

1. Given the temporary key, $tk^{(A)}$, the cloud derives the three keys, $k_{1,j}$, $k_{2,j}$ and $k_{3,j}$, for each bin $\mathtt{HT}_j$, where $1 \leq j \leq h$. Using the three keys, the cloud regenerates the set of pseudorandom values $a_{j,i}$ ($\forall i, 1 \leq i \leq n$) and the two pseudorandom polynomials, $\omega_j^{(A)}(x)$ and $\omega_j^{(B)}(x)$, for each bin $\mathtt{HT}_j$, where $1 \leq j \leq h$ (see step c.4a-c.6).
2. The cloud computes the result as follows. For each bin, it fetches the clients' outsourced datasets $\vec{\boldsymbol{o}}_j^{(A)}$ and $\vec{\boldsymbol{o}}_j^{(B)}$. Next, it computes the result vector $\vec{\boldsymbol{t}}_j$ for the bin.
   $\forall j, 1 \leq j \leq h$ and $\forall i, 1 \leq i \leq n$:

$$t_{j,i} = \omega_j^{(A)}(x_i) \cdot o_{j,i}^{(A)} + \omega_j^{(B)}(x_i) \cdot (o_{j,i}^{(B)} + \mathtt{PRF}(tk_j^{(B)}, i)) + a_{j,i}$$

   where $tk_j^{(B)} = \mathtt{PRF}(tk^{(B)}, j)$ and key $tk^{(B)}$ was sent to the cloud in step c.2.
3. The cloud sends $\vec{\boldsymbol{t}} = [\vec{\boldsymbol{t}}_1, ..., \vec{\boldsymbol{t}}_h]$ to client B.

e. **Set Intersection: Client-Side Result Retrieval**

1. Client B removes the blinding factors from each vector $\vec{\boldsymbol{t}}_j$ ($\forall j, 1 \leq j \leq h$), using the corresponding vector $\vec{\boldsymbol{q}}_j$ (provided by client A in step c.8). The result is vectors $\vec{\boldsymbol{g}}_j$ computed as follows.

$\forall j, 1 \leq j \leq h,\ \forall i, 1 \leq i \leq n$:

$$g_{j,i} = t_{j,i} - q_{j,i} = \omega_j^{(A)}(x_i) \cdot \tau_j^{(A)}(x_i) + \omega_j^{(B)}(x_i) \cdot \tau_j^{(B)}(x_i).$$

2. Given vectors $\vec{\boldsymbol{g}_j}$ and $\vec{\boldsymbol{x}}$, it interpolates the polynomials, $\phi_j(x)$ ($\forall j, 1 \leq j \leq h$).
3. It extracts the roots of each polynomial, and considers the union of the (valid) roots as the intersection of the sets.

Remark 1: In EO-PSI, the client needs to find the roots of h polynomials of degree $2d$, where d is a fixed value picked by the cloud and it is much smaller than the maximum number of elements, c. In contrast, in O-PSI the client receives only one polynomial of degree $2c$. As we explained in section 2.7, finding roots of h polynomials of small degree $2d$ is much faster than finding the roots of one polynomial of large degree $2c$.

Remark 2: In both EO-PSI and O-PSI, the cloud-side setup is performed only once, when the cloud comes online. After that, no further computation is needed in this step.

4.3.3 Extensions

4.3.3.1 Multi-client EO-PSI

With minor adjustments, the protocol can support $m > 2$ number of clients. In the following, we outline how this can be done. Here, we denote the result recipient by client B and the other clients by A_q, where $\forall q, 1 \leq q \leq y$ and $y = m-1$. In this setting, client B sends a single key to the cloud and it sends the same message as it does in the two-client case to the other clients. Also, each client A_q sends to the cloud a temporary key $tk^{(A_q)}$ that lets the cloud generate for each bin $\mathtt{HT}_j$ the set of pseudorandom values $a_{j,i}^{(A_q)}$ and two pseudorandom polynomials $\omega_j^{(A_q)}(x)$ and $\omega_j^{(B_q)}(x)$. However, as its shown below, the cloud-side computation in step d.2 is slightly changed.
$\forall j, 1 \leq j \leq h$ and $\forall i, 1 \leq i \leq n$:

$$t_{j,i} = \omega_j^{(B)}(x_i) \cdot (o_{j,i}^{(B)} + \mathtt{PRF}(tk_j^{(B)}, i)) + \sum_{q=1}^{y} a_{j,i}^{(A_q)} + \sum_{q=1}^{y} \omega_j^{(A_q)}(x_i) \cdot o_{j,i}^{(A_q)},$$

where $\omega_j^{(B)}(x) = \sum_{q=1}^{y} \omega_j^{(Bq)}(x)$.

Note that in the above step, the cloud first adds all the pseudorandom polynomials, $\omega_j^{(Bq)}(x)$, together and then it evaluates the resulting polynomial at every element in $\vec{\boldsymbol{x}}$. Consequently, client B in step e.1 removes the blinding factors from vector $\vec{\boldsymbol{t}}_j$ as follows.
$\forall j, 1 \leq j \leq h$ and $\forall i, 1 \leq i \leq n$:

$$\begin{aligned} g_{j,i} &= t_{j,i} - \sum_{q=1}^{y} q_{j,i}^{(Aq)} \\ &= \omega_j^{(B)}(x_i) \cdot \tau_j^{(B)}(x_i) + \sum_{q=1}^{y} \omega_j^{(Aq)}(x_i) \cdot \tau_j^{(Aq)}(x_i). \end{aligned}$$

In the multiple clients EO-PSI, even if client B colludes with $y-1$ clients, it cannot learn any information about the non-colluding client's set elements. The reason is that, as it is shown in [75], the polynomial $\omega_j^{(B)}(x)$ is always a uniformly random polynomial even if only one of the polynomials $\omega_j^{(Bq)}(x)$ is uniformly random, and unknown to client B.

Remark 1: In the multi-client EO-PSI, the communication and computation complexities for those clients who authorize the computation (i.e. clients A_q) are independent of the number of clients participating in the protocol. In other words, the computation and communication complexities for client A in the two-client case are similar to client A_j's in the multiple clients case. Note that the same holds for multi-client O-PSI.

Remark 2: Another advantage of using a hash table is that the operations on the bins can be carried out in parallel. This results in faster computation run-time when multiple cores are used (especially at the cloud-side). In addition to that, such approach allows each of client A_q to have local storage equal to $2d+1$, that is much less than the upper bound of the sets' cardinality, c.

4.3.3.2 How to Avoid Client-to-client Interaction in EO-PSI

In EO-PSI, with minor adjustments, the direct communication between the clients can be avoided. Note that in the protocol, if client B, in step c.2, sends vector $\vec{\boldsymbol{r}}^{(B)} = [\vec{\boldsymbol{r}}_1^{(B)}, ..., \vec{\boldsymbol{r}}_h^{(B)}]$ to the cloud, instead of client A, then the cloud would be able to learn

client B's set elements, because the cloud who knows key $tk^{(B)}$, can unblind each value $r_{j,i}^{(B)} = z_{j,i}^{(B)} + \texttt{PRF}(tk_j^{(B)}, i)$, where $r_{j,i}^{(B)} \in \vec{\boldsymbol{r}}_j^{(B)}, tk_j^{(B)} = \texttt{PRF}(tk^{(B)}, j)$, and recover the blinding factors $z_{j,i}^{(B)}$ used to blind the client's y-coordinates. Also, if client A in step c.8, sends $\vec{\boldsymbol{q}} = [\vec{\boldsymbol{q}}_1, ..., \vec{\boldsymbol{q}}_h]$, where $q_{j,i} = \omega_j^{(A)}(x_i) \cdot z_{j,i}^{(A)} + \omega_j^{(B)}(x_i) \cdot r_{j,i}^{(B)} + a_{j,i}$ and $q_{j,i} \in \vec{\boldsymbol{q}}_j$, to the cloud instead of client B, then the cloud who knows $tk^{(A)}$ might be able to learn the clients blinding factors. As a result, it may be able to figure out the clients set elements.

We now outline how we can slightly adjust the protocol to let clients avoid communicating directly with each other without leaking any information. Each client first picks a symmetric key, and uses it to encrypt the vector that used to be sent to the other client in the original protocol. Then, it encrypts the symmetric key under the other client's public key, and uploads the encrypted vector and the encrypted key to the cloud. Next, each client downloads the encrypted key and vector uploaded by the other client and decrypts them. The rest of the protocol remains the same. In the above scheme, since the keys and vectors are encrypted, the cloud cannot learn anything about clients' set elements. It should be noted that the number of public key operations in the above scheme is constant; in particular, each client encrypts/decrypts only one value using public key encryption. Therefore, although the computation and communication cost slightly increases, the overall computation and communication complexity remains the same.

4.3.4 EO-PSI Security Proof

Now we sketch the protocol security proof in the presence of static semi-honest adversaries. We analyze the security of the protocol in the three cases where one of the parties is corrupted. We use the model presented in 4.2.4 for the analysis.

Theorem 3. *If* PRF *is a collision-resistant pseudorandom function, then the EO-PSI protocol is secure in the presence of a semi-honest adversary.*

Proof. In order to prove the above theorem, we consider the case where each of the parties has been corrupted. In each case, we invoke a simulator with the corresponding party's input and output. Our focus is on the case where party A wants to engage in the computation of the intersection, i.e. it authorizes the computation. If party A does not want to proceed with the protocol, the views can be simulated in the same way up to the point where the execution stops.

Case 1: Corrupted Cloud. In this case, we show that we can construct a simulator, SIM_C, that can produce a computationally indistinguishable view to the real model view. In the real execution, the cloud's view is:

$$\mathsf{VIEW}_C(\Lambda, S^{(A)}, S^{(B)}) = \{\Lambda, r_C, \vec{\boldsymbol{o}}^{(A)}, \vec{\boldsymbol{o}}^{(B)}, tk^{(A)}, tk^{(B)}, \mathbf{ID}^{(A)}, \mathbf{ID}^{(B)}, \textbf{Compute}, \Lambda\}.$$

In the above view, r_C is the outcome of internal random coins of the cloud, $\vec{\boldsymbol{o}}^{(A)}$, $\vec{\boldsymbol{o}}^{(B)}$ are the hash tables that comprise the blinded set representations of A's and B's sets, respectively. Also, $tk^{(A)}$ and $tk^{(B)}$ are l-bit random keys used in the protocol to generate the pseudorandom polynomials and the blinding factors used to mask the result generated by the cloud. To simulate this view, SIM_C does the following:

1. Creates an empty view and appends to it Λ and uniformly at random chosen coins r'_C.
2. Uses the public parameters and the hash function to construct two hash tables $\mathtt{HT}'^{(A)}$ and $\mathtt{HT}'^{(B)}$. Then, it fills each bin of the hash tables with n uniformly random values picked from the same field $\mathbb{F}_p$, so each bin $\mathtt{HT}'^{(I)}_j$ $(\forall I, I \in \{A, B\})$ contains a vector $\vec{\boldsymbol{o}}'^{(I)}_j$ of n random values.
3. Chooses two keys, $tk'^{(A)}$ and $tk'^{(B)}$, chosen uniformly at random and appends vectors $\vec{\boldsymbol{o}}'^{(A)} = [\vec{\boldsymbol{o}}'^{(A)}_1, ..., \vec{\boldsymbol{o}}'^{(A)}_h]$, $\vec{\boldsymbol{o}}'^{(B)} = [\vec{\boldsymbol{o}}'^{(B)}_1, ..., \vec{\boldsymbol{o}}'^{(B)}_h]$ and the keys to the view.
4. Constructs strings $\mathbf{ID}^{(A)}$, $\mathbf{ID}^{(B)}$, **Compute**, appends them along with Λ to the view and outputs the view.

We now show why the two views are computationally indistinguishable. In both views, the input parts are identical (i.e. both are Λ) and the random coins (r_C and r'_C) are both uniformly random, so they are indistinguishable. In the real model, the elements in each vector $\vec{\boldsymbol{o}}^{(I)}_j$ $(\forall I, I \in \{A, B\})$ are blinded with the outputs of a pseudorandom function. Also, the elements in $\vec{\boldsymbol{o}}'^{(I)}_j$ are random elements of the field. As the blinded values and random value are indistinguishable, vectors $\vec{\boldsymbol{o}}^{(I)}_j$ and $\vec{\boldsymbol{o}}'^{(I)}_j$ are not distinguishable. Thus, the vectors $\vec{\boldsymbol{o}}^{(I)}$ and $\vec{\boldsymbol{o}}'^{(I)}$ are indistinguishable. Furthermore, as keys $tk^{(A)}, tk^{(B)}, tk'^{(A)}$ and $tk'^{(B)}$ are picked uniformly at random, they are computationally indistinguishable as well. Also, $\mathbf{ID}^{(A)}$, $\mathbf{ID}^{(B)}$, and **Compute** in both models are identical, and the output parts in both views are identical (i.e. both are Λ). We conclude that the two views are indistinguishable.

Case 2: Corrupted Client A. In the real execution, client A's view is as follows:

$$\mathsf{VIEW}_A(\Lambda, S^{(A)}, S^{(B)}) = \{S^{(A)}, r_A, \vec{\boldsymbol{r}}^{(B)}, \mathsf{ID}^{(B)}, \Lambda\},$$

where $\vec{\boldsymbol{r}}^{(B)} = [\vec{\boldsymbol{r}}_1^{(B)}, ..., \vec{\boldsymbol{r}}_h^{(B)}]$ is the message sent by client B to client A. The simulator, SIM_A, who receives the party's input, $S^{(A)}$, and output, Λ, does the following.

1. Creates an empty view and appends the party's input, $S^{(A)}$, to it.
2. Inserts coins r'_A, chosen uniformly at random, to the view.
3. Constructs vector $\vec{\boldsymbol{r}}'^{(B)} = [\vec{\boldsymbol{r}}'^{(B)}_1, ... \vec{\boldsymbol{r}}'^{(B)}_h]$ where each of the vectors, $\vec{\boldsymbol{r}}'^{(B)}_j$, contains n uniformly random elements of the field. It appends $\vec{\boldsymbol{r}}'^{(B)}$ to the view.
4. Adds the strings, $\mathsf{ID}^{(B)}$ and Λ, to the view, and outputs it.

In the following, we explain why the two views are computationally indistinguishable. In both models $S^{(A)}$ is identical. Moreover, both r_A and r'_A are picked uniformly at random so they are indistinguishable. The elements of vectors $\vec{\boldsymbol{r}}_j^{(B)}$ are blinded by the uniformly random elements that are outputs of a pseudorandom function. So the blinded values are distributed uniformly at random over the field. On the other hand, vectors $\vec{\boldsymbol{r}}'^{(B)}_j$ contain random elements of the field. Since, pseudorandom values of a field are indistinguishable from random values of the same field, vectors $\vec{\boldsymbol{r}}_i^{(B)}$ and $\vec{\boldsymbol{r}}'^{(B)}_i$ are indistinguishable. Therefore, vectors $\vec{\boldsymbol{r}}^{(B)}$ and $\vec{\boldsymbol{r}}'^{(B)}$ are indistinguishable as well. Moreover, $\mathsf{ID}^{(B)}$ and Λ are identical in both models. Therefore, the two views are computationally indistinguishable in this case.

Case 3: Corrupted Client B. In the real execution, client B's view is as follows:

$$\mathsf{VIEW}_B(\Lambda, S^{(A)}, S^{(B)}) = \{S^{(B)}, r_B, \vec{\boldsymbol{g}}, \vec{\boldsymbol{q}}, f_\cap(S^{(A)}, S^{(B)})\}.$$

The simulator, SIM_B, who receives the party's input, $S^{(B)}$, and output, $f_\cap(S^{(A)}, S^{(B)})$, performs as follows.

1. Creates an empty view, then it appends $S^{(B)}$ and uniformly at random chosen coins r'_B to it.
2. Chooses two sets $S'^{(A)}$ and $S'^{(B)}$ such that $|S'^{(A)}|, |S'^{(B)}| \leq c$ and $S'^{(A)} \cap S'^{(B)} = f_\cap(S^{(A)}, S^{(B)})$.
3. Constructs hash tables $\mathtt{HT}'^{(A)}$ and $\mathtt{HT}'^{(B)}$ using the public parameters, and maps the elements in $S'^{(A)}$ and $S'^{(B)}$ to the bins of $\mathtt{HT}'^{(A)}$ and $\mathtt{HT}'^{(B)}$, respectively.

$\forall I, I \in \{A, B\},\ \forall s_i'^{(I)} \in S'^{(I)}$:

$$\mathtt{H}(s_i'^{(I)}) = j,\ s_i'^{(I)} \rightarrow \mathtt{HT}_j'^{(I)},$$

where $1 \leq j \leq h$.

4. For each bin constructs a polynomial representing its elements. If a bin contains less than d elements, it is padded with random values to d elements.
   $\forall I, I \in \{A, B\},\ \forall j, 1 \leq j \leq h$:

$$\tau_j'^{(I)}(x) = \prod_{m=1}^{d}(x - e_m^{(I)}),$$

   where $e_m^{(I)} \in \mathtt{HT}_j'^{(I)}$, $e_m^{(I)}$ is either the set element or a random value.
5. Assigns a random polynomial, $\omega_j'^{(I)}$, of degree d to each bin, $\mathtt{HT}_j'^{(I)}$ ($\forall j, 1 \leq j \leq h$), of the hash table, $\mathtt{HT}'^{(I)}$ ($\forall I, I \in \{A, B\}$).
6. Constructs vectors $\vec{\boldsymbol{g}}_j'$ whose elements are computed as follows.
   $\forall j, 1 \leq j \leq h,\ \forall i, 1 \leq i \leq n$:

$$g_{j,i}' = \tau_j'^{(A)}(x_i) \cdot \omega_j'^{(A)}(x_i) + \tau_j'^{(B)}(x_i) \cdot \omega_j'^{(B)}(x_i),$$

   where $\tau_j'^{(I)}(x)$ is the polynomial representing the set of elements contained in bin $\mathtt{HT}_j^{(I)}$.
7. Picks a key, mk', and derives h keys, k_j', from it as below.

$$\forall j, 1 \leq j \leq h : k_j' = \mathtt{PRF}(mk', j).$$

8. Uses each key k_j' to generate vector $\vec{\boldsymbol{q}}_j'$ whose elements are computed as follows.

$$\forall j, 1 \leq j \leq h,\ \forall i, 1 \leq i \leq n : q_{j,i}' = \mathtt{PRF}(k_j', i).$$

9. Adds $\vec{\boldsymbol{g}}' = [\vec{\boldsymbol{g}}_1', ..., \vec{\boldsymbol{g}}_h']$ and $\vec{\boldsymbol{q}}' = [\vec{\boldsymbol{q}}_1', ..., \vec{\boldsymbol{q}}_h']$ to the view. Also, it inserts to the view the party's output, $f_\cap(S^{(A)}, S^{(B)})$, and outputs it.

In the following we show why the two views are computationally indistinguishable. In both models $S^{(B)}$ is identical. As r_B and r_B' are chosen uniformly at random, they are indistinguishable as well. Furthermore, in the real model, given each unblinded vector $\vec{\boldsymbol{g}}_j$, the adversary interpolates a polynomial of the form $\phi(x)_j = \omega_j^{(A)}(x) \cdot \tau_j^{(A)}(x) +$

Property	EO-PSI	O-PSI	[68]	[73]	[81]	[74]	[123]	[99]
Private Against the Cloud	✓	✓	✓	✓	×	✓	×	×
Client-to-client Computation Authorization	✓	✓	✓	✓	✓	✓	✓	✓
Non-interactive Client-side Setup	✓	✓	×	×	✓	✓	✓	✓
Secure Repeated PSI Delegation	✓	✓	×	×	×	×	×	×
Multiple Clients	✓	✓	✓	✓	✓	×	✓	✓
Not Requiring a Trusted Third Party	✓	✓	✓	✓	✓	✓	×	×
Communication Complexity	$O(hd)$	$O(c)$	$O(c)$	$O(c^2)$	$O(c)$	$O(c^2)$	$O(k)$	$O(k)$
Computation Complexity	$O(hd^2)$	$O(c)$	$O(c)$	$O(c^2)$	$O(c^2)$	$O(c^2)$	$O(c)$	$O(c)$

Table 4.1: Comparison of different delegated PSI protocols. Set cardinality upper bound and intersection cardinality are denoted by c and k, respectively. Also, h is the length of hash table and d is a bin's maximum load.

$\omega_j^{(B)}(x) \cdot \tau_j^{(B)}(x) = \mu_j \cdot gcd(\tau_j^{(A)}(x), \tau_j^{(B)}(x))$. Similarly, in the ideal model, each polynomial $\phi'_j(x)$ interpolated from vector $\vec{\boldsymbol{g}}'_j$ has the form $\phi'_j(x) = \omega_j^{\prime(A)}(x) \cdot \tau_j^{\prime(A)}(x) + \omega_j^{\prime(B)}(x) \cdot \tau_j^{\prime(B)}(x) = \mu'_j \cdot gcd(\tau_j^{\prime(A)}(x), \tau_j^{\prime(B)}(x))$. As we discussed in section 2.5 (and it is proven in [75, 20]), μ_j and μ'_j are uniformly random polynomials and indistinguishable. Besides, only with negligible probability their roots represent set elements.

Moreover, polynomials $gcd(\tau_j^{\prime(A)}(x), \tau_j^{\prime(B)}(x))$ and $gcd(\tau_j^{(A)}(x), \tau_j^{(B)}(x))$ represent the intersection, $f_\cap(S_j^{(A)}, S_j^{(B)})$. Therefore, $\vec{\boldsymbol{g}}'$ and $\vec{\boldsymbol{g}}$ are computationally indistinguishable. In the real model, the elements in $\vec{\boldsymbol{q}}_j$ are blinded by the outputs of a pseudorandom function. So the blinded elements are uniformly random values. On the other hand, in the ideal model the elements in $\vec{\boldsymbol{q}}'_j$ are the outputs of a pseudorandom function. So, the elements in both vectors $\vec{\boldsymbol{q}}$ and $\vec{\boldsymbol{q}}'$ are computationally indistinguishable. Moreover, values x_i are not equal to any set elements (as shown in section 2.6). Finally, in both views the output part (i.e. $f_\cap(S^{(A)}, S^{(B)})$) is the same. Hence, the two views are computationally indistinguishable. Combining the above, we conclude the protocol is secure and complete our proof. □

4.4 Delegated PSI Protocol Comparison

We first evaluate EO-PSI and O-PSI by comparing their properties to those provided by other protocols that delegate PSI computation to a cloud. We also compare these protocols in terms of communication and computation complexity. Table 4.1 summarizes the results.

Properties. When PSI computation is delegated to the cloud that is not fully trusted, protecting the privacy of the computation inputs and output from it, is crucial. Protocols like the size-hiding variation of [68], those in [73, 74], O-PSI and EO-PSI protect the privacy of the computation inputs and output. However, as we discussed in section 3.3.2, the protocols in [81, 123, 99] do not fully preserve data privacy.

Another desirable security property is that PSI computation is only possible with the explicit authorization from all the clients. But, this property is not fully satisfied in [81, 123, 99] and the cloud can use the outsourced data to compute PSI without clients' permission. Although the cloud cannot decrypt the data and the result, it can learn some information about the intersection. In particular, if the intersection between the sets of client A and B is computed, followed by that between the sets of client A and C, then the cloud will also find out whether some elements are common in the sets of client B and C without their consent. However, this is not the case for the other protocols. Because these protocols are fully secure and protect clients' data and the computation result from the cloud. Furthermore, all the protocols can enforce client-to-client authorization by using an infrastructure such as PKI.

The protocols in [68, 73] require clients to interact with each other at setup. In [68] clients need to jointly compute the key for the pseudorandom permutation used to encode the datasets, while in [73] they need to jointly compute the parameters used to encrypt their datasets. In contrast to these protocols, in [81, 123, 74, 99], O-PSI and EO-PSI the clients can independently prepare and outsource their private datasets.

Moreover, only O-PSI and EO-PSI support secure repeated PSI delegation, so clients can store their private data in the cloud and delegate the computation to the cloud an unlimited number of times without leaking any information about the set elements. Nevertheless, this is not the case for any of the other protocols. Because they either support one-off PSI delegation or leak information about clients' set elements to the cloud (see section 3 for an elaborate discussion).

As we illustrated in sections 4.2.3.1 and 4.3.3.1, O-PSI and EO-PSI can be extended to support multiple clients. The same holds for [68, 73, 81, 123, 99]. In contrast, [74] does not support multiple clients, as it requires an additional logical operation that is not supported by the homomorphic encryption scheme it uses. The protocols in [99, 123] require a trusted third party to initialize, and distribute a set of (private) parameters among parties, whereas the rest of the protocols do not need such assistance.

Communication Complexity. The communication complexity of two-client O-PSI for the client who receives the result, client B, is $O(c)$, where c is the upper bound of the set cardinality (or the set maximum size). Because client B sends, to client A, $n = 2c + 1$ encrypted random values $E_{pk_B}(r_i^{(B)})$ for $1 \leq i \leq n$, in step c.2. The communication complexity for client A, who authorizes the operation on its dataset, is also $O(c)$, as it sends n values of the form $E_{pk_B}(r_i^{(B)} \cdot (r_i^{(A)})^{-1})$, $1 \leq i \leq n$ to the cloud, in step c.4. The communication complexity for the cloud is $O(c)$ too. Because, it sends to client B the result vector $\vec{t}$ of size n, in step d.5. Moreover, in multi-client O-PSI, client B's communication cost is $O(c \cdot (m-1))$ as it sends n encrypted values to each client A_q, where $1 \leq q \leq m-1$ and m is total number of clients. However, the communication cost of client A and the cloud remains the same as that of in the two-client case.

In the following, we evaluate EO-PSI communication complexity. In step c.2, client B sends a single key to the cloud, and h bins to client A, where each bin contains $n = 2d+1$ elements. So its communication cost is $O(hd)$. Client A sends a single key to the cloud, and h bins to client B in step c.8, where each bin contains n elements. Therefore, client A's communication complexity is $O(hd)$. The cloud's communication complexity is $O(hd)$ as well, because in step d.3, it sends h bins to client B, where each bin comprises n elements. Note that each message in this protocol is a random element of the field (i.e. an output of a symmetric key encryption). In multi-client EO-PSI, the communication cost of the cloud and the clients who authorize the computation remains the same as that of in two-client case. But, in multi-client EO-PSI, client B's communication cost is $O(hd(m-1))$ as it sends h bins to each client A_q, where each bin contains $n = 2d + 1$ elements.

In [73] for each set intersection, the client engages in a two-round protocol, one round to upload its elements in the form of RSA ciphertexts to the cloud with $O(c)$ communication complexity, and another to interactively compute the private set intersection with the cloud with $O(c^2)$ communication complexity. For the protocol in [74], the communication complexity is also quadratic $O(sc^2)$, where s is the number of hash function used for the bloom filter, and the messages contain BGN encryption ciphertexts. On the other hand, the protocols in [68, 81] have $O(c)$ communication complexity with messages containing symmetric key encryption ciphertexts. Finally, the protocols in [123, 99] have $O(k)$ complexity with messages containing public key encryption, where k is the intersection size.

In conclusion, the two protocols in [123, 99] have smaller communication complexity but they are not fully secure. The protocol in [81] has linear communication complexity but it is not fully private too. On the other hand, similar to the secure PSI protocol [68], O-PSI and EO-PSI have linear communication complexity. However, the message size in O-PSI is larger than the message sizes in [68] and EO-PSI.

Computation Complexity. We evaluate the computation cost of O-PSI by counting the number of exponentiation operations, as their cost dominates that of other operations in the protocol. In two-client O-PSI, client B performs n exponentiations to encrypt the random values in step c.1, and needs another n exponentiations to decrypt the polynomial sent by the cloud in step e.1. So, in total it carries out $2n$ exponentiations. Client A performs n exponentiations to authorize the set intersection in step c.3. The cloud carries out n exponentiations to encrypt client B's dataset and n exponentiations to operate on client A's dataset in step d.2 and d.3. Furthermore, it performs n exponentiations in step d.4. It is interesting to note that although using the point-value representation moderately increases the overall storage costs at the cloud-side, it brings a significant decrease in the computational costs, from $O(c^2)$ (when using encrypted coefficients such as in [75]) to $O(c)$. Furthermore, in multi-client O-PSI, clients computation cost remain the same as their cost in two-client case. But, in multi-client case, the cloud computation cost is $O(cm)$. Because, it performs n exponentiation operations on each client A_q dataset, performs n exponentiation operations to encrypt client B dataset, and then combines them together.

