UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Multiparty Protocols for Tree Classifiers

Anselme Kemgne Tueno

Dissertation eingereicht an der Fakultät für Informatik und Mathematik der Universität Passau zur Erlangung des Grades eines Doktors der Naturwissenschaften

Vorsitzender: Prof. Dr. Jens Zumbrägel
1. Gutachter: Prof. Dr. Stefan Katzenbeisser
2. Gutachter: Prof. Dr. Florian Kerschbaum
3. Gutachter: Prof. Dr. Josep Domingo-Ferrer
Schriftführer: Prof. Dr. Joachim Posegga

Einreichungsdatum: Dezember 2019
Tag des Rigorosums: 3. Juni 2020

Abstract

Cryptography is the scientific study of techniques for securing information and communication against adversaries. It is about designing and analyzing encryption schemes and protocols that protect data from unauthorized reading. However, in our modern information-driven society with highly complex and interconnected information systems, encryption alone is no longer enough as it makes the data unintelligible, preventing any meaningful computation without decryption. On the one hand, data owners want to maintain control over their sensitive data. On the other hand, there is a high business incentive for collaborating with an untrusted external party.

Modern cryptography encompasses different techniques, such as secure multiparty computation, homomorphic encryption or order-preserving encryption, that enable cloud users to encrypt their data before outsourcing it to the cloud while still being able to process and search on the outsourced and encrypted data without decrypting it. In this thesis, we rely on these cryptographic techniques for computing on encrypted data to propose efficient multiparty protocols for order-preserving encryption, decision tree evaluation and k^{th}-ranked element computation.

We start with Order-preserving encryption (OPE) which allows encrypting data, while still enabling efficient range queries on the encrypted data. However, OPE is symmetric limiting, the use case to one client and one server. Imagine a scenario where a Data Owner (DO) outsources encrypted data to the Cloud Service Provider (CSP) and a Data Analyst (DA) wants to execute private range queries on this data. Then either the DO must reveal its encryption key or the DA must reveal the private queries. We overcome this limitation by allowing the equivalent of a public-key OPE.

Decision trees are common and very popular classifiers because they are explainable. The problem of evaluating a private decision tree on private data consists of a server holding a private decision tree and a client holding a private attribute vector. The goal is to classify the client's input using the server's model such that the client learns only the result of the classification, and the server learns nothing. In a first approach, we represent the tree as an array and execute only d interactive comparisons (instead of 2^d as in existing solutions), where d denotes the depth of the tree. In a second approach, we delegate the complete tree evaluation to the server using somewhat or fully homomorphic encryption where the ciphertexts are encrypted under the client's public key.

A generalization of a decision tree is a random forest that consists of many decision trees. A classification with a random forest evaluates each decision tree in the forest and outputs the classification label which occurs most often. Hence, the classification labels are ranked by their number of occurrences and the final result is the best ranked one. The best ranked element is a special case of the k^{th}-ranked element. In this thesis, we consider the secure computation of the k^{th}-ranked element in a distributed setting with applications in benchmarking and auctions. We propose different approaches for privately computing the k^{th}-ranked element in a star network, using either garbled circuits or threshold homomorphic encryption.

Kurzfassung

Kryptographie ist die wissenschaftliche Untersuchung von Techniken zum Schutz von Daten und Kommunikation durch Verschlüsselung und Protokolle. In unserer modernen informationsgetriebenen Gesellschaft mit hochkomplexen und miteinander verbundenen Informationssystemen reicht Verschlüsselung allein nicht mehr aus, da die Daten dadurch unverständlich werden und keine sinnvolle Berechnung ohne Entschlüsselung mehr möglich ist. Einerseits möchten Dateneigentümer die Kontrolle über ihre sensiblen Daten behalten. Andererseits besteht ein hoher geschäftlicher Anreiz für die Zusammenarbeit mit einer nicht vertrauenswürdigen externen Partei.

Die moderne Kryptographie umfasst verschiedene andere Techniken, wie sichere Mehrparteienberechnung, homomorphe Verschlüsselung oder ordnungserhaltende Verschlüsselung, die es Cloud-Benutzern ermöglichen, ihre Daten vor dem Auslagern in die Cloud zu verschlüsseln, während sie die ausgelagerten und verschlüsselten Daten weiterhin verarbeiten und durchsuchen können, ohne sie zu entschlüsseln. In dieser Arbeit stützen wir uns auf diese kryptografischen Techniken zur Verarbeitung verschlüsselter Daten, um effiziente Mehrparteienprotokolle für ordnungserhaltende Verschlüsselung, Entscheidungsbaumauswertung und die Berechnung des k-kleinsten Elements vorzuschlagen.

Wir starten mit der ordnungserhaltenden Verschlüsselung (Order-Preserving Encryption im englischen, kurz OPE), die das Verschlüsseln von Daten ermöglicht, während weiterhin effiziente Bereichsabfragen auf den verschlüsselten Daten möglich sind. OPE ist jedoch symmetrisch und beschränkt den Anwendungsfall auf einen Client und einen Server. Wir betrachten ein Szenario, in dem ein Data Owner (DO) verschlüsselte Daten an den Cloud Service Provider (CSP) auslagert und ein Data Analyst (DA) private Bereichsabfragen auf diese Daten ausführen möchte. Dann muss entweder der DO seinen Verschlüsselungsschlüssel offenlegen oder der DA muss die privaten Abfragen offenlegen. Wir überwinden diese Einschränkung, indem wir das Äquivalent eines OPE mit öffentlichem Schlüssel konstruieren.

Entscheidungsbäume sind gängige und sehr beliebte Klassifikatoren, da sie erklärbar sind. Das Problem der Auswertung eines privaten Entscheidungsbaums auf private Daten besteht aus einem Server mit privatem Entscheidungsbaum und einem Client mit privatem Attributvektor. Das Ziel ist es, die Eingaben des Clients mithilfe des Servermodells so zu klassifizieren, dass der Client nur das Ergebnis lernt und der Server nichts lernt. In einem ersten Ansatz stellen wir den Baum als Array dar und führen nur d interaktive Vergleiche durch (anstatt 2^d als in existierenden Lösungen), wobei d die Tiefe des Baums bezeichnet. In einem zweiten Ansatz delegieren wir die vollständige Baumbewertung unter Verwendung einer homomorphen Verschlüsselung an den Server, wobei die Chiffretexte unter dem öffentlichen Schlüssel des Clients verschlüsselt werden.

Eine Verallgemeinerung eines Entscheidungsbaums ist ein Random Forest, das aus vielen Entscheidungsbäumen besteht. Eine Klassifizierung mit einem Random Forest wertet jeden Entscheidungsbaum in dem Forest aus und gibt das am häufigsten vorkommende klassifizierte Label aus. Das Element mit der höchsten Häufigkeit ist ein Spezialfall des k-kleinsten Elements. In dieser Arbeit betrachten wir die Berechnung des k-kleinsten Elements in einer verteilten Umgebung mit Anwendungen in Benchmarking und Auktionen. Wir schlagen verschiedene Ansätze vor, um das k^{th}-Element in einem Sternennetz privat zu berechnen.

Acknowledgements

I am indebted to several people, without whom this dissertation would not have been possible.

First and foremost, I am very grateful to my advisors Stefan Katzenbeisser and Florian Kerschbaum for your great guidance and technical support that significantly contribute to shaping my research career. You trusted my abilities to do research and helped me withstand several difficulties.

Special thanks go to the members of the thesis committee Jens Zumbrägel, Joachim Posegga, and Josep Domingo-Ferrer for your valued participation in the examination process at this particular time of COVID-19.

I am particularly thankful to my SAP supervisors Roger Gutbrod, Detlef Plümper, and Mathias Kohler for granting me enough research freedom and supporting my professional development.

I would also like to thank my fellow doctoral students and friends Andreas, Benny, Benjamin, Daniel, Florian H., and Jonas for our constant active discussions as well as for your feedback and personal support.

A great thank you also to my master students Mubashir and Yordan for your valuable contribution to some implementations of this thesis.

My special thanks also go to my SAP team colleagues Martin H., Martin J., Peter, Susan for your helpful advice and feedback.

Last but not least, I would like to thank my parents Thérèse and Aristide; my brother Thiery; my Sisters Edwige, Edith, Ida; my cousins Constant, Harold, Robert; my friend Josue; my sister-in-law Judith, and my dear wife Jeanne D'Arc for your love and constant encouragement throughout this adventure.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we propose efficient multiparty protocols for order-preserving encryption, decision tree evaluation, and k^{th}-ranked element computation. We start with the motivation behind our results. This chapter is structured as follows. In Section 1.1, we describe the motivation of computing on encrypted data and survey the cryptographic techniques relevant to our work. In Section 1.2, we describe our contributions which have applications in database range queries, machine learning classification, benchmarking and auction. Section 1.3 lists the papers published throughout the process of writing this thesis. In Section 1.4, we describe the methodology followed in our main chapters. Finally, we describe the structure of the remaining thesis in Section 1.5.

1.1 Motivation

Cryptography is the scientific study of techniques for securing information and communication against adversaries. It is about designing and analyzing encryption schemes and protocols that protect data from unauthorized reading. However, in our modern information-driven society with highly complex and interconnected information systems, encryption alone is no longer enough as it makes the data unintelligible, preventing any meaningful computation without decryption. On the one hand, data owners want to maintain control over their sensitive data. On the other hand, there is a high business incentive for collaborating with an untrusted external party.

In cloud computing, companies use a network of remote servers hosted by a service provider on the Internet to store, manage, and process data, rather than a local server or a personal computer. By doing so, they delegate the maintenance of their computer system and can focus on their core business. To protect their data, data owners can encrypt it before outsourcing. However, this would imply that they must give up the functionality of processing the data because encryption makes the data unintelligible, and outsourcing the encryption keys is not an option. Therefore, companies are reluctant to migrate their sensitive data to the cloud.

The cloud provider might want to offer to its customers a cloud service which requires computation on data belonging to the provider itself and the customers. On the one hand, the cloud service might be based on a proprietary algorithm containing business sensitive data, e.g., a trained neural network, or the cloud provider might be constrained by law not to share certain information. On the other hand, the input of customers to this cloud service and even the result of the computation might be sensitive such that customers are not willing to use this service at all.

In other settings, different data owners might be interested in sharing the result of a computation done on their individual data. A trivial solution would be to perform the computation by an external trusted third party, which receives the input data, runs the computation, outputs the result, and deletes all sensitive data. However, the

confidentiality of the data and the lack of trust in any other party might prevent data owners to participate in this kind of joint computation.

Modern cryptography encompasses different techniques, such as secure multiparty computation [58, 90, 137, 140, 161, 195], homomorphic encryption [66, 84, 159, 186] or order-preserving encryption [3, 30, 31, 123, 124, 163, 164], that enable cloud users to encrypt their data before outsourcing it to the cloud, while still being able to process and search on the outsourced and encrypted data without decrypting it. Secure multiparty computation can emulate the trusted third party mentioned above, such that the computation reveals nothing beyond the result.

Secure multiparty computation (SMC) is a cryptographic technique that allows several parties to compute a function on their private inputs without revealing any information other than the function's output. The basic idea is to generically represent the function to be computed as a Boolean or arithmetic circuit and then run an interactive protocol among the parties to privately evaluate the circuit. While generic solutions are theoretically interesting in showing that a solution exists, they are in general impractical for large size problems, since the number of cryptographic operations is usually proportional to the number of gates in the circuit. Specialized protocols exploit the domain knowledge of the problem at hand and make use of generic techniques only where it is necessary, reducing the number of gates that require cryptographic operations and, therefore, resulting in more efficient solutions.