Now we analyze the computation complexity of EO-PSI. In our analysis, we do not consider the pseudorandom function invocation cost as it is a fast operation, and dominated by the other operations (e.g. modular arithmetic, interpolation and factorization) in our protocol. Client A, performs $2hn$ modular multiplication and $2hn$ modular addition operations to blind the values, in step c.7. Also, in order to evaluate the two polynomials allocated to every bin, it carries out $2hnd$ modular multiplication and $2hnd$ modular addition operations, in step c.7. So, the computation complexity of client A is $O(hd^2)$. Also, the cloud carries out $2hn$ modular multiplication and $3hn$ modular addition operations to blind the values in step d.2. Moreover, for the cloud to evaluate the two polynomials assigned to every bin, it performs $2hnd$ modular multiplication and $2hnd$ modular addition operations, in step d.2. Therefore, the computation complexity for the cloud is $O(hd^2)$. Client B executes hn modular addition operations to blind the messages in step c.1. Moreover, it performs hn modular addition operations

to unblind the cloud's response, in step e.1. Furthermore, in step e.2, it interpolates h polynomials where each polynomial interpolation costs $O(d)$. In step e.3, it factorizes h polynomials where each polynomial factorization costs $O(d^2)$. Hence, in total client B's computation cost is $O(hd^2)$. Also, in multi-client EO-PSI, client B's computation cost is $O(hd^2m)$. As, in order for it to remove the blinding factors, it needs to combine the messages sent by the other clients and this introduces $O(hd^2(m-1))$ cost in addition to its cost in two-client setting. The cloud's computation cost in multi-client setting is $O(hd^2m)$, because it needs to perform on each client message separately and then it combines them together to compute the result. However, the other clients (i.e. A_q) computation cost remain the same as in two-client case. Shortly, we will show that in our protocol (with the right choice of parameters) $d = 100$ and $hd \leq 4c$.

The semi-honest variant of the protocol in [68] has linear complexity $O(c)$, as the client computing the result and the cloud invokes the pseudorandom permutation (PRP) c times, while the other client invokes the PRP, $2c$ times. On the other hand, the computational overhead in [73] is quadratic and it involves $O(c^2)$ RSA encryption operations. The protocol in [74] also has quadratic complexity, and it involves $O(c^2)$ BGN public key encryption operations. In [81] the client performs $O(c)$ modular additions, while the cloud carries out $O(c^2)$ operations to compare the expanded sets of the users. The protocol in [123] is based on bilinear maps and requires $6c$ pairings at the cloud-side and $2k$ exponentiations at the client side, resulting in $O(c)$ and $O(k)$ computation complexity at the cloud and client side respectively. The protocol in [99] is also based on bilinear maps, it requires 6 exponentiation operations at client-side, k pairing for decrypting the result at the client-side, and $8c$ pairing at the server-side. So the overall computation cost of the protocol is $O(c)$.

We highlight that there exists a generic protocol in [82] that allows clients to independently outsource both data and computation to the cloud. But, the protocol is based on FHE scheme that is computationally very expensive; and the overall number of FHE operations, in this protocol, is linear to the inputs size, c. So it involves at least $O(c)$ fully homomorphic operations. Furthermore, the communication cost of the protocol is at least $O(c)$ where each message is the FHE ciphertext (whose length is higher than partially homomorphic encryption ciphertext's length). However, as the protocol is designed for generic computation, a precise complexity evaluation for PSI has not been defined.

In conclusion, O-PSI and EO-PSI, similar to [82, 123, 81, 99, 73] have linear computation complexity. However, the protocols in [123, 81, 99] are not fully secure. The

computation in [82] involves expensive fully homomorphic encryption, the computation in O-PSI involves partially homomorphic encryption that is cheaper. Moreover, the dominant operation in EO-PSI is polynomial factorization and in [73] is symmetric key encryption, and the latter protocol is more efficient than the other protocols, however it lacks some properties that O-PSI and EO-PSI offer.

4.5 Performance Evaluation

In this section, we first show how in EO-PSI, with the right choice of parameters, we can keep the overall cost optimal. After that, we outline the code design of O-PSI and EO-PSI, and then we provide the performance analysis of the protocols.

4.5.1 Choice of Parameters

In EO-PSI, with the right choice of parameters, the cloud can keep the overall costs optimal, while keeping the overflow probability negligibly small, e.g. 2^{-40}. In this section, we show how the parameters can be chosen. As before, let c be the upper bound of the set cardinality, d be the bin size, and h be the number of bins. Recall that in our protocol, as we have shown, the overall (computation, communication and storage) cost depends on the product, hd (i.e. the total number of elements, including set elements and random values stored in the hash table). Furthermore, the computation cost is dominated by factorizing h polynomials of degree $n = 2d + 1$. In order for the cloud to keep the costs optimal, given c, it uses inequality 2.4, to find the right balance between parameters d and h, in the sense that the cost of factorizing a polynomial of degree $n = 2d + 1$ is minimal, while hd is *close* to c.

At a high level, we do the following to find the right parameters. First, we measure the average time, t, taken to factorize a degree $n = 2d + 1$ polynomial for different values of n (see Fig. 4.4). Then, for each c, we compute h for different values of d (see Fig. 4.5). Next, for each d we compute ht, after that for each c we find minimal d whose ht is at the lowest level (see Fig. 4.6). In the experiments, similar to [96, 38], the number of set elements upper bound ranges over $[2^{10}, 2^{20}]$, and the bit length of set elements is 32. Furthermore, we use 80 bits for padding. Therefore, the bit length of p would be 112, i.e. $|p| = 32 + 80 = 112$. In this experiment, the pad, b_i, for each element, s_i, is computed as $b_i := \mathtt{H}(s_i) \bmod 2^{80}$, where $|b_i| = 80$ and $\mathtt{H}(.)$ is a fixed cryptographic hash function (as we stated in 2.5). Then, each set element is encoded

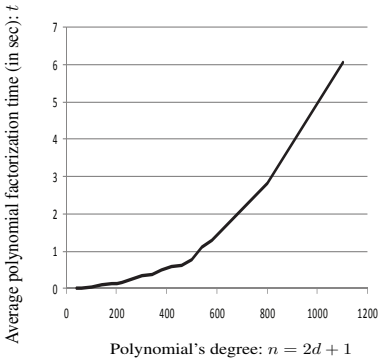

Figure 4.4: The average time taken to factorize polynomials of degree n defined over $\mathbb{F}_p$, where p is an 112-bit prime number.

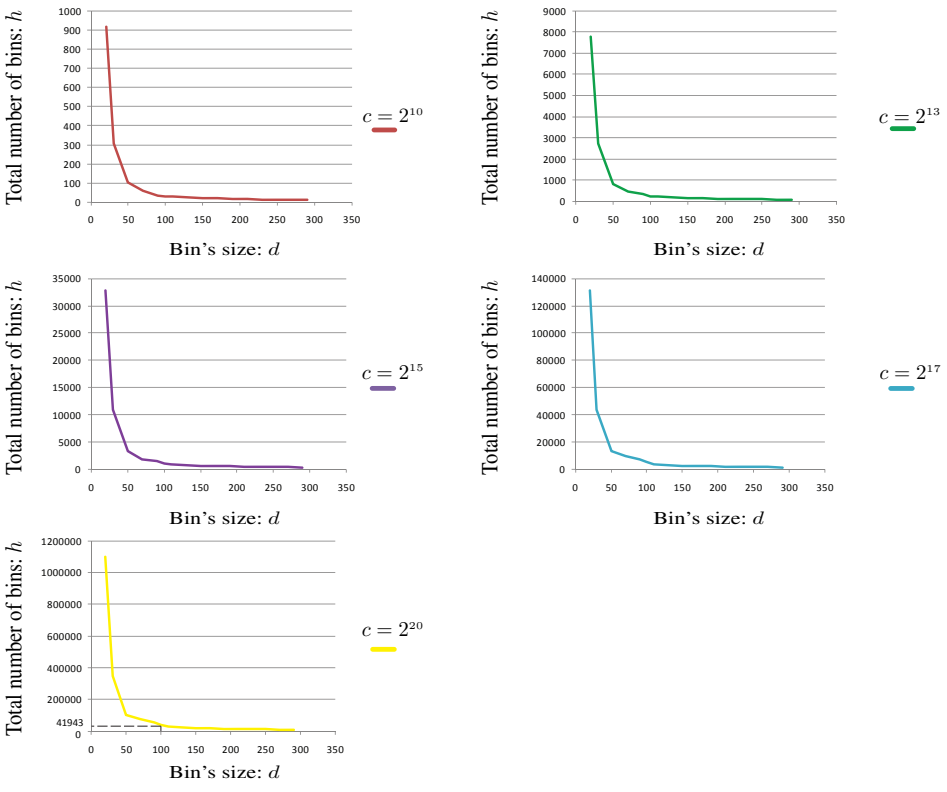

Figure 4.5: The relation between the number of bins, h, and the size of each bin, d, for different set size upper bounds, c.

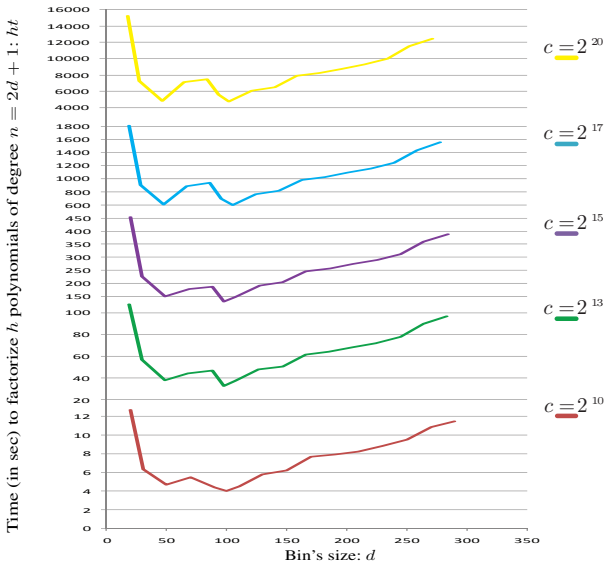

Figure 4.6: The average time taken to factorize h polynomials of degree $n = 2d + 1$, for different set size upper bounds, c. The polynomials are defined over the field $\mathbb{F}_p$, where p is an 112-bit prime number.

as $s'_i = s_i||b_i$. In this setting, the probability that an arbitrary element of the field represents the encoded elements of the correct form is at most 2^{-80}.

First, we determine the running time of polynomial factorization, t. To calculate the running time, we factorize random polynomials of degree $n = 2d+1$, for different values of n, where $n \in [20, 1000]$, so n can have small and large values. The result of the experiment [1] is depicted in Fig 4.4. As we can see in the figure, when $n > 600$ the time grows rapidly. This means, if we put so many elements in a bin, it would take too long to factorize a bin's polynomial. So we set $n < 600$ (i.e. $d < 300$).

Second, for each value of $c \in \{2^{10}, 2^{13}, 2^{15}, 2^{17}, 2^{20}\}$, and $d \in [20, 290]$, we use the inequality to find their corresponding number of bins, h; while keeping the probability (of bin overloading) below 2^{-40}. The result is depicted in Fig 4.5. Interestingly, as it is evident in the figure, (for all $c \in \{2^{10}, 2^{13}, 2^{15}, 2^{17}, 2^{20}\}$) h grows rapidly when $d < 50$ decreases; however, such growth is much slower when $d > 100$ reduces.

Third, given t and h, for each c we calculate the time of factorizing h polynomial

[1] The experiment is implemented in C++, and conducted on Ubuntu 14.04 desktop PC with an Intel Core $i5$-3570@3.4 GHz CPU and 16 GB RAM. We use the NTL library for factorizing the polynomials defined over a field, $\mathbb{F}_p$, where p is an 112-bit prime number. Furthermore, to obtain the average time, for each polynomial's degree, n, we run the experiment 100 times using distinct random polynomials whose coefficients are picked uniformly at random from the field.

of degree $n = 2d + 1$ for different d (i.e. we calculate ht). The result is illustrated in Fig 4.6. It is evident in the figure that, for all c, when $100 \leq d \leq 110$ the computation cost is at the lowest level, so we set $d = 100$. In this case, as can be seen in figure 4.5, $hd \leq 4c$. For example, as we show in the graph at the bottom left of the figure (i.e. $c = 2^{20} = 1048576$), for $d = 100$, we have $h = 41943$, so $hd = 4194300 \approx 4c$.

In conclusion, in the protocol, the cloud can set $d = 100$ for all values of c. In this setting, hd is at most $4c$ and only with negligibly small probability, 2^{-40}, a bin may receive more than d elements.

4.5.2 Implementation

In the following, we provide an abstract overview of the O-PSI and EO-PSI code design. The protocols have been implemented in C++. The implementations use the NTL [1] and GMP [2] libraries for polynomial factorization and big integer operations, respectively. The running time of polynomial factorization for the NTL library is a factor of 2-3 faster than the FLINT library [3]. The implementation of O-PSI also utilizes a Paillier homomorphic encryption library [4].

The O-PSI protocol implementation has three main classes `Client`, `Server` and `Polynomial` as well as a `Random` class that provides functions to generate truly random values. Furthermore, the protocol implementation contains three structures carrying the set of messages exchanged between the parties. The structures are: (1) `CompPerm_Request` that contains the messages sent by client B to client A when client B wants to obtain its permission, (2) `GrantComp_Info` carrying the messages sent by client A to the cloud to delegate computation to it, and (3) `Server_Result` which comprises the result sent to client B by the (cloud) server. The class diagram for O-PSI is provided in appendix A.

On the other hand, the EO-PSI implementation has five classes `Client`, `Server`, `Polynomial`, `Hashtable` and `Random`. The implementation involves four structures where `CompPerm_Request`, `GrantComp_Info` and `Server_Result` have the same responsibilities as they do in O-PSI, but they carry different types of messages than in O-PSI. In EO-PSI, there is an additional structure called `Client_Dataset`

[1] http://www.shoup.net/ntl/

[2] https://www.gmplib.org/

[3] Detailed evaluation can be found on http://www.shoup.net/ntl/. We also ran an experiment and compared the two libraries' performances for the factorization, and reached the same conclusion.

[4] Advanced Crypto Software Collection: http://acsc.cs.utexas.edu/libpaillier/

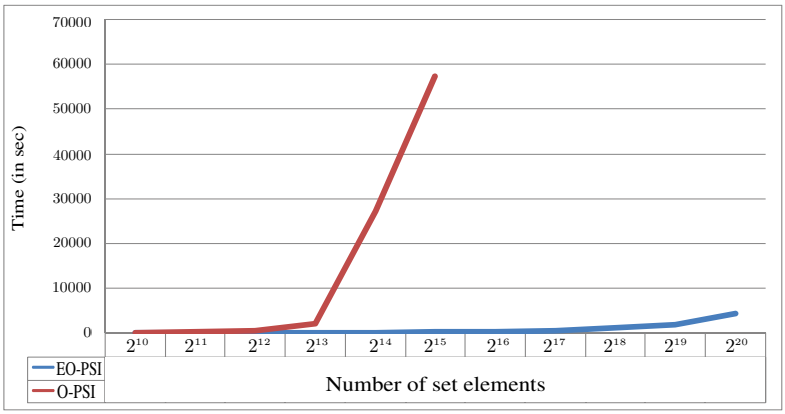

Figure 4.7: Performance comparison of EO-PSI and O-PSI protocols

that contains the blinded polynomials, in the hash table, outsourced by the client to the server. It should be noted that the member functions of `Client` and `Server` classes in O-PSI and EO-PSI have different implementations (as the underlying protocols are different), even though they may have the same names and responsibilities in both protocols implementations. The class diagram for EO-PSI is provided in appendix B.

4.5.3 Performance Comparison

In the following, we report the performance analysis of the two protocols based on the experiments we conducted. The experiments were carried out on Ubuntu 14.04 desktop PC with an Intel Core $i5$-$3570@3.4$ GHz CPU and 16 GB RAM. All the polynomials and plaintexts are defined over the field, $\mathbb{F}_p$, where p is an 112-bit prime number. Also, for O-PSI that uses Paillier encryption, we set the size of the public key, N, 512 bits. In the experiments, the number of set elements ranges over $[2^{10}, 2^{20}]$, whose size is 32-bit. For EO-PSI, as we discussed in section 4.5.1, we set the hash table bin size, d, to 100 and the probability of overflow to less than 2^{-40}.

In Fig. 4.7, we show the performance comparison of the two protocols. It is evident in the figure that the performance of EO-PSI is much better than OPSI. In particular, EO-PSI is about 1-2 orders of magnitude faster than O-PSI. For O-PSI we skipped tests with set size over 2^{15} as the running time would be too long. As we highlighted earlier, there are two reasons why EO-PSI outperforms O-PSI, (1) no use of public key encryption, (2) the result recipient factorizes a set of smaller degree polynomials.

Furthermore, in Table 4.2, we show the time taken for each main step of the EO-

Main Steps	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Data Outsourcing	0.08	0.17	0.33	0.68	1.37	2.81	7.76	16.63	37.56	74.54	258.44
Computation Delegation	0.21	0.44	0.88	1.77	3.53	7.13	19.19	38.18	76.02	152.94	302.35
Cloud-side Computation	0.18	0.38	0.76	1.53	3.06	6.18	16.58	32.99	65.69	137.78	261.67
Client-side Result Retrieval	3.06	6.31	11.55	24.51	49.96	97.47	208.83	434.65	858.23	1386.47	3413.57
Total	3.53	7.30	13.52	28.47	57.92	113.59	252.36	522.45	1037.50	1751.73	4236.03

Table 4.2: EO-PSI Performance: performance time (in second) for each step.

PSI protocol. As the table and Fig 4.7 show, the performance time increases when the number of set elements grows. The growth is steady when the set size is smaller than 2^{19}, but faster when the set size becomes larger than 2^{19}. Moreover, the performance in each of the first three steps is faster than the last step, the result retrieval. In the last step, client B unblinds the data received, interpolates the polynomials, and extracts roots from the polynomials by factorizing them. In this process, the polynomial factorization takes most of the time (the other operations are very fast).

4.6 Concluding Remarks

In this chapter, we presented two protocols, O-PSI and EO-PSI, that support secure repeated PSI delegation. They allow multiple clients to independently outsource their private data to the cloud and at any point in time, can delegate PSI computation on their data to it. In this process, the cloud learns nothing about the dataset elements, the intersection, and the intersection cardinality. Furthermore, the protocols ensure that the cloud can compute the intersection only when all the clients agree, and the clients can securely delegate PSI computation on the outsourced datasets an unlimited number of times with no need to download and re-prepare the datasets. We showed that our protocols are secure in the presence of semi-honest parties. O-PSI is based on point-value polynomial representation, a blinding technique and partially homomorphic encryption (i.e. Paillier encryption) that is widely used in secure multi-party computation protocols. Although O-PSI allows secure outsourcing of data storage and

the computation, it utilizes a public key encryption scheme which is not very efficient in general. Therefore, we proposed EO-PSI that is much more efficient than O-PSI for two reasons. First, it does not use any public key encryption, and second it lets clients retrieve the result faster by utilizing a hash table. We implemented both protocols and analyzed their performance. The analysis showed that EO-PSI is 1-2 orders of magnitude faster than O-PSI.

Chapter 5

Delegated PSI on Outsourced Dynamic Private Datasets

5.1 Introduction

In the previous chapter, we proposed two protocols, O-PSI and EO-PSI, that support secure repeated PSI delegation. The protocols allow clients to outsource their private datasets to the cloud and later on they can delegate PSI computation on the data to it. The protocols ensure that the privacy of client's data and computation result are preserved in the cloud and the result recipient cannot learn anything beyond the intersection. However, these protocols are suitable for static data. While the static model fits some application scenarios (e.g. libraries and scientific datasets), it will impose large communication, storage and computation costs when big data needs to be updated regularly.

In this chapter, we present UEO-PSI, an efficient protocol that supports secure repeated PSI delegation on *dynamic* data. It lets multiple clients independently outsource their private data to the cloud; and over time, they can *efficiently update* their outsourced data. At any point in time, they can get together and delegate PSI computation to the cloud. In these processes, the cloud cannot learn anything about the clients' set elements and the computation output [1]. The protocol preserves the desirable features and efficiency of EO-PSI, as it does not require any (expensive) public key encryption, and in practice it could also yield efficient implementation.

[1] Although the cloud learns the query and access pattern in this process, it cannot infer anything about the computation input and output from the patterns.

The rest of this chapter is organized as follows. Section 5.2 starts with a discussion about update operations in the delegated PSI protocols we designed so far, and continues with an overview and a detailed description of UEO-PSI including the protocol's extensions. In section 5.3, we define the security model for our protocol followed by section 5.4 that contains the security analysis of UEO-PSI. In section 5.5, we compare the protocol with the other secure delegated PSI protocols. Finally, in section 5.6, we provide concluding remarks of this chapter.

5.2 UEO-PSI: Efficient Delegated Private Set Intersection on Dynamic Outsourced Private Data

In this section, we first explain why our two previous protocols cannot directly support efficient data update, and then we provide an overview of UEO-PSI protocol followed by a detailed description of it. After that, we outline the rationale behind the protocol design, and the protocol's extensions.

5.2.1 Data Update in O-PSI and EO-PSI

Although the two delegated PSI protocols, O-PSI and EO-PSI, we designed so far can securely delegate PSI computation on the outsourced data to the cloud, they cannot efficiently support data update and therefore they are suitable for static data.

In O-PSI protocol, the entire set is represented as one blinded polynomial that is outsourced to the cloud. Since a client does not have any local copy of the data, it does not know whether the elements it wants to update (i.e. insert or delete) exists in the outsourced data. Hence, in order for the client to update the outsourced data, it needs to download the entire outsourced dataset, check whether the update is required, and if it is needed, update the data locally, and upload it to the cloud. But this is not efficient (in terms of communication, computation and storage costs).

Recall that our EO-PSI protocol uses a hash table and at first glance in order for a client to update its outsourced data, it can efficiently update only one bin (i.e. it retrieves a bin, checks if the update is needed, if so it updates it locally and sends it back to the cloud). However, in this protocol, if the client naively updates one bin, the cloud might be able to learn some of its set elements. In particular, since the hash table bins are in their original order, and each bin's address is the hash value of an element in

the bin, if the client retrieves one bin, then the cloud would learn what element is being updated in a bin with a non-negligible probability when the set universe is not big. In this case, the cloud can carry out a brute force attack. To do that, it can enumerate the elements of the set universe and then it can determine to which bin each element is mapped. This will allow the cloud to figure out exactly which elements and how many of them are mapped to a specific bin. Therefore, in the update phase, if a client accesses a bin and updates it, the cloud would learn with probability at least $\frac{1}{v}$ that a specific element is being updated, where v is the total number of elements mapped to that bin. Therefore, in EO-PSI too, the secure way for a client to update the outsourced dataset is to retrieve the entire dataset, update it locally and send it back to the cloud.

5.2.2 An Overview of UEO-PSI

We design UEO-PSI on top of the EO-PSI protocol. In UEO-PSI, in order for a client, to hide the original address of the hash table bins from the cloud, it pseudorandomly permutes the bins and then sends them to the cloud.

As in EO-PSI, for each client to prepare its set, it constructs a hash table whose parameters are published by the cloud. Then, it maps the set elements into the hash table bins and pads the bins with random elements to the bin's maximum size. Afterwards, it represents the elements in each bin as a blinded point-value polynomial. The difference is that each client now needs to assign a unique random label to each bin, where each label is a (pseuo)random string of length l, where l is a security parameter. Then, it pseudorandomly permutes the hash table bins, and the labels. The client retains the label-to-bin mapping and the permutation, by keeping only two secret keys. Then, it sends the permuted hash table (with the bin labels) to the cloud. The random mapping and permutation prevent the cloud from learning anything about the bins' original order.

In order for the client to insert/delete an element, it first determines to which bin the element belongs. After that, the client recomputes the corresponding label, sends the label to the cloud and asks it to send the bin tagged with the label. Upon receiving the bin, the client decodes it and checks if the element exists in the bin. It updates the elements (if it is needed), pads and re-encodes the bin's content, and sends it back to the cloud. Therefore, the client needs to access only one bin to update it.

Now we explain how the clients delegate the computation to the cloud. When client B wants the intersection of its set and client A's set, it sends a message to client A to

obtain its permission. If client A agrees, it generates two sets of messages, one for client B and the other for the cloud. It sends a message containing unblinding vectors to client B, and a message including a permutation map to the cloud. The vectors help client B to unblind the cloud's response. The map lets the cloud associate one client's bins to the other client's bin, while it cannot learn their original order. The cloud uses client A's message and the outsourced datasets to compute a set of blinded polynomials and sends them to client B. Given the polynomials and client A's message, client B unblinds them and retrieves the intersection of the sets.

5.2.3 UEO-PSI Protocol

In this section, without loss of generality, we consider the two-client case, where client A, client B and the cloud engage in the protocol.

a. **Cloud-Side Setup.** The cloud in this phase, similar to EO-PSI protocol, sets c as an upper bound of set cardinality, and it sets the parameters for the hash (h, $\mathtt{H}$ and d). Also, it picks pseudorandom functions $\mathtt{PRF} : \{0,1\}^b \times \{0,1\}^l \rightarrow \mathbb{F}_p$ and $\mathtt{PRF}' : \{0,1\}^b \times \{0,1\}^l \rightarrow \{0,1\}^l$, where $|p| = l'$ and l, l' are the security parameters. Also, it picks a vector $\vec{\boldsymbol{x}}$ of $n = 2d + 1$ distinct non-zero values, x_i. Also, it chooses a pseudorandom shuffle, π, that permutes the elements of an input vector pseudorandomly. The cloud publishes c, the parameters of the hash table, the description of the field, value n, vector $\vec{\boldsymbol{x}}$, pseudorandom functions $\mathtt{PRF}, \mathtt{PRF}'$, pseudorandom shuffle π along with hash function $\mathtt{H}$.

b. **Client-Side Setup and Data Outsourcing.** Let client $I \in \{A, B\}$ have a set $S^{(I)}$, where $|S^{(I)}| < c$. Each client I performs the following:

   1. In this step, the client checks that value x_i are not equal to its set elements, inserts its set elements to a hash table and encodes the elements in each bin as a point-value polynomial representation. This step is similar to steps b.1-b.3b in EO-PSI protocol and here we briefly explain. The client constructs a hash table $\mathtt{HT}^{(I)}$ and inserts its set elements into the table. Then, it assigns a key to each bin, by picking a master key $mk^{(I)}$, and then generating h pseudorandom keys $k_j^{(I)}$. After that, it pads every bin with random elements to d elements (if needed), then it constructs polynomial $\tau_j^{(I)}(x)$ representing the elements in the bin, and evaluates each polynomial at every elements of $\vec{\boldsymbol{x}}$. This yields a vector of n y-coordinates: $\tau_j^{(I)}(x_i)$, for each bin.

2. Blinds every value $\tau_j^{(I)}(x_i)$. To do so, first it generates a pseudorandom value $z^{(I)}_{j,c_j^{(I)},i} = \mathtt{PRF}(k_j^{(I)}, i)$, where key $k_j^{(I)}$ was generated in step b.1. Then, it computes $o^{(I)}_{j,c_j^{(I)},i}$ as follows.

$$\forall i, 1 \leq i \leq n : o^{(I)}_{j,c_j^{(I)},i} = \tau_j^{(I)}(x_i) + z^{(I)}_{j,c_j^{(I)},i}.$$

Note, $c_j^{(I)}$ ($\forall j, 1 \leq j \leq h$) is a counter initially set to 0 and is incremented in the update phase when the client accesses the contents of $\mathtt{HT}_j^{(I)}$. At the end of this step, the elements in bin $\mathtt{HT}_j^{(I)}$ are represented as a vector $\vec{\boldsymbol{o}}^{(I)}_{j,c_j^{(I)}}$ containing elements $o^{(I)}_{j,c_j^{(I)},i}$.

3. Assigns a pseudorandom label to each bin.

$$\forall j, 1 \leq j \leq h : l_j^{(I)} = \mathtt{PRF}'(lk^{(I)}, j),$$

where $lk^{(I)}$ is a fresh label key.