A Homomorphic Encryption (HE) scheme allows computations on ciphertexts by generating a new ciphertext encrypting the result of a function on the plaintexts. It can allow addition or multiplication to be performed homomorphically on plaintexts and is, therefore, said to be additively or multiplicatively homomorphic. A somewhat HE allows both addition and multiplication but with a limited number of operations. The encryption adds noise to ciphertexts that grows during homomorphic operations. If the noise in a ciphertext exceeds a given bound then correct decryption is no longer possible. Fully homomorphic encryption can be built using somewhat HE by adding a bootstrapping procedure that re-encrypts the ciphertext with a smaller noise without decrypting it.

Order-preserving encryption (OPE) allows encrypting data, while still enabling efficient range queries on the encrypted data. Moreover, it does not require any change to the database management system, because the same comparison operator for plaintexts is used for ciphertexts. This makes order-preserving encryption schemes very suitable for data outsourcing in cloud computing scenarios. OPE can be stateless or stateful. Stateless OPE fails to provide ideal security, which requires that OPE should reveal only the order and nothing else. Stateful OPE can provide ideal security but even an ideal secure OPE can remain vulnerable to plaintext guessing attacks.

In this thesis, we rely on the above mentioned cryptographic techniques for computing on encrypted data to propose efficient SMC protocols to specific problems that are defined in the next section.

1.2 Contributions

The contributions of this thesis have applications in databases, machine learning, benchmarking, and auctions. This section is, therefore, structured around these applications. We then close the section with a list of publications.

1.2.1 Databases: Order-Preserving Encryption

Databases allow storing data in a structured way such that queries can easily be executed on the data. In this thesis, we are particularly interested in range queries that require a comparison between values. Since we want the computation to be done on encrypted data, we encrypt the data using order-preserving encryption before outsourcing. OPE is necessarily symmetric limiting the use case to one client and one server. Imagine a scenario where a Data Owner (DO) outsources encrypted data to the Cloud Service Provider (CSP) and a Data Analyst (DA) wants to execute private range queries on this data. Then either the DO must reveal its encryption key or the DA must reveal the private queries. We overcome this limitation by allowing the equivalent of a public-key OPE. We present a secure multiparty protocol that enables secure range queries for multiple users. In this scheme, the DA cooperates with the DO and the CSP to order-preserving encrypt the private range queries without revealing any other information to the parties. This work has been published in [181] and is discussed in detail in Chapter 4. The contribution is summarized as follows:

- We introduce a novel notion of oblivious order-preserving encryption (OOPE). This scheme allows a DA to execute private range queries on an order-preserving encrypted database.
- We propose an oblivious OPE protocol based on mutable OPE schemes by Popa et al. [163] and Kerschbaum and Schröpfer [124].
- Since the schemes [124, 163] are deterministic, we also consider the case where the underlying OPE scheme is probabilistic such as frequency-hiding OPE [123] or OPE based on an efficiently searchable encrypted data structure [125].
- Finally, we implement and evaluate our scheme in LAN and WAN settings.

A concrete application of OOPE mentioned in the previous section can be found in collaborative data analysis and machine learning. Imagine a supply chain scenario, where a Data Analyst is a supplier (manufacturer) owning a private machine learning model and is interested in optimizing its manufacturing process using private data owned by its buyer (another supplier or distributor). As a machine learning model, we are particularly interested in decision tree models. A decision tree model consists of decision nodes, each marked with a test condition, and leaf nodes, each marked with a classification label. As a result, if each test condition in a decision node consists of a greater-than comparison, i.e., a range query, then each path in the decision tree is a conjunction of range queries, which OOPE allows to execute while preserving the privacy of both Data Owner and Data Analyst. Additional contributions related to this work have been published in [125, 176, 182].

1.2.2 Machine Learning: Decision Tree Classification

Decision trees are common and very popular classifiers because their result is easy to explain. In this thesis, we are interested in another scenario of decision trees, namely, the problem of evaluating private decision trees on private data. The scenario consists of a server holding a private decision tree and a client holding a private attribute vector. The goal is to classify the client's attribute vector using the server's decision tree model such that the result of the classification is revealed only to the client and nothing else is revealed neither to the client nor the server. We propose in this thesis two solution approaches to the problem.

Previous solutions to this problem either transform the whole decision tree to an oblivious program or perform as many comparisons as there are decision nodes. In our first approach, the main idea of our novel solution is to represent the tree as an array. Then we execute only d comparisons, where d denotes the depth of the tree. The result of each comparison allows to obliviously compute the index of the next node, which is never revealed to any party in clear. The comparison and the computation of the next node are performed using a small garbled circuit, which is independent of the position in the tree and its size. To actually select the next node in the tree, we use a primitive called oblivious array indexing which, given an array and secret shares on an index, returns secret shares of the indexed element to the parties. This work has been published in [183] and is discussed in detail in Chapter 5. The contribution is summarized as follows:

- We represent the tree as an array, where each element contains a node of the tree and pointers to its child nodes. Then, while traversing the tree, we obliviously select the next node and secret-share it to the parties. The comparison at each node is performed using a garbled circuit, that takes secret-share of that node and the corresponding attribute value and returns secret-shares of the index of the next node.
- We instantiate our protocol by the different indexing procedures mentioned above. In particular, instantiating our protocol with oblivious transfer is more efficient for small to mid-size trees. With ORAM [189] our scheme has sublinear communication.
- Finally, we implement and evaluate our scheme and demonstrate its practicality regarding runtime and bandwidth. For small size trees, our scheme competes with previous protocols but outperforms them if the size of the decision tree becomes larger – using oblivious transfer for small to mid-size trees and ORAM for very large trees.

Existing privacy-preserving protocols that address this problem use or combine different generic secure multiparty computation approaches resulting in several interactions between the client and the server. In the second approach, we design and implement a novel client-server protocol that delegates the complete tree evaluation to the server while preserving privacy and reducing the overhead. The idea is to use fully (somewhat) homomorphic encryption and evaluate the tree on ciphertexts encrypted under the client's public key. However, since current somewhat homomorphic encryption schemes have high overhead, we combine efficient data representations with different algorithmic optimizations to keep the computational overhead and the communication cost low. As a result, we are able to provide the first non-interactive protocol, that allows the client to delegate the evaluation to the server by sending an encrypted input and receiving only the encryption of the result. This work is published in [180] and is discussed in detail in Chapter 6. The contribution is summarized as follows:

- We propose a non-interactive protocol for private decision tree evaluation. Our scheme allows the client to delegate the evaluation to the server by sending an encrypted input and receiving only the encryption of the result.
- We propose Pdt-Bin which is an instantiation of the main protocol with a binary representation of the input. Then we combine efficient data representations with different algorithmic optimizations to keep the computational overhead and the communication cost low.

- We first propose a modified variant of the Lin-Tzeng comparison protocol [135]. Then, we propose PDT-INT which is an instantiation of the main protocol using an arithmetic circuit, where the values are compared using our modified variant of Lin-Tzeng comparison protocol [135].
- We provide correctness and security proofs of our scheme. Finally, we implement and benchmark both instantiations using HElib [100] and TFHE [52].

Both approaches can be extended to random forests which consist of many decision trees, such that the final result is the classification label that occurs most often among the individual decision tree evaluations.

1.2.3 Benchmarking and Auction: The k^{th}-Ranked Element

As mentioned above, a generalization of a decision tree is a random forest that consists of many decision trees. Classification with a random forest evaluates each decision tree in the forest and outputs the classification label which occurs most often. Hence, the classification labels are ranked by their number of occurrences and the final result is the best ranked one. The best ranked element is a special case of the k^{th}-ranked element. In this part of the thesis, we consider the computation of the k^{th}-ranked element in a distributed setting. That is, given n parties each holding a private integer, the problem is to securely compute the element ranked k (for a given k such that $1 \leq k \leq n$) among these n integers. The goal is to reveal to the parties only the k^{th}-ranked element (or the party holding it) and nothing else. The computation of the k^{th}-ranked element has applications in benchmarking, where a company is interested in knowing how well it is doing compared to others, or in auctions where bidders are interested in knowing the highest bid. Previous secure protocols for the k^{th}-ranked element require a communication channel between each pair of parties. A server model naturally fits with the client-server architecture of Internet applications in which clients are connected to the server and not to other clients. It simplifies secure computation by reducing the number of rounds and improves its performance and scalability. We propose different approaches for privately computing the k^{th}-ranked element in the server model, using either garbled circuits or threshold homomorphic encryption. This work has been published in [184] and is discussed in detail in Chapter 7. The contribution is summarized as follows:

- Our first scheme KRE-YGC uses Yao's garbled circuits [20, 139] to compare clients' inputs and additively HE (AHE) to compute the rank of each input without revealing any sensitive data.
- Our second scheme KRE-AHE1 is based on threshold additively HE (AHE). The server uses our modified variant of the Lin-Tzeng comparison protocol [135] to compare inputs encrypted with AHE.
- In our third scheme KRE-AHE2, we continue with threshold AHE, however, we perform the comparison interactively using the DGK protocol [63].
- The fourth scheme KRE-SHE is based on somewhat HE and allows the server to non-interactively compute the k^{th}-ranked element such that the clients only interact to jointly decrypt the result.

Our schemes have a constant number of rounds. KRE-YGC is suitable for a setting where the server is non-colluding and clients cannot fail. If collusion and failure are an issue, then either KRE-AHE2 or KRE-SHE is suitable. KRE-SHE has the best

asymptotic complexity, however, it is practically less efficient because of the high overhead of homomorphic encryption.

1.3 Publications

The contributions of this thesis have been published in the work listed below:

- [181]: Anselme Tueno and Florian Kerschbaum.
  Efficient Secure Computation of Order-Preserving Encryption.
  Proceedings of the 15th ACM ASIA Conference on Computer and Communications Security (ASIACCS), 2020

- [125]: Florian Kerschbaum and Anselme Tueno.
  An efficiently searchable encrypted data structure for range queries.
  Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS), 2019

- [176]: Fabian Taigel, Anselme K. Tueno, and Richard Pibernik.
  Privacy-preserving condition-based forecasting using machine learning.
  Journal of Business Economics, Jan 2018

- [182]: Anselme Tueno, Florian Kerschbaum, Daniel Bernau and Sara Foresti.
  Selective access for supply chain management in the cloud.
  Proceedings of the IEEE Conference on Communications and Network Security (CNS), 2017

- [183]: Anselme Tueno, Florian Kerschbaum and Stefan Katzenbeisser.
  Private Evaluation of Decision Trees using Sublinear Cost.
  Proceedings of the 19th Privacy Enhancing Technologies Symposium (PoPETS), 2019

- [180]: Anselme Tueno, Yordan Boev and Florian Kerschbaum.
  Non-Interactive Private Decision Tree Evaluation.
  Proceedings of the 34th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec), 2020

- [184]: Anselme Tueno, Florian Kerschbaum, Stefan Katzenbeisser, Yordan Boev and Mubashir Qureshi.
  Secure Computation of the kth-Ranked Element in a Star Network.
  Proceedings of the 24th International Conference on Financial Cryptography and Data Security (FC), 2020

1.4 Methodology

This section describes the methodology that we follow throughout our main chapters, i.e., Chapters 4 to 7. Our methodology consists of five phases: problem definition, functionality, construction, analysis, and evaluation. We start by describing the problem to solve and defining the functionality that computes the problem. Then we construct a protocol that implements the defined functionality, and prove that it is correct and secure. The protocol is then theoretically analyzed in computation and communication. Finally, we implement and evaluate the protocol.

1.4.1 Problem Definition

In the first phase, we describe in detail the problem that we want to solve and the weaknesses in existing solutions. Then, we describe possible applications and briefly sketch our solution approach. Finally, we conclude by briefly comparing our solution to existing ones.

1.4.2 Functionality

Before designing our secure multiparty protocols, we start by defining the functionality that is to be computed to solve the problem at hand. A multiparty functionality is a random process that maps a given number n of inputs to n outputs [90]. The n inputs are sensitive and belong to n different parties that want to perform the computation such that the inputs remain private and each party receives only its designated outputs. For each functionality, we, therefore, also specify, which are the parties, what are their inputs, and which party receives which output of the computation. In some situations, a party might have no input to – or even no output from – the computation. Since we want to preserve the confidentiality of sensitive data, we define when a protocol implementing a functionality can be considered secure. In this thesis, we are particularly interested in the semi-honest security model, which requires parties to follow the protocol specification. Although the malicious security model provides a stronger security, it requires in general much more overhead. Additionally, we specify additional assumptions if any, such as the existence of a public key infrastructure.