4. Chooses a permutation key, $pk^{(I)}$, and then constructs

$$\overrightarrow{\boldsymbol{po}}^{(I)} = \pi(pk^{(I)}, \vec{\boldsymbol{o}}^{(I)}),\ \overrightarrow{\boldsymbol{pl}}^{(I)} = \pi(pk^{(I)}, \vec{\boldsymbol{l}}^{(I)}),$$

where $\vec{\boldsymbol{o}}^{(I)} = [\vec{\boldsymbol{o}}^{(I)}_{1,c_1^{(I)}}, ..., \vec{\boldsymbol{o}}^{(I)}_{h,c_h^{(I)}}]$ and $\vec{\boldsymbol{l}}^{(I)}$ contains the labels generated in step b.3.

5. Sends $\overrightarrow{\boldsymbol{po}}^{(I)}$ and $\overrightarrow{\boldsymbol{pl}}^{(I)}$ to the cloud.
6. Constructs and maintains a set $C^{(I)}$ of (small sized) counters $c_i^{(I)}$ initially set to zero. Also, it picks a fresh counter key, $ck^{(I)}$. Each counter $c_i^{(I)}$ keeps track of the number of times a bin $\mathtt{HT}_i^{(I)}$ in the outsourced hash table has been retrieved by the client for an update. The client after a number of updates can reset all the counters $c_i^{(I)}$ to zero (shortly we will show how it can do that). The counter and the key allow the client to regenerate the most recent blinding factors used to secure the outsourced data. The use of them will become clearer at the update and set intersection phases.

c. **Update.** In this phase, client I wants to insert/delete element $s^{(I)}$ to/from its outsourced dataset. It does as follows.

1. Recomputes the label for the corresponding bin by generating the bin's

index:

$$\mathtt{H}(s^{(I)}) = j,$$

and then computing the label:

$$l_j^{(I)} = \mathtt{PRF}'(lk^{(I)}, j).$$

Sends $l_j^{(I)}$ to the cloud and receives (the contents of) the corresponding bin, $\vec{\boldsymbol{o}}^{(I)}_{j,c_j^{(I)}}$.

2. Removes the blinding factors from $\vec{\boldsymbol{o}}^{(I)}_{j,c_j^{(I)}}$ as follows.
   (a) Regenerates the blinding factors for the bin.

   If $c_j^{(I)} = 0$, then $\forall i, 1 \leq i \leq n : z^{(I)}_{j,c_j^{(I)},i} = \mathtt{PRF}(k_j^{(I)}, i)$.

   If $c_j^{(I)} \neq 0$, then $\forall i, 1 \leq i \leq n : z^{(I)}_{j,c_j^{(I)},i} = \mathtt{PRF}(k_j^{(I)} + \mathtt{PRF}(ck_j^{(I)}, c_j^{(I)}), i)$, where $ck_j^{(I)} = \mathtt{PRF}(ck^{(I)}, j)$ and $c_j^{(I)}$ is the counter for bin $\mathtt{HT}_j$.
   (b) Removes the blinding factors.

   $$\forall i, 1 \leq i \leq n : y_{j,i}^{(I)} = o^{(I)}_{j,c_j^{(I)},i} - z^{(I)}_{j,c_j^{(I)},i}.$$

   At the end of this step, the client has a vector $\vec{\boldsymbol{y}}_j^{(I)}$ of the unblinded values, $y_{j,i}^{(I)}$.
3. Uses the n pairs of $(y_{j,i}^{(I)}, x_i)$ to interpolate a polynomial, $\psi_j(x)$.
4. Increments the corresponding counter by one: $c_j'^{(I)} = c_j^{(I)} + 1$.
5. Checks whether $s^{(I)}$ is a root of the polynomial, by checking: $\psi_j(s^{(I)}) \stackrel{?}{=} 0$. If the update is element **insertion**:
   (a) If $\psi_j(s^{(I)}) \neq 0$, then it does the following. First, it extracts the existing set elements from $\psi_j(x)$, i.e. it finds the roots of polynomial $\psi_j(x)$. Then, using the existing set elements (if there are any), and the element to be inserted (and random elements to pad the elements in the bin to d, if necessary) it constructs polynomial

   $$\alpha_j^{(I)}(x) = \prod_{m=1}^{d}(x - s_m'^{(I)}),$$

   where $s_m'^{(I)} \in \mathtt{HT}_j^{(I)}$, i.e. $s_m'^{(I)}$ is a set element or random value. Next, it

evaluates the polynomial at every element in $\vec{\boldsymbol{x}}$. This yields a vector, $\vec{\boldsymbol{u}}^{(I)}_{j,c'^{(I)}_j}$, of elements $u^{(I)}_{j,c'^{(I)}_j,i}$.

(b) If $\psi_j(s^{(I)}) = 0$, then it constructs vector $\vec{\boldsymbol{u}}^{(I)}_{j,c'^{(I)}_j} = \vec{\boldsymbol{y}}^{(I)}_j$, where $\vec{\boldsymbol{y}}^{(I)}_j$ was generated in step c.2b. Note, in this case the element already exists in the set, so no element insertion is required.

If the update is element **deletion**:

(a) If $\psi_j(s^{(I)}) \neq 0$, then it constructs vector $\vec{\boldsymbol{u}}^{(I)}_{j,c'^{(I)}_j} = \vec{\boldsymbol{y}}^{(I)}_j$, where $\vec{\boldsymbol{y}}^{(I)}_j$ was generated in step c.2b. Note, in this case the element does not exist in the set, so no element deletion is required.

(b) If $\psi_j(s^{(I)}) = 0$, then it does the following. First, it constructs two polynomials

$$\alpha^{(I)}_j(x) = (x - s^{(I)}),\ \beta^{(I)}_j(x) = (x - r^{(I)}),$$

where $r^{(I)}$ is a fresh random value. Next, it removes the set element and inserts the random value. This yields vector $\vec{\boldsymbol{u}}^{(I)}_{j,c'^{(I)}_j}$ of elements $u^{(I)}_{j,c'^{(I)}_j,i}$ computed as follows.

$$\forall i, 1 \leq i \leq n : u^{(I)}_{j,c'^{(I)}_j,i} = y^{(I)}_{j,i} \cdot (\alpha^{(I)}_j(x_i))^{-1} \cdot \beta^{(I)}_j(x_i).$$

6. Blinds the elements of vector $\vec{\boldsymbol{u}}^{(I)}_{j,c'^{(I)}_j}$ generated in step c.5, as follows.

(a) Computes fresh pseudorandom values.

$$\forall i, 1 \leq i \leq n : z^{(I)}_{j,c'^{(I)}_j,i} = \mathtt{PRF}(k^{(I)}_j + \mathtt{PRF}(ck^{(I)}_j, c'^{(I)}_j), i),$$

where $k^{(I)}_j = \mathtt{PRF}(mk^{(I)}, j)$ and $ck^{(I)}_j = \mathtt{PRF}(ck^{(I)}, j)$.

(b) Constructs vector $\vec{\boldsymbol{o}}^{(I)}_{j,c'^{(I)}_j}$ containing the blinded values.

$$\forall i, 1 \leq i \leq n : o^{(I)}_{j,c'^{(I)}_j,i} = u^{(I)}_{j,c'^{(I)}_j,i} + z^{(I)}_{j,c'^{(I)}_j,i}.$$

7. Sends $\vec{\boldsymbol{o}}^{(I)}_{j,c'^{(I)}_j}, l^{(I)}_j$ along with an **Update** message to the cloud who replaces the bin contents with the new ones.

d. **Set Intersection: Computation Delegation.** This phase starts when client B wants the intersection of its set and client A's set.

1. Client B regenerates vector $\vec{\boldsymbol{z}}^{(B)} = [\vec{\boldsymbol{z}}^{(B)}_{1,c_1^{(B)}}, ..., \vec{\boldsymbol{z}}^{(B)}_{h,c_h^{(B)}}]$ where the elements of vector $\vec{\boldsymbol{z}}^{(B)}_{j,c_j^{(B)}}$ were used by the client to blind its outsourced set. Then, it computes a shuffled vector: $\pi(pk^{(B)}, \vec{\boldsymbol{z}}^{(B)})$.
2. Client B blinds the shuffled vector with blinding factors that are independent of the original ordering of the $\vec{\boldsymbol{z}}^{(B)}$ elements. To do that, it picks a temporary key $tk^{(B)}$, and computes vectors $\vec{\boldsymbol{r}}^{(B)}_g$ whose elements $r^{(B)}_{g,i}$ are generated as follows.

$$\forall g, 1 \leq g \leq h,\ \forall i, 1 \leq i \leq n : r^{(B)}_{g,i} = z^{(B)}_{a,c_a^{(B)},i} + \mathtt{PRF}(tk^{(B)}_g, i),$$

   where $tk^{(B)}_g = \mathtt{PRF}(tk^{(B)}, g)$ and vector $\vec{\boldsymbol{z}}^{(B)}_a$ at position a ($1 \leq a \leq h$) in $\vec{\boldsymbol{z}}^{(B)}$ has been moved to position g after shuffling, in step d.1. Note, vectors $\vec{\boldsymbol{r}}^{(B)}_g$ allow client A to compute a vector that will be used, later on, by client B to unblind the result sent by the cloud. But, vector $\vec{\boldsymbol{r}}^{(B)}_g$ will not leak any information about client B's update pattern to client A.
3. Client B sends $lk^{(B)}, pk^{(B)}, \vec{\boldsymbol{r}}^{(B)} = [\vec{\boldsymbol{r}}^{(B)}_1, ..., \vec{\boldsymbol{r}}^{(B)}_h]$, and its id, $\mathsf{ID}^{(B)}$, to client A. It sends $tk^{(B)}$ to the cloud.
4. Upon receiving client B's message, client A computes a mapping vector that allows the cloud to match one client's bins to the other client's bin. To do so, it first generates vector $\vec{\boldsymbol{m}}_{A\rightarrow B}$ whose elements, m_i, are computed as follows.

$$\forall i, 1 \leq i \leq h : l^{(A)}_i = \mathtt{PRF}'(lk^{(A)}, i), l^{(B)}_i = \mathtt{PRF}'(lk^{(B)}, i), m_i = (l^{(A)}_i, l^{(B)}_i).$$

   Then, client A randomly permutes the elements of vector $\vec{\boldsymbol{m}}_{A\rightarrow B}$. This yields mapping vector $\overrightarrow{\boldsymbol{pm}}_{A\rightarrow B}$.
5. In this step, client A assigns n pseudorandom values $a_{g,i}$ and two polynomials $\omega^{(A)}_g(x)$ and $\omega^{(B)}_g(x)$ to each index g, where $1 \leq g \leq h$. This step is similar to steps c.4-c.6 in EO-PSI protocol and in the following we briefly outline how it can be done. It first picks a temporary key $tk^{(A)}$, then it derives three keys k_1, k_2 and k_3. Next, it uses each k_t to compute h pseudorandom values $k_{1,g}$. For each index g, client A generates a set of pseudorandom values $a_{g,i}$ by using key $k_{1,g}$. Moreover, it uses keys $k_{2,g}$ and $k_{3,g}$ to generate two degree d pseudorandom polynomials $\omega^{(A)}_g(x)$ and $\omega^{(B)}_g(x)$ for that index.

6. Client A regenerates vector $\vec{\boldsymbol{z}}^{(A)} = [\vec{\boldsymbol{z}}^{(A)}_{1,c_1^{(A)}}, ..., \vec{\boldsymbol{z}}^{(A)}_{h,c_h^{(A)}}]$ where the elements of each vector $\vec{\boldsymbol{z}}^{(A)}_{j,c_j^{(A)}}$ were used by the client to blind its outsourced set. After that, it computes shuffled vector $\pi(pk^{(A)}, \vec{\boldsymbol{z}}^{(A)})$.
7. Client A multiplies each element at position g ($1 \leq g \leq h$) in $\pi(pk^{(A)}, \vec{\boldsymbol{z}}^{(A)})$ and $\vec{\boldsymbol{r}}^{(B)}$, by the pseudorandom polynomials $\omega_g^{(A)}(x)$ and $\omega_g^{(B)}(x)$, respectively. To do so, it generates $\vec{\boldsymbol{v}}^{(I)} = [\vec{\boldsymbol{v}}_1^{(I)}, ..., \vec{\boldsymbol{v}}_h^{(I)}]$ where the elements of each vector $\vec{\boldsymbol{v}}_g^{(I)}$, (where $I \in \{A, B\}$) are computed as follows. $\forall g, 1 \leq g \leq h$ and $\forall i, 1 \leq i \leq n$:

$$v_{g,i}^{(A)} = \omega_g^{(A)}(x_i) \cdot z^{(A)}_{j,c_j^{(A)},i}$$
$$v_{g,i}^{(B)} = \omega_g^{(B)}(x_i) \cdot r_{g,i}^{(B)} = \omega_g^{(B)}(x_i) \cdot (z^{(B)}_{a,c_a^{(B)},i} + \mathtt{PRF}(tk_g^{(B)}, i)),$$

   where vector $\vec{\boldsymbol{z}}_j^{(A)}$ at position j ($1 \leq j \leq h$) in $\vec{\boldsymbol{z}}^{(A)}$ has been moved to position g after shuffling, in step d.6.
8. Given keys $pk^{(A)}$ and $pk^{(B)}$, for each vector $\vec{\boldsymbol{v}}_g^{(A)} \in \vec{\boldsymbol{v}}^{(A)}$ it finds vector $\vec{\boldsymbol{v}}_e^{(B)} \in \vec{\boldsymbol{v}}^{(B)}$ such that $v_{g,i}^{(A)} = \omega_g^{(A)}(x_i) \cdot z^{(A)}_{j,c_j^{(A)},i}$ and $v_{e,i}^{(B)} = \omega_e^{(B)}(x_i) \cdot r_{e,i}^{(B)} = \omega_e^{(B)}(x_i) \cdot (z^{(B)}_{j,c_j^{(B)},i} + \mathtt{PRF}(tk_e^{(B)}, i))$, i.e. blinding factors $z^{(A)}_{j,c_j^{(A)},i}$ and $z^{(B)}_{j,c_j^{(B)},i}$ belong to the same index j. Next, it computes vectors $\vec{\boldsymbol{q}}_e$ as follows. $\forall i, 1 \leq i \leq n$:

$$\begin{aligned} q_{e,i} &= v_{g,i}^{(A)} + v_{e,i}^{(B)} + a_{g,i} \\ &= \omega_g^{(A)}(x_i) \cdot z^{(A)}_{j,c_j^{(A)},i} + \omega_e^{(B)}(x_i) \cdot (z^{(B)}_{j,c_j^{(B)},i} + \mathtt{PRF}(tk_e^{(B)}, i)) + a_{g,i}. \end{aligned}$$

   Note that in the above, values $z^{(A)}_{j,c_j^{(A)},i}$ and $z^{(B)}_{j,c_j^{(B)},i}$ belong to the same bin: $\mathtt{HT}_j$, before they were permuted. Moreover, vectors $\vec{\boldsymbol{q}}_e$ will allow client B to remove the blinding factors from the cloud's response without learning the pseudorandom polynomials.
9. Client A sends $\vec{\boldsymbol{q}} = [\vec{\boldsymbol{q}}_1, ..., \vec{\boldsymbol{q}}_h]$ to client B. Also, it sends $tk^{(A)}$, $\mathsf{ID}^{(B)}$, $\mathsf{ID}^{(A)}$, $\overrightarrow{\boldsymbol{pm}}_{A \to B}$ along with a **Compute** message to the cloud.

e. **Set Intersection: Cloud-Side Result Computation.**

1. Given key $tk^{(A)}$, the cloud derives three keys $k_{1,g}$, $k_{2,g}$ and $k_{3,g}$ for each index g, where $1 \leq g \leq h$. Then, it uses the keys to regenerate the set of pseudorandom values $a_{g,i}$ ($\forall i, 1 \leq i \leq n$) and two pseudorandom polynomials $\omega_g^{(A)}(x)$ and $\omega_g^{(B)}(x)$ for each index g, where $1 \leq g \leq h$ (similar to

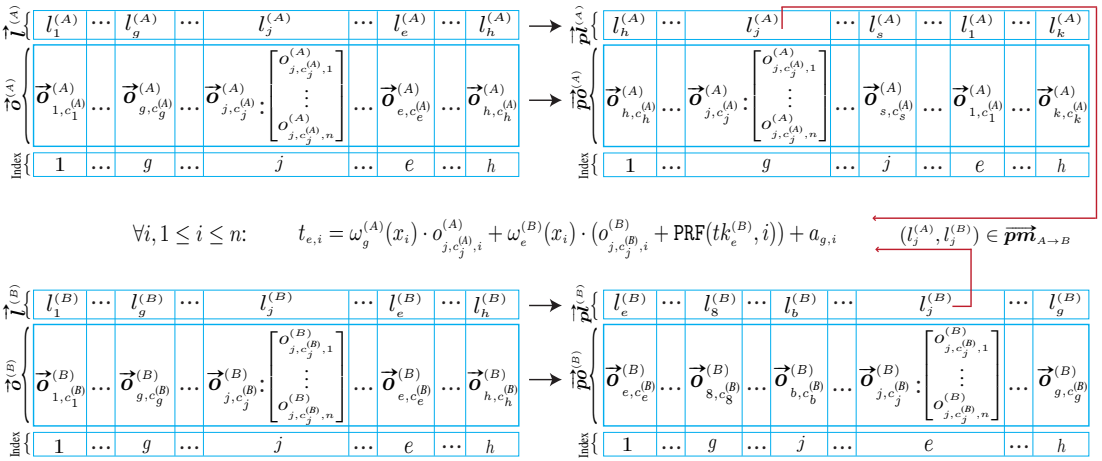

Figure 5.1: Cloud-Side Computation, step e.2: given the permutation map and the clients' permuted datasets, the cloud matches one client's bins to the other client's bins, such that matched bins had the same index before they were shuffled. Note, in the figure, the left hand-side tables are not given to the cloud.

step d.5).

2. To generate the computation result, the cloud first uses the mapping vector, $\overrightarrow{pm}_{A\to B}$, that allows it for each index g in $\overrightarrow{po}^{(A)}$ to find the corresponding index e in $\overrightarrow{po}^{(B)}$, where $1 \le g, e \le h$. As it is depicted in Fig 5.1, if the cloud behaves honestly, then $\overrightarrow{po}^{(A)}_{g} = \vec{o}^{(A)}_{j,c_j^{(A)}}$ and $\overrightarrow{po}^{(B)}_{e} = \vec{o}^{(B)}_{j,c_j^{(B)}}$ (for some $j, 1 \le j \le h$). Next, it computes result vectors $\vec{t}_e$.
$\forall g, 1 \le g \le h, \forall i, 1 \le i \le n$:

$$t_{e,i} = \omega_g^{(A)}(x_i) \cdot o^{(A)}_{j,c_j^{(A)},i} + \omega_e^{(B)}(x_i) \cdot (o^{(B)}_{j,c_j^{(B)},i} + \mathtt{PRF}(tk_e^{(B)}, i)) + a_{g,i},$$

where $tk_e^{(B)} = \mathtt{PRF}(tk^{(B)}, e)$, $o^{(A)}_{j,c_j^{(A)},i} \in \vec{o}^{(A)}_{j,c_j^{(A)}}$, $o^{(B)}_{j,c_j^{(B)},i} \in \vec{o}^{(B)}_{j,c_j^{(B)}}$, and $tk^{(B)}$ was given to the cloud in step d.3.
3. The cloud sends $\vec{t} = [\vec{t}_1, ..., \vec{t}_h]$ to client B.

f. **Set Intersection: Client-Side Result Retrieval**. In this phase, client B wants to find out the computation result, so it operates as follows.

1. Removes the blinding factors from each vector $\vec{t}_e$ ($\forall e, 1 \le e \le h, \vec{t}_e \in \vec{t}$) by using the corresponding vector $\vec{q}_e$ (given by client A in step d.9). This yields vectors $\vec{f}_e$ whose elements are computed follows.

$\forall e, 1 \leq e \leq h,\ \forall i, 1 \leq i \leq n$:

$$f_{e,i} = t_{e,i} - q_{e,i} = \omega_g^{(A)}(x_i) \cdot u^{(A)}_{j,c_j^{(A)},i} + \omega_e^{(B)}(x_i) \cdot u^{(B)}_{j,c_j^{(B)},i},$$

where $1 \leq g, j \leq h$.

2. Interpolates polynomial $\phi_e(x)$ ($\forall e, 1 \leq e \leq h$), given vectors $\vec{\boldsymbol{f}}_e$ and $\vec{\boldsymbol{x}}$.
3. Extracts the roots of each polynomial and considers the union of the (valid) roots as the intersection of the sets.

Remark 1: The counter set, $C^{(I)}$, helps client I to regenerate the blinding factors. The client can always *reset* its counter (i.e. it sets all its counters $c_j^{(I)}$ to zero). To do so, it regenerates the vector $\vec{\boldsymbol{z}}^{(I)} = [\vec{\boldsymbol{z}}^{(I)}_{1,c_1^{(I)}}, ..., \vec{\boldsymbol{z}}^{(I)}_{h,c_h^{(I)}}]$ where the elements of each vector $\vec{\boldsymbol{z}}^{(I)}_{j,c_j^{(I)}}$ were used by the client to blind its outsourced set (i.e. $\forall j, i,\ 1 \leq j \leq h, 1 \leq i \leq n : z^{(I)}_{j,c_j^{(I)},i} \in \vec{\boldsymbol{z}}^{(I)}_{j,c_j^{(I)}}$). Also, it picks a fresh master key $mk'^{(I)}$ and generates vector $\vec{\boldsymbol{z}}'^{(I)} = [\vec{\boldsymbol{z}}'^{(I)}_{1,c_1^{(I)}}, ..., \vec{\boldsymbol{z}}'^{(I)}_{h,c_h^{(I)}}]$ such that $\forall j, i,\ 1 \leq j \leq h, 1 \leq i \leq n : z'^{(I)}_{j,c_j^{(I)},i} \in \vec{\boldsymbol{z}}'^{(I)}_{j,c_j^{(I)}}$ and fresh pseudorandom values $z'^{(I)}_{j,c_j^{(I)},i}$ are generated the same way as the ones in step b.2, with the difference that here the master key $mk'^{(I)}$ is used. After that, it computes vector $\vec{\boldsymbol{z}}''^{(I)} = [\vec{\boldsymbol{z}}''^{(I)}_{1,c_1^{(I)}}, ..., \vec{\boldsymbol{z}}''^{(I)}_{h,c_h^{(I)}}]$ where values $z''^{(I)}_{j,c_j^{(I)},i}$ are computed as follows.

$$\forall j, 1 \leq j \leq h,\ \forall i, 1 \leq i : z''^{(I)}_{j,c_j^{(I)},i} = z'^{(I)}_{j,c_j^{(I)},i} - z^{(I)}_{j,c_j^{(I)},i}.$$

It sends $\vec{\boldsymbol{b}} = \pi(pk^{(I)}, \vec{\boldsymbol{z}}''^{(I)})$ to the cloud and asks it to sum (component-wise) each elements in the vector with the elements in the outsourced dataset. The cloud performs as below.

$$\forall g, 1 \leq g \leq h, \forall i, 1 \leq i \leq n : o_{g,i} + b_{g,i} = u^{(I)}_{g,c_g^{(I)},i} + z'^{(I)}_{g,c_g^{(I)},i}.$$

As it is evident above, the old blinding factors are replaced with the fresh ones. Finally, the client needs to keep the new master key, $mk'^{(I)}$, and discard the old one, $mk^{(I)}$, and set its entire counter to zero. It should be noted that, although the number of counters (i.e. h in total) is linear to hash table length, each counter bit-size is independent of (and smaller than) the bit-size of each element in the hash table.

Remark 2: In the update phase, the time taken to insert, delete or check element membership (but doing no element update) varies. So, by timing the client's response, the cloud might be able to learn what kind of updates the client performed. To mitigate such attack, the client can delay its response and send the message after a fixed time.

Remark 3: Since the client, in the update phase, refreshes the blinding factors each time it retrieves a bin, the cloud cannot figure out whether it inserts or deletes an element. Also, as the original index of each bin is hidden from the cloud, it cannot learn which element has been updated in the bin.

Remark 4: Since the clients do not interact with each other at setup and they permute their datasets independently, the permuted datasets do not have the same order. For instance, as Fig 5.1 shows, client A's and B's bin at position j may reside at position g and e, respectively, after the permutation. In order for the cloud to compute the correct result, it needs to combine the right bins that had the same index before they were permuted, e.g. both belong to index j. To this end, client A provides the permutation map, $\overrightarrow{pm}_{A\rightarrow B}$, that enables the cloud to match the correct bins and perform computation on them.

Remark 5: In the protocol, the client does not need to recompute the hash table as long as each client's set cardinality remains smaller than the upper bound: c, irrespective of the number of updates performed on its outsourced data. Note that only in the case where a bin exceeds its capacity the client would need to recompute the hash table. However, as shown in section 2.7, given the upper bound c, we can set the hash table parameters (i.e. total number of bins and bin capacity) in such a way that a bin overflows only with negligibly small probability. Thus, if (at any point in time) the total number of the client's set elements in the hash table remains smaller than c, then the hash table does not need to be restructured.

5.2.4 Extensions

In this section, we first show how the two-client UEO-PSI protocol can be extended to a multi-client one. Recall that if client B runs a two-client PSI protocol multiple times with different clients it would learn more information than the information it can learn if it engages in a multi-client PSI protocol (see section 4.2.3.1 for a detailed dis-

cussion). Therefore, it is vital that a PSI protocol supports multiple clients. Then, we outline how we can reduce the storage space that a client who authorizes the computation requires, in UEO-PSI .

5.2.4.1 Multi-client UEO-PSI

In the following, we explain how the two-client UEO-PSI can be modified to support *m*-client UEO-PSI, where $m > 2$. Here, we denote the result recipient by client B and the other clients by A_l, where $1 \leq l \leq y$, $y = m - 1$, and m is the total number of clients.

Similar to the two-client case, here each client A_l sends to the cloud a temporary key $tk^{(A_l)}$. To generate the computation result, the cloud uses each $\overrightarrow{\boldsymbol{pm}}_{A_l \to B}$ that allows it for each index e in $\overrightarrow{\boldsymbol{po}}^{(B)}$ to find an index $g^{(A_l)}$ in $\overrightarrow{\boldsymbol{po}}^{(A_l)}$, where $1 \leq e, g^{(A_l)} \leq h$. The cloud uses each $tk^{(A_l)}$ to generate values $a^{(A_l)}_{g^{(A_l)},i}$ ($\forall i, 1 \leq i \leq n$) for each index $g^{(A_l)}$. Additionally, using key $tk^{(A_l)}$, it generates two pseudorandom polynomials $\omega^{(A_l)}_{g^{(A_l)}}(x)$ and $\omega^{(B_l)}_{e}(x)$ for each index $g^{(A_l)}$ and e, respectively. Next, it computes result vectors $\vec{\boldsymbol{t}}_e$ as follows. $\forall e, 1 \leq e \leq h,\ \forall i, 1 \leq i \leq n$:

$$t_{e,i} = \omega^{(B)}_{e}(x_i) \cdot (o^{(B)}_{e,i} + \mathtt{PRF}(tk^{(B)}_{e}, i)) + \sum_{l=1}^{y} a^{(A_l)}_{g^{(A_l)},i} + \sum_{l=1}^{y} \omega^{(A_l)}_{g^{(A_l)}}(x_i) \cdot o^{(A_l)}_{g^{(A_l)},i},$$

where $\omega^{(B)}_{e}(x) = \sum_{l=1}^{y} \omega^{(B_l)}_{e}(x)$, $o^{(B)}_{e,i} \in \overrightarrow{\boldsymbol{po}}^{(B)}$, $o^{(A_l)}_{g^{(A_l)},i} \in \overrightarrow{\boldsymbol{po}}^{(A_l)}$, $o^{(B)}_{e,i} = u^{(B)}_{j,c^{(B)}_j,i} + z^{(B)}_{j,c^{(B)}_j,i}$ and $o^{(A_l)}_{g^{(A_l)},i} = u^{(A_l)}_{j,c^{(A_l)}_j,i} + z^{(A_l)}_{j,c^{(A_l)}_j,i}$ for some j ($1 \leq j \leq h$). So, client B in step f.1 removes the blinding factors from each vector $\vec{\boldsymbol{t}}_e$ as follows. $\forall e, 1 \leq e \leq h,\ \forall i, 1 \leq i \leq n$:

$$f_{e,i} = t_{e,i} - \sum_{l=1}^{y} q^{(A_l)}_{e,i} = \omega^{(B)}_{e}(x_i) \cdot u^{(B)}_{j,c^{(B)}_j,i}(x_i) + \sum_{l=1}^{y} \omega^{(A_l)}_{g^{(A_l)}}(x_i) \cdot u^{(A_l)}_{j,c^{(A_l)}_j,i}(x_i).$$

It should be noted that in the multi-client case, even if client B colludes with $y-1$ clients, it cannot learn anything about the non-colluding client's set elements. Because, as it is shown in [75], polynomial $\omega^{(B)}_{e}(x) = \sum_{l=1}^{y} \omega^{(B_l)}_{e}(x)$ is always a uniformly random polynomial even if only one of the polynomials $\omega^{(B_l)}_{e}(x)$ is a uniformly random polynomial unknown to client B.

Remark: In our multi-client protocol, the communication and computation complexity of those clients who authorize the computation (i.e. client A_l) is independent of the

number of clients in the protocol. So, the computation and communication complexity of client A in the two-client case is similar to client A_l's in the multi-client case.

5.2.4.2 Reducing Authorizer's Required Storage Space

Now, we show how, with minor adjustment, we can reduce the storage space that client A who authorizes the computation requires. Here, for the sake of simplicity we consider the two clients case, but the modification can directly be applied to the multi-client case. The main idea is that client B sends one bin at a time to client A when they want to delegate the computation. The client A can work on only one bin, and generate the corresponding messages to be sent to client B and the cloud.

In step d.1, client B for each vector $\vec{\boldsymbol{z}}^{(B)}_{j,c_j^{(B)}}$, where $\vec{\boldsymbol{z}}^{(B)}_{j,c_j^{(B)}} \in \vec{\boldsymbol{z}}^{(B)}$, finds index e that is the index of vector $\vec{\boldsymbol{z}}^{(B)}_{j,c_j^{(B)}}$ after $\vec{\boldsymbol{z}}^{(B)}$ is permuted. Next, in step d.2, for each $\vec{\boldsymbol{z}}^{(B)}_{j,c_j^{(B)}}$ it computes $\vec{\boldsymbol{r}}^{(B)}_e$ whose elements are computed as follows.

$$\forall i, 1 \leq i \leq n : r^{(B)}_{e,i} = z^{(B)}_{j,c_j^{(B)},i} + \mathtt{PRF}(tk^{(B)}_e, i).$$

Then, it constructs a vector of h tuples $(\vec{\boldsymbol{r}}^{(B)}_e, j, e)$ (where $1 \leq e, j \leq h$), and permutes the vector. Client B, in step d.3 sends $lk^{(B)}, pk^{(B)}$ and $\mathsf{ID}^{(B)}$ to client A. Then, it sends each tuple in the permuted vector to client A. Therefore, in this process, instead of sending the entire vector, $\vec{\boldsymbol{r}}^{(B)} = [\vec{\boldsymbol{r}}^{(B)}_1, ..., \vec{\boldsymbol{r}}^{(B)}_h]$, it sends a tuple (including one of the vectors in $\vec{\boldsymbol{r}}^{(B)}$) at a time. Client A finds index g the position where a value at index j (after permutation) would move to, when client A's permutation key is used. Then, it regenerates vector $\vec{\boldsymbol{z}}^{(A)}_{j,c_A^{(A)}}$. After that, in step d.7, it computes values $v^{(A)}_{g,i}$ and $v^{(B)}_{e,i}$ as follows.

$\forall i, 1 \leq i \leq n :$

$$v^{(A)}_{g,i} = \omega^{(A)}_g(x_i) \cdot z^{(A)}_{j,c_j^{(A)},i} + a_{g,i},$$
$$v^{(B)}_{e,i} = \omega^{(B)}_e(x_i) \cdot r^{(B)}_{e,i} = \omega^{(B)}_e(x_i) \cdot (z^{(B)}_{j,c_j^{(B)},i} + \mathtt{PRF}(tk^{(B)}_e, i)).$$

Also, client A in step d.8, computes vector $\vec{\boldsymbol{q}}_e$ whose elements are computed as below.

$$\forall i, 1 \leq i \leq n : q_{e,i} = v^{(A)}_{g,i} + v^{(B)}_{e,i}.$$

It sends $\vec{\boldsymbol{q}}_e$ to client B. Finally, client A computes $(l^{(A)}_j, l^{(B)}_j)$ and sends it to the

cloud. Note that client B sends the tuples in a random order to client A, therefore the tuples that the cloud receives (i.e. $(l_j^{(A)}, l_j^{(B)})$) are also in a random order.

Hence, the above modification reduces the required storage space at the client-side from $O(hd)$ to $O(d)$.

5.3 Security Definition

In this section, we provide the security definition for our protocol. For UEO-PSI, similar to O-PSI and EO-PSI, we consider a static semi-honest adversary who controls one of the parties at a time (i.e. non-colluding semi-honest adversaries) [52, 67]. However, the protocol's security definition is more involved than the security definition provided for O-PSI and EO-PSI. Here, for the sake of simplicity and without loss of generality we assume there are three parties, cloud C and clients A and B, engaging in the protocol, where client A authorizes the computation and client B is interested in the result. Here also, similar to O-PSI and EO-PSI, we assume there exists an infrastructure, e.g. PKI, via which client A can authenticate, and then authorize the other client. To retain efficiency, we allow some information to be leaked to the cloud. This approach is widely used in the literature, e.g. [115, 69, 59, 110]. We say the protocol is secure as long as the cloud does not learn anything about the computation inputs and output despite this leakage and client does not learn anything beyond the intersection about the other client's set elements.

The leakage includes the query and access patterns. The protocol involves two types of operations: dataset update: Upd, and delegated PSI computation: D-PSI. We define the query pattern $\vec{\boldsymbol{T}}$ to be a vector of strings such that $|\vec{\boldsymbol{T}}| = poly(\lambda) = \Upsilon$, where λ is a security parameter, the vector element is defined as $T_i \in \{\mathsf{Upd}_t^{(I)}, \mathsf{PSI\text{-}Com}\}$, $1 \leq t \leq \mathbf{p}$, value $\mathbf{p}$ is the total number of update queries issued by each client and $I \in \{A, B\}$. So, $\vec{\boldsymbol{T}}$ contains clients A and B request strings for update as well as request string for the computation issued by client A. Intuitively, in the protocol clients need to ask explicitly the cloud for a certain operation, so the cloud learns, whether the client's i^{th} query is for update or PSI computation, i.e. the query pattern.

The access pattern is more complex. In our protocol, a dataset is encoded as a hash table and each bin of the hash table is tagged with a unique *deterministic* label, in a form of pseudorandom binary string of length l, where l is a security parameter. Without loss of generality, we assume in each update query only one element is inserted/removed to/from the set. Each update query always involves the client sending

a label to the cloud, receiving the bin (tagged with the label), and then rewriting the contents of the bin. Therefore, in the update process the cloud (from the sequence of update queries sent by clients) can see what part of the outsourced data is updated. Nevertheless, it cannot associate that part with the sets elements (or the computation output). In particular, given a sequence of client's queries, the cloud can see that a bin is updated but it does not learn the original address of the bin (or the hash value of the set elements mapped to the bin) because the bins are pseudorandomly permuted. Also, 9it cannot figure out whether the update is an insertion or deletion.

Definition 14. *(Access Pattern) Let* $\mathtt{HT}^{(I)}$ *be client* I*'s hash table containing* h *bins where each bin,* $\mathtt{HT}_i^{(I)}$*, is tagged with a unique label* $l_i^{(I)}$*. Moreover, let* $\vec{\boldsymbol{o}}'^{(I)} = \pi(k^{(I)}, \vec{\boldsymbol{o}}^{(I)})$ *be shuffled data, where* $k^{(I)}$ *is a secret key and* $\vec{\boldsymbol{o}}^{(I)} = [(\mathtt{HT}_1^{(I)}, l_1^{(I)}), ..., (\mathtt{HT}_h^{(I)}, l_h^{(I)})]$*. The access pattern, for the shuffled data, induced by* $\boldsymbol{p}$*-update queries is a symmetric binary matrix* $\mathcal{M}^{(I)}$ *such that for* $1 \leq i, j \leq \boldsymbol{p}$*, the element in the* i^{th} *row and* j^{th} *column is* 1*,* $\mathcal{M}_{i,j}^{(I)} = 1$*, if the* i^{th} *query equals* j^{th} *query (i.e. both queries have the same label) and* 0 *otherwise (where* $I \in \{A, B\}$*).*

Similar to O-PSI and EO-PSI, here the delegated PSI computation involves the three parties. The three-party delegated PSI protocol, D-PSI, computes a function that maps the inputs to some outputs. We define this function as $F : \Lambda \times 2^{\mathfrak{U}} \times 2^{\mathfrak{U}} \rightarrow \Lambda \times \Lambda \times f_{\cap}$, where Λ denotes the empty string, $2^{\mathfrak{U}}$ denotes the powerset of the set universe and $f_{\cap}$ denotes the set intersection function. For every tuple of inputs Λ, $S^{(A)}$ and $S^{(B)}$ belonging to C, A and B respectively, the function outputs nothing to C and A, and outputs $f_{\cap}(S^{(A)}, S^{(B)}) = S^{(A)} \cap S^{(B)}$ to B. Note, for PSI computation we do not have any data leakage. In the security model, we define the leakage function as $\mathsf{leak}(\vec{\boldsymbol{o}}'^{(I)}, \overrightarrow{\boldsymbol{upd}}^{(I)}) = [\mathcal{M}^{(I)}, \vec{\boldsymbol{T}}]$ that captures precisely what is being leaked by the update operation. The function takes as input clients' outsourced data, and **p** update queries. It outputs two different types of information; namely, the access pattern (i.e. the matrix) and the query pattern for clients A and B.

We say the protocol is secure if (1) nothing beyond the leakage is revealed to the cloud; (2) whatever can be computed by a client in the protocol can be obtained from its input and output only. This is formalized by the simulation paradigm. We require a client's *view* during an execution of D-PSI to be simulatable given its input and output. As one client's update pattern is not leaked to the other client, the scheme is secure as long as the PSI computation result does not leak any information to the client. Also, we require that the cloud's view can be simulated given the leakage. The party I's view

on input tuple (x, y, z) is denoted by $\mathsf{VIEW}_I^{\mathbf{t}}(x, y, z)$ (as it is defined in section 2.9), and if $I \in \{A, B\}$ then $\mathbf{t}$: D-PSI; if $I = C$, then $\mathbf{t}$: UEO-PSI.

Definition 15. *Let UEO-PSI = (Upd, D-PSI) be the scheme defined above. We say that UEO-PSI is secure at the client side in the presence of static semi-honest adversaries if there exist probabilistic polynomial-time algorithms SIM_A and SIM_B that given the input and output of a client, can simulate a view that is computationally indistinguishable from the client's view in the protocol:*

$$\{\mathsf{SIM}_A(S^{(A)}, \Lambda)\}_{S^{(A)},S^{(B)}} \stackrel{c}{\equiv} \{\mathsf{VIEW}_A^{\text{D-PSI}}(\Lambda, S^{(A)}, S^{(B)})\}_{S^{(A)},S^{(B)}}$$

$$\{\mathsf{SIM}_B(S^{(B)}, f_\cap(S^{(A)}, S^{(B)}))\}_{S^{(A)},S^{(B)}} \stackrel{c}{\equiv} \{\mathsf{VIEW}_B^{\text{D-PSI}}(\Lambda, S^{(A)}, S^{(B)})\}_{S^{(A)},S^{(B)}},$$

where D-PSI was defined above. Also, UEO-PSI is secure, at the cloud side, in the presence of static semi-honest adversaries if there exists probabilistic polynomial-time algorithm SIM_C that given the leakage function, can simulate a view that is computationally indistinguishable from the cloud's view in the protocol:

$$\{\mathsf{SIM}_C^{\text{leak}()}(\Lambda, \Lambda)\}_{S^{(A)},S^{(B)}} \stackrel{c}{\equiv} \{\mathsf{VIEW}_C^{\text{UEO-PSI}}(\Lambda, S^{(A)}, S^{(B)})\}_{S^{(A)},S^{(B)}}$$

5.4 UEO-PSI Security Proof

In this section, we sketch the protocol's security proof in the presence of static semi-honest adversaries. We conduct the security analysis for the three cases where one of the parties is corrupted at a time.

Theorem 4. *If* $\mathtt{PRF}$ *and* $\mathtt{PRF}'$ *are collision-resistant pseudorandom functions, and π is a pseudorandom permutation, then UEO-PSI protocol is secure in the presence of a semi-honest adversary.*

Proof. We will prove the theorem by considering in turn the case where each of the parties has been corrupted. In each case, we invoke the simulator with the corresponding party's input and output. Our focus is on the case where party A wants to engage in the computation of the intersection, i.e. it authorizes the computation. If party A does not want to proceed with the protocol, the views can be simulated in the same way up to the point where the execution stops.

Case 1: Corrupted Cloud. In this case, we show that given the leakage function we can construct a simulator SIM_C that can produce a view computationally indistinguishable from the one in the real model. In the real execution, the cloud's view is:

$$\mathsf{VIEW}_C^{\mathsf{UEO\text{-}PSI}}(\Lambda, S^{(A)}, S^{(B)}) = \{\Lambda, r_C, \overrightarrow{\boldsymbol{po}}^{(A)}, \overrightarrow{\boldsymbol{pl}}^{(A)}, \overrightarrow{\boldsymbol{po}}^{(B)}, \overrightarrow{\boldsymbol{pl}}^{(B)}, (Q_1, ..., Q_\Upsilon), \Lambda\}.$$

In the above view, r_C is the outcome of internal random coins of the cloud, ($\forall I, I \in \{A, B\}$) $\overrightarrow{\boldsymbol{po}}^{(I)}$ are the permuted hash tables containing the clients' blinded datasets and $\overrightarrow{\boldsymbol{pl}}^{(I)}$ are the permuted labels. Moreover, if the query is for PSI delegation then

$$Q_i = \{tk^{(B)}, tk^{(A)}, \mathsf{ID}^{(A)}, \mathsf{ID}^{(B)}, \overrightarrow{\boldsymbol{pm}}_{A\rightarrow B}, \mathsf{Compute}\},$$

otherwise (if the client I's query is for update),

$$Q_i = \{\overrightarrow{\boldsymbol{o}}^{(I)}_{j,c_j^{(I)}}, l_j^{(I)}, \mathsf{Update}\},$$

for some $j, 1 \leq j \leq h$.

Now we construct the simulator SIM_C in the ideal model which executes as follows.

1. Creates an empty view and appends Λ and uniformly random coin r'_C to it.
2. Uses the public parameters and the hash function to construct two hash tables $\mathtt{HT}'^{(A)}$ and $\mathtt{HT}'^{(B)}$. It fills each bin of the hash tables with n uniformly random values picked from the field $\mathbb{F}_p$. So each bin $\mathtt{HT}'^{(I)}_j$ ($\forall I, I \in \{A, B\}$) contains a vector $\overrightarrow{\boldsymbol{o}}'^{(I)}_j$ of n random values.
3. Assigns a pseudorandom label to each bin $\mathtt{HT}'^{(I)}_j$ ($\forall I, I \in \{A, B\}$). To do so, it picks fresh label-keys, $lk'^{(I)}$, and computes the labels as $\forall I, I \in \{A, B\}$ and $\forall j, 1 \leq j \leq h : l'^{(I)}_j = \mathtt{PRF}'(lk'^{(I)}, j)$.
4. Constructs two vectors of the form $[(\overrightarrow{\boldsymbol{o}}'^{(I)}_1, l'^{(I)}_1), ..., (\overrightarrow{\boldsymbol{o}}'^{(I)}_h, l'^{(I)}_h)]$ for each I, where $I \in \{A, B\}$. Then, it randomly permutes each vector. Next, it inserts each element $\overrightarrow{\boldsymbol{o}}'^{(I)}_g$ ($\forall g, 1 \leq g \leq h$) of the permuted vector into $\overrightarrow{\boldsymbol{po}}'^{(I)}$. Also, it inserts each element $l'^{(I)}_g$ of the permuted vector into $\overrightarrow{\boldsymbol{pl}}'^{(I)}$. Therefore, it has constructed four vectors: $\overrightarrow{\boldsymbol{po}}'^{(A)}$, $\overrightarrow{\boldsymbol{pl}}'^{(A)}$, $\overrightarrow{\boldsymbol{po}}'^{(B)}$ and $\overrightarrow{\boldsymbol{pl}}'^{(B)}$. It appends the four vectors to the view.
5. Given the leakage function $[\mathcal{M}^{(I)}, \overrightarrow{\boldsymbol{T}}]$, it first uses each matrix $\mathcal{M}^{(I)}$ to construct the corresponding vector $\overrightarrow{\boldsymbol{v}}^{(I)}$ that will contain a set of labels $l'^{(I)} \in \overrightarrow{\boldsymbol{l}}'^{(I)}$, and has the same access pattern as the one indicated by the matrix. In order for it

to generate this vector, it first constructs vector $\vec{\boldsymbol{v}}^{(I)}$ of zeros, where $|\vec{\boldsymbol{v}}^{(I)}| = \mathbf{p}$. Then, for every row i ($1 \leq i \leq \mathbf{p}$) of the matrix $\mathcal{M}^{(I)}$, it performs the following:

(a) If there exists at least one element set to 1 in the row and if the i^{th} element in the vector $\vec{\boldsymbol{v}}^{(I)}$ is zero, then it finds a set G such that $\forall g \in G : \mathcal{M}^{(I)}_{i,g} = 1$. Next, it picks a label $l'^{(I)} \in \vec{\boldsymbol{l}}'^{(I)}$ and inserts it into all i^{th}, g^{th} positions of the vector $\vec{\boldsymbol{v}}^{(I)}$. The label must be distinct from the ones used for the previous rows i', where $i' < i$. Otherwise, if the i^{th} element in the vector is non-zero, it moves on to the next row.

(b) If all the elements in the row are zero and if the i^{th} element in the vector $\vec{\boldsymbol{v}}^{(I)}$ is zero, then it picks a label $l'^{(I)} \in \vec{\boldsymbol{l}}'^{(I)}$ and inserts it at position i^{th} of the vector, where the label is distinct from the ones used for the previous rows i', where $i' < i$. Otherwise, if the i^{th} element in the vector is non-zero, it moves on to the next row.

6. Uses $\vec{\boldsymbol{T}}$ and checks T_i ($\forall i, 1 \leq i \leq \Upsilon$) as follows:

(a) if $T_i = \mathsf{PSI\text{-}Com}$, then sets:

$$Q'_i = \{tk'^{(B)}, tk'^{(A)}, \mathbf{ID}^{(A)}, \mathbf{ID}^{(B)}, \overrightarrow{\boldsymbol{pm}}'_{A\rightarrow B}, \mathbf{Compute}\},$$

where $\overrightarrow{\boldsymbol{pm}}'_{A\rightarrow B}$ contains tuples $(l'^{(A)}_i, l'^{(B)}_i)$ randomly permuted. Also, $tk'^{(B)}$ and $tk'^{(A)}$ are fresh random keys, and labels $l'^{(I)}_i$ were generated step 3. Next, it appends Q'_i to the view.

(b) if $T_i = \mathsf{Upd}^{(I)}_t$, then sets:

$$Q'_i = \{\vec{\boldsymbol{o}}''^{(I)}, l'^{(I)}_t, \mathbf{Update}\},$$

where $\vec{\boldsymbol{o}}''^{(I)}$ contains n uniformly random values picked from the field $\mathbb{F}_p$ and $l'^{(I)}_t$ is the t^{th} element in the vector $\vec{\boldsymbol{v}}^{(I)}$. After that, it appends Q'_i to the view.

7. Appends Λ to its view and outputs the view.

We are ready now to show why the two views are indistinguishable. In both views, the input and output parts (i.e. Λ) are identical and the random coins are both uniformly random, and so they are indistinguishable. Each vector $\overrightarrow{\boldsymbol{po}}^{(I)}_i \in \overrightarrow{\boldsymbol{po}}^{(I)}$, ($\forall i, 1 \leq i \leq h$ and $\forall I, I \in \{A, B\}$), contains n values blinded with pseudorandom values (that are the outputs of a pseudorandom function), also each vector $\overrightarrow{\boldsymbol{po}}'^{(I)}_i \in \overrightarrow{\boldsymbol{po}}'^{(I)}$ contains n

random values sampled uniformly from the same field. Since the blinded values and random values are not distinguishable, the elements of vectors $\overrightarrow{\boldsymbol{po}}^{(I)}$ and $\overrightarrow{\boldsymbol{po}}'^{(I)}$ are indistinguishable too. Also, labels $l_i^{(I)} \in \overrightarrow{\boldsymbol{pl}}^{(I)}$ and $l_i'^{(I)} \in \overrightarrow{\boldsymbol{pl}}'^{(I)}$, ($\forall i, 1 \leq i \leq h$ and $\forall I, I \in \{A, B\}$), are the outputs of a pseudorandom function and they are indistinguishable. Therefore, the elements of vectors $\overrightarrow{\boldsymbol{pl}}^{(I)}$ and $\overrightarrow{\boldsymbol{pl}}'^{(I)}$ are indistinguishable. Furthermore, since a pseudorandom permutation is indistinguishable from a random permutation, permuted vectors $\overrightarrow{\boldsymbol{po}}^{(I)}$ and $\overrightarrow{\boldsymbol{pl}}^{(I)}$, in the real model, and permuted vectors $\overrightarrow{\boldsymbol{po}}'^{(I)}$ and $\overrightarrow{\boldsymbol{pl}}'^{(I)}$, in the ideal model, are indistinguishable.

Since sequence $Q'_1, ..., Q'_\Upsilon$ is generated given the leakage function, its access and query patterns are identical to the access and query patterns of $Q_1, ..., Q_\Upsilon$. Now we show that Q_i is indistinguishable from Q'_i, ($\forall i, 1 \leq i \leq \Upsilon$). First we consider the case where $T_i = \mathsf{PSI\text{-}Com}$. In this case, keys $tk^{(I)}$ and $tk'^{(I)}$, ($\forall I, I \in \{A, B\}$), are random values, so they are indistinguishable. Also, messages $\mathbf{ID}^{(A)}$, $\mathbf{ID}^{(B)}$ and **Compute** are identical in both views. Moreover, in the real model each pair in randomly permuted vector $\overrightarrow{\boldsymbol{pm}}_{A\to B}$ has the form $(l_g^{(A)}, l_g^{(B)})$ where ($\forall g, 1 \leq g \leq h$ and $\forall I, I \in \{A, B\}$) $l_g^{(I)} \in \overrightarrow{\boldsymbol{l}}^{(I)}$ and each $l_g^{(I)}$ is a pseudorandom string. Similarly, in the ideal model each pair in randomly permuted vector $\overrightarrow{\boldsymbol{pm}}'_{A\to B}$ has the form $(l_{g'}'^{(A)}, l_{g'}'^{(B)})$ where $l_{g'}'^{(I)} \in \overrightarrow{\boldsymbol{l}}'^{(I)}$ and each $l_{g'}'^{(I)}$ is a pseudorandom string, so $\overrightarrow{\boldsymbol{pm}}_{A\to B}$ and $\overrightarrow{\boldsymbol{pm}}'_{A\to B}$ are indistinguishable. Hence, Q_i is indistinguishable from Q'_i.

Now we move on to the case where $T_i = \mathsf{Upd}_t^{(I)}$. In the real model, $\vec{\boldsymbol{o}}^{(I)}_{j,c_j^{(I)}}$ contains n elements blinded with pseudorandom values, while in the ideal model $\vec{\boldsymbol{o}}''^{(I)}$ comprises n random values. Since the random values and blinded values are indistinguishable vectors $\vec{\boldsymbol{o}}^{(I)}_{j,c_j^{(I)}}$ and $\vec{\boldsymbol{o}}''^{(I)}$ are indistinguishable. In the real model, $l_j^{(I)}$ is a pseudorandom string and it belongs to vector $\vec{\boldsymbol{l}}^{(I)}$. In the ideal model, $l_j'^{(I)}$ is a pseudorandom string and it belongs to vector $\vec{\boldsymbol{l}}'^{(I)}$. Therefore, the labels have the same distribution and are indistinguishable. Also, message **Update** is identical in both models. So, Q'_i and Q_i are indistinguishable in this case too. From the above, we conclude that the views are indistinguishable.

Case 2: Corrupted Client B. In the real execution client B's view is defined as:

$$\mathsf{VIEW}_B^{\mathsf{D\text{-}PSI}}(\Lambda, S^{(A)}, S^{(B)}) = \{S^{(B)}, r_B, \vec{\boldsymbol{q}}, \vec{\boldsymbol{f}}, f_\cap(S^{(A)}, S^{(B)})\}.$$

The simulator SIM_B who receives $pk^{(B)}, lk^{(B)}, S^{(B)}$ and $f_\cap(S^{(A)}, S^{(B)})$) does the following:

1. Creates an empty view, and appends $S^{(B)}$ and uniformly at random chosen coins r'_B to it. Then, it chooses two sets $S'^{(A)}$ and $S'^{(B)}$ such that $S'^{(A)} \cap S'^{(B)} = f_\cap(S^{(A)}, S^{(B)})$ and $|S'^{(A)}|, |S'^{(B)}| \leq c$.
2. Constructs $\mathtt{HT}'^{(A)}$ and $\mathtt{HT}'^{(B)}$ using the public parameters. Next, it maps the elements in $S'^{(A)}$ and $S'^{(B)}$ to the bins of $\mathtt{HT}'^{(A)}$ and $\mathtt{HT}'^{(B)}$, respectively. $\forall I, I \in \{A, B\}$ and $\forall s_i'^{(I)} \in S'^{(I)}$: $\mathtt{H}(s_i'^{(I)}) = j$, then $s_i'^{(I)} \rightarrow \mathtt{HT}_j'^{(I)}$, where $1 \leq j \leq h$.
3. Constructs a polynomial representing the d elements of each bin. If a bin contains less than d elements first it is padded with random values to d elements. $\forall I, I \in \{A, B\}$ and $\forall j, 1 \leq j \leq h$: $\tau_j'^{(I)}(x) = \prod_{m=1}^{d}(x - e_m^{(I)})$, where $e_m^{(I)} \in \mathtt{HT}_j'^{(I)}$.
4. Assigns a random polynomial $\omega_j'^{(I)}$ of degree d to each bin $\mathtt{HT}_j'^{(I)}$ ($\forall I, I \in \{A, B\}$). Next, it constructs vectors $\vec{\boldsymbol{f}}'_j$ whose elements are computed as $\forall j, 1 \leq j \leq h$ and $\forall i, 1 \leq i \leq n$: $f'_{j,i} = \tau_j'^{(A)}(x_i) \cdot \omega_j'^{(A)}(x_i) + \tau_j'^{(B)}(x_i) \cdot \omega_j'^{(B)}(x_i)$, where $n = 2d + 1$.
5. Generates vector $\vec{\boldsymbol{q}}' = [\vec{\boldsymbol{q}}'_1, ..., \vec{\boldsymbol{q}}'_h]$ where each vector $\vec{\boldsymbol{q}}'_i$ contains n random values picked from field $\mathbb{F}_p$. Then, it appends $\vec{\boldsymbol{q}}'' = \pi(pk^{(B)}, \vec{\boldsymbol{q}}')$, $\vec{\boldsymbol{f}}'' = \pi(pk^{(B)}, \vec{\boldsymbol{f}}')$ and $f_\cap(S^{(A)}, S^{(B)})$ to the view and outputs it.

Now we show that the two views are computationally indistinguishable. The entries $S^{(B)}$ and Λ are identical in both views. In the real model, the elements in $\vec{\boldsymbol{q}}_j$ are blinded with pseudorandom values, so the blinded elements are uniformly random values. On the other hand, in the ideal model the elements in $\vec{\boldsymbol{q}}'_j$ are random values drawn from the same field. Moreover, both vectors are permuted in the same way. Hence, the vectors $\vec{\boldsymbol{q}}$ and $\vec{\boldsymbol{q}}'$ are computationally indistinguishable.