1.4.3 Construction

After defining the functionality, we provide a step by step construction of the protocol implementing the functionality under the defined security requirement. For some protocols, we start by proposing a basic solution that is later optimized in possibly different ways or extended to a larger problem. In other cases, we provide different variants using different sub-protocols. To design our protocols, we rely on basic techniques that allow computation on encrypted data, such as homomorphic encryption, garbled circuit, and secret sharing. In some protocols, we even combine some of these techniques to achieve our goal. We rely on special protocols such as protocols for oblivious transfer, ORAM, or integer comparison.

1.4.4 Analysis

The analysis consists of three different aspects. First, we discuss the correctness of the protocol showing that after the computation each party receives an output that is correct with respect to the functionality. The second aspect is proving the security of the protocol, where we define for each party its view of the protocol which consists of the messages received by that party during the protocol execution. Then we define a simulator that simulates the adversary's view given only the inputs and outputs of corrupted parties. This proof technique is called *simulation paradigm* [90] and is briefly discussed in Section 2.1.5. The third and last aspect of the analysis is the complexity of the protocol. The complexity consists of computing the number of rounds, the number of cryptographic operations, and the number of bits sent during the protocol execution. In some cases, we compare the complexity of possibly different variants of our protocol with existing protocols.

1.4.5 Evaluation

The evaluation consists of describing evaluation details, and the results of our evaluation. In the evaluation details, we mention things such as software, programming language, API, and the hardware used for the evaluation. In the results part, we report the results of our evaluation using graphs and tables and possibly comparing with the related work.

1.5 Structure

After this introductory chapter, we briefly review some relevant cryptographic building blocks in Chapter 2. Chapter 3 is dedicated to reviewing related work on secure multiparty party computation, secure integer comparison, search over encrypted data, private decision tree evaluation, and secure computation of the k^{th}-ranked element. Chapters 4 to 7 contain the main contributions of the thesis. In Chapter 4, we describe our oblivious OPE that allows the equivalent of a public-key encryption for range queries. Chapters 5 and 6 describe our protocols for privately evaluating a decision tree model on private data. While the scheme in Chapter 5 is interactive and sublinear in the tree size, the scheme in Chapter 6 is non-interactive and delegates the complete tree evaluation to the server. In Chapter 7, we describe different approaches to compute the k^{th}-ranked element in a star network using either garbled circuit or threshold homomorphic encryption. We, finally, conclude this thesis in Chapter 8.

Chapter 2

Preliminaries

In this chapter, we briefly review relevant cryptographic definitions, primitives, and tools in Section 2.1 and the application domains of the protocols developed in this thesis in Section 2.2. Finally, Section 2.3 defines the notation used throughout this thesis.

2.1 Cryptographic Backgrounds

This section sets the foundation which consists of relevant cryptographic definitions and basic primitives such as pseudorandom generators, encryption schemes, secret sharing, and homomorphic encryption. Then it describes the concept of multiparty computation, presents basic protocols, such as oblivious transfer and oblivious Random Access Machine (RAM), and the garbled circuit protocol that are used in this thesis to design secure protocols. We follow Katz and Lindell's [115] and Goldreich's [89, 90] comprehensive textbooks.

2.1.1 Definitions

Encryption schemes can be perfectly secure in the sense that even with unlimited resources no adversary can break the security. However, any perfectly secure encryption requires a key space that is at least as large as the message space [115]. This motivates the development of schemes that are practically unbreakable in the sense that the security is only preserved against an efficient adversary which can carry out computation in *Probabilistic Polynomial Time (PPT)*.

Definition 2.1.1 (Probabilistic Polynomial Time)**.** *An algorithm $\mathcal{A}$ is said to be* probabilistic *if it has access to a source of randomness. An algorithm $\mathcal{A}$ is said to run in* polynomial time *if there exists a polynomial $p(\cdot)$ such that, for every input $x \in \{0,1\}^*$, the computation of $\mathcal{A}(x)$ terminates within at most $p(|x|)$ steps, where $|x|$ denotes the length of the string x.*

Hence, an efficient adversary has limited computing resources. However, it can still succeed to break the encryption scheme with a very small, i.e., *negligible* probability.

Definition 2.1.2 (Negligible)**.** *A function f is* negligible *if for every polynomial $p(\cdot)$ there exists an integer N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$.*

Equivalently a function f is negligible if for all constants c there exists an integer N such that for all $n > N$ it holds that $f(n) < n^{-c}$. One typically denotes an arbitrary negligible function by negl. Examples of negligible functions are 2^{-n}, $2^{-\sqrt{n}}$, $n^{-\log n}$. The sum of two negligible functions as well as the product of a negligible function with a positive polynomial are negligible functions.

Two probability distributions $X = \{X_n\}_{n\in\mathbb{N}}$ and $Y = \{Y_n\}_{n\in\mathbb{N}}$ are *computationally indistinguishable* (denoted $X \overset{c}{\equiv} Y$) if no probabilistic polynomial time (PPT) algorithm can distinguish them except with negligible probability.

Definition 2.1.3 (Probability Ensemble)**.** *Let I be a countable index set. A* probability ensemble indexed by *I is a sequence of random variables indexed by I. Namely, any $X = \{X_n\}_{n\in I}$, where each X_n is a random variable, is a probability ensemble indexed by I.*

Definition 2.1.4 (Computational Indistinguishability)**.** *Two probability ensembles $X = \{X_n\}_{n\in\mathbb{N}}$ and $Y = \{Y_n\}_{n\in\mathbb{N}}$ are* computationally indistinguishable*, denoted $X \overset{c}{\equiv} Y$, if for every probabilistic polynomial-time distinguisher $\mathcal{D}$, there exists a negligible function* negl *such that:*

$$|Pr[\mathcal{D}(X_n, 1^n) = 1] - Pr[\mathcal{D}(Y_n, 1^n) = 1]| \leq \mathsf{negl}(n)\,.$$

To prove that two probability ensembles $X = \{X_n\}_{n\in\mathbb{N}}$ and $Y = \{Y_n\}_{n\in\mathbb{N}}$ are computationally indistinguishable, one uses a standard proof technique called the *hybrid argument* [89, 90].

2.1.2 Basic Primitives

Cryptographic schemes depend on randomness whenever it is required to avoid the predictability of the outcome. For example, the output of an algorithm that generates encryption keys should not be predictable. As no deterministic process can produce true randomness, one uses mathematical approaches to produce pseudorandomness.

Pseudorandom Generator. A *Pseudorandom Generator (PRG)* is a deterministic algorithm that generates a distribution that is computationally indistinguishable from the uniform distribution over strings of a certain length. A PRG takes a short uniformly distributed string, known as the *seed*, and outputs a longer string that cannot be efficiently distinguished from a uniformly distributed string of the same length.

Definition 2.1.5 (Pseudorandom Ensembles)**.** *An ensemble $X = \{X_n\}_{n\in\mathbb{N}}$ is* pseudorandom*, if there exists a polynomial $l(n)$ such that X is computationally indistinguishable from the ensemble $U = \{U_{l(n)}\}_{n\in\mathbb{N}}$, where $U_{l(n)}$ is the uniform distribution over $\{0,1\}^{l(n)}$*

Definition 2.1.6 (Pseudorandom Generator)**.** *Let $l(\cdot)$ be a polynomial and let G be a (deterministic) polynomial-time algorithm such that upon any input s, algorithm G outputs a string of length $l(|s|)$. We say that G is a* pseudorandom generator (PRG) *if the following two conditions hold:*

- *Expansion: For every n it holds that $l(n) > n$,*
- *Pseudorandomness: The ensemble $\{G(s_n)\}_{n\in\mathbb{N}}$, where $s_n \leftarrow \{0,1\}^n$ is pseudorandom.*

Pseudorandom Function. A *keyed function* is a function $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ where the first input, denoted by k and called the *key*, is chosen and fixed such that we have the single-input function $F_k : \{0,1\}^* \to \{0,1\}^*$ defined by $F_k(x) = F(k,x)$. A keyed function F_k is *length-preserving*, if the key, the input, and the output of F all have the same length.

Definition 2.1.7. *Let* $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ *be an efficient, length-preserving, keyed function. We say* F *is a* Pseudorandom Function (PRF) *if for all probabilistic polynomial-time distinguisher* D*, there exists a negligible function* negl *such that:*

$$|Pr[D^{F_k(\cdot)}(1^n) = 1] - Pr[D^{f_n(\cdot)}(1^n) = 1]| \leq \mathsf{negl}(n)\,,$$

where $k \leftarrow \{0,1\}^n$ *is chosen uniformly at random and* f_n *is chosen uniformly at random from the set of functions mapping* n*-bit strings to* n*-bit strings.*

Encryption Scheme. Encryption schemes are used to secure communication between two parties. We distinguish between private-key schemes which use the same key for encryption and decryption, and public-key schemes which use different keys for encryption and decryption.

Definition 2.1.8. *A* private-key encryption scheme *is a tuple of polynomial-time algorithms (*KGen*,* Enc*,* Dec*) such that:*

- $\mathsf{sk} \leftarrow \mathsf{KGen}(\lambda)$*: the probabilistic* key-generation algorithm KGen *takes as input the security parameter* λ *and outputs a key* sk*.*
- $\mathsf{c} \leftarrow \mathsf{Enc}(\mathsf{sk}, \mathsf{m})$*: the* encryption algorithm Enc *takes as input a key* sk *and a plaintext message* $\mathsf{m} \in \{0,1\}^*$*, and outputs a ciphertext* c*.*
- $\mathsf{m}' \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{c})$*: the* decryption algorithm Dec *takes as input a key* sk *and a ciphertext* c*, and outputs a message* m'*.*

The encryption scheme is correct iff for every λ*, every key* sk *output by* $\mathsf{KGen}(1^\lambda)$*, and every message* $\mathsf{m} \in \{0,1\}^*$*, it holds that* $\mathsf{Dec}(\mathsf{sk}, Enc(\mathsf{sk}, \mathsf{m})) = \mathsf{m}$*.*

Definition 2.1.9. *A* public-key encryption scheme *is a tuple of polynomial-time algorithms (*KGen*,* Enc*,* Dec*) such that:*

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(\lambda)$*: the probabilistic* key-generation algorithm KGen *takes as input the security parameter* λ *and outputs a* public key pk *and a* private key sk*.*
- $\mathsf{c} \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$*: the* encryption algorithm Enc *takes as input a public key* pk *and a plaintext message* $\mathsf{m} \in \{0,1\}^*$*, and outputs a ciphertext* c*.*
- $\mathsf{m}' \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{c})$*: the* decryption algorithm Dec *takes as input a private key* sk *and a ciphertext* c*, and outputs a message* m'*.*

The encryption scheme is correct iff for every λ*, every key* sk *output by* $\mathsf{KGen}(1^\lambda)$*, and every message* $\mathsf{m} \in \{0,1\}^*$*, it holds that* $\mathsf{Dec}(\mathsf{sk}, Enc(\mathsf{pk}, \mathsf{m})) = \mathsf{m}$*.*

An important security requirement for encryption schemes is *semantic security*. An encryption scheme is semantically secure if it is infeasible to learn anything about the plaintext from the ciphertext. Semantic security is equivalent to *indistinguishability under chosen-plaintext attack* (usually denoted by IND-CPA) and requires that encryption schemes produce ciphertexts that are infeasible to tell apart. Loosely speaking, the encryption algorithm Enc must be probabilistic and return different ciphertexts for the same plaintext. Several public-key schemes such as ElGamal [76], Goldwasser-Micali [92], Paillier [159] have been proven to be semantically secure. For more details, we refer to [90, 115]. Unless explicitly stated, we require the encryption schemes used throughout this thesis to be semantically secure.

2.1.3 Secret Sharing

A *secret sharing* scheme consists of two algorithms: the first algorithm specifies how to *share* the secret s in *shares*, and the second one specifies how to *reconstruct* s from the shares or a subset thereof. Assume the secret s is a bit string of length μ. Then a simple way to share s in n secret shares $\langle s\rangle_1, \ldots, \langle s\rangle_n$ is to choose $n-1$ random strings $r_1, \ldots, r_{n-1}$ of length μ and set $\langle s\rangle_1 = r_1, \ldots, \langle s\rangle_{n-1} = r_{n-1}$ and $\langle s\rangle_n = s \oplus r_1 \oplus \ldots \oplus r_{n-1}$. It is easy to see that $\langle s\rangle_1 \oplus \ldots \oplus \langle s\rangle_n = s$ holds. We will refer to this sharing scheme as *exclusive-or sharing*. If s is an integer, then the sharing algorithm chooses $n-1$ random numbers $r_1, \ldots, r_{n-1}$ modulo a number p with $|p| \geq |s|$ and sets $\langle s\rangle_1 = r_1, \ldots, \langle s\rangle_{n-1} = r_{n-1}$ and $\langle s\rangle_n = s - r_1 - \ldots - r_{n-1} \bmod p$. Again we easily have $\sum_{i=1}^{n} \langle s\rangle_i \bmod p = s$. We will refer to this sharing scheme as *additive sharing*. Assume there are n parties $P_1, \ldots, P_n$. The goal of the sharing scheme is to distribute share $\langle s\rangle_i$ to party P_i. These simple ways to share a secret will be used in Chapters 4, 5 and 7.

An important property of a secret sharing scheme is *linearity* which allows to share the sum of two secrets by just adding their respective secret shares. For example, if two secrets s_1, s_2 have been additively shared in $\langle s_1\rangle_i$, $\langle s_2\rangle_i$ then the parties can securely compute the shares $s_1 + s_2$ by just locally adding their shares. In a *linear scheme*, a secret s is viewed as an element of a finite field, and the shares are obtained by applying a linear mapping to the secret and several independent random field elements [18]. An example is the so-called (t, n)–*threshold* scheme, which assumes that among the parties that receive the secret shares, at most t parties can fail to deliver a result or cooperate to defeat the protocol, where $t < n$. If at least $t+1$ valid shares are available, the secret can be reconstructed. Therefore, a subset of parties with cardinality at least $t+1$ is referred to as *authorized*.

In this thesis, we will use Shamir's (t, n)–threshold scheme [170]. Let n, t be integers such that $t \leq n$. Shamir's scheme shares a secret s by choosing a random polynomial:

$$f(x) = s + \sum_{i=1}^{t-1} a_i x^i \tag{2.1}$$

and computing secret shares $\langle s\rangle_i = f(i), 1 \leq i \leq n$. Let $L_i(x)$ and l_i be defined as:

$$L_i(x) = \prod_{j=1, j\neq i}^{t} \frac{x-j}{i-j} \text{ and } l_i = L_i(0) = \prod_{j=1, j\neq i}^{t} \frac{-j}{i-j}. \tag{2.2}$$

Given t shares $(1, \langle s\rangle_1), \ldots, (t, \langle s\rangle_t)$, the polynomial $g(x) = \sum_{i=1}^{t} L_i(x) \cdot \langle s\rangle_i$ is the same as $f(x)$, since both have degree at most $t-1$ and match at t points. Therefore $s = f(0) = g(0) = \sum_{i=1}^{t} \langle s\rangle_i \cdot l_i$. The numbers l_i are called *Lagrange coefficients*. Shamir's scheme can be used in threshold decryption described in the next section.

2.1.4 Homomorphic Encryption

A *Homomorphic Encryption (HE)* allows computations on ciphertexts by generating an encrypted result whose decryption matches the result of a function on the plaintexts.

HE Algorithms. A HE scheme consists of the following algorithms:

- $\mathsf{pk}, \mathsf{sk}, \mathsf{ek} \leftarrow \mathsf{KGen}(\lambda)$: This probabilistic algorithm takes a security parameter λ and outputs public, private, and evaluation keys pk, sk, and ek.

- $\mathsf{c} \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$: This algorithm takes pk and a message m and outputs a ciphertext c. We will use $[\![m]\!]$ as a shorthand notation for $\mathsf{Enc}(\mathsf{pk}, \mathsf{m})$.
- $\mathsf{c} \leftarrow \mathsf{Eval}(\mathsf{ek}, f, \mathsf{c}_1, \ldots, \mathsf{c}_n)$: This algorithm takes ek, an n-ary function f and n ciphertexts $\mathsf{c}_1, \ldots \mathsf{c}_n$ and outputs a ciphertext c.
- $\mathsf{m}' \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{c})$: This deterministic algorithm takes sk and a ciphertext c and outputs a message m'.

We require IND-CPA security and the following correctness conditions $\forall \mathsf{m}_1, \ldots, \mathsf{m}_n$:

- $\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, \mathsf{m}_i)) = \mathsf{Dec}(\mathsf{sk}, [\![\mathsf{m}_i]\!]) = \mathsf{m}_i$,
- $\mathsf{Dec}(\mathsf{sk}, \mathsf{Eval}(\mathsf{ek}, f, [\![\mathsf{m}_1]\!], \ldots, [\![\mathsf{m}_n]\!])) = \mathsf{Dec}(\mathsf{sk}, [\![f(\mathsf{m}_1, \ldots, \mathsf{m}_n)]\!])$.

Additively HE. If the scheme supports only addition, then it is called *additivelly homomorphic encryption (AHE)*. Schemes such as [127, 159] are additively homomorphic and have the following properties:

- Addition: $\forall \mathsf{m}_1, \mathsf{m}_2, [\![\mathsf{m}_1]\!] \cdot [\![\mathsf{m}_2]\!] = [\![\mathsf{m}_1 + \mathsf{m}_2]\!]$,
- Multiplication with constant: $\forall \mathsf{m}_1, \mathsf{m}_2, [\![\mathsf{m}_1]\!]^{\mathsf{m}_2} = [\![\mathsf{m}_1 \cdot \mathsf{m}_2]\!]$,
- Xor: $\forall a, b \in \{0,1\}, \textsc{Xor}([\![a]\!], b) = [\![a \oplus b]\!] = [\![1]\!]^b \cdot [\![a]\!]^{(-1)^b}$.

The scheme of Paillier [159] has been used to evaluate protocols described in Chapters 4, 5, and 7.

Somewhat/Fully Homomorphic Encryption. *Somewhat HE (SHE)* and *Fully HE (FHE)* allow arithmetic circuits to be evaluated directly on ciphertexts. For SHE the depth of the circuit must be small enough to avoid decryption error. In this thesis, we focus on lattice-based SHE/FHE schemes that allow many chained additions and multiplications to be computed on plaintexts homomorphically. These schemes are usually defined over a ring $\mathbb{Z}[X]/(X^N+1)$, where N might be a power of 2. The encryption algorithm Enc adds "noise" to the ciphertext which increases during homomorphic evaluation. While the addition of ciphertexts increases the noise slightly, the multiplication increases it rapidly [36]. If the noise becomes too large then correct decryption is no longer possible. To prevent this from happening, one can either keep the circuit's depth of the function f low enough or use the *refresh* algorithm. This algorithm consists either of a *bootstrapping* procedure, which takes a ciphertext with large noise and outputs a ciphertext of the same message with a fixed amount of noise; or a *key-switching procedure*, which takes a ciphertext under one key and outputs a ciphertext of the same message under a different key [4]. For performance reasons, it is usual to keep the circuit's depth low by designing protocols using so-called leveled fully homomorphic encryption. A *leveled homomorphic encryption* is an FHE with an extra parameter L such that the scheme can evaluate all circuits of depth at most L without bootstrapping.

Homomorphic Operations. We assume a BGV type homomorphic encryption scheme [36] and describe homomorphic operations as implemented in HElib [100], which will be used to evaluate some protocols of this thesis. Plaintexts can be encrypted using an integer representation (an integer x_i is encrypted as $[\![x_i]\!]$) or a binary representation (each bit of the bit representation $x_i^{\mathsf{b}} = x_{i\mu} \ldots x_{i1}$ is encrypted). We

describe below homomorphic operations in the binary representation (i.e., arithmetic operations mod 2). They work similarly in the integer representation.

The FHE scheme might support Smart and Vercauteren's ciphertext packing (SVCP) technique [173] to pack many plaintexts in one ciphertext. Using SVCP, a ciphertext consists of a fixed number s of slots, each capable of holding one plaintext, i.e. $[\![\cdot|\cdot|\ldots|\cdot]\!]$. The encryption of a bit b replicates b to all slots, i.e., $[\![b]\!] = [\![b|b|\ldots|b]\!]$. However, we can also pack the bits of x_i^{b} in one ciphertext and will denote it by $[\![\vec{x}_i]\!] = [\![x_{i\mu}|\ldots|x_{i1}|0|\ldots|0]\!]$.

The computation relies on some built-in routines, that allow homomorphic operations on encrypted data. The relevant routines for our scheme are: addition (SheAdd), multiplication (SheMult) and comparison (SheCmp). These routines are compatible with the ciphertext packing technique (i.e., operations are replicated on all slots in a SIMD manner).

Addition: The routine SheAdd takes two or more ciphertexts and performs a component-wise addition modulo two, i.e., we have:

$$\textsc{SheAdd}([\![b_{i1}|\ldots|b_{is}]\!], [\![b_{j1}|\ldots|b_{js}]\!]) = [\![b_{i1} \oplus b_{j1}|\ldots|b_{is} \oplus b_{js}]\!].$$

Multiplication: Similarly, SheMult performs component-wise multiplication modulo two, i.e., we have:

$$\textsc{SheMult}([\![b_{i1}|\ldots|b_{is}]\!], [\![b_{j1}|\ldots|b_{js}]\!]) = [\![b_{i1} \cdot b_{j1}|\ldots|b_{is} \cdot b_{js}]\!].$$

We will also denote addition and multiplication by $\boxplus$ and $\boxdot$, respectively.

Greater-Than Comparison: Let x_i, x_j be two integers, $b_{ij} = [x_i > x_j]$ and $b_{ji} = [x_j > x_i]$, the routine SheCmp takes $[\![x_i^{\mathsf{b}}]\!], [\![x_j^{\mathsf{b}}]\!]$, compares x_i and x_j and returns $[\![b_{ij}]\!]$, $[\![b_{ji}]\!]$:

$$([\![b_{ij}]\!], [\![b_{ji}]\!]) \leftarrow \textsc{SheCmp}([\![x_i^{\mathsf{b}}]\!], [\![x_j^{\mathsf{b}}]\!]).$$

Note that, if the inputs to SheCmp encrypt the same value, then the routine outputs two ciphertexts of 0. This routine implements the comparison circuit described in [45, 46, 47].

Full Adder: Let $b_{i1}, \ldots, b_{in}$ be n bits such that $r_i = \sum_{j=1}^{n} b_{ij}$ and let $r_i^b = r_{i\log n}, \ldots, r_{i1}$ be the bit representation of r_i. The routine SheFadder implements a full adder on $[\![b_{i1}]\!], \ldots, [\![b_{in}]\!]$ and returns $[\![r_i^b]\!] = ([\![r_{i\log n}]\!], \ldots, [\![r_{i1}]\!])$.