Furthermore, in the real model, given each unblinded vector $\vec{\boldsymbol{f}}_j$, the adversary interpolates a $2d$-degree polynomial of the form $\phi_j(x) = \omega_j^{(A)}(x) \cdot \tau_j^{(A)}(x) + \omega_j^{(B)}(x) \cdot \tau_j^{(B)}(x) = \mu_j \cdot gcd(\tau_j^{(A)}(x), \tau_j^{(B)}(x))$, where polynomial $gcd(\tau_j^{(A)}(x), \tau_j^{(B)}(x))$ represents intersection of the sets in the corresponding bin, $\mathtt{HT}_j$. Similarly, in the ideal model, each $2d$-degree polynomial $\phi'_j(x)$ interpolated from vector $\vec{\boldsymbol{f}}'_j$ has the form $\phi'_j(x) = \omega_j'^{(A)}(x) \cdot \tau_j'^{(A)}(x) + \omega_j'^{(B)}(x) \cdot \tau_j'^{(B)}(x) = \mu'_j \cdot gcd(\tau_j'^{(A)}(x), \tau_j'^{(B)}(x))$, where $gcd(\tau_j'^{(A)}(x), \tau_j'^{(B)}(x))$ represents the sets intersection in the corresponding bin, $\mathtt{HT}'_j$. Also, as we discussed in section 2.5, μ_j and μ'_j are uniformly random polynomials and the probability that their roots represent set elements is negligible, thus $\phi_j(x)$ and $\phi'_j(x)$ only contain information about the set intersection and have the same distribution in both models [75, 20]. Moreover, since the same hash table parameters were used, the same elements would reside in the same bins in both models, therefore polynomials

$gcd(\tau_j^{(A)}(x), \tau_j^{(B)}(x))$ and $gcd(\tau_j'^{(A)}(x), \tau_j'^{(B)}(x))$ represent the set elements of the intersection for that bin. Moreover, both vectors $\vec{\boldsymbol{f}}$ and $\vec{\boldsymbol{f}}'$ are permuted in the same way. So, $\vec{\boldsymbol{f}}$ and $\vec{\boldsymbol{f}}'$ are indistinguishable as well. Also, the output, $f_{\cap}(S^{(A)}, S^{(B)})$, is identical in both views. Thus, the two views are computationally indistinguishable.

Case 3: Corrupted Client A**.** In the real execution client A's view is defined as:

$$\mathsf{VIEW}_A^{\mathsf{D\text{-}PSI}}(\Lambda, S^{(A)}, S^{(B)}) = \{S^{(A)}, r_A, lk^{(B)}, pk^{(B)}, \vec{\boldsymbol{r}}^{(B)}, \mathsf{ID}^{(B)}, \Lambda\}.$$

The simulator, SIM_A, who receives $S^{(A)}$ performs as follows. It constructs an empty view, and adds $S^{(A)}$ and uniformly at random chosen coins r'_A to the view. It picks two random keys $lk'^{(B)}$, $pk'^{(B)}$ and adds them to the view. Then, it constructs $\vec{\boldsymbol{r}}'^{(B)} = [\vec{\boldsymbol{r}}'^{(B)}_1, ..., \vec{\boldsymbol{r}}'^{(B)}_h]$, where each vector $\vec{\boldsymbol{r}}'^{(B)}_i$ contains n random values picked from the field. It also appends $\vec{\boldsymbol{r}}'^{(B)}$, $\mathsf{ID}^{(B)}$ and Λ to the view and outputs the view.

In the following, we will show why the two views are indistinguishable. In both views, $S^{(A)}$, $\mathsf{ID}^{(B)}$ and Λ are identical. Also r_A and r'_A are chosen uniformly at random so they are indistinguishable. Moreover, values $lk^{(B)}, pk^{(B)}, lk'^{(B)}$ and $pk'^{(B)}$ are the keys picked uniformly at random, so they are indistinguishable, too. In the real model, each vector $\vec{\boldsymbol{r}}^{(B)}_i$ contains n values blinded with pseudorandom values. On the other hand, in the ideal model, each vector $\vec{\boldsymbol{r}}'^{(B)}_i$ comprises n random elements of the field. Since the random values and blinded values are indistinguishable, vectors $\vec{\boldsymbol{r}}^{(B)}$ and $\vec{\boldsymbol{r}}'^{(B)}$ are indistinguishable. Hence, the two views are indistinguishable. □

5.5 Updatable Delegated PSI Protocol Comparison

We compare the properties and costs of the UEO-PSI scheme with EO-PSI, O-PSI and [82], i.e. the protocols that support secure repeated PSI delegation and can support update. In table 5.1, we summarize the comparison results. All the four protocols are fully private and they preserve clients' data privacy; therefore, the cloud cannot learn anything about the client's data. Also, they ensure that only an authorized client receives the computation result. Moreover, the protocols support secure repeated PSI delegation, and support multiple clients. However, UEO-PSI and EO-PSI (unlike the other two protocols) do not use any public key encryption.

It should be noted that the analysis we carried out, in section 4.5.1, to determine the right parameters for the hash table in EO-PSI, can be applied to UEO-PSI as well, as

Property	UEO-PSI	[82]	O-PSI	EO-PSI
Private Against the Cloud	✓	✓	✓	✓
Client-to-client Computation Authorization	✓	✓	✓	✓
Non-interactive Client-side Setup	✓	✓	✓	✓
Secure Repeated PSI Delegations	✓	✓	✓	✓
Multiple Clients	✓	✓	✓	✓
Without Involving Public Key Encryption	✓	×	×	✓
Overall Communication Complexity	$O(hd)$	$O(c)$	$O(c)$	$O(hd)$
Update Communication Complexity	$O(d)$	–	$O(c)$	$O(hd)$
Overall Computation Complexity	$O(hd^2)$	$O(c)$	$O(c)$	$O(hd^2)$
Update Computation Complexity	$O(d^2)$	–	$O(c)$	$O(hd^2)$

Table 5.1: Comparison of the properties and costs of the protocols supporting secure repeated PSI delegation. We denote set cardinality's upper bound by c. In the table, h denotes the hash table length and d denotes bin's maximum load.

the hash tables in both protocols have the same structure. Therefore, for both of them we can set $d = 100$. In this setting, $hd \leq 4c$ and the probability that a bin receives more than d elements is lower than 2^{-40}.

Communication Complexity. First we evaluate the communication cost of UEO-PSI protocol when PSI is delegated. Client B, in step d.3, sends a single value $tk^{(B)}$ to the cloud. In the same step, it sends two constant values, $pk^{(B)}$ and $lk^{(B)}$, and vector $\vec{\boldsymbol{r}}^{(B)}$ of h bins to client A, where each bin contains $n = 2d + 1$ elements. Therefore, client B's communication cost is $O(hd)$. In step d.9, client A sends single value $tk^{(A)}$ and vector $\pi(pk^{(A)}, \vec{\boldsymbol{m}}_{A\to B})$ containing $2h$ elements to the cloud. In the same step, it sends vector $\vec{\boldsymbol{q}}$ containing h bins to client B, where each bin contains n elements. So, client A's total communication complexity is $O(hd)$. The cloud's communication complexity is $O(hd)$, as in step e.3, it sends vector $\vec{\boldsymbol{t}}$ of h bins to client B, where each bin contains n elements. Moreover, multi-client UEO-PSI's overall communication complexity is similar to the multi-client EO-PSI's.

Now we evaluate the communication complexity of UEO-PSI in the update phase. For client I ($I \in \{A, B\}$) to update its outsourced set, it sends to the cloud two labels, one in each of steps c.1 and c.7. Moreover, it sends vector $\vec{\boldsymbol{o}}^{(I)}_{j,c_j^{(I)}}$ of $n = 2d+1$ elements in steps c.7. So, in total $O(d)$ elements are sent by the client. Also, the cloud in step c.1 sends vector $\vec{\boldsymbol{o}}^{(I)}_{j,c_j^{(I)}}$ of n elements to the client. Therefore, the cloud's communication cost is $O(d)$, too.

The overall communication cost of O-PSI and EO-PSI are $O(c)$ and $O(hd)$ respec-

tively. Also, the protocol in [82] has at least linear communication cost, $O(c)$. As we stated earlier in section 5.2.1, in O-PSI protocol, in order for a client to update the data it needs to download them, update locally, and upload them to the cloud, so its update cost is $O(c)$. Also, in EO-PSI, if client simply updates one bin it would leak non-negligible information about set elements to the cloud (see section 5.2.1 for discussion). So, in EO-PSI, the secure way for the client to update the outsourced dataset is to retrieve the entire dataset and update it locally and send it back to the cloud. However, such approach imposes $O(hd)$ communication cost. On the other hand, the protocol in [82] that utilizes FHE can support data update by using the FHE's capabilities. But, the protocol is designed for generic computation and the complexity of update in delegated PSI has not been defined. Note that each message in EO-PSI and UEO-PSI protocols is a random element of a field (e.g. p is about 112 bits), whereas the messages in the other two protocols are ciphertexts of public key encryption that operate in a much larger group/ring.

Hence, only in UEO-PSI the client can securely update its dataset with communication cost of $O(d)$, while such operation cost is $O(c)$ and $O(hd)$ for O-PSI and EO-PSI respectively, where $d < c < hd$.

Computation Complexity. We first analyze the computation cost of UEO-PSI protocol when the clients delegate PSI to the cloud. In our analysis, we do not count the pseudorandom function invocation cost, as it is a fast operation, and dominated by the other operations (e.g. modular arithmetic, shuffling and factorization) in our protocol. In step d.1, client B executes h modular additions, and shuffles a vector of h elements where the cost of such operation is $O(h)$. Also, it performs nh modular additions in each of steps d.2 and f.1 to blind and unblind the hash table elements, respectively. Furthermore, in step f.2 it interpolates h polynomials where each interpolation costs $O(d)$. In step f.3, it factorizes h polynomials where each factorization costs $O(d^2)$. So in total client B's computation cost is $O(hd^2)$. Client A in step d.4 shuffles a vector of h elements that costs $O(h)$ and also in step d.6 it does h modular additions, and shuffles a vector of h elements. In step d.7, it performs $2hnd$ modular multiplications and $2hnd$ modular additions to evaluate the pseudorandom polynomials. In the same step, to generate vectors $\vec{\boldsymbol{v}}^{(A)}$ and $\vec{\boldsymbol{v}}^{(B)}$, it performs $2nh$ modular multiplications. In step d.8, it carries out $2nh$ modular additions to generate vectors $\vec{\boldsymbol{q}_e}$. Thus, in total $O(hd^2)$ modular operations are involved. The cloud in step e.2 does $2hnd$ modular multiplications and $2hnd$ modular additions to evaluate the pseudorandom polynomials. In

the same step, it performs $2nh$ modular multiplications and $3nh$ modular additions to generate the vectors $\vec{t_e}$. Thus, in total the cloud's computation involves $O(hd^2)$ modular operations. Also, multi-client UEO-PSI's computation complexity is similar to multi-client EO-PSI's.

Now we analyze the computation complexity of our scheme in the update phase. Client I carries out $n+1$ modular additions in step c.2. It interpolates a polynomial in step c.3 which costs $O(d)$. In step c.5, it executes d modular additions and d modular multiplications to evaluate the polynomial at $s^{(I)}$. In the same step, when the update is element insertion, the client extracts a bin's set elements that costs $O(d^2)$, and carries out nd modular multiplications and nd modular additions to evaluate the polynomial. In this case, its total computation cost is $O(d^2)$. While, if the update is element deletion it performs $2n$ modular additions to evaluate the two polynomials and $2n$ modular multiplications to generate values $u^{(I)}_{j,c^{(I)}_j,i}$. Finally, in step c.6, the client performs $n+1$ modular additions. Therefore, in this case, the client's computation complexity is $O(d)$.

To approximate the overall time taken for the update operation in UEO-PSI, we can focus on the average time needed to factorize the polynomial in a bin. The reason is that the polynomial factorization is the dominant operation in this process and the rest of operations are very fast. As Fig 4.4 indicates, this operation takes only 0.25 seconds (recall that $d = 100$), and this time is for the element insertion operation that involves polynomial factorization. Nonetheless, the element deletion process takes even less than that, as there is no factorization operation involved and all the operations are very fast.

O-PSI protocol uses Paillier homomorphic encryption and its computation cost is dominated by the encryption operations. The number of such operations is linear to c, so it is $O(c)$. Furthermore, EO-PSI computation complexity is $O(hd^2)$ and involves no public key encryption operations. Also, the overall computation complexity of the protocol in [82] is dominated by FHE operations and the overall number of such operations is linear to the inputs size, c. So the protocol's computation cost is at least $O(c)$.

In O-PSI, the update cost for a client to update an element, it needs to download the dataset, check the element membership in the set, construct a polynomial, evaluate it at $m = 2c+1$ values, encode it and then upload it. So, the protocol's computation cost for the update is $O(c)$. Moreover, as we discussed earlier, in EO-PSI, for the client to securely update its outsourced data, it needs to download the entire dataset and apply

the update locally and this costs $O(hd^2)$. As the protocol in [82] is based on FHE, it can support data update due to the nature of FHE. But, the protocol is designed for generic computation and the computation complexity of update in delegated PSI has not been defined for the protocol.

Although EO-PSI and UEO-PSI have the same complexities, EO-PSI is slightly more efficient than UEO-PSI. The reason is that in UEO-PSI, when the computation is delegated, client A sends two sets of messages to the cloud while in EO-PSI it sends only one set. Also, in UEO-PSI, client A pseudorandomly permutes two sets of messages and this introduces additional computational cost to the client while this computation is not needed in EO-PSI (however pseudorandom permutation is a fast operation).

Thus, EO-PSI and UEO-PSI are computationally more efficient than the two protocols using a public key encryption scheme. Moreover, UEO-PSI supports update operations efficiently with a cost of at most $O(d^2)$, while this is not the case for the other three protocols.

5.6 Concluding Remarks

In this chapter, we presented a protocol that efficiently supports secure repeated PSI delegation on dynamic data. In the protocol, multiple clients can independently store their private data in the cloud, efficiently update the outsourced data at any time, and delegate computation of private dataset intersection to the cloud (an unlimited number of times). The clients prepare their data independently without interacting with each other. Once the clients outsourced the data, they do not need to keep a local copy of it to perform the computation. The protocol guarantees the cloud learns no information about the computation inputs and output, and the result recipient only learns the intersection. It utilizes a combination of a hash table and permutation techniques to support efficient data update without leaking information about the set elements. Another advantage of the protocol is that it does not use any public key encryption operation which makes it more efficient. We defined the security model in the presence of semi-honest adversaries, and analyzed the protocol's security in the standard model. The analysis showed that the protocol is secure in the presence of semi-honest adversaries.

Chapter 6

Verifiable Delegated PSI on Outsourced Private Datasets

6.1 Introduction

In chapters 4 and 5, we proposed protocols that support secure repeated PSI delegation, and enable clients to outsource data storage and PSI computation to the cloud. The main focus of those protocols is on data privacy. However, (as we discussed in section 3.1) there are cases where the cloud may arbitrary misbehave, and provide an incorrect result.

In this chapter, we provide VD-PSI, a protocol that, in addition to supporting secure repeated PSI delegation, allows clients to *efficiently verify* the correctness of the computation result provided by the cloud. If the cloud misbehaves (e.g. by deviating from the protocol, tampering with the clients' data or computation result, etc) and provides an incorrect result, it would be detected by the client who receives the result. The main novelty of VD-PSI is a lightweight verification mechanism that allows a client to efficiently verify the result correctness without having access to its outsourced dataset or having any knowledge of the other client's dataset. The VD-PSI protocol has been published in a paper titled "*VD-PSI: Verifiable Delegated Private Set Intersection on Outsourced Private Datasets*" [2].

The rest of this chapter is organized as follows. Section 6.2 includes a brief explanation of the challenges that VD-PSI should address, an overview of the protocol, an elaborate description of two-client VD-PSI, and its extensions. In section 6.3, we provide the security definition of VD-PSI followed by section 6.4 that contains a se-

curity analysis of the protocol. In section 6.5, we provide a comparison between the verifiable delegated PSI protocols and in section 6.6 we conclude the chapter.

6.2 VD-PSI: Verifiable Delegated Private Set Intersection on Outsourced Private Datasets

We start this section by briefly outlining a set of challenges that the protocol must deal with. First off, the protocol needs to make sure the cloud uses datasets corresponding to the participating clients, as a malicious cloud may use a dataset belonging to a different client who is not participating in the protocol, or it may use its own dataset. Also, it needs to ensure that the cloud uses the clients' original datasets intact, because a malicious cloud may use the participating clients' datasets but it may modify them. Moreover, the protocol needs to ensure that the cloud executes the protocol correctly, as the cloud may use the clients' intact datasets but it may misbehave during the computation of the result and generate an incorrect result. As the clients, in the data preparation phase in VD-PSI, do not interact with each other, they do not know (and should not know) what exactly the other client's set elements are. This makes the previous challenges even more difficult to address. Roughly speaking, the client who receives the result must be able to detect the cloud if the cloud misbehaves, and provides an incorrect result. In addition to that, the privacy of the datasets must be protected from the cloud and the result recipient should not learn anything beyond the intersection about the other client's set elements.

6.2.1 An Overview of VD-PSI

VD-PSI's main building blocks include Paillier homomorphic encryption, point-value polynomial representation of sets, a blinding technique, and a verification mechanism that leverages a value that acts as a trap to detect the cloud's misbehavior. However, the way the elements are blinded in VD-PSI is different than they are in O-PSI, EO-PSI or UEO-PSI. The reason is that in VD-PSI both data integrity and privacy are security concerns; whereas, in the other three protocols only data privacy is. So, in VD-PSI we blind the elements in a way that can help to satisfy both needs.

We first consider the two-client case, where the cloud along with client A and B engage in the protocol. The interaction between the parties is depicted in Fig 6.1.

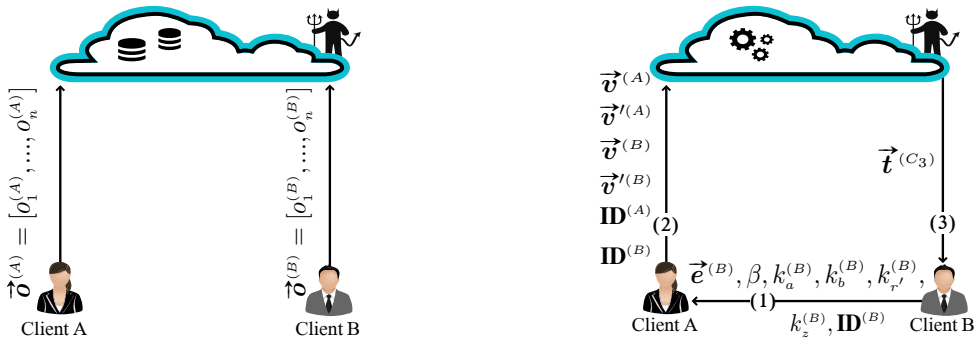

Figure 6.1: The left-hand side figure: party interactions at data outsourcing phase; the right-hand side figure: party interactions at computation delegation phase.

Similar to the other protocols we designed, in the setup phase each client independently prepares and stores its dataset in the cloud. Later on, when client B becomes interested in the intersection of its dataset and client A's, it obtains client A's permission by sending a message to it. Client A authorizes the computation by using the message sent by client B to compute a new message that it sends to the cloud. The cloud uses the message and the clients' outsourced datasets to compute the intersection, and sends the result to client B. When client B receives the cloud's response it decodes the elements, retrieves the intersection and checks its correctness. If the result is correct then the client accepts it; otherwise, it realizes the cloud has misbehaved.

The main novelty of VD-PSI is its lightweight verification mechanism that allows a client to efficiently verify the correctness of the result without having access to its own outsourced dataset and having any knowledge of the other client's dataset. To achieve this, when the clients decide to delegate the computation of the set intersection, they agree on a secret value β and provide β in an encoded form to the cloud, while the encoded data reveals nothing about β to the cloud. The cloud uses the clients' outsourced datasets and the encoded β to generate the result. In fact, the value, β, acts as a "trap" which means if the cloud behaves honestly, β will be inserted into the intersection and the client can retrieve it. However, if the cloud misbehaves or tampers with the result, then after unblinding client B gets a random set, which would not contain β with a high probability. In particular, in order for the cloud to modify or replace the client(s) data, it needs to know the blinding factors the clients used to protect the datasets but it does not. Due to the way the data and the intermediate message are blinded, and our vital observation (stated in section 6.3, Lemma 3) any tampering with the data (without knowing the blinding factors) makes the result recipient receive a set of random elements among which value β would not be, with a high probability. Furthermore, the

case where the cloud does not carry out the computation correctly, would be equivalent to the previous scenario (i.e. data tampering) and has the same consequence. Thus, by checking whether β is included in the result, client B knows whether the result set is correct. The verification is very lightweight because the only overhead is to check whether β is included in the result.

6.2.2 VD-PSI Protocol

What follows is a detailed description of VD-PSI protocol followed by the rationale behind it. We first consider the two-client case, where client A, client B and the cloud engage in the protocol. We use $E_{pk_I}(h)$ and $D_{sk_I}(h)$ to say that value h is encrypted using client I's public key, and decrypted using its secret key, respectively.

a. **Cloud-Side Setup.** The cloud picks a public parameter: c, that is an upper bound of the set cardinality. It constructs a field $\mathbb{F}_p$ and a pseudorandom function PRF (similar to our previous protocols). Also, it constructs a vector $\vec{\boldsymbol{x}}$ containing $n = 2c + 3$ distinct non-zero random values: x_i. Note that after inserting β in the protocol, the result polynomial (in point-value form) becomes of degree $2c+2$; therefore, client needs $n = 2c+3$ distinct x_i values to interpolate a correct polynomial (in coefficient form), at the end of the protocol. The cloud publishes the description of the field, value n, vector $\vec{\boldsymbol{x}}$ along with pseudorandom function PRF.

b. **Client-Side Setup and Data Outsourcing.** Let client $I \in \{A, B\}$ have a set $S^{(I)}$, where $|S^{(I)}| \leq c$. Each client I performs as follows.

   1. Computes a key pair (pk_I, sk_I) for Paillier encryption and publishes the public key pk_I. Also, it picks two random private keys, $k_r^{(I)}$ and $k_z^{(I)}$ for the pseudorandom function, PRF. All keys are generated according to given security parameters. Furthermore, it makes sure values x_i are not equal to its set elements.
   2. Generates a polynomial representation of the set.

$$\tau^{(I)}(x) = \prod_{m=1}^{|S^{(I)}|} (x - s_m^{(I)}),$$

   where $s_m^{(I)} \in S^{(I)}$.

3. Represents the polynomial in point-value form by evaluating $\tau^{(I)}(x)$ at every element x_i in vector $\vec{\boldsymbol{x}}$. This yields a vector of y-coordinates: $\tau^{(I)}(x_i)$, where $x_i \in \vec{\boldsymbol{x}}$.
4. Blinds every y-coordinate, $\tau^{(I)}(x_i)$. To do that, it first computes pseudorandom values $r_i^{(I)} = \mathtt{PRF}(k_r^{(I)}, i)$ and $z_i^{(I)} = \mathtt{PRF}(k_z^{(I)}, i)$, and then uses them to blind the y-coordinates.

$$\forall i, 1 \leq i \leq n : o_i^{(I)} = r_i^{(I)} \cdot (\tau^{(I)}(x_i) + z_i^{(I)}).$$

5. Sends the blinded dataset, $\vec{\boldsymbol{\sigma}}^{(I)} = [o_1^{(I)}, ..., o_n^{(I)}]$, to the cloud.

c. **Set Intersection: Computation Delegation.** This phase starts when client B wants the intersection of its set and client A's set.

1. Client B picks a uniformly random value $\beta \xleftarrow{R} \mathbb{F}_p^*$ that will be inserted into the two datasets and chooses three fresh keys $k_a^{(B)}$, $k_b^{(B)}$ and $k_{r'}^{(B)}$ that are used to blind the messages sent by client A to the cloud.
2. Client B constructs a vector, $\vec{\boldsymbol{e}}^{(B)}$. Later on, client A will modify the vector, and send it to the cloud who can utilize the vector to insert β into client A's dataset and switch the client's blinding factors.

$$\forall i, 1 \leq i \leq n : e_i^{(B)} = E_{pk_B}(\sigma(x_i) \cdot r_i'^{(B)} \cdot r_i^{(B)}),$$

   where $\sigma(x_i) = (x_i - \beta)$, values $r_i^{(B)}$ are the blinding factors used by client B in step b.4 and $r_i'^{(B)} = \mathtt{PRF}(k_{r'}^{(B)}, i)$.
3. Client B sends to client A: $\vec{\boldsymbol{e}}^{(B)}$, β, $k_a^{(B)}$, $k_b^{(B)}$, $k_{r'}^{(B)}$, $k_z^{(B)}$, and its id, $\mathsf{ID}^{(B)}$.
4. Client A uses the multiplicative homomorphism of the encryption scheme to generate $\vec{\boldsymbol{v}}^{(A)}$ and $\vec{\boldsymbol{v}}^{(B)}$ that enable the cloud to multiply each client's dataset by a random polynomial and insert β into it. Also, $\vec{\boldsymbol{v}}^{(A)}$ allows the cloud to switch the blinding factors of client A's dataset.
   $\forall i, 1 \leq i \leq n :$

$$\begin{aligned} v_i^{(A)} &= (e_i^{(B)})^{\omega^{(A)}(x_i) \cdot (r_i^{(A)})^{-1}} \\ &= E_{pk_B}(r_i^{(B)} \cdot r_i'^{(B)} \cdot \omega^{(A)}(x_i) \cdot \sigma(x_i) \cdot (r_i^{(A)})^{-1}), \\ v_i^{(B)} &= \omega^{(B)}(x_i) \cdot \sigma(x_i) \cdot r_i'^{(B)}, \end{aligned}$$

   where $r_i'^{(B)} = \mathtt{PRF}(k_{r'}^{(B)}, i)$, key $k_{r'}^{(B)}$ was sent by client B in step c.3, $r_i^{(A)}$

are the blinding factors used by client A in step b.4, $\omega^{(A)}(x)$ and $\omega^{(B)}(x)$ are two random polynomials of degree $c+1$ and $\sigma(x_i) = (x_i - \beta)$.

5. Client A generates $\vec{\boldsymbol{v}}'^{(A)}$ and $\vec{\boldsymbol{v}}'^{(B)}$ (using the multiplicative homomorphism of the encryption scheme). The vectors enable the cloud to preserve the correctness of the result.
   $\forall i, 1 \leq i \leq n$:

$$\begin{aligned} v_i'^{(A)} &= (e_i^{(B)})^{\omega^{(A)}(x_i)\cdot(-z_i^{(A)})+a_i} \\ &= E_{pk_B}((-z_i^{(A)}) \cdot r_i^{(B)} \cdot r_i'^{(B)} \cdot \omega^{(A)}(x_i) \cdot \sigma(x_i) + c_i), \\ v_i'^{(B)} &= (e_i^{(B)})^{\omega^{(B)}(x_i)\cdot(-z_i^{(B)})+b_i} \\ &= E_{pk_B}((-z_i^{(B)}) \cdot r_i^{(B)} \cdot r_i'^{(B)} \cdot \omega^{(B)}(x_i) \cdot \sigma(x_i) + d_i), \end{aligned}$$

   where $c_i = a_i \cdot r_i^{(B)} \cdot r_i'^{(B)} \cdot \sigma(x_i)$, $d_i = b_i \cdot r_i^{(B)} \cdot r_i'^{(B)} \cdot \sigma(x_i)$, $a_i = \mathtt{PRF}(k_a^{(B)}, i)$, $b_i = \mathtt{PRF}(k_b^{(B)}, i)$, keys $k_a^{(B)}$ and $k_b^{(B)}$ were sent by client B in step c.3, and $z_i^{(I)}$ are the values used by client $I \in \{A, B\}$ in step b.4.
6. Client A sends to the cloud: $\vec{\boldsymbol{v}}^{(A)}$, $\vec{\boldsymbol{v}}'^{(A)}$, $\vec{\boldsymbol{v}}^{(B)}$, $\vec{\boldsymbol{v}}'^{(B)}$, $\mathsf{ID}^{(B)}$, $\mathsf{ID}^{(A)}$, and a request message, **Compute**.

d. **Set Intersection: Cloud-Side Computation.** In this phase, the cloud leverages the additive and multiplicative homomorphism of the encryption scheme to combine the clients' datasets with the messages client A sent, and generate the result.