Equality Testing: There is no built-in routine for an equality check in HElib [100]. We implemented it using SheCmp and SheAdd. Let x_i and x_j be two $|k|$-bit integers. We use SheEqual to denote the equality check and implement $\textsc{SheEqual}([\![x_i^{\mathsf{b}}]\!], [\![x_j^{\mathsf{b}}]\!])$ by computing:

- $([\![b_i']\!], [\![b_i'']\!]) = \textsc{SheCmp}([\![x_i^{\mathsf{b}}]\!], [\![x_j^{\mathsf{b}}]\!])$ and
- $[\![\beta_i]\!] = ([\![b_i']\!] \boxplus [\![b_i'']\!] \boxplus [\![1]\!])$, which results in $\beta_i = 1$ if $x_i = x_j$ and $\beta_i = 0$ otherwise.

Shift: If ciphertext packing is enabled, then we also assume that HE supports shift operations. Given a packed ciphertext $[\![b_1|\ldots|b_s]\!]$, the *shift left* operation shifts all slots to the left by a given offset, using zero-fill, i.e., shifting $[\![b_1|\ldots|b_s]\!]$ by i positions returns $[\![b_i|\ldots|b_s|0|\ldots|0]\!]$. The *shift right* operation is defined similarly for shifting to the right.

Threshold HE. A *threshold homomorphic encryption (THE)* [32, 57] allows to share the private key to the parties using a threshold secret sharing scheme such that a subset of parties is required for decryption. Hence, instead of sk as above, the

key generation outputs a set of shares $\mathbb{SK} = \{\langle \mathsf{sk} \rangle_1, \ldots, \langle \mathsf{sk} \rangle_n\}$ which are distributed to the clients. The decryption algorithm is replaced by the following algorithms:

- $\mathsf{m}'_i \leftarrow \mathsf{Dec}_p(\langle \mathsf{sk} \rangle_i, \mathsf{c})$: The deterministic partial decryption algorithm takes a ciphertext c and a share $\langle \mathsf{sk} \rangle_i \in \mathbb{SK}$ of the private key and outputs a partial decryption result m'_i.
- $\mathsf{m}' \leftarrow \mathsf{Dec}_f(\mathbb{M}_t)$: The deterministic final decryption algorithm takes a subset $\mathbb{M}_t = \{\mathsf{m}'_{j_1}, \ldots, \mathsf{m}'_{j_t}\} \subseteq \{\mathsf{m}'_1, \ldots, \mathsf{m}'_n\}$ of partial decryption shares and outputs a message m'.

We refer to it as *threshold decryption*. It is correct if for all $\mathbb{M}_t = \{\mathsf{m}'_{j_1}, \ldots, \mathsf{m}'_{j_t}\}$ such that $|\mathbb{M}_t| \geq t$ and $\mathsf{m}'_{j_i} = \mathsf{Dec}_p(\langle \mathsf{sk} \rangle_{j_i}, [\![\mathsf{m}]\!])$, it holds $\mathsf{m} = \mathsf{Dec}_f(\mathbb{M}_t)$.
When used in a protocol, we denote by *combiner* the party which is responsible to execute the algorithm $\mathsf{Dec}_f()$. Depending on the protocol, the combiner can be any party. It receives a set $\mathbb{M}_t = \{\mathsf{m}'_{j_1}, \ldots, \mathsf{m}'_{j_t}\}$ of partial decryption results, runs $\mathsf{m} \leftarrow \mathsf{Dec}_f(\mathbb{M}_t)$ and publishes the result or moves to the next step of the protocol specification.

For the protocol described in Chapter 7, we have implemented a variant of ElGamal encryption [76] on elliptic curves [127, 128] that is additively homomorphic. The threshold decryption has been implemented using Shamir's scheme [170] as described in the online book of Boneh and Shoup [33].

2.1.5 Multiparty Computation

Secure multiparty computation (SMC) is a cryptographic technique that allows several parties to compute a function on their private inputs without revealing any information other than the function's output. A classic example in the literature is the so-called *Yao's Millionaire's problem* introduced in [195]. Two millionaires are interested in knowing which of them is richer without revealing their actual wealth. Formally, let there be a set of n parties $P_1, \ldots, P_n$.

Definition 2.1.10. *An n-ary* functionality, *denoted $\mathcal{F} : (\{0,1\}^*)^n \rightarrow (\{0,1\}^*)^n$, is a random process mapping sequences of the form $x = (x_1, \ldots, x_n)$ into sequences of random variables, $\mathcal{F}(x) = (\mathcal{F}_1(x), ..., \mathcal{F}_n(x))$. The semantics is that for every i, the party P_i, initially holds an input x_i, and wishes to obtain the i-th element in $\mathcal{F}(x_1, \ldots, x_n)$, denoted $\mathcal{F}_i(x_1, \ldots, x_n)$ [90].*

Functionalities considered in this thesis are search tree, decision tree and ranked-based statistics such as minimum, median, maximum. The goal of a secure protocol is to reveal to each party P_i the output $f_i(x_1, \ldots, x_n)$ and nothing else. The view of a party defines what it can learn from the protocol execution.

Definition 2.1.11. *The view of the i-th party during an execution of the protocol on input $(x_1, \ldots, x_n)$ is denoted by: $\mathsf{View_i}(x_1, \ldots, x_n) = \{x_i, r_i, m_{i1}, m_{i2}, \ldots\}$, where r_i represents the outcome of the i-th party's internal coin tosses, and m_{ij} represents the j-th message it has received or sent.*

The security of SMC protocols is often defined by comparison to an *ideal model*. In that model parties privately send their input to a trusted third party (TTP). Then the TTP computes the outcome of the function on their behalf, sends the corresponding result to each party, and deletes the private inputs. In the *real model*, parties emulate the ideal model by executing a cryptographic protocol to perform the computation. At the end, only the result should be revealed and nothing else. An SMC protocol is

then said to be secure if the adversary can learn only the result of the computation and data that can be deduced from this result and known inputs [58, 82, 90].

An important issue to consider when defining the security of SMC is the adversary's power. There exist many security models, but the *semi-honest* and the *malicious* adversary model are the most popular [58, 90]. In the semi-honest (a.k.a *honest-but-curious*) model parties behave *passively* and follow the protocol specification, but the adversary tries to learn more than allowed by inspecting the internal state of corrupted parties. A protocol is proven secure if the views of the adversary can be efficiently simulated given only inputs and outputs of corrupted parties. This proof technique is called *simulation paradigm* [90].

Definition 2.1.12. *Let $\mathcal{F} : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ be a n-ary functionality, where $\mathcal{F}_i(x_1,\ldots,x_n)$ is the i-th element of $\mathcal{F}(x_1,\ldots,x_n)$. For $I = \{i_1,\ldots,i_t\} \subseteq \{1,\ldots,n\}$ (i.e., I contains indexes of corrupted parties), we let*

$$\mathcal{F}^I(x_1,\ldots,x_n) = (\mathcal{F}_{i_1}(x_1,\ldots,x_n),\ldots,\mathcal{F}_{i_t}(x_1,\ldots,x_n)).$$

A protocol Π t-privately computes the functionality $\mathcal{F}$ in the semi-honest model if there exists a polynomial-time simulator $\mathsf{Sim}_I^{\mathsf{f}}$ such that: for every $I \subseteq \{1,\ldots,n\}, |I| = t$, it holds:

$$\mathsf{Sim}_I^{\mathsf{f}}((x_{i_1},\ldots,x_{i_t}),\mathcal{F}^I(x_1,\ldots,x_n)) \stackrel{c}{\equiv} \mathsf{View}_I^{\Pi}(x_1,\ldots,x_n),$$

where $\mathsf{View}_i^{\Pi}(x_1,\ldots,x_n)$ denote the view of the i-th party in protocol Π and

$$\mathsf{View}_I^{\Pi}(x_1,\ldots,x_n) = (\mathsf{View}_{i_1}^{\Pi}(x_1,\ldots,x_n),\ldots,\mathsf{View}_{i_t}^{\Pi}(x_1,\ldots,x_n)).$$

In contrast, a malicious adversary is *active* and instructs corrupted parties to deviate from the protocol specification. Besides trying to learn more information, it can delete, add, modify messages to influence the outcome of the protocol, or prevent the protocol to terminate correctly. Any protocol secure against a semi-honest adversary can be transformed into a protocol secure against a malicious adversary using Goldreich's compiler [88, 90]. The compiler works by transforming semi-honest instructions into new instructions that preserve the correctness and force each party to either behave in a semi-honest manner or to be detected. In case a party is detected cheating, the protocol aborts. The malicious adversary model is preferred because it guarantees that no adversary will successfully attack the protocol. However, maliciously secure protocols are in general less efficient than semi-honestly secure ones.

Often a functionality f can be privately reducible to a functionality g meaning that a protocol Π_f privately implementing f uses a protocol Π_g privately implementing g as sub-protocol. The composition theorem helps to simplify the security proof of Π_f and loosely states that given a proven sub-protocol Π_g, it is only necessary to prove the security of the composed protocol Π_f using Π_g as a black box [90].

Theorem 2.1.13. *Let f be functionality that is privately reducible to a functionality g and let there be a protocol Π_g for privately computing g. Then there exists a protocol Π_f for privately computing f.*

2.1.6 Basic Protocols

Oblivious Transfer. Oblivious Transfer (OT) is a fundamental cryptographic protocol that allows a receiver to choose 1 out of n values held by a sender, without

revealing to the sender which value was chosen, while revealing to the receiver only the chosen value [90].

Definition 2.1.14 (OT functionality)**.** *Let $n, n \geq 2$ be an integer, $i \leq n$ be an index and $X = (x_1, \ldots, x_n) \in (\{0,1\}^l)^n$ be an array. An* Oblivious transfer (OT) *is a functionality that takes the array X from a party called* sender*, the index i from a party called* receiver*, and outputs x_i to the receiver and nothing to the sender. Formally:*

$$\mathcal{F}_{\mathsf{OT}}((x_1, \ldots, x_n), i) \to (\varepsilon, x_i),$$

where ε denotes the empty string.

Definition 2.1.15 (OT Security)**.** *Let $n, n \geq 2$ be an integer, $i \leq n$ be the input of the receiver and $X = (x_1, \ldots, x_n) \in (\{0,1\}^l)^n$ be the input of the sender in the OT functionality $\mathcal{F}_{\mathsf{OT}}$. A protocol Π_{OT} privately computes the OT functionality $\mathcal{F}_{\mathsf{OT}}$ in the semi-honest model if the following conditions hold:*

- *there exists a probabilistic polynomial time algorithm $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{ot}}$ that simulates the sender's view $\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{OT}}}$ of the protocol Π_{OT} given $X = (x_1, \ldots, x_n)$ only:*

$$\mathsf{Sim}_{\mathsf{S}}^{\mathsf{ot}}((x_1, \ldots, x_n), \varepsilon) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{OT}}}((x_1, \ldots, x_n), i).$$

- *there exists a probabilistic polynomial time algorithm $\mathsf{Sim}_{\mathsf{R}}^{\mathsf{ot}}$ that simulates the receiver's view $\mathsf{View}_{\mathsf{R}}^{\Pi_{\mathsf{OT}}}$ of the protocol Π_{OT} given i and x_i only:*

$$\mathsf{Sim}_{\mathsf{R}}^{\mathsf{ot}}(i, x_i) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{R}}^{\Pi_{\mathsf{OT}}}((x_1, \ldots, x_n), i).$$

A special case of OT is 1-out-of-2 OT (OT_2^1). In a 1-out-of-2 OT, the sender has two inputs x_0 and x_1 and the receiver has an index $i \in \{0,1\}$. After the protocol the receiver learns only x_i and the sender learns nothing. In this thesis, we used the scheme of Naor and Pinkas [153] as implemented in the frameworks SCAPI [75] and ObliVM [144]. The scheme of Naor and Pinkas is illustrated in Protocol 2.1. It is possible to extend a few oblivious transfers into many. The idea is to generate few seed OTs using public key cryptography which can then be extended to any number of OTs using only symmetric key primitives. This can be seen as the OT equivalent of hybrid encryption and is referred to as *OT extension protocol*. The efficient OT extension protocols of Ishai et al. [107] and Asharov et al. [9] have been used to evaluate some protocols described in this thesis. The protocol described in Chapter 5 can be instantiated with 1-out-of-n OT. We focus on the 1-out-of-n OT scheme of Naor and Pinkas [154] that uses $\log n$ OT_2^1 protocols to realize OT_n^1. We briefly review Naor and Pinkas's 1-out-of-n OT scheme and refer to [154] for details. Let $X = [x_1, \ldots x_n]$ be an array (sender input), i an index (receiver input), $h = \log(n)$ and F_k a PRF. Let ι be an index in $[1, n]$ and $\iota_1 \ldots \iota_h$ its bit representation. The sender generates h pairs $(k_1^0, k_1^1), \ldots, (k_h^0, k_h^1)$ of symmetric keys. For each $x_j \in X$, the sender uses the keys $k_1^{j_1}, \ldots, k_h^{j_h}$ to generate a symmetric ciphertext c_j. Then the sender sends all ciphertexts $c_1, \ldots, c_n$ to the receiver. Both parties then run h times an OT_2^1 allowing the receiver to select $k_u^{i_u}$ among (k_u^0, k_u^1). Finally, the receiver uses the keys $k_1^{i_1}, \ldots, k_h^{i_h}$ to decrypt the ciphertext c_i. In this scheme, sender and receiver perform $\mathcal{O}(n)$ and $\mathcal{O}(\log(n))$ operations respectively. It is described in Protocol 2.2.

Oblivious RAM. Oblivious RAM (ORAM) is a cryptographic primitive that allows a client to outsource its encrypted storage to an untrusted server and to hide the data

Sender		**Receiver**
Input: (x_0, x_1)		Input: $b \in \{0,1\}$
Output: ε		Output: x_b
$C \leftarrow_\$ \mathbb{Z}_p$		$k \leftarrow_\$ \mathbb{Z}_p$
	$\xrightarrow{C}$	
		$h \leftarrow (1-b) \cdot g^k + C \cdot b \cdot g^{-k}$
	$\xleftarrow{h}$	
$r \leftarrow_\$ \{1, \ldots, p\}$		
$e_0 \leftarrow x_0 \oplus \mathsf{H}(h^r, 0)$		
$e_1 \leftarrow x_1 \oplus \mathsf{H}((C/h)^r, 1)$		
	$\xrightarrow{g^r, e_0, e_1}$	
		$x_b \leftarrow \mathsf{H}((g^r)^k, b) \oplus e_b$

Protocol 2.1: Naor and Pinkas 1-out-of-2 OT [153]

Sender		**Receiver**
Input: $(x_1, \ldots, x_n)$		Input: $i \in \{1, \ldots, n\}$
Output: ε		Output: x_i
	Let $h \leftarrow \log n$	
	For $j \in \{1, \ldots, n\}$, let $j = \sum_{u=1}^{h} j_u 2^{u-1}$	
Let F_k be a PRF and		
$k_u^0, k_u^1 \leftarrow_\$ \{0,1\}^\lambda, 1 \le u \le h$		
$c_j \leftarrow x_j \oplus \oplus_{u=1}^{h} F_{k_j^{j_u}}(j), 1 \le j \le n$		
	$\xrightarrow{c_1, \ldots, c_n}$	
	k_u^0, k_u^1 / i_u / OT_2^1 $1 \le u \le h$ / $k_u^{i_u}$	
		$x_i \leftarrow c_i \oplus \bigoplus_{u=1}^{h} F_{k_u^{i_u}}(i)$

Protocol 2.2: Naor and Pinkas 1-out-of-n OT [154]

access patterns of the client from the server. The protocol described in Chapter 5 can be instantiated with 1-out-of-n OT as well as with ORAM. We will adopt tree-based ORAM which has sublinear costs and has been introduced by Shi et al. [171].

Definition 2.1.16 (ORAM functionality)**.** *Let D be a set of n data blocks. For each data block $d \in D$ let $u \in \{1, \ldots, n\}$ denote the block identifier. An* Oblivious RAM *functionality is a finite composition sequence of the two following functionalities between a client and a server :*

- $\mathsf{ReadAndRemove}$*: This functionality takes a private block identifier u from the client and the data set $D = [d_1, \ldots, d_u, \ldots, d_n]$ from the server. It retrieves the data block d_u, removes it from the data set D and returns d_u to the client:*

$$\mathcal{F}_{\mathsf{RaR}}([d_1, \ldots, d_u, \ldots, d_n], u) \rightarrow ([d_1, \ldots, \varepsilon, \ldots, d_n], d_u).$$

- Add*: This functionality takes a private block identifier u, a new private data block d'_u from the client and the data set $D = [d_1, \ldots, d_u, \ldots, d_n]$ from the server. It retrieves the old data block d_u, replaces it by the new data block d'_u in the data set D and returns the old data bock d_u to the client:*

$$\mathcal{F}_{\mathsf{Add}}([d_1, \ldots, d_u, \ldots, d_n], (u, d'_u)) \rightarrow ([d_1, \ldots, d'_u, \ldots, d_n], d_u).$$

A data request *is a tuple* $(\mathsf{op}, \mathsf{arg})$ *where:*

- *the first element* op *denotes an operation in* $\{\mathsf{ReadAndRemove}, \mathsf{Add}\}$ *and*
- *the second element* arg *is a block identifier u if* $\mathsf{op} = \mathsf{ReadAndRemove}$*. Otherwise, if* $\mathsf{op} = \mathsf{Add}$*, then* arg *consists of a tuple (u, d_u) where u is a block identifier and d is a data block.*

A data request sequence *of length m is a sequence $\vec{y} = ((\mathsf{op}_1, \mathsf{arg}_1), \ldots, (\mathsf{op}_m, \mathsf{arg}_m))$ of m data requests such that if $(\mathsf{op}_i, \mathsf{arg}_i) = (\mathsf{Add}, (u_i, d_{u_i}))$ then the $i-1$-st data request must satisfy $(\mathsf{op}_{i-1}, \mathsf{arg}_{i-1}) = (\mathsf{ReadAndRemove}, u_i)$.*

Definition 2.1.17 (ORAM Security)**.** *Let the ORAM functionality be defined as above. For a data request sequence $\vec{y} = ((\mathsf{op}_1, \mathsf{arg}_1), \ldots, (\mathsf{op}_m, \mathsf{arg}_m))$, let* ops *denote the sequence of operations associated with $\vec{y}$, i.e.,* $\mathsf{ops}(\vec{y}) = (\mathsf{op}_1, \ldots, \mathsf{op}_m)$*. Moreover, let* $\mathsf{View}(\vec{y})$ *denote the* access pattern *that leaks during the execution of $\vec{y}$. A protocol* Π_{ORAM} *securely implements the ORAM functionality if for two data request sequences $\vec{y}$ and $\vec{z}$ of the same length with* $\mathsf{ops}(\vec{y}) = \mathsf{ops}(\vec{z})$*, their access patterns* $\mathsf{View}^{\Pi_{\mathsf{ORAM}}}(\vec{y})$ *and* $\mathsf{View}^{\Pi_{\mathsf{ORAM}}}(\vec{z})$ *are computationally indistinguishable by anyone but the client:*

$$\mathsf{View}^{\Pi_{\mathsf{ORAM}}}(\vec{y}) \stackrel{c}{\equiv} \mathsf{View}^{\Pi_{\mathsf{ORAM}}}(\vec{z}).$$

We now briefly review tree-based ORAM and refer to the literature [171, 189] for more details.

Data Structure. Let n be the number of data blocks. The data is stored as *blocks* at the server in a complete binary tree of depth $L = \log n + 1$. A node of the tree is called *bucket* containing a fixed number Z of blocks. Each block is stored encrypted using a semantically secure encryption scheme and consists of an *index* idx of the block (its logical address), a *label* specifying a leaf identifier of the path on which the block resides, and the actual *data*: $\{\mathsf{idx}||\mathsf{label}||\mathsf{data}\}$. A *position map* is stored at the client and maps each block idx to the corresponding leaf label. The leaf labels are chosen randomly between $0, \ldots, n-1$ such that each label is associated with a path from

u	v	w
0	0	0
0	1	0
1	0	0
1	1	1

u	v	w
k_u^0	k_v^0	k_w^0
k_u^0	k_v^1	k_w^0
k_u^1	k_v^0	k_w^0
k_u^1	k_v^1	k_w^1

Garbled Table
$Enc_{k_u^0,k_v^0}(k_w^0)$
$Enc_{k_u^0,k_v^1}(k_w^0)$
$Enc_{k_u^1,k_v^0}(k_w^0)$
$Enc_{k_u^1,k_v^1}(k_w^1)$

Table 2.1: Garbled Table Illustration

the root to the corresponding leaf. Block labels are reassigned as they are accessed. Tree-based ORAM maintains the invariant that a block resides on the path from the root to the corresponding leaf.

Operations. Tree-based ORAM supports three operations. Given an index idx and the corresponding label, the *read and remove* operation $\mathsf{ReadAndRemove}$ fetches and removes the block idx from the path. The retrieved block is eventually updated, re-encrypted and written back using the *add* operation Add. The *eviction* operation Evict randomly evicts blocks to child nodes of the node where they reside. Tree-based ORAM schemes differ mostly on their eviction strategy but adopt a similar data structure as described above.

Recursion. Instead of storing the position map at the client, it can be stored recursively in smaller ORAMs at the server using the same data structure as above. As a result, the client stores only a constant amount of metadata locally.

Tree-based ORAM has been improved by other works, including [189] which has been implemented in ObliVM [144], the secure computation framework used in Chapter 5 of this thesis.

2.1.7 Garbled Circuits

A Garbled Circuit (GC) can be used to execute any function privately between two parties. In this section, we recall the idea of GC protocol and refer to [20, 75, 139, 140, 161, 195] for more details. Let f be a function over two inputs x_i and x_j, then a garbling scheme consists of a five-tuple of algorithms $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De}, \mathsf{ev})$.

- On input f, a random seed s and a security parameter $\lambda \in \mathbb{N}$, the garbling algorithm Gb returns a triple of strings $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, s, f)$.
- The string e describes an encoding function, $\mathsf{En}(e, \cdot)$, that maps the bit representation $x_i^\mathsf{b} = x_{i\mu} \dots x_{i1}$ and $x_j^\mathsf{b} = x_{j\mu} \dots x_{j1}$ of the initial inputs x_i, x_j to garbled inputs $\bar{x}_i = \mathsf{En}(e, x_i^\mathsf{b})$, $\bar{x}_j = \mathsf{En}(e, x_j^\mathsf{b})$.
- The string F describes a garbled function, $\mathsf{Ev}(F, \cdot, \cdot)$, that maps each pair of garbled inputs $(\bar{x}_i, \bar{x}_j)$ to a garbled output $\bar{y} = \mathsf{Ev}(F, \bar{x}_i, \bar{x}_j)$.
- The string d describes a decoding function, $\mathsf{De}(d, \cdot)$, that maps a garbled output $\bar{y}$ to a final output $y = \mathsf{De}(d, \bar{y})$.
- The original function f is encoded as a circuit that can be evaluated by the function:

$$\mathsf{ev}(f, \cdot, \cdot) : \{0,1\}^\mu \times \{0,1\}^\mu \to \{0,1\}^\nu$$

The garbling scheme is correct if $\mathsf{De}(d, \mathsf{Ev}(F, \mathsf{En}(e, x_i), \mathsf{En}(e, x_j))) = \mathsf{ev}(f, x_i, x_j)$ [20].