1. When the cloud receives client A's message, it uses $\vec{\boldsymbol{v}}^{(A)}$, $\vec{\boldsymbol{v}}'^{(A)}$ and client A's outsourced dataset $\vec{\boldsymbol{o}}^{(A)}$ to switch the dataset's blinding factors, insert β to the dataset, and multiply it by a random polynomial: $\omega^{(A)}(x)$. This results in $\vec{\boldsymbol{t}}^{(C_1)}$ whose elements are computed as follows.
   $\forall i, 1 \leq i \leq n$:

$$\begin{aligned} t_i^{(C_1)} =& (v_i^{(A)})^{o_i^{(A)}} \cdot v_i'^{(A)} \\ =& E_{pk_B}((-z_i^{(A)}) \cdot r_i^{(B)} \cdot r_i'^{(B)} \cdot \omega^{(A)}(x_i) \cdot \sigma(x_i) + \\ & z_i^{(A)} \cdot r_i^{(B)} \cdot r_i'^{(B)} \cdot \omega^{(A)}(x_i) \cdot \sigma(x_i) \cdot (r_i^{(A)})^{-1} \cdot r_i^{(A)} + \\ & r_i^{(B)} \cdot r_i'^{(B)} \cdot \sigma(x_i) \cdot (r_i^{(A)})^{-1} \cdot r_i^{(A)} \cdot (\omega^{(A)}(x_i) \cdot \tau^{(A)}(x_i) + a_i))) \end{aligned}$$

2. The cloud uses $\vec{\boldsymbol{o}}^{(B)}$, $\vec{\boldsymbol{v}}^{(B)}$, $\vec{\boldsymbol{v}}'^{(B)}$ to insert β into client B's dataset, and multiply it by a random polynomial: $\omega^{(B)}(x)$. This yields $\vec{\boldsymbol{t}}^{(C_2)}$.

$\forall i, 1 \leq i \leq n$:

$$\begin{aligned} t_i^{(C_2)} =& v_i'^{(B)} \cdot E_{pk_B}(v_i^{(B)} \cdot o_i^{(B)}) \\ =& E_{pk_B}((-z_i^{(B)}) \cdot r_i^{(B)} \cdot r_i'^{(B)} \cdot \omega^{(B)}(x_i) \cdot \sigma(x_i) + \\ & z_i^{(B)} \cdot r_i^{(B)} \cdot r_i'^{(B)} \cdot \omega^{(B)}(x_i) \cdot \sigma(x_i) + \\ & r_i^{(B)} \cdot r_i'^{(B)} \cdot \sigma(x_i) \cdot (\omega^{(B)}(x_i) \cdot \tau^{(B)}(x_i) + b_i))) \end{aligned}$$

3. The cloud combines the values computed in steps d.2 and d.1 to produce the final result $\vec{\boldsymbol{t}}^{(C_3)}$.

$$\forall i, 1 \leq i \leq n : t_i^{(C_3)} = t_i^{(C_1)} \cdot t_i^{(C_2)}.$$

4. The cloud sends to client B vector $\vec{\boldsymbol{t}}^{(C_3)} = [t_1^{(C_3)}, ..., t_n^{(C_3)}]$.

e. **Set Intersection: Client-Side Result Verification and Retrieval.**

1. Client B decrypts the cloud's response, $\vec{\boldsymbol{t}}^{(C_3)}$. Then, it unblindes the decrypted values using its knowledge of $(-c_i)$, $(-d_i)$, $(r_i^{(B)})^{-1}$ and $(r_i'^{(B)})^{-1}$. This yields vector $\vec{\boldsymbol{g}}$ of the following elements.
$\forall i, 1 \leq i \leq n$:

$$\begin{aligned} g_i &= (D_{sk_B}(t_i^{(C_3)}) + (-c_i) + (-d_i)) \cdot (r_i^{(B)})^{-1} \cdot (r_i'^{(B)})^{-1} \\ &= \omega^{(B)}(x_i) \cdot \sigma(x_i) \cdot \tau^{(B)}(x_i) + \omega^{(A)}(x_i) \cdot \sigma(x_i) \cdot \tau^{(A)}(x_i) \end{aligned}$$

2. Client B interpolates a polynomial, $\phi(x)$, using n point-value pairs (x_i, g_i), extracts its roots, and checks whether β is among them. If it is, it considers the rest of the (valid) roots as elements of the intersection; otherwise, it aborts.

Remark 1: In step b.4, client $I \in \{A, B\}$ blinds its private data $\tau^{(I)}(x_i)$ as $o_i^{(I)} = r_i^{(I)} \cdot (\tau^{(I)}(x_i) + z_i^{(I)})$ to preserve their privacy and to detect unauthorized modifications. In our protocol, the way each blinded y-coordinate is computed has some similarity to the way an information-theoretic message authentication code (MAC) is generated, e.g. in [13], with the difference that we use pseudorandom values rather than truly random ones. It should be noted that if the client does not blind $\tau^{(I)}(x_i)$, the cloud can interpolate the polynomial $\tau^{(I)}(x)$ and find the client's set elements. After blinding,

every $o_i^{(I)}$ is a uniformly random value and does not leak any information about $\tau^{(I)}(x_i)$. As we will show shortly in section 6.3, if the cloud changes a subset of elements in $\vec{\boldsymbol{o}}^{(I)}$, in step e.1 after client B unblinds the cloud's response, it would get vector $\vec{\boldsymbol{g}}$ of elements some of which would become uniformly random values. However, in this case, the polynomial interpolated from n pairs of (x_i, g_i) will not have root β (with a high probability) if some of g_i values are random values. Therefore, the client can detect the misbehavior. Also, the case where the cloud deviates from the protocol would be similar to the above scenario and the client can detect it too.

Remark 2: We set $n = 2c+3$, because in step e.2, polynomial $\phi(x)$ is of degree $2c+2$ and at least $2c + 3$ pairs of (x_i, y_i) are required to interpolate it. Therefore, given n pairs of (x_i, y_i), if they are computed correctly, client B can interpolate $\phi(x)$.

Remark 3: In section 2.5, we saw that the set of all roots of polynomial $\omega^{(B)}(x) \cdot \tau^{(B)}(x) + \omega^{(A)}(x) \cdot \tau^{(A)}(x)$ represents the intersection: $S^{(A)} \cap S^{(B)}$. Note that β is also a root of polynomial $\phi(x) = \sigma(x) \cdot (\omega^{(B)}(x) \cdot \tau^{(B)}(x) + \omega^{(A)}(x) \cdot \tau^{(A)}(x))$ that client B gets after unblinding, where $\sigma(x) = x - \beta$. Hence, a correctly computed result contains the intersection and value β.

Remark 4: Similar to our previous protocols, for each computation, fresh random polynomials $\omega^{(A)}(x)$ and $\omega^{(B)}(x)$ are used, so the result recipient cannot find out anything beyond the intersection about the other client's set. Also, the cloud cannot learn the exact number of elements in the client's set; it only knows the upper bound of the set cardinality (i.e. c).

Remark 5: Every client I, after outsourcing its private dataset needs to keep locally only two secret keys, $k_r^{(I)}$ and $k_z^{(I)}$. Moreover, client B who is interested in the result generates keys $k_a^{(B)}$, $k_b^{(B)}$, $k_{r'}^{(B)}$ and value β on the fly for each run of the protocol and it can discard them after the protocol ends. Furthermore, given p each client I can always independently generate its own public key N_I, such that $N_I > p + 2p^2 + p^3$ to preserve the computation correctness (i.e. to prevent any overflow during homomorphic operations). To determine the lower bound of N_I, we can calculate the maximal value that message m_i in $E_{pk_I}(m_i)$ may take on as a result of the homomorphic operations in the protocol. To do so, we start from step c.2 and calculate upper bound of value m_i in each step (note that $o_i^{(I)} \in \mathbb{F}_p$). We proceed this up to step d.4, and then we set the

lower bound of N_I to maximal upper bound of m_i, that is $p + 2p^2 + p^3$.

Remark 6: We stress that in VD-PSI (similar to O-PSI, EO-PSI and UEO-PSI) the clients' outsourced datasets remain unchanged. Also, at the end of the protocol all parties can discard all intermediate messages they received.

Remark 7: Similar to most PSI protocols, e.g. [68, 56, 81, 123, 99], we considered a static malicious cloud. However, there are cases where the cloud can be corrupted by a stronger, dynamic (or adaptive) malicious adversary. For example, an external adversary (e.g. a hacker) corrupts the cloud and then a client or the cloud penetrates the other client's machine, during the protocol execution. We highlight that at the current stage, our protocol cannot withstand such adversaries. For instance, if the cloud could get access to client B's machine, then it would be able to learn both clients outsourced set elements, the computation result and it could tamper with the computation result without being detected. It would be desirable to have a protocol that is secure against adaptive malicious adversaries. However, since the adversary is more powerful, it is much harder to protect against. In consequence, protocols secure against these adversaries can be more complex and less efficient. Also, in some cases, it is even impossible to protect against these adversaries [56]. At this stage, we have to make a trade-off between security and efficiency, and assume the adversary is static. We hope we can address this problem in future work.

Remark 8: VD-PSI does not deal with malicious clients. The protocol would have provided stronger security, if it could withstand malicious clients too. A malicious client may behave arbitrarily e.g. to affect the result correctness. For instance, in the protocol, a malicious client A can set the messages, sent to the cloud, in a way that the intersection always looks empty to client B while the verification output indicates the result has been computed correctly. Therefore, as a result of clients misbehaviours, the result correctness can be affected. In order to make a multi-party protocol secure against malicious parties, researchers usually utilize techniques such as zero-knowledge proofs [75] or cut-and-choose approach [80] that make the parties follow the protocol correctly and VD-PSI can adopt these techniques too. Nevertheless, these techniques often introduce high (communication or computation) costs [45].

6.2.3 Extensions

In this section, we first outline how the two-client VD-PSI protocol can be extended to multi-client VD-PSI. After that, we show how in our VD-PSI we can reduce the storage space that client A who authorizes the computation needs.

6.2.3.1 Multi-client VD-PSI

In the following, we show how we can turn two-client VD-PSI into *m*-client VD-PSI, where $m > 2$. We denote the result recipient by client B and the other clients by A_l, where $1 \leq l \leq y$, $y = m - 1$.

In step c.3, client B sends to every client A_l the same message as it does in the two-client setting. Also, each client A_l takes the same steps as it does in the previous setting, except steps c.4 and c.5. In step c.4, it replaces $\vec{\boldsymbol{v}}^{(B)}$ with $\vec{\boldsymbol{v}}^{(B_l)}$ containing elements $v_i^{(B_l)} = E_{pk_B}(\omega^{(B_l)}(x_i) \cdot \sigma(x_i) \cdot r_i'^{(B)})$. Moreover, in step c.5, it replaces $\vec{\boldsymbol{v}}'^{(B)}$ with $\vec{\boldsymbol{v}}'^{(B_l)}$ that comprises elements $v_i'^{(B_l)} = E_{pk_B}((-z_i^{(B)}) \cdot r_i^{(B)} \cdot r_i'^{(B)} \cdot \omega^{(B_l)}(x_i) \cdot \sigma(x_i) + d_i)$. Note that $\omega^{(B_l)}(x)$ is a random polynomial picked by client A_l. Similar to the two-client case, in step d.1, the cloud computes $\vec{\boldsymbol{t}}_l^{(C_1)}$ for each client A_l. Also, in step d.2, the cloud computes $\vec{\boldsymbol{t}}^{(C_2)}$ as follows.

$$\forall i, 1 \leq i \leq n : t_i^{(C_2)} = v_i'^{(B)} \cdot (v_i^{(B)})^{o_i^{(B)}},$$

where $v_i'^{(B)} = \prod_{l=1}^{y} v_i'^{(B_l)}$ and $v_i^{(B)} = \prod_{l=1}^{y} v_i^{(B_l)}$. We highlight that in the above, client B's polynomial, $\tau^{(B)}(x)$, is multiplied by the sum of the random polynomials picked by the other clients (i.e. client A_l, $\forall l, 1 \leq l \leq y$). Accordingly, in step d.3, the cloud computes the elements of $\vec{\boldsymbol{t}}^{(C_3)}$ as follows.

$$\forall i, 1 \leq i \leq n : t_i^{(C_3)} = t_i^{(C_2)} \cdot \prod_{l=1}^{y} t_{l,i}^{(C_1)}$$

Then, it sends $\vec{\boldsymbol{t}}^{(C_3)}$ to client B. Finally, in step e.1, client B computes $\vec{\boldsymbol{g}}$ whose elements are generated as below. $\forall i, 1 \leq i \leq n$:

$$\begin{aligned} g_i &= (D_{sk_B}(t_i^{(C_3)}) + \mathbf{y} \cdot (-c_i) + \mathbf{y} \cdot (-d_i)) \cdot (r_i^{(B)})^{-1} \cdot (r_i'^{(B)})^{-1} \\ &= \omega^{(B)}(x_i) \cdot \sigma(x_i) \cdot \tau^{(B)}(x_i) + \sum_{l=1}^{y} \omega^{(A_l)}(x_i) \cdot \sigma(x_i) \cdot \tau^{(A_l)}(x_i), \end{aligned}$$

where $\omega^{(B)}(x) = \sum_{l=1}^{y} \omega^{(B_l)}(x)$. The rest of the steps remain unchanged. As we explained in section 4.2.3.1 and it is proven in [75], in the multi-client setting, even if client B colludes with $y-1$ clients, it cannot learn any information (beyond the intersection) about the non-colluding client's set elements.

Remark 1: In the multi-client case, each client encrypts the elements of vector $\vec{\boldsymbol{v}}_j^{(B)}$; whereas, in the two-client case it does not need to do that. Nonetheless, regardless of the number of clients, every client's computation complexity is $O(c)$.

Remark 2: Interestingly, verification complexity at the verifier-side is independent of the number of clients. Also, the number of messages every client, except the client who is interested in the result, sends and receives is independent of the number of clients, too. However, the client who is interested in the result sends the same message to all other clients.

6.2.3.2 Reducing Authorizer's Required Storage Space

In VD-PSI, similar to EO-PSI and UEO-PSI, we can leverage a hash table to reduce the storage space that client A needs to authorize the computation (also the use of a hash table reduces the computation cost of the result retrieval for client B). In the following, we briefly outline how this can be done. For the sake of simplicity we consider the two-client case, but the adjustments can be directly applied to the multi-client setting. In the beginning, the hash table parameters are picked and made public by the cloud. In the client-side setup phase, each client first maps its set elements to the hash table bins, pads each bin up to d elements and then encodes the elements of each bin in the same way as they do in VD-PSI. In this case, the clients (when they delegate the computation) insert a random β_j in each outsourced bin $\mathtt{HT}_j$, to ensure the operation on each bin is performed correctly. In order for the clients to generate β_j, they can use a shared key, βk, where $\beta_j = \mathtt{PFR}(\beta k, j)$. Client B keeps the key that allows it to regenerate β_j in the verification phase. When it receives the result from the cloud, it checks whether each bin $\mathtt{HT}_j$ contains β_j. This setting enables client B, in step c.3, to send only one bin at a time to client A who operates on the bin and forwards it to the cloud who similarly operates on each bin and sends the result to client B. As a result, the storage space client A needs to authorize the computation reduces from

$O(c)$ to $O(d)$ (as we showed in section 4.5.1, $d = 100$ and $d < c$).

6.3 Security Definition

In this chapter, similar to our previous protocols, we consider a static adversary who controls one of the parties at a time (i.e. non-colluding static adversaries). The definition and model are according to [52, 67]. Without loss of generality, we consider three parties engaging in the protocol, a cloud C, and two clients, A and B, where client A authorizes the computation and client B is interested in the result. Similar to O-PSI and EO-PSI, we assume there exists an infrastructure, e.g. PKI, via which client A can authenticate, and then authorize the other client. We allow an adversary who corrupts C to be malicious. So it may arbitrarily deviate from the prescribed protocol. Moreover, we allow an adversary who corrupts a client to be semi-honest.

We define a three-party protocol, VD-PSI, computing function F where $F : \Lambda \times 2^{\mathfrak{U}} \times 2^{\mathfrak{U}} \rightarrow \Lambda \times \Lambda \times f_{\cap}$. Also, Λ denotes the empty string, $2^{\mathfrak{U}}$ denotes the powerset of the set universe and $f_{\cap}$ denotes the set intersection function. For every tuple of inputs $\Lambda, S^{(A)}$ and $S^{(B)}$ belonging to C, A and B respectively, the function outputs nothing to C and A, and outputs $f_{\cap}(S^{(A)}, S^{(B)}) = S^{(A)} \cap S^{(B)}$ to B. To show the protocol is secure, we define an ideal model, which satisfies all the security needs. In the ideal model, there is an incorruptible trusted third party (TTP) which helps with the functionality. As it is defined in section 2.9, a protocol is said to be secure if for every adversary in the real model there is an adversary in the ideal model that can simulate the real model's adversary.

Real Model: In this setting, protocol VD-PSI is executed between parties A, B, C and an adversary denoted by $\mathcal{A}_J$ that is allowed to corrupt one party, where $J \in \{A, B, C\}$. In the beginning of the protocol, each party $I \in \{A, B\}$ receives its private input $S^{(I)}$, the protocol's public parameters, random coins r, and an auxiliary input z, while the cloud C receives the public parameters, a set of random coins r, and an auxiliary input z. At the end of the execution, an honest party outputs whatever is prescribed by the protocol and the adversary outputs its view. The joint output of the real model execution of VD-PSI between the parties in the presence of the adversary $\mathcal{A}_J$ is defined as $\mathsf{REAL}^{\text{VD-PSI}}_{\mathcal{A}_J(z)}(\Lambda, S^{(A)}, S^{(B)})$.

Ideal Model: The ideal model takes place between parties A, B, C and a simulator

SIM_J that is allowed to corrupt at most one party at a time. Each party receives the same input as the corresponding party in the real model. An honest party always sends its input to the TTP. The corrupted party may abort or send arbitrary input. The cloud, C, receives c (i.e. $c \geq |S^{(I)}|, I \in \{A, B\}$) from the TTP. The TTP computes the set intersection and sends the result to B. If the TTP receives an abort message as an input, it sends B the special symbol $\perp$. The joint output of the parties in the ideal model in the presence of SIM_J is defined as $\mathsf{IDEAL}^{F}_{\mathsf{SIM}_J(z)}(\Lambda, S^{(A)}, S^{(B)})$.

Definition 16. *Let VD-PSI be a protocol and F a deterministic function defined as above. Protocol VD-PSI is said to securely compute F in the presence of static adversaries if for every probabilistic polynomial time (PPT) adversary $\mathcal{A}_J$ in the real model, there exists a PPT adversary SIM_I in the ideal model such that $\forall J, J \in \{A, B, C\}$:*

$$\{\mathit{IDEAL}^{F}_{\mathsf{SIM}_J(z)}(\Lambda, S^{(A)}, S^{(B)})\}_{S^{(A)},S^{(B)}} \overset{c}{\equiv} \{\mathit{REAL}^{\text{VD-PSI}}_{\mathcal{A}_J(z)}(\Lambda, S^{(A)}, S^{(B)})\}_{S^{(A)},S^{(B)}}$$

6.4 VD-PSI Security Proof

In this section, we sketch the security proof of the protocol. To this end, first we show that the cloud's misbehaviors can be detected with a high probability, and then we provide the main theorem.

Recall, the client encodes its set as blinded y-coordinates having the following form:

$$o_i = r_i \cdot (\tau(x_i) + z_i),$$

where $o_i \neq 0$, $r_i = \mathtt{PRF}(k_r, i)$ and $z_i = \mathtt{PRF}(k_z, i)$. If $o_i = 0$ the client replaces k_r and k_z with new random keys and encodes $\tau(x_i)$ again until $\forall i, 1 \leq i \leq n : o_i \neq 0$. Note that o_i is uniformly distributed in $\mathbb{F}_p^*$. We show that if the cloud applies any change to o_i, this will make the y-coordinate a uniformly random value.

Lemma 2. *Given $o_i = r_i \cdot (\tau(x_i) + z_i)$, where r_i and z_i are two independent pseudorandom values that are unknown to the cloud, if the cloud changes o_i to o'_i, then $\tau'(x_i) = r_i^{-1} \cdot o'_i - z_i$ becomes a uniformly random value.*

Proof. Since values r_i and z_i are picked uniformly at random and independently of each other, their (multiplicative and additive) inverse are uniformly random values too. Therefore, when $o_i \neq o'_i$, value $\tau'(x_i)$ is a uniformly random value in $\mathbb{F}_p$. □

In the protocol, in step e.1, client B after decrypting the cloud's response obtains blinded values of the form $f_i = e_i \cdot g_i + z_i$, where e_i and z_i are pseudorandom values. If the cloud misbehaves (e.g. deviates from the protocol, modifies the outsourced client datasets, etc), some f_i are changed to f'_i, and Lemma 2 implies that $g'_i = e_i^{-1} \cdot (f'_i - z_i)$ will be a uniformly random value. So, any misbehavior of the cloud turns some of values g_i into uniformly random values.

Now, we show that given a set of y-coordinates some of which are uniformly random values, client B interpolates a polynomial (in step e.2 in the protocol), that will not contain the specific root, β, with a high probability.

Lemma 3. *Let polynomial $\tau(x)$ be interpolated from $S = \{(x_1, y_1), ..., (x_n, y_n)\}$, and have root β (where $\beta \in \mathbb{F}_p^*$) such that $\forall i, 1 \leq i \leq n : \beta \neq x_i$. Let $S' = \{(x_1, y'_1), ..., (x_n, y'_n)\}$, where at least one of y'_i is a uniformly random value and the rest of them are equal to the y-coordinates in S (i.e. $y'_j = y_j$). Let polynomial $\tau'(x)$ be interpolated from S'. The probability that $\tau'(x)$ has root β is negligible.*

Proof. Given S', we interpolate a unique polynomial, $\tau'(x)$, of degree at most $n-1$. According to the Lagrange interpolation, the polynomial is

$$\tau'(x) = \sum_{i=1}^{n} y'_i \cdot \prod_{\substack{k=1 \\ i \neq k}}^{n} \frac{x - x_k}{x_i - x_k}$$

We evaluate $\tau'(x)$ at β:

$$\tau'(\beta) = \sum_{i=1}^{n} y'_i \cdot \prod_{\substack{k=1 \\ i \neq k}}^{n} \frac{\beta - x_k}{x_i - x_k}$$

As $\forall i, 1 \leq i \leq n : \beta \neq x_i$, we would have $\prod_{\substack{k=1 \\ i \neq k}}^{n} \frac{\beta - x_k}{x_i - x_k} \neq 0$. Since, at least one of y'_i is uniformly random, value $y'_i \cdot \prod_{\substack{k=1 \\ i \neq k}}^{n} \frac{\beta - x_k}{x_i - x_k}$ is uniformly random. Therefore, $\tau'(\beta)$ is uniformly random. Thus, $Pr[\tau'(\beta) = 0] = \frac{1}{p}$ which is negligible.

□

Now we are ready to prove that the client can detect cloud misbehavior with high probability.

Theorem 5. *Let clients A and B have sets $S^{(A)}$ and $S^{(B)}$ respectively, also let $S_\cap = S^{(A)} \cap S^{(B)}$. In the VD-PSI protocol, if the cloud sends S' (where $S' \neq S_\cap$) to client, the client can detect it with high probability.*

Proof. Due to Lemma 2, the cloud's misbehavior turns some of the y-coordinates (representing $S_\cap$) into uniformly random values. Also, in the protocol, β is chosen uniformly at random from $\mathbb{F}_p^*$, so the probability that $\beta = x_k$ for some $k, 1 \leq k \leq n$, is negligible. Due to Lemma 3, if the client interpolates a polynomial by using a set of y-coordinates where at least one of them is a uniformly random value, the probability that the polynomial would have β as a root is negligible. Thus, if the cloud computes an incorrect intersection the client can detect this with high probability through the absence of β from the intersection. □

Finally, we prove our main theorem.

Theorem 6. *If the homomorphic encryption scheme is semantically secure and* PRF *is a collision-resistant pseudorandom function, then the protocol is secure in the presence of (1) a malicious cloud and honest clients, (2) a semi-honest client and honest cloud.*

Proof. We consider three cases where each party is corrupted at a time. We consider the case where all parties want to engage in the computation of the intersection. If one party does not want to continue in the protocol, the views can be simulated in the same way up to the point where the execution stops. This includes the case where party A wants to engage in the computation of the intersection, i.e. it authorizes the computation. If party A does not want to proceed with the protocol, the views can be simulated in the same way up to the point where the execution stops.

Case 1: Cloud is Corrupted. We construct a simulator, SIM_C, in the ideal model that uses the adversary, $\mathcal{A}_C$, as a subroutine. Simulator SIM_C executes as follows.

1. Picks two random sets $S^{(E)}$ and $S^{(D)}$, where $|S^{(E)}|, |S^{(D)}| \leq c$. Also, it chooses keys $k_r^{(E)}$, $k_z^{(E)}$, $k_r^{(D)}$, $k_z^{(D)}$, $k_b^{(E)}$, $k_a^{(E)}$, $k_{r'}^{(E)}$.
2. Generates polynomials $\tau^{(E)}(x)$ and $\tau^{(D)}(x)$ representing the sets.

$$\tau^{(I)}(x) = \prod_{m=1}^{|S^{(I)}|} (x - s_m^{(I)}),$$

where $I \in \{D, E\}$ and $s_m^{(I)} \in S^{(I)}$. Then, it evaluates the polynomials at every element in $\vec{\boldsymbol{x}}$ and blinds the evaluated values. The results are two vectors $\vec{\boldsymbol{o}}^{(E)}$ and $\vec{\boldsymbol{o}}^{(D)}$ whose elements are generated as below.
$\forall i, 1 \leq i \leq n$:

$$r_i^{(I)} = \mathtt{PRF}(k_r^{(I)}, i),\ z_i^{(I)} = \mathtt{PRF}(k_z^{(I)}, i),\ o_i^{(I)} = r_i^{(I)} \cdot (\tau^{(I)}(x_i) + z_i^{(I)}).$$

3. Picks a random non-zero value: β', and constructs polynomial $\sigma'(x) = (x - \beta')$. Then, it picks two random polynomials, $\omega^{(E)}$ and $\omega^{(D)}$, of degree $c + 1$, and computes $\vec{\boldsymbol{v}}^{(E)}$ and $\vec{\boldsymbol{v}}^{(D)}$ as follows. $\forall i, 1 \leq i \leq n$:

$$\begin{aligned} v_i^{(D)} &= E_{pk_E}(r_i^{(E)} \cdot r_i'^{(E)} \cdot \omega^{(D)}(x_i) \cdot \sigma'(x_i) \cdot (r_i^{(D)})^{-1}), \\ v_i^{(E)} &= r_i'^{(E)} \cdot \omega^{(E)}(x_i) \cdot \sigma'(x_i), \end{aligned}$$

where $r_i'^{(E)} = \mathtt{PRF}(k_{r'}^{(E)}, i)$.

4. Computes two vectors $\vec{\boldsymbol{v}}'^{(E)}$ and $\vec{\boldsymbol{v}}'^{(D)}$ whose elements are computed as follows.
$\forall i, 1 \leq i \leq n$:

$$\begin{aligned} a_i^{(E)} &= \mathtt{PRF}(k_a^{(E)}, i),\ b_i^{(E)} = \mathtt{PRF}(k_b^{(E)}, i),\ z_i^{(I)} = \mathtt{PRF}(k_z^{(I)}, i), \\ v_i'^{(D)} &= E_{pk_E}((-z_i^{(D)}) \cdot r_i^{(E)} \cdot r_i'^{(E)} \cdot \omega^{(D)}(x_i) \cdot \sigma'(x_i) + c_i^{(E)}), \\ v_i'^{(E)} &= E_{pk_E}((-z_i^{(E)}) \cdot r_i^{(E)} \cdot r_i'^{(E)} \cdot \omega^{(E)}(x_i) \cdot \sigma'(x_i) + d_i^{(E)}), \end{aligned}$$

where $I \in \{D, E\}$, $c_i^{(E)} = a_i^{(E)} \cdot r_i^{(E)} \cdot r_i'^{(E)} \cdot \sigma'(x_i)$, $d_i^{(E)} = b_i^{(E)} \cdot r_i^{(E)} \cdot r_i'^{(E)} \cdot \sigma'(x_i)$.