A GC protocol is a 2-party protocol consisting of a generator (Gen) and an evaluator (Eva) with input x_i and x_j respectively. It consists of the following steps:

- Garbling: On input f, s and λ, Gen runs $(F, e, d) \leftarrow \mathsf{Gb}(1^\lambda, s, f)$ and parses e as $(k^0_{u_1}, k^1_{u_1}, \cdots, k^0_{u_\mu}, k^1_{u_\mu}, k^0_{v_1}, k^1_{v_1}, \cdots, k^0_{v_\mu}, k^1_{v_\mu})$ and transforms x_i into a garbled input $\bar{x}_i$ using $x^{\mathsf{b}}_i = x_{i\mu} \dots x_{i1}$ and $\mathsf{En}(e, \cdot)$, i.e., $\bar{x}_i = (k^{x_{i\mu}}_{u_1}, \cdots, k^{x_{i1}}_{u_n}) \leftarrow \mathsf{En}(e, x^{\mathsf{b}}_i)$.
- Sending GC and garbled input: Then Gen sends F, d and $\bar{x}_i$ to Eva.
- Oblivious Transfer: Now the parties execute an OT protocol with Eva having selection string $x^{\mathsf{b}}_j = x_{j\mu} \dots x_{j1}$ and Gen having inputs $(k^0_{v_1}, k^1_{v_1}, \cdots, k^0_{v_n}, k^1_{v_n})$. As a result, Eva obtains $\bar{x}_j = (k^{x_{j\mu}}_{v_1}, \cdots, k^{x_{j1}}_{v_n})$ and Gen learns nothing.
- Evaluation: Finally, Eva evaluates and outputs $y = \mathsf{De}(d, \mathsf{Ev}(F, \bar{x}_i, \bar{x}_j))$.

To garble the function f the generator first transforms f into a Boolean circuit C_f. Assume each gate in C_f has input wires u, v and output wire w. The generator chooses for each wire two symmetric keys representing bits 0 or 1: (k^0_u, k^1_u), (k^0_v, k^1_v), (k^0_w, k^1_w). Then she uses the input wires' keys to encrypt the output wires' keys according to the truth table of the gate as illustrated in Table 2.1, where $Enc_{k_u,k_v}(k_w)$ is a double-key deterministic symmetric encryption, e.g.: $Enc_{k_u,k_v}(k_w) = \mathsf{H}(k_u \| k_v) \oplus k_w$, where H is a cryptographic hash function and $k_u \| k_v$ stands for the concatenation of keys k_u, k_v [105, 131]. The rows are then randomly permuted resulting in a *garbled gate* or *garbled table*. The set of all garbled tables constitutes the garbled circuit that is sent to the evaluator. To evaluate the garbled circuit, Eva uses the keys in $\bar{x}_i$ and $\bar{x}_j$ to successively decrypt the garbled tables. There exist many optimizations that make garbled circuit protocol efficient including point-and-permute [17], row reduction [155], FreeXOR [131], fixed-key block cipher [19], HalfGate [197]. The garbled circuit protocol has been used in Chapters 4, 7, and 5 as implemented in the secure computation frameworks SCAPI [75] and ObliVM [144].

2.2 Applications

The protocols described in this thesis have applications in machine learning and benchmarking. In this section, we briefly review these application domains and refer to the respective literature for more details.

2.2.1 Machine Learning

Our digital world overwhelms us with information. The gap between generation of data and our understanding of it is growing rapidly. Fortunately, machine learning (ML) techniques allow us to find patterns in data. A machine learning process consists of two phases. In the first phase or *learning phase*, a model or classifier is built on a possibly large set of training data. In the second phase, the model can then be used to classify new data.

Due to their simplicity and ease of use, decision trees are widespread machine learning models used for data classification with many applications in areas such as health care, remote diagnostic, spam filtering, etc. A decision tree consists of two types of nodes. Internal nodes are *decision nodes* that are used to compare an attribute to a constant. *Leaf nodes* give a classification that applies to all instances that reach the leaf. To classify an unknown instance, the tree is traversed according to the values of the attributes tested in successive nodes, and when a leaf is reached the instance is classified according to the class assigned to that leaf [192].

Definition 2.2.1 (Decision Tree)**.** *An* attribute vector *is an array of integers each array element measuring a specific feature or attribute. A* decision tree *is a binary tree consisting of* internal nodes *(or* decision nodes*) and* leaf nodes*. Each decision node consists of a* threshold value *that is compared with a corresponding* attribute value *while traversing the decision tree. Moreover, a decision node has two child nodes which maybe decision node as well or leaf node. Each leaf node consists of a* label *that is returned as a result of the classification if the tree traversal reaches that leaf. Leaf nodes have no child node.*

Machine learning classifiers are valuable tools in many areas such as health care, finance, spam filtering, intrusion detection, remote diagnosis, supply chain, etc. On the one hand, the classifier requires access to user's data, which is most of the time sensitive information such as medical or financial data, to perform its task. On the other hand, the model itself may contain intellectual property that needs to be protected. For example, a bank that uses a decision tree for credit assessment of its customers may not want to reveal any internal information about the model, because competitors may benefit from it. Moreover, the model may have been built on sensitive data. It is known that white-box access and sometimes even black-box access to a machine learning model may allow so-called *model inversion attacks* [81, 179, 194]. In a black-box setting, the adversary is only allowed to query the model, whereas in a white-box setting the adversary is in possession of the model. Therefore, it is crucial to investigate techniques that would allow benefiting from machine learning classifiers while preserving the privacy of the data. In this thesis, we address the problem of evaluating decision tree classifier on private data.

2.2.2 Benchmarking

Benchmarking is a management process where a company compares its *key performance indicator (KPI)* to the statistics of the same KPIs of a group of competitors from a peer group. A key performance indicator (KPI) is a statistical quantity measuring the performance of a business process. Examples of KPIs from different company operations are make cycle time (manufacturing), cash flow (financial), and employee fluctuation rate (human resources). A peer group is a group of similar companies, usually competitors, wanting to compare against each other. Examples formed along different characteristics include car manufacturers (industry sector), Fortune 500 companies in the United States (revenue and location).

Let assume that there are n companies $C_1, \ldots, C_n$ each with a value x_i and let $x = (x_1, \ldots, x_n)$ and $\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_n)$ be the sorted the vector of sorted values of x, i.e., $\tilde{x}_1 < \ldots < \tilde{x}_n$. A benchmarking platform may provide the following statistics:

- Aggregate Statistics:
  - the Mean: $\textsc{Mean}(x) = \frac{1}{n}\sum_{i=n}^{n} x_i$,
  - the Variance: $\textsc{Var}(x) = \frac{1}{n}\sum_{i=n}^{n}(x_i - \textsc{Mean}(x))^2 = \frac{1}{n}\sum_{i=n}^{n} x_i^2 - \textsc{Mean}(x)^2$.
- Rank-based Statistics:
  - the Median: $\textsc{Median}(x) = \tilde{x}_{\lceil \frac{n}{2} \rceil}$,
  - the Minimum: $\textsc{Min}(x) = \tilde{x}_1$,
  - the Maximum: $\textsc{Max}(x) = \tilde{x}_n$,
  - the Top Quartile: $\textsc{Toq}(x) = \tilde{x}_{\lfloor \frac{n}{4} \rfloor - 1}$,

– the Bottom Quartile: $\textsc{Boq}(x) = \tilde{x}_{\lfloor \frac{3n}{4} \rfloor + 1}$.

In this thesis, we address the problem of computing ranked-based statistics using a communication model called server model. In this model, there are communication channels only between each client and the server and only clients have inputs to the computation and get the output of the computation.

2.3 Notation

In this section, we introduce some notations that will be used in this thesis. A summary is illustrated in Table 2.2. Further notations, that are not general for the whole thesis, may be introduced later in the respective chapter.

Parameters. A security parameter is a way of measuring how hard it is for an adversary to break a cryptographic scheme. We can distinguish between computational security parameters and statistical security parameters. On the one hand, a statistical security parameter, denoted by σ, i.e., $\sigma = 32$, determines the probability with which an adversary can break a cryptographic scheme given unlimited computing resources. On the other hand, a computational security parameter, denoted by λ, i.e., $\lambda = 128$, determines the bitlength of cryptographic keys and random values used in the scheme. It limits the success probability of a PPT adversary in breaking a cryptographic scheme. The computational security parameter also determines the bitlength of the associated ciphertexts. We will use κ to denote the bitlength of a ciphertext, i.e., $\kappa = 1024$ or $\kappa = 4096$. Hence, both bitlength of keys and bitlength of ciphertext are functions of λ. Both may equal the computational security parameter. This is the case in symmetric encryption schemes such as AES. The bitlength of ciphertext will help to determine the concrete communication complexity of a protocol. Another parameter that is necessary for complexity analysis is the bitlength of integer values, which will be denoted by μ, i.e., $\mu = 32$ or $\mu = 64$.

Parties. Protocols presented in this thesis involve two types of parties: the server and the client. The server will be denoted by S and the client by C. If a protocol consists of several clients then we will denote clients by C_i, C_j using their indexes or by C_a, C_o using their initial (i.e., C_a for data analyst or C_o for data owner). When referring to a protocol party that may be client or server, we use P_i, P_j.

Inputs/Outputs. Given some private inputs, a secure protocol performs some computations and produces some outputs. Input and outputs are integer values and will be denoted by x_i, y_i, z_i, where i may be the index of the corresponding party or just the i-th value of some party's input. We will use $|x_i|$ to denote the size of x_i, i.e., $|x_i| = \mu = 32$. As a protocol may perform computations that depend on single bits of an input we will use $x_i^{\mathsf{b}} = x_{i\mu} \dots x_{i1}$ to denote bit representation of x_i, where the b in the exponent stands for bit or binary, $x_{i\mu}$ is the most significant bit (MSB) and x_{i1} is the least significant bit (LSB).

Ciphertext. Our protocols require public key encryption schemes that have a homomorphic property. We will, therefore, use $\mathsf{pk}, \mathsf{sk}, \mathsf{ek}$ to denote public, private, and evaluation keys, respectively. If the keys belong to a party with index i, we will denote them by $\mathsf{pk}_i, \mathsf{sk}_i, ek_i$. The encryption of x_i under pk, respectively pk_i, will be denoted

by $[\![x_i]\!]$, respectively $[\![x_i]\!]_i$. For operations that perform computations on single encrypted bits, we will use $[\![x_i^{\mathsf{b}}]\!] = ([\![x_{i\mu}]\!], \ldots, [\![x_{i1}]\!])$ to denote the bitwise encryption of x_i. Ciphertext packing allows encrypting several plaintexts in the same ciphertext and will be denoted by $[\![\alpha_1|\ldots|\alpha_s]\!]$, where each α_i is a single plaintext and s is the number of plaintexts that can be packed in a single ciphertext. However, we can also pack the bits of x_i^{b} in one ciphertext and will denote it by $[\![\vec{x}_i]\!] = [\![x_{i\mu}|\ldots|x_{i1}|0|\ldots|0]\!]$.

Secret Shares. Alternatively to encrypting a value, we may use secret sharing and represent the secret sharing of a variable v by $\langle v \rangle$. To indicate that a specific secret share belongs to a party with index i, we use $\langle v \rangle_i$. If the variable v is a vector $v = (v_1, \ldots, v_k)$ consisting of several values then we will write $\langle v \rangle_i = \langle v_1, \ldots, v_k \rangle_i = (\langle v_1 \rangle_i, \ldots, \langle v_k \rangle_i)$.

Expressions. Let s_1 and s_2 be two bit strings of the same length, we use $s_1 \oplus s_2$ to denote the bitwise exclusive-or of s_1 and s_2. Let *bexpr* be a Boolean expression. We will use $[bexpr]$ (the Iverson bracket [97]) to denote the truth value (represented by 0 or 1) of the Boolean expression *bexpr*, i.e., $[x_i > x_j]$ is 1 if x_i is larger than x_j and 0 otherwise. We use $[bexpr]?expr_1 : expr_0$ to denote a conditional expression that returns the result of $expr_1$ if $bexpr = \text{true}$ and the result of $expr_0$ otherwise. Let $\mathbb{S}$ be any set, we use $a \leftarrow_{\$} \mathbb{S}$ to choose uniformly a random element a from $\mathbb{S}$. We also use $\{a_1, \ldots, a_t\} \leftarrow_{\$} \mathbb{S}$, to choose uniformly t random distinct elements from set $\mathbb{S}$. Variable assignment will be represented by $x \leftarrow expr$, for assigning the result of *expr* to x, i.e., $x \leftarrow a + b$. We will use $P_i \rightarrow P_j : \mathsf{msg}$ to indicate that party P_i sends a message msg to party P_j.