5. Invokes $\mathcal{A}_C$ and feeds it with $\vec{\boldsymbol{o}}^{(D)}$, $\vec{\boldsymbol{o}}^{(E)}$, $\vec{\boldsymbol{v}}^{(D)}$, $\vec{\boldsymbol{v}}^{(E)}$, $\vec{\boldsymbol{v}}'^{(D)}$, $\vec{\boldsymbol{v}}'^{(E)}$, $\mathsf{ID}^{(A)}$, $\mathsf{ID}^{(B)}$, message **Compute**. Then, it receives $\vec{\boldsymbol{t}}^{(C)}$ from $\mathcal{A}_C$, decrypts the elements, and removes the blinding factors. This yields vector $\vec{\boldsymbol{g}}'$.
6. Interpolates a polynomial using the n point-value pairs (x_i, g_i') (where $g_i' \in \vec{\boldsymbol{g}}'$), and extracts the roots of the polynomial. It checks whether β' is among the roots. If it is not, aborts and instructs the TTP to send the abort message, $\perp$, to client B. Otherwise, it asks TTP to send the result to the client.
7. Outputs whatever the adversary outputs and terminates.

First, we consider the adversary's output. In the real model its view contains $\vec{\boldsymbol{o}}^{(A)}$, $\vec{\boldsymbol{o}}^{(B)}$, $\vec{\boldsymbol{v}}^{(B)}$, $\vec{\boldsymbol{v}}^{(A)}$,$\vec{\boldsymbol{v}}'^{(A)}$,$\vec{\boldsymbol{v}}'^{(B)}$, $\mathsf{ID}^{(A)}$, $\mathsf{ID}^{(B)}$, **Compute** and Λ. In the real model, the elements in $\vec{\boldsymbol{o}}^{(A)}$, $\vec{\boldsymbol{o}}^{(B)}$, $\vec{\boldsymbol{v}}^{(B)}$ are blinded by the outputs of a pseudorandom function using random secret keys. The same is true in the ideal model for the elements in $\vec{\boldsymbol{o}}^{(D)}$, $\vec{\boldsymbol{o}}^{(E)}$,

$\vec{\boldsymbol{v}}^{(E)}$. Since the outputs of the pseudorandom functions are computationally indistinguishable, the distributions of $\vec{\boldsymbol{o}}^{(A)}$, $\vec{\boldsymbol{o}}^{(B)}$, $\vec{\boldsymbol{v}}^{(B)}$ and $\vec{\boldsymbol{o}}^{(D)}$, $\vec{\boldsymbol{o}}^{(E)}$, $\vec{\boldsymbol{v}}^{(E)}$ are computationally indistinguishable, too. If the homomorphic encryption is semantically secure then $\vec{\boldsymbol{v}}^{(A)}$,$\vec{\boldsymbol{v}}'^{(A)}$,$\vec{\boldsymbol{v}}'^{(B)}$ and $\vec{\boldsymbol{v}}^{(D)}$,$\vec{\boldsymbol{v}}'^{(D)}$,$\vec{\boldsymbol{v}}'^{(E)}$ are computationally indistinguishable, as they contain the elements encrypted using the homomorphic encryption scheme. Moreover, in both models, the protocol outputs Λ (i.e. empty) to the adversary. Furthermore, strings $\mathsf{ID}^{(A)}$, $\mathsf{ID}^{(B)}$ and **Compute** are identical in both models, so they are computationally indistinguishable. We conclude that the adversary's outputs in both models are computationally indistinguishable.

Now we consider client B's output. We show the honest client B aborts with the same probability in both models. In the ideal model, if the cloud misbehaves, SIM_C would detect it with a high probability according to Theorem 5. In this case, it will send $\perp$ to the client and accordingly the client will abort. Note that in this case, SIM_C has not found the value β' in the intersection. In the real model, since the client knows the value β it can do the same checks as SIM_C does. So in both models, the client aborts with the same probability if the cloud misbehaves. Finally, since client A has no output, its output is identical in both models.

From the above we conclude that:

$$\left\{\mathsf{IDEAL}^{F}_{\mathsf{SIM}_{C(z)}}(\Lambda, S^{(A)}, S^{(B)})\right\}_{S^{(A)},S^{(B)}} \stackrel{c}{\equiv} \left\{\mathsf{REAL}^{\text{VD-PSI}}_{\mathcal{A}_{C(z)}}(\Lambda, S^{(A)}, S^{(B)})\right\}_{S^{(A)},S^{(B)}}.$$

Case 2: Client B is Corrupted. In this case, we consider a semi-honest adversary that controls client B. In the real execution, the joint outputs of the parties include only client B's view containing vector $\vec{\boldsymbol{t}}^{(C_3)}$, where the vector comprises the set intersection. Now we construct a simulator, SIM_B, in the ideal model that uses adversary $\mathcal{A}_B$ as a subroutine. The simulator executes as follows.

1. Invokes adversary $\mathcal{A}_B$, and receives $\vec{\boldsymbol{e}}'^{(B)}, S^{(B)}, \beta', k^{(B)}_{a'}, k^{(B)}_{b'}, k^{(B)}_{r''}, k^{(B)}_{z'}$ from it.
2. Sends $S^{(B)}$ to TTP and receives the result $f_{\cap}(S^{(A)}, S^{(B)})$. It picks two random sets $S^{(E)}$ and $S^{(D)}$, where $S^{(E)} \cap S^{(D)} = f_{\cap}(S^{(A)}, S^{(B)})$ and $|S^{(D)}|, |S^{(E)}| \leq c$. It constructs two polynomials $\tau^{(E)}(x)$ and $\tau^{(D)}(x)$ representing set $S^{(E)}$ and $S^{(D)}$, respectively.
3. Picks two uniformly random polynomials, $\omega^{(E)}(x)$ and $\omega^{(D)}(x)$ of degree $c+1$. Moreover, it picks two keys $k^{(A)}_{r'}$ and $k^{(A)}_{z'}$.
4. Computes two vectors $\vec{\boldsymbol{v}}^{(D)}$ and $\vec{\boldsymbol{v}}'^{(D)}$ whose elements are computed as follows.

$\forall i, 1 \leq i \leq n$:

$$v_i^{(D)} = (e_i'^{(B)})^{\omega^{(D)}(x_i)\cdot(r_i'^{(A)})^{-1}},$$
$$v_i'^{(D)} = (e_i'^{(B)})^{\omega^{(D)}(x_i)\cdot(-z_i'^{(A)})+a_i'},$$

where $a_i' = \mathtt{PRF}(k_{a'}^{(B)}, i)$, $r_i'^{(A)} = \mathtt{PRF}(k_{r'}^{(A)}, i)$ and $z_i'^{(A)} = \mathtt{PRF}(k_{z'}^{(A)}, i)$.

5. Computes vector $\vec{\boldsymbol{t}}'^{(C_1)}$ whose elements are computed as follows.

$$\forall i, 1 \leq i \leq n : t_i'^{(C_1)} = v_i'^{(D)} \cdot (v_i^{(D)})^{r'^{(A)}\cdot(\tau^{(D)}(x_i)+z_i'^{(A)})}$$

6. Generates vectors $\vec{\boldsymbol{v}}^{(E)}$ and $\vec{\boldsymbol{v}}'^{(E)}$ whose elements are computed as below. $\forall i, 1 \leq i \leq n$:

$$v_i^{(E)} = (e_i'^{(B)})^{\omega^{(E)}(x_i)\cdot(\tau^{(E)}(x_i)+z_i'^{(B)})}$$
$$v_i'^{(E)} = (e_i'^{(B)})^{\omega^{(E)}(x_i)\cdot(-z_i'^{(B)})+b_i'},$$

where $b_i' = \mathtt{PRF}(k_{b'}^{(B)}, i)$ and $z_i'^{(B)} = \mathtt{PRF}(k_{z'}^{(B)}, i)$.

7. Computes vector $\vec{\boldsymbol{t}}'^{(C_2)}$ comprising the following elements.

$$\forall i, 1 \leq i \leq n : t_i'^{(C_2)} = v_i^{(E)} \cdot v_i'^{(E)}.$$

8. Generates $\vec{\boldsymbol{t}}'^{(C_3)}$ containing elements $t_i'^{(C_3)}$ computed as follows.

$$\forall i, 1 \leq i \leq n : t_i'^{(C_3)} = t_i'^{(C_1)} \cdot t_i'^{(C_2)}.$$

9. Feeds $\vec{\boldsymbol{t}}'^{(C_3)}$ to $\mathcal{A}_B$. Then, it outputs whatever the adversary outputs.

Since the other parties have output Λ (i.e. empty), we only need to consider the adversary's view. In the real model, given vector $\vec{\boldsymbol{t}}^{(C_3)}$, the adversary decrypts and unblinds it to get vector $\vec{\boldsymbol{g}}$. Then, the adversary interpolates a polynomial of degree $2d+3$ of the form: $\phi(x) = \omega^{(A)}(x) \cdot \tau^{(A)}(x) + \omega^{(B)}(x) \cdot \tau^{(B)}(x) = \mu \cdot gcd(\tau^{(A)}(x), \tau^{(B)}(x))$, where polynomial $gcd(\tau^{(A)}(x), \tau^{(B)}(x))$ represents the intersection. Similarly, in the ideal model, given $\vec{\boldsymbol{t}}'^{(C_3)}$, the adversary decrypts, unblinds it and then interpolates a polynomial of degree $2d + 3$ which has form $\phi'(x) = \omega^{(D)}(x) \cdot \tau^{(D)}(x) + \omega^{(E)}(x) \cdot \tau^{(E)}(x) = \mu' \cdot gcd(\tau^{(A)}(x), \tau^{(B)}(x))$. Also, as it was proven in [75, 20], μ and μ' are uniformly random polynomials and the probability that their roots represent set elements is negligible; thus, $\phi(x)$ and $\phi'(x)$ only contain information about the set intersection

and have the same distribution in both models. Furthermore, random values β and β' have the same distribution in both models, as they are picked by a semi-honest adversary. Finally, the output, $f_{\cap}(S^{(A)}, S^{(B)})$, is identical in both views. From the above we conclude that:

$$\left\{\mathsf{IDEAL}^{F}_{\mathsf{SIM}_{B(z)}}(\Lambda, S^{(A)}, S^{(B)})\right\}_{S^{(A)},S^{(B)}} \stackrel{c}{\equiv} \left\{\mathsf{REAL}^{\text{VD-PSI}}_{\mathcal{A}_{B(z)}}(\Lambda, S^{(A)}, S^{(B)})\right\}_{S^{(A)},S^{(B)}}.$$

Case 3: Client A is Corrupted. In this case, we consider a semi-honest adversary which controls client A. We construct a simulator, SIM_A, that can simulate client A's view in the ideal model. The simulator uses adversary $\mathcal{A}_A$ as a subroutine. Then, we show that the views in the real and ideal model are indistinguishable. The simulator performs as follows.

1. Picks n values and encrypts them using the encryption scheme. The result is vector $\vec{\boldsymbol{e}}'^{(B)}$ of n encrypted values.
2. Chooses four random keys: $k^{(B)}_{a'}, k^{(B)}_{b'}, k^{(B)}_{r''}$ and $k^{(B)}_{z'}$. Also, it picks a value, β', uniformly at random from the field: $\beta' \stackrel{R}{\leftarrow} \mathbb{F}^{*}_{p}$.
3. Invokes adversary $\mathcal{A}_A$, and feeds it with $k^{(B)}_{a'}, k'^{(B)}_{b'}, k^{(B)}_{r''}, k^{(B)}_{z'}, \beta'$ and $\mathsf{ID}^{(B)}$.
4. Outputs whatever the adversary outputs.

Now we explain why the two views are indistinguishable. In the real model, keys $k^{(B)}_{a}, k^{(B)}_{b}, k^{(B)}_{r'}$ and $k^{(B)}_{z}$ are random keys for a pseudorandom function. Similarly, in the ideal model, keys $k^{(B)}_{a'}, k^{(B)}_{b'}, k^{(B)}_{r''}$ and $k^{(B)}_{z'}$ are random values for the pseudorandom function. So, the keys are computationally indistinguishable. Moreover, both β and β' are picked uniformly at random from the same field, therefore they are indistinguishable. Furthermore, the id, $\mathsf{ID}^{(B)}$, is identical in both views. Also, the client receives empty output from the protocol in both models. From the above discussion we conclude that:

$$\left\{\mathsf{IDEAL}^{F}_{\mathsf{SIM}_{A(z)}}(\Lambda, S^{(A)}, S^{(B)})\right\}_{S^{(A)},S^{(B)}} \stackrel{c}{\equiv} \left\{\mathsf{REAL}^{\text{VD-PSI}}_{\mathcal{A}_{A(z)}}(\Lambda, S^{(A)}, S^{(B)})\right\}_{S^{(A)},S^{(B)}}.$$

□

Property	VD-PSI	[68]	[82]
Private Against the Cloud	✓	✓	✓
Client-to-client Computation Authorization	✓	✓	✓
Non-interactive Client-side Setup	✓	×	✓
Secure Repeated PSI Delegation	✓	×	✓
Multiple Clients	✓	✓	✓
Verifying the Computation Integrity	✓	✓	✓
Not Using Expensive Generic Proof Systems (e.g. Zero Knowledge)	✓	✓	×
Overall Communication Complexity	$O(c)$	$O(c)$	$O(c)$
Overall Computation Complexity	$O(c)$	$O(c)$	$O(c)$
Verification Computation Complexity	$O(k)$	$O(\lambda k)$	$O(c)$

Table 6.1: Comparison of the properties of verifiable delegated PSI protocols. We denote the set cardinality upper bound by c, set intersection cardinality by k, and the security parameter by λ.

6.5 Verifiable Delegated PSI Protocol Comparison

We evaluate VD-PSI by comparing its properties to those protocols that support verifiable delegated PSI [68, 82] and preserve the privacy of the intersection in the cloud. We also compare the protocols in terms of communication, computation and verification complexity. Table 6.1 summarises the results.

Properties. All the three protocols protect clients data privacy, ensure that the computation can be carried out with the clients' consent, and only an authorized client receives the result. In VD-PSI and [82], clients can independently prepare and upload their private data, while this is not the case in [68], because it requires clients to interact with each other in order to jointly generate a key for the pseudorandom function used to encode the datasets. Both VD-PSI and [82] support secure repeated PSI delegation, so clients can prepare and upload their private data to the cloud once, but delegate the computation to it an unlimited number of times without leaking any information to the cloud. Nevertheless, [68] supports only one-off PSI delegation, so clients need to locally re-prepare their data each time they delegate the computation. Also, all the protocols support multiple clients, and allow the result recipient to verify the correctness of the computation result.

Communication Complexity. In VD-PSI, the communication complexity for client B who receives the result is $O(c)$, where c is the upper bound of set cardinality. Because client B sends to client A vector $\vec{\boldsymbol{e}}^{(B)}$ containing $n = 2c + 3$ encrypted values, in

step c.3. The communication complexity for client A who grants the computation is $O(c)$, because the client, in step c.6, sends to the cloud $\vec{\boldsymbol{v}}^{(A)}$, $\vec{\boldsymbol{v}}'^{(A)}$, $\vec{\boldsymbol{v}}'^{(B)}$, $\vec{\boldsymbol{v}}^{(B)}$ where each of the first three vectors contains n encrypted elements and the last one contains n random elements of the field. The communication complexity for the cloud is $O(c)$. As, in step d.4, it sends to client B vector $\vec{\boldsymbol{t}}^{(C_3)}$ that contains n encrypted elements. Hence, the overall communication complexity of our protocol is $O(c)$. The parties overall communication complexity in multi-client VD-PSI is also linear and similar to theirs in multi-client O-PSI.

The protocol in [68] has also $O(c)$ communication complexity. In [82], two protocols dealing with a malicious adversary are proposed. The overall communication complexity of each protocol is linear to the total number of computation inputs, so it is $O(c)$. In one of the protocols, the cloud broadcasts all the encrypted inputs to the clients, while in the other, the cloud broadcasts the hash value of the encrypted inputs.

Although all the three protocols have overall linear communication complexity, most messages in VD-PSI are ciphertext of Paillier encryption, in [82] ciphertext of fully homomorphic encryption, and in [68] ciphertext of symmetric key encryption.

Computation Complexity. Since computation complexity of VD-PSI is dominated by the exponentiation operations, we evaluate its computational cost by counting the number of such operations. Client B in step c.2 carries out n exponentiations to encrypt the elements of $\vec{\boldsymbol{e}}^{(B)}$. Furthermore, in step e.1 it performs n exponentiations to decrypt the elements of $\vec{\boldsymbol{t}}^{(C_3)}$. Client A carries out n exponentiations in steps c.4 and $2n$ exponentiations in step c.5. The cloud carries out $2n$ exponentiations in step d.1, $2n$ exponentiations in step d.2, and n exponentiations in step d.3. In total, $10n$ exponentiation operations are carried out, so the overall computation complexity is $O(c)$. In VD-PSI, the verifier only checks whether β is among the elements of the intersection. Therefore, the verification computation complexity involves at most $O(k)$ comparison operations. The parties overall computation complexity in multi-client VD-PSI is linear too and it is similar to the parties computation complexity in multi-client O-PSI.

The computation complexity of the two protocols dealing with a malicious adversary in [82] is dominated by fully homomorphic encryption operations. In each protocol, the overall number of such operations is linear to the size of the inputs, $O(c)$. Note, the protocols are designed for generic computation and the exact computation complexity for PSI is not clear. Therefore, the overall computation complexity for each protocol is at least $O(c)$. Moreover, in order for each client to verify the com-

putation correctness, it needs to access all the (encrypted) inputs and perform generic proof system operations linear to the number of inputs. Therefore, the verification complexity at the verifier-side is $O(c)$, too. While, in [68] each participant's overall computation complexity is $O(c)$. In order for the client to verify the integrity of the result, it checks whether λ copies of all intersection elements exist in the result. So, its verification complexity is $O(\lambda k)$ where λ and k are the security parameter and intersection cardinality, respectively.

Thus, the overall computation complexity of all the schemes is linear, where VD-PSI uses Paillier encryption, [82] uses fully homomorphic and [68] uses symmetric key encryption. The verification mechanisms in [82] is based on expensive generic proof systems, while [68] and VD-PSI utilize lightweight mechanisms.

It should be noted that, in VD-PS, after client outsources its dataset, it has to keep locally only two secret keys. During the computation, the client who is interested in the result generates four values that it needs to keep until it retrieves the result. At the end of the protocol, it can discard the four values. In contrast, a variant of the protocol in [82] requires clients to have the hash value of their encrypted inputs for verification. This introduces storage overhead linear to the number of its outsourced inputs. Similar to VD-PSI, clients in [68] need to locally store only the secret keys for a pseudorandom function in order to verify the computation result.

Moreover, we highlight that even though O-PSI and VD-PSI have the same (communication and) computation complexities, O-PSI is more efficient than VD-PSI. The reason is that in VD-PSI, in the computation delegation phase, client A sends four vectors to the cloud, while in O-PSI it sends only one vector. Therefore, in VD-PSI, the cloud receives more messages from the clients and it performs more homomorphic operations on them. So its computation cost is more than its cost in O-PSI.

In conclusion, although [68] is faster than the other two protocols, VD-PSI enjoys two properties that [68] lacks. First, it allows clients to outsource their datasets once but verifiably delegate the computation to the cloud an unlimited number of times (i.e. it supports secure repeated PSI delegation). Second, it supports non-interactive setup at the client-side. As we explained in chapter 1, these properties are vital in the real world cloud computing setting. Compared to [82], VD-PSI offers the same security properties much more efficiently.

6.6 Concluding Remarks

In this chapter, we proposed VD-PSI, a protocol that supports secure repeated PSI delegation and also allows clients to efficiently verify the correctness of the computation result that the cloud generates. As we have shown, the protocol also protects the confidentiality of the outsourced data and the computation result. The protocol allows clients to independently outsource their private data to the cloud, and later on verifiably delegate the computation of PSI to it. VD-PSI enables a result recipient to efficiently verify the correctness of the result, without the need to know its own outsourced dataset and the other clients' datasets. VD-PSI offers a combination of features that allows businesses to get the full benefits of the cloud in a more cost-effective way.

Chapter 7

Conclusions

7.1 Contributions

Outsourcing data storage and computation to the cloud are becoming common practice. PSI is a key protocol with a broad range of applications that parties want to outsource. Nonetheless, existing PSI protocols cannot be run securely on outsourced private data. In particular, traditional PSI protocols require parties to have their sets locally and jointly compute the result. Furthermore, those PSI protocols designed to take advantage of the cloud's computation capabilities, support only one-off PSI delegation, so clients need to either download, locally re-prepare, and upload the data each time the computation is delegated or have a local copy of the data. Moreover, those protocols that support repeated PSI delegation are either very inefficient or not fully private and leak information to the cloud. Therefore, there is a pressing need for efficient protocols that enable the clients to run PSI on their outsourced data while protecting data privacy from the cloud.

The main contribution of this thesis is the design of four protocols that support repeated PSI delegation. Our first protocol, O-PSI [1], enables multiple clients to independently store their private data in the cloud. Later on, the clients can get together and delegate to the cloud the computation of PSI on their private data. The protocol also allows the outsourced sets to be utilized securely an unlimited number of times without the need to download and re-prepare them again. In this process, the cloud cannot figure out the clients' set elements, the intersection and the intersection cardinality, and also the result recipient cannot learn anything beyond the intersection. The protocol enables the clients to prepare, and outsource their data independently without interacting with each other or having any knowledge of others. Our protocol ensures

that the intersections can only be computed with the clients' consent. The protocol uses a single cloud and it does not involve a trusted third party.

Our second protocol, EO-PSI [3], is more efficient than O-PSI, as it does not use any public key encryption scheme and the result recipient can retrieve the intersection faster than it can do in O-PSI. We implemented both protocols and analyzed their performance. The analysis indicates that EO-PSI is 1-2 orders of magnitude faster than O-PSI. The third protocol is UEO-PSI, a protocol that supports dynamic data, and allows clients to update their outsourced data with low communication and computation costs.

We also proposed VD-PSI [2], a protocol that lets clients efficiently verify the integrity of the computation delegated to the cloud. The protocol considers scenarios where the cloud may arbitrarily misbehave (e.g. deviate from the protocol, tamper with the clients' outsourced data, the computation result) and provide an incorrect result. The main novelty of this protocol is a lightweight verification mechanism.

7.2 Directions for Future Research

One direction of future research is to improve the performance of the protocols. One possible technique is to use permutation-based hashing [95] to reduce the bit-length of the set elements. In the following, we briefly explain the technique and our suggestion. Let $e = e_1||e_2$ be the bit representation of element e, $\mathtt{H}$ be a hash function and $\oplus$ be the XOR operation. To insert element e into the hash table, we compute its address as: $j = e_1 \oplus \mathtt{H}(e_2)$, and store e_2 in that address. Thus, instead of inserting the entire element, we store a shorter representation of it in the table. Permutation-based hashing allows us to use a smaller field to represent elements. With a smaller field, the polynomial operations, especially factorization, would be faster.

Another possible future research direction is to extend UEO-PSI protocol in a way that can support computation verification without leveraging any public key encryption scheme (i.e. a combination of VD-PSI and UEO-PSI that does not use a public key encryption scheme). One way to do is to use a (non-colluding) proxy server that is not necessarily trusted. In this setting, the clients distribute the messages, sent in the computation delegation phase, over the cloud and the proxy server. This will eliminate the need for a public key encryption scheme used in VD-PSI and prevent the cloud from learning information from the messages sent to it. In the new variant, the client

will be able to securely update its outsourced data, delegate PSI to the cloud and efficiently verify the computation result. To verify the integrity of a bin's contents, the client can pick random value β, encode it and send it to the cloud. The cloud combines the encoded message with the content of the related bin and sends the result back to the client. Given the result, the client decodes it, interpolates a polynomial and evaluates the polynomial at β. If the result is zero, the bin's content is intact, otherwise it has been tampered with.

Another potential direction is to design delegated privacy preserving protocols that support more set operations on outsourced private data, such as delegated set union that has applications such as data collection for statistics. For example, a research group wants to collect information from different hospitals about patients with a sparse disease, while the hospitals want to protect their data privacy [63] and do not want to reveal what parts of the information belong to them. A combination of polynomial properties and homomorphic encryption (e.g. partially or somewhat homomorphic encryption) will be a suitable building block for the delegated set union protocol.

Furthermore, strengthening the security of the protocols is another direction for future research. In particular, we can improve our protocols to withstand malicious clients (in addition to the malicious cloud) who may deviate from the protocol. Techniques like zero-knowledge proofs could help in dealing with malicious parties but impose high costs. Designing mechanisms that could provide similar guarantees in a more efficient manner would be desirable. Moreover, we could extend the protocols to deal with dynamic adversaries who are stronger than the static ones which we considered in this research.

References

[1] AYDIN ABADI, SOTIRIOS TERZIS, AND CHANGYU DONG. O-PSI: delegated private set intersection on outsourced datasets. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Germany.*, pages 3–17, 2015. 6, 47, 133

[2] AYDIN ABADI, SOTIRIOS TERZIS, AND CHANGYU DONG. VD-PSI: verifiable delegated private set intersection on outsourced private datasets. In *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Barbados.*, 2016. 7, 110, 134

[3] AYDIN ABADI, SOTIRIOS TERZIS, ROBERTO METERE, AND CHANGYU DONG. Efficient delegated private set intersection on outsourced private datasets. *IEEE Transactions on Dependable and Secure Computing*, Submitted on July 18, 2016. 6, 48, 134

[4] ALFRED V. AHO AND JOHN E. HOPCROFT. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974. 18

[5] YURIY ARBITMAN, MONI NAOR, AND GIL SEGEV. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 787–796, 2010. 36

[6] GILAD ASHAROV, YEHUDA LINDELL, THOMAS SCHNEIDER, AND MICHAEL ZOHNER. More efficient oblivious transfer and extensions for faster secure computation. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 535–548, 2013. 34, 35

[7] GIUSEPPE ATENIESE, RANDAL C. BURNS, REZA CURTMOLA, JOSEPH HERRING, LEA KISSNER, ZACHARY N. J. PETERSON, AND DAWN XIAODONG SONG. Provable data possession at untrusted stores. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS'07, Alexandria, Virginia, USA, October 28-31, 2007*, pages 598–609, 2007. 4

[8] GIUSEPPE ATENIESE, EMILIANO DE CRISTOFARO, AND GENE TSUDIK. (if) size matters: Size-hiding private set intersection. In *Public Key Cryptography - PKC 2011 - 14th International Conference on Practice and Theory in Public Key Cryptography, Taormina, Italy, March 6-9, 2011. Proceedings*, pages 156–173, 2011. 2

[9] GIUSEPPE ATENIESE, KEVIN FU, MATTHEW GREEN, AND SUSAN HOHENBERGER. Improved proxy re-encryption schemes with applications to secure distributed storage. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005. 43

[10] PIERRE BALDI, ROBERTA BARONIO, EMILIANO DE CRISTOFARO, PAOLO GASTI, AND GENE TSUDIK. Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 691–702, 2011. 2

[11] MAURO BARNI, PIERLUIGI FAILLA, VLADIMIR KOLESNIKOV, RICCARDO LAZZERETTI, AHMAD-REZA SADEGHI, AND THOMAS SCHNEIDER. Secure evaluation of private linear branching programs with medical applications. In *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, pages 424–439, 2009. 21

[12] MIHIR BELLARE AND PHILLIP ROGAWAY. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73, 1993. 30

[13] RIKKE BENDLIN, IVAN DAMGÅRD, CLAUDIO ORLANDI, AND SARAH ZAKARIAS. Semi-homomorphic encryption and multiparty computation. In *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Con-*

ference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings, pages 169–188, 2011. 116

[14] PETRA BERENBRINK, ARTUR CZUMAJ, ANGELIKA STEGER, AND BERTHOLD VÖCKING. Balanced allocations: the heavily loaded case. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 745–754, 2000. 20

[15] SAUL J BERMAN, LYNN KESTERSON-TOWNES, ANTHONY MARSHALL, AND ROHINI SRIVATHSA. How cloud computing enables process and business model innovation. *Strategy & Leadership*, pages 27–35, 2012. 1

[16] JEAN-PAUL BERRUT AND LLOYD N. TREFETHEN. Barycentric lagrange interpolation. *SIAM Review*, pages 501–517, 2004. 18

[17] ERROL A BLAKE. Network and database security: Regulatory compliance, network, and database security-a unified process and goal. *The Journal of Digital Forensics, Security and Law: JDFSL*, page 77, 2007. 2

[18] BURTON H. BLOOM. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, pages 422–426, 1970. 33

[19] SONIA BOGOS, JOHN GASPOZ, AND SERGE VAUDENAY. Cryptanalysis of a homomorphic encryption scheme. Cryptology ePrint Archive, Report 2016/775, 2016. 12

[20] DAN BONEH, CRAIG GENTRY, SHAI HALEVI, FRANK WANG, AND DAVID J. WU. Private database queries using somewhat homomorphic encryption. In *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, pages 102–118, 2013. 16, 55, 60, 72, 104, 127

[21] DAN BONEH, EU-JIN GOH, AND KOBBI NISSIM. Evaluating 2-dnf formulas on ciphertexts. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, pages 325–341, 2005. 39

[22] JOSHUA BRODY, AMIT CHAKRABARTI, RANGANATH KONDAPALLY, DAVID P. WOODRUFF, AND GRIGORY YAROSLAVTSEV. Beyond set disjoint-

ness: the communication complexity of finding the intersection. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 106–113, 2014. 1

[23] RAN CANETTI, OMER PANETH, DIMITRIOS PAPADOPOULOS, AND NIKOS TRIANDOPOULOS. Verifiable set operations over outsourced databases. In *17th IACR International Conference on Theory and Practice of Public-Key Cryptography*, pages 113–130, 2014. 1

[24] FRED H CATE. Eu data protection directive, information privacy, and the public interest, the. *Iowa Law Review*, page 431, 1994. 2

[25] SANJIT CHATTERJEE, SAYANTAN MUKHERJEE, AND GOVIND PATIDAR. Efficient protocol for authenticated email search. In *Security, Privacy, and Applied Cryptography Engineering - 5th International Conference, SPACE 2015, Jaipur, India, October 3-7, 2015, Proceedings*, pages 1–20, 2015. 1

[26] DAVID CHAUM. Blind signatures for untraceable payments. In *Advances in Cryptology: Proceedings of CRYPTO '82, Santa Barbara, California, USA, August 23-25, 1982.*, pages 199–203, 1982. 32

[27] DAVID CHAUM AND TORBEN P. PEDERSEN. Wallet databases with observers. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, pages 89–105, 1992. 30

[28] BO CHEN AND REZA CURTMOLA. Auditable version control systems. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014. 4

[29] SEUNG GEOL CHOI, JONATHAN KATZ, RANJIT KUMARESAN, AND CARLOS CID. Multi-client non-interactive verifiable computation. In *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, pages 499–518, 2013. 56

[30] JEAN-SÉBASTIEN CORON, JACQUES PATARIN, AND YANNICK SEURIN. The random oracle model and the ideal cipher model are equivalent. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference,*

Santa Barbara, CA, USA, August 17-21, 2008. Proceedings, pages 1–20, 2008. 30

[31] EMILIANO DE CRISTOFARO, PAOLO GASTI, AND GENE TSUDIK. Fast and private computation of cardinality of set intersection and union. In *Cryptology and Network Security, 11th International Conference, CANS 2012, Darmstadt, Germany, December 12-14, 2012. Proceedings*, pages 218–231, 2012. 2

[32] EMILIANO DE CRISTOFARO, JIHYE KIM, AND GENE TSUDIK. Linear-complexity private set intersection protocols secure in malicious model. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security*, pages 213–231, 2010. 32

[33] EMILIANO DE CRISTOFARO AND GENE TSUDIK. Practical private set intersection protocols with linear complexity. In *14th International Conference on Financial Cryptography and Data Security*, pages 143–159, 2010. 28, 32

[34] DANA DACHMAN-SOLED, TAL MALKIN, MARIANA RAYKOVA, AND MOTI YUNG. Efficient robust private set intersection. In *Applied Cryptography and Network Security, 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings*, pages 125–142, 2009. 2

[35] IVAN DAMGÅRD AND MADS JURIK. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proceedings*, pages 119–136, 2001. 30

[36] IVAN DAMGÅRD AND NIKOS TRIANDOPOULOS. Supporting non-membership proofs with bilinear-map accumulators. *IACR Cryptology ePrint Archive*, page 538, 2008. 43

[37] CHANGYU DONG, LIQUN CHEN, JAN CAMENISCH, AND GIOVANNI RUSSELLO. Fair private set intersection with a semi-trusted arbiter. In *Data and Applications Security and Privacy XXVII - 27th Annual IFIP WG 11.3 Conference, DBSec 2013, USA.*, pages 128–144, 2013. 15, 17

[38] CHANGYU DONG, LIQUN CHEN, AND ZIKAI WEN. When private set intersection meets big data: an efficient and scalable protocol. In *20th ACM Conference on Computer and Communications Security*, pages 789–800, 2013. 2, 32, 33, 34, 35, 77

[39] ZEKERIYA ERKIN, MARTIN FRANZ, JORGE GUAJARDO, STEFAN KATZENBEISSER, INALD LAGENDIJK, AND TOMAS TOFT. Privacy-preserving face recognition. In *Privacy Enhancing Technologies, 9th International Symposium, PETS 2009, Seattle, WA, USA, August 5-7, 2009. Proceedings*, pages 235–253, 2009. 48

[40] SHIMON EVEN, ODED GOLDREICH, AND ABRAHAM LEMPEL. A randomized protocol for signing contracts. *Commun. ACM*, pages 637–647, 1985. 33

[41] A. FERDOWSI. The dropbox blog: Yesterdays authentication bug @ . https://blog.dropbox.com/?p=821, 2011. 2, 28

[42] DARIO FIORE, ROSARIO GENNARO, AND VALERIO PASTRO. Efficiently verifiable computation on encrypted data. In *21st ACM Conference on Computer and Communications Security, Scottsdale, AZ, USA*, pages 844–855, 2014. 1

[43] MARC FISCHLIN. Pseudorandom function tribe ensembles based on one-way permutations: Improvements and applications. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, pages 432–445, 1999. 14

[44] PIERRE-ALAIN FOUQUE, GUILLAUME POUPARD, AND JACQUES STERN. Sharing decryption in the context of voting or lotteries. In *Financial Cryptography, 4th International Conference, FC 2000 Anguilla, British West Indies, February 20-24, 2000, Proceedings*, pages 90–104, 2000. 31

[45] MICHAEL J. FREEDMAN, CARMIT HAZAY, KOBBI NISSIM, AND BENNY PINKAS. Efficient set intersection with simulation-based security. *J. Cryptology*, pages 115–155, 2016. 30, 118

[46] MICHAEL J. FREEDMAN, KOBBI NISSIM, AND BENNY PINKAS. Efficient private matching and set intersection. In *EUROCRYPT 2004, International Confer-*

ence on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, pages 1–19, 2004. 2, 15, 17, 28, 29, 48, 53

[47] TAHER EL GAMAL. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, pages 10–18, 1984. 12

[48] CRAIG GENTRY. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178, 2009. 12

[49] CRAIG GENTRY, SHAI HALEVI, AND NIGEL P. SMART. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 850–867, 2012. 12

[50] ESHA GHOSH, OLGA OHRIMENKO, DIMITRIOS PAPADOPOULOS, ROBERTO TAMASSIA, AND NIKOS TRIANDOPOULOS. Zero-knowledge accumulators and set operations. *IACR Cryptology ePrint Archive*, page 404, 2015. 1

[51] ODED GOLDREICH. *The Foundations of Cryptography - Volume 1, Basic Techniques*. Cambridge University Press, 2001. 9

[52] ODED GOLDREICH. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004. 21, 22, 43, 55, 98, 121

[53] SHAFI GOLDWASSER AND SILVIO MICALI. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 365–377, 1982. 12

[54] MICHAEL T. GOODRICH, DUY NGUYEN, OLGA OHRIMENKO, CHARALAMPOS PAPAMANTHOU, ROBERTO TAMASSIA, NIKOS TRIANDOPOULOS, AND CRISTINA VIDEIRA LOPES. Efficient verification of web-content searching through authenticated web crawlers. *PVLDB*, pages 920–931, 2012. 1

[55] S. DOV GORDON, JONATHAN KATZ, FENG-HAO LIU, ELAINE SHI, AND HONG-SHENG ZHOU. Multi-client verifiable computation with stronger secu-

rity guarantees. In *Theory of Cryptography - 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II*, pages 144–168, 2015. 56

[56] S. DOV GORDON, JONATHAN KATZ, FENG-HAO LIU, ELAINE SHI, AND HONG-SHENG ZHOU. Multi-client verifiable computation with stronger security guarantees. In *12th Theory of Cryptography Conference, TCC , Poland.*, pages 144–168, 2015. 118

[57] THE GUARDIAN. Playstation network hack: Why it took sony seven days to tell the world @ http://www.guardian.co.uk/technology/gamesblog/2011/apr/27/playstation-network-hack-sony, 2011. 2, 28

[58] HAKAN HACIGÜMÜS, SHARAD MEHROTRA, AND BALAKRISHNA R. IYER. Providing database as a service. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 29–38, 2002. 1

[59] FLORIAN HAHN AND FLORIAN KERSCHBAUM. Searchable encryption with secure and efficient updates. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 310–320, 2014. 98

[60] CARMIT HAZAY AND YEHUDA LINDELL. *Efficient Secure Two-Party Protocols - Techniques and Constructions*. Information Security and Cryptography. Springer, 2010. 2, 21

[61] CARMIT HAZAY AND MUTHURAMAKRISHNAN VENKITASUBRAMANIAM. On the power of secure two-party computation. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, pages 397–429, 2016. 21

[62] WILKO HENECKA, STEFAN KÖGL, AHMAD-REZA SADEGHI, THOMAS SCHNEIDER, AND IMMO WEHRENBERG. TASTY: tool for automating secure two-party computations. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 451–462, 2010. 48

[63] JEONGDAE HONG, JUNG WOO KIM, JIHYE KIM, KUNSOO PARK, AND JUNG HEE CHEON. Constant-round privacy preserving multiset union. *IACR Cryptology ePrint Archive*, page 138, 2011. 135

[64] MOHAMMAD SAIFUL ISLAM, MEHMET KUZU, AND MURAT KANTARCIOGLU. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012. 2, 28

[65] S. ISLAM, M. OUEDRAOGO, C. KALLONIATIS, H. MOURATIDIS, AND S. GRITZALIS. Assurance of security and privacy requirements for cloud deployment model. *IEEE Transactions on Cloud Computing*, pages 1–1, 2015. 27

[66] MAHESH KALLAHALLA, ERIK RIEDEL, RAM SWAMINATHAN, QIAN WANG, AND KEVIN FU. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*, 2003. 5

[67] SENY KAMARA, PAYMAN MOHASSEL, AND MARIANA RAYKOVA. Outsourcing multi-party computation. *IACR Cryptology ePrint Archive*, page 272, 2011. 12, 55, 98, 121

[68] SENY KAMARA, PAYMAN MOHASSEL, MARIANA RAYKOVA, AND SAEED SADEGHIAN. Scaling private set intersection to billion-element sets. In *18th International Conference on Financial Cryptography and Data Security*, pages 863–874, 2014. 2, 28, 40, 41, 55, 72, 73, 74, 75, 76, 118, 129, 130, 131

[69] SENY KAMARA AND CHARALAMPOS PAPAMANTHOU. Parallel and dynamic searchable symmetric encryption. In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, pages 258–274, 2013. 12, 98

[70] MURAT KANTARCIOGLU. A survey of privacy-preserving methods across horizontally partitioned data. In *Privacy-Preserving Data Mining - Models and Algorithms*, pages 313–335. 2008. 1

[71] JONATHAN KATZ AND YEHUDA LINDELL. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007. 11, 14, 30, 40, 43

[72] KIRAN S. KEDLAYA AND CHRISTOPHER UMANS. Fast polynomial factorization and modular composition. *SIAM J. Comput.*, pages 1767–1802, 2011. 17

[73] FLORIAN KERSCHBAUM. Collusion-resistant outsourcing of private set intersection. In *27th ACM Symposium on Applied Computing, Riva, Trento, Italy*, pages 1451–1456, 2012. 37, 72, 73, 74, 76, 77

[74] FLORIAN KERSCHBAUM. Outsourced private set intersection using homomorphic encryption. In *Computer and Communications Security, ASIACCS '12.*, pages 85–86, 2012. 39, 72, 73, 74, 76

[75] LEA KISSNER AND DAWN XIAODONG SONG. Privacy-preserving set operations. In *CRYPTO 2005, 25th International Cryptology Conference*, pages 241–257, 2005. 15, 16, 17, 28, 30, 31, 53, 60, 67, 72, 75, 96, 104, 118, 120, 127

[76] DONALD E. KNUTH. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981. 15, 18

[77] VLADIMIR KOLESNIKOV, RANJIT KUMARESAN, AND ABDULLATIF SHIKFA. Efficient verification of input consistency in server-assisted secure function evaluation. In *Cryptology and Network Security, 11th International Conference, CANS 2012, Darmstadt, Germany, December 12-14, 2012. Proceedings*, pages 201–217, 2012. 55

[78] AHMED E. KOSBA, DIMITRIOS PAPADOPOULOS, CHARALAMPOS PAPAMANTHOU, MAHMOUD F. SAYED, ELAINE SHI, AND NIKOS TRIANDOPOULOS. TRUESET: faster verifiable set computations. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 765–780, 2014. 1, 15

[79] YEHUDA LINDELL AND BENNY PINKAS. Privacy preserving data mining. In *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, pages 36–54, 2000. 48

[80] YEHUDA LINDELL AND BENNY PINKAS. Secure two-party computation via cut-and-choose oblivious transfer. *J. Cryptology*, **25**[4]:680–722, 2012. 118

[81] FANG LIU, WEE KEONG NG, WEI ZHANG, DO HOANG GIANG, AND SHUGUO HAN. Encrypted set intersection protocol for outsourced datasets. In *IEEE International Conference on Cloud Engineering*, IC2E '14, pages 135–140, Washington, DC, USA, 2014. IEEE Computer Society. 42, 44, 72, 73, 74, 75, 76, 118

[82] ADRIANA LÓPEZ-ALT, ERAN TROMER, AND VINOD VAIKUNTANATHAN. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Symposium on Theory of Computing Conference, USA.*, pages 1219–1234, 2012. 45, 46, 76, 77, 105, 106, 107, 108, 109, 129, 130, 131

[83] SEAN MARSTON, ZHI LI, SUBHAJYOTI BANDYOPADHYAY, AND ANAND GHALSASI. Cloud computing - the business perspective. In *44th Hawaii International International Conference on Systems Science (HICSS-44 2011), USA*, pages 1–11, 2011. 1, 28

[84] M. A. H. MASUD, J. YONG, AND X. HUANG. Cloud computing for higher education: A roadmap. In *2012 IEEE 16th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 552–557, May 2012. 28

[85] LUCA MELIS, GEORGE DANEZIS, AND EMILIANO DE CRISTOFARO. Efficient private statistics with succinct sketches. *CoRR*, 2015. 45

[86] GHITA MEZZOUR, ADRIAN PERRIG, VIRGIL D. GLIGOR, AND PANOS PAPADIMITRATOS. Privacy-preserving relationship path discovery in social networks. In *Cryptology and Network Security, 8th International Conference, CANS 2009, Kanazawa, Japan, December 12-14, 2009. Proceedings*, pages 189–208, 2009. 1

[87] T. MOYO AND J. BHOGAL. Investigating security issues in cloud computing. In *Complex, Intelligent and Software Intensive Systems (CISIS), 2014 Eighth International Conference on*, pages 141–146, July 2014. 28

[88] ARVIND NARAYANAN, NARENDRAN THIAGARAJAN, MUGDHA LAKHANI, MICHAEL HAMBURG, AND DAN BONEH. Location privacy via private proximity testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011. 15

[89] LAN NGUYEN. Accumulators from bilinear pairings and applications. In *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings*, pages 275–292, 2005. 43

[90] JESPER BUUS NIELSEN, PETER SEBASTIAN NORDHOLT, CLAUDIO ORLANDI, AND SAI SHESHANK BURRA. A new approach to practical active-secure two-party computation. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 681–700, 2012. 35

[91] PASCAL PAILLIER. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic*, pages 223–238, 1999. 12

[92] DIMITRIOS PAPADOPOULOS, STAVROS PAPADOPOULOS, AND NIKOS TRIANDOPOULOS. Taking authenticated range queries to arbitrary dimensions. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 819–830, 2014. 1

[93] BRYAN PARNO, MARIANA RAYKOVA, AND VINOD VAIKUNTANATHAN. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *Theory of Cryptography - 9th Theory of Cryptography Conference, TCC 2012, Taormina, Sicily, Italy, March 19-21, 2012. Proceedings*, pages 422–439, 2012. 4

[94] ANDREAS PETER, ERIK TEWS, AND STEFAN KATZENBEISSER. Efficiently outsourcing multiparty computation under multiple keys. *IEEE Trans. Information Forensics and Security*, pages 2046–2058, 2013. 45

[95] 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, Washington, D.C., USA, August 12-14, 2015.*, pages 515–530, 2015. 21, 32, 36, 134

[96] BENNY PINKAS, THOMAS SCHNEIDER, AND MICHAEL ZOHNER. Faster private set intersection based on OT extension. In *23rd USENIX Security Symposium, San Diego, CA, USA*. USENIX, 2014. 2, 21, 32, 34, 36, 77

[97] TINA PIPER. *Dalhousie Law Journal*, page 253, 2000. 2

[98] RALUCA ADA POPA, EMILY STARK, STEVEN VALDEZ, JONAS HELFER, NICKOLAI ZELDOVICH, AND HARI BALAKRISHNAN. Building web applications on top of encrypted data using mylar. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 157–172, 2014. 45

[99] S. QIU, J. LIU, Y. SHI, M. LI, AND W. WANG. Identity-based private matching over outsourced encrypted datasets. *Cloud Computing, IEEE Transactions on*, pages 1–1, 2015. 44, 72, 73, 74, 75, 76, 118

[100] MARTIN RAAB AND ANGELIKA STEGER. Balls into bins - A simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science, Second International Workshop, RANDOM'98, Barcelona, Spain*, pages 159–170, 1998. 20

[101] MICHAEL O RABIN. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, page 187, 2005. 33

[102] MARIANA RAYKOVA, BINH VO, STEVEN M. BELLOVIN, AND TAL MALKIN. Secure anonymous database search. In *First ACM Cloud Computing Security Workshop, Chicago, IL, USA*, pages 115–126, 2009. 55

[103] THOMAS RISTENPART, ERAN TROMER, HOVAV SHACHAM, AND STEFAN SAVAGE. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009. 2, 28

[104] RONALD L RIVEST, LEN ADLEMAN, AND MICHAEL L DERTOUZOS. On data banks and privacy homomorphisms. *Foundations of secure computation*, pages 169–180, 1978. 12

[105] RONALD L. RIVEST, ADI SHAMIR, AND LEONARD M. ADLEMAN. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, pages 120–126, 1978. 12, 38

[106] BRUCE SCHNEIER. *Applied cryptography - protocols, algorithms, and source code in C (2. ed.)*. Wiley, 1996. 33

[107] ELAINE SHI, T.-H. HUBERT CHAN, EMIL STEFANOV, AND MINGFEI LI. Oblivious RAM with o((logn)3) worst-case cost. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, pages 197–214, 2011. 20

[108] NIGEL P. SMART AND FREDERIK VERCAUTEREN. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography - PKC 2010, 13th International Conference on Practice and Theory in Public Key Cryptography, Paris, France, May 26-28, 2010. Proceedings*, pages 420–443, 2010. 12

[109] NIGEL P. SMART AND FREDERIK VERCAUTEREN. Fully homomorphic SIMD operations. *Des. Codes Cryptography*, pages 57–81, 2014. 12

[110] EMIL STEFANOV, CHARALAMPOS PAPAMANTHOU, AND ELAINE SHI. Practical dynamic searchable encryption with small leakage. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014. 98

[111] EMIL STEFANOV AND ELAINE SHI. Multi-cloud oblivious storage. In *20th ACM Conference on Computer and Communications Security, Berlin, Germany*, pages 247–258, 2013. 55

[112] SUBASHINI SUBASHINI AND VEERARUNA KAVITHA. A survey on security issues in service delivery models of cloud computing. *Journal of network and computer applications*, pages 1–11, 2011. 27

[113] MARTEN VAN DIJK, CRAIG GENTRY, SHAI HALEVI, AND VINOD VAIKUNTANATHAN. Fully homomorphic encryption over the integers. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, pages 24–43, 2010. 12

[114] MARTEN VAN DIJK AND ARI JUELS. On the impossibility of cryptography alone for privacy-preserving cloud computing. In *5th USENIX Workshop on*

Hot Topics in Security, HotSec'10, Washington, D.C., USA, August 10, 2010, 2010. 12

[115] PETER VAN LIESDONK, SAEED SEDGHI, JEROEN DOUMEN, PIETER H. HARTEL, AND WILLEM JONKER. Computationally efficient searchable symmetric encryption. In *Secure Data Management, 7th VLDB Workshop, SDM 2010, Singapore, September 17, 2010. Proceedings*, pages 87–100, 2010. 98

[116] ALKA VARMA CITRIN, DAVID E SPROTT, STEVEN N SILVERMAN, AND DONALD E STEM JR. Adoption of internet shopping: the role of consumer innovativeness. *Industrial management & data systems*, pages 294–300, 2000. 5

[117] TOBY VELTE, ANTHONY VELTE, AND ROBERT ELSENPETER. *Cloud Computing, A Practical Approach*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010. 26, 27

[118] BOYANG WANG, MING LI, SHERMAN S. M. CHOW, AND HUI LI. A tale of two clouds: Computing on data encrypted under multiple keys. In *IEEE Conference on Communications and Network Security, CNS 2014, San Francisco, CA, USA, October 29-31, 2014*, pages 337–345, 2014. 55

[119] ZHANG XU, HAINING WANG, ZICHEN XU, AND XIAORUI WANG. Power attack: An increasing threat to data centers. In *The Proceedings of the 2014 Network and Distributed System Security Symposium, NDSS*, 2014. 2, 28

[120] YU YANG, CHENGGUI ZHAO, AND TILEI GAO. Cloud computing: Security issues overview and solving techniques investigation. In *Intelligent Cloud Computing - First International Conference, ICC 2014, Muscat, Oman, February 24-26, 2014, Revised Selected Papers*, pages 152–167, 2014. 28

[121] XUN YI, RUSSELL PAULET, AND ELISA BERTINO. *Homomorphic Encryption and Applications*. Springer Briefs in Computer Science. Springer, 2014. 11

[122] YINQIAN ZHANG, ARI JUELS, MICHAEL K. REITER, AND THOMAS RISTENPART. Cross-vm side channels and their use to extract private keys. In *19th ACM Conference on Computer and Communications Security*, pages 305–316, 2012. 2, 28

[123] QINGJI ZHENG AND SHOUHUAI XU. Verifiable delegated set intersection operations on outsourced encrypted data. *IACR Cryptology ePrint Archive*, page 178, 2014. 43, 44, 45, 72, 73, 74, 75, 76, 118

Appendix A

O-PSI Implementation Class Diagram

As it is shown in Fig A.1, the O-PSI implementation has four classes `Client`, `Server`, `Polynomial` and `Random`. Furthermore, the protocol implementation contains three structures carrying the set of messages exchanged between the parties. The structures are: (1) `CompPerm_Request`, (2) `GrantComp_Info`, and (3) `Server_Result`.

A. O-PSI Implementation Class Diagram

Figure A.1: O-PSI protocol implementation class diagram

Appendix B

EO-PSI Implementation Class Diagram

As it is depicted in Fig B.1, the EO-PSI implementation has five classes `Client`, `Server`, `Polynomial`, `Hashtable` and `Random`. Moreover, the implementation involves four structures: (1) `CompPerm_Request`, (2) `GrantComp_Info`, (3) `Server_Result`, and (4) `Client_Dataset`.

B. EO-PSI Implementation Class Diagram

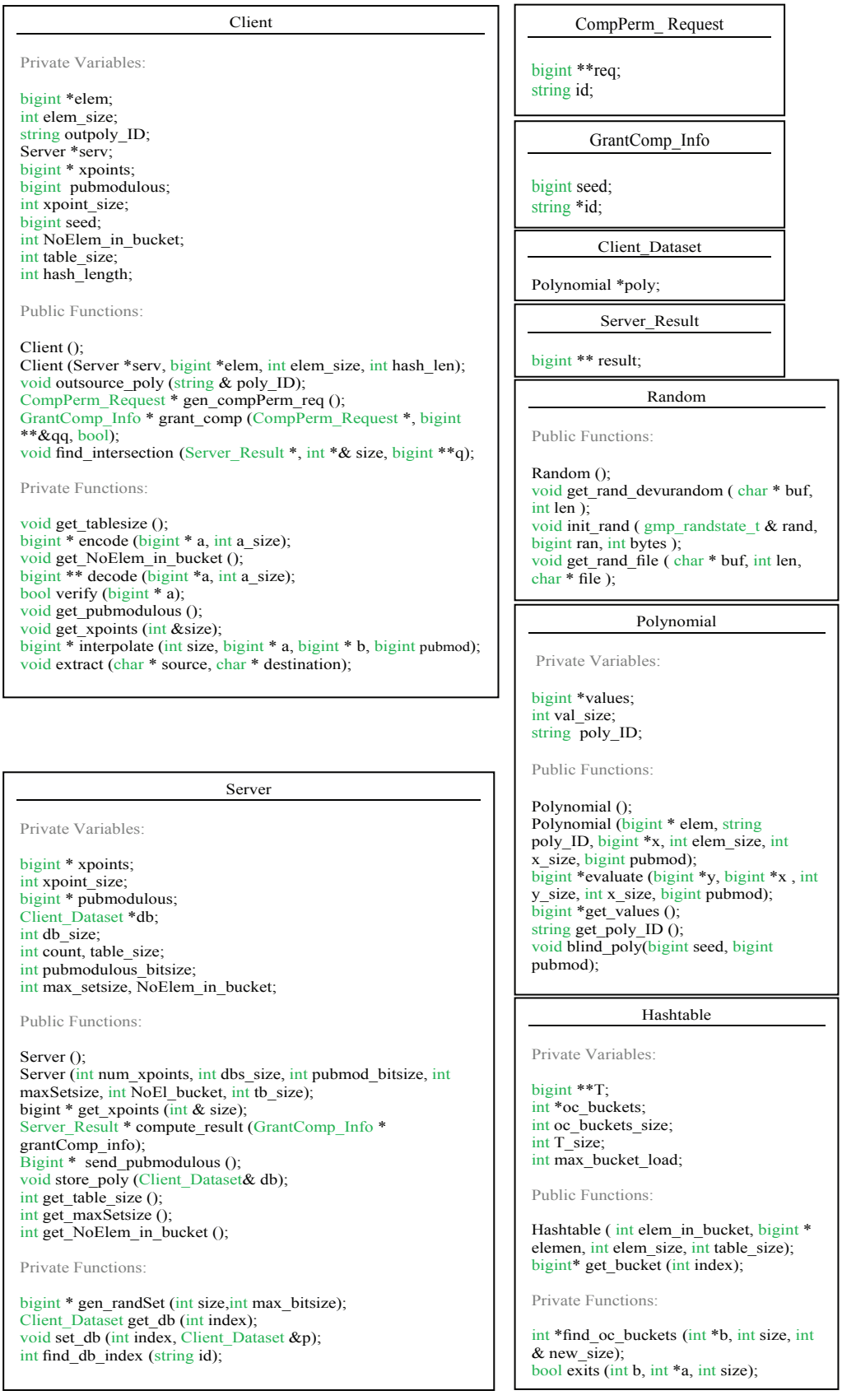

Figure B.1: EO-PSI protocol implementation class diagram