Miscellaneous. We let $\{0,1\}^l$ to denote the set of all bit strings of length l and ε to denote the empty string. We let $\mathbb{N}, \mathbb{Z}$ denote the set of natural numbers and integers respectively. For $a, b \in \mathbb{Z}$, we let $[a, b]$ denote the set of all integers from a to b. For a positive integer n, we write $\mathbb{Z}_n$ to denote the set of integers modulo n. We write $\mathbb{Z}_n^*$ to denote the set of integers modulo n that are coprime with n. We let $\mathfrak{S}_n$ denote the set of all permutations of $\{1, \ldots, n\}$. Elements of $\mathfrak{S}_n$ will be denoted by π or $\pi_1, \pi_2, \ldots$ when several permutations are required.

Symbol	Interpretation
λ	Computational Security parameter
σ	Statistical Security parameter
κ	Bitlength of ciphertext
μ	Bitlength of inputs
S	Symbol for the Server
C, C_i, C_j, C_a, C_o	Symbols for Clients
P_i, P_j	Symbols for arbitrary parties
x_i, y_i, z_i	Inputs/Outputs
$\lvert x_i \rvert$	Bitlength of integer x_i, e.g., $\lvert x_i \rvert = \mu$
$x_i^{\mathsf{b}} = x_{i\mu} \dots x_{i1}$	Bit representation of x_i with MSB $x_{i\mu}$ and LSB x_{i1}
$\mathsf{pk}, \mathsf{sk}, \mathsf{ek}$	public, private, evaluation keys
$\mathsf{pk}_i, \mathsf{sk}_i, ek_i$	public, private, evaluation keys of party with index i
$[\![x_i]\!]$	x_i's ciphertext under public key pk
$[\![x_i]\!]_j$	x_i's ciphertext under public key pk_j
$[\![x_i^{\mathsf{b}}]\!]$	Bitwise encryption $([\![x_{i\mu}]\!], \dots, [\![x_{i1}]\!])$ of x_i
$[\![\alpha_1 \lvert \dots \lvert \alpha_s]\!]$	Packed ciphertext containing plaintexts $\alpha_1, \dots, \alpha_s$
$[\![\vec{x}_i]\!]$	Packed ciphertext $[\![x_{i\mu} \lvert \dots \lvert x_{i1} \lvert 0 \lvert \dots \lvert 0]\!]$ of x_i^{b}
$\langle v \rangle$	Secret sharing of a variable v
$\langle v \rangle_i$	Secret share of v that belongs to the party with index i
$\langle v \rangle_i = \langle v_1, \dots, v_k \rangle_i = (\langle v_1 \rangle_i, \dots, \langle v_k \rangle_i)$	Secret sharing of a variable $v = (v_1 \dots, v_k)$
$s_1 \oplus s_2$	Bitwise exclusive-or of bit strings s_1 and s_2
$[bexpr]$	Truth value of Boolean expression $bexpr$
$[bexpr]?expr_1 : expr_0$	if ($bexpr$ = true) return $expr_1$ else return $expr_0$
$a \leftarrow\!\$\, \mathbb{S}$	Choose uniformly a random element a in set $\mathbb{S}$
$\{a_1, \dots, a_t\} \leftarrow\!\$\, \mathbb{S}$	Choose uniformly t random distinct elements in $\mathbb{S}$
$x \leftarrow expr$	Assign the value of $expr$ to variable x
$P_i \rightarrow P_j : \mathsf{msg}$	Party P_i sends a message msg to parties P_j
$\{0,1\}^l$	Set of all bit strings of length l
ε	The empty string
$\mathbb{N}, \mathbb{Z}$	Set of natural numbers and integers
$[a, b]$	Set of all integers from $a \in \mathbb{Z}$ to $b \in \mathbb{Z}$
$\mathbb{Z}_n$	Set of integers modulo n
$\mathbb{Z}_n^*$	Set of integers modulo n that are coprime with n
$\mathfrak{S}_n$	Set of all permutations of $\{1, \dots, n\}$
π, π_1, π_2	Permutations

Table 2.2: Notations

Chapter 3

Related Work

In this chapter, we review work that is related to this thesis. We start in Section 3.1 by reviewing related work to secure multiparty computation in the standard model and in the so-called server model. Implementing secure computation protocols is a difficult task. We will conclude the Section 3.1 by reviewing some frameworks that have been developed to help implementing efficient protocols. In Section 3.2, we review related protocols to OT and ORAM for which security definition and schemes relevant for this thesis have been presented in the preliminaries. Secure integer comparison is an important building block for our own protocols. Therefore, we first describe in Section 3.3 comparison protocols that are closely related to the protocols described in this thesis. Then, we briefly review other comparison protocols found in the literature. Searching over encrypted data is one of the topics addressed in this thesis. Related to Chapter 4, we describe in Section 3.4 order-preserving encryption schemes and other primitives that allow searching on encrypted data without decrypting it. Two other topics considered in the thesis are classification with private decision trees (Chapters 5 and 6) and computation of the k^{th}-ranked element (Chapter 7), for which related work will be described in Section 3.5 and Section 3.6, respectively.

3.1 Secure Multiparty Computation

3.1.1 Standard Model

Secure multiparty computation (SMC) allows several parties to compute a publicly known function on their private inputs without revealing any information other than the function's output. To privately compute a function, the parties run an interactive protocol without requiring the help of any third party. We will refer to this as the *standard model* of SMC. SMC has been introduced for the two-party case by Yao in his breakthrough papers [195, 196]. It has been extended for the multiparty case by Goldreich et al. [87], Ben-Or et al. [23], and Chaum et al. [43], where they also showed feasibility results. Since then, a large number of works have been done to refine the security definitions, strengthen the adversarial model, and improve the efficiency. Therefore, SMC has evolved from a theoretical curiosity to a powerful tool for building privacy-preserving applications. For more details, we refer to foundational references on SMC such as [20, 58, 90, 101]. We also refer to the paper of Evans et al. [77] which gives a state-of-the-art overview of the main constructions and the most currently active research areas. They also provide insights into SMC problems that are practically solvable today and how different threat models and assumptions impact the practicality of different approaches.

3.1.2 Server Model

In a multiparty party computation, it is sometimes convenient to rely on a server that helps the real parties (the clients) in the computation. We will refer to this as the *server model*. In this model, the server has no input to the computation and receives no output, but makes its computational resources available to the clients [114, 121].

The server model for generic secure computation was introduced in [78]. In [120], Kerschbaum proposed an approach allowing a service provider to offer a SMC service by himself. The cryptographic study of the server model was initiated in [113, 114].

In [155], Naor et al. considered another setting requiring two mutually distrustful servers and designed a specific protocol for secure auctions. Relying on multiple non-colluding servers requires a different business model for the service provider of a privacy-preserving service. The service provider has to share benefits with an almost equal peer offering its computational power [42, 120].

3.1.3 Secure Computation Frameworks

Despite being a powerful tool for privacy-preserving applications, SMC incurs large overheads and requires expert knowledge to develop efficient protocols. Therefore a variety of frameworks (the software tools implementing the underlying – or a combination of – generic SMC protocol(s)) has been developed to address these problems.

Garbled Circuit Frameworks Some frameworks are tailored for Garbled Circuits. They provide a GC-backend that allows the generator to transform a Boolean circuit into a garbled circuit that can be evaluated by the evaluator. Examples are: FastGC [105], SCAPI [75], ObliVM-GC [144], JustGarble [19], LEGO [157], WMCrypt [146].

Secret Sharing Frameworks Some frameworks are based on secret sharing. They allow sharing the parties' inputs among many computing parties using additive secret sharing. Addition of secret values can be evaluated by just adding the shares locally. Multiplication requires interactions and can be sped up using a pre-processing based on Beaver technique [16]. So-called multiplication triples are generated in an offline phase and later used in the online protocol to reduce the number of interactions. Examples of such tools are: Sharemind [29], VIFF [64], SPDZ [69], SPDZ-2 [67], TinyOT [39], MASCOT [117].

Hybrids Frameworks There are hybrid frameworks that combine garbled, additively homomorphic encryption, and secret sharing. Examples are: ABY [71], TASTY [103], Chameleon [169], SCALE–MAMBA [5].

Compilers Some frameworks have a compiler that can transform programs written in a high-level language, such as Java, C/C++, etc., into a secure representation. We have for example: FairPlay [147], FairPlayMP [22], CBMC-GC [80], ObliVM-Lang [144], SCVM [143], PCF [132], Wysteria [168], TinyGarble [174], PICCO [198].

3.2 Building Blocks Protocols

We first review the literature on OT and ORAM. Then, we review oblivious data structures and private information retrieval that are related to ORAM and OT, respectively. We conclude the section with the server model of secure computation that

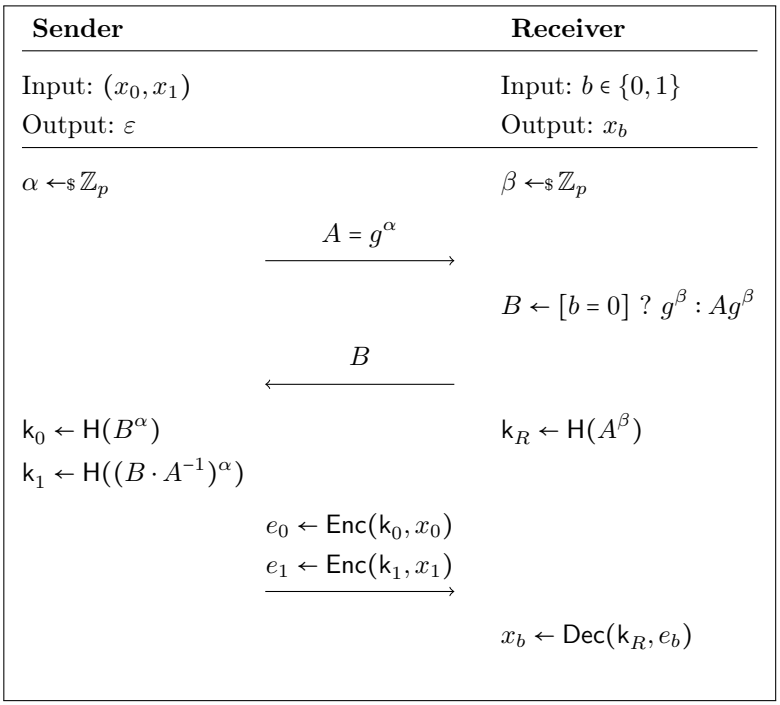


Figure 3.1: Simplest OT [54]

has an additional party to the protocol, namely the server, which however has neither input nor output.

3.2.1 Oblivious Transfer

Oblivious Transfer (OT) has been introduced by Rabin [167] using public key cryptography. There are several efficient OT protocols including Bellare and Micali [21], Naor and Pinkas [153], Chou and Orlandi [54]. The scheme of Chou and Orlandi is the simplest and most efficient protocol to date and is illustrated in Figure 3.1. Nevertheless, all these schemes require some public key operations. In fact, Impagliazzo and Rudich [106] showed that a black-box reduction from OT to a one-way function (or a one-way permutation) would imply $\mathsf{P} \neq \mathsf{NP}$. Their result strongly suggests that we cannot base OT on one-way functions. However, Beaver [15] later showed that it is possible to extend a few oblivious transfers into many. The idea is to generate few seed OTs using public key cryptography which can then be extended to any number of OTs using only symmetric key primitives. It can be seen as the OT equivalent of hybrid encryption. Unfortunately, Beaver's scheme is inefficient in practice and has later been improved in [107] which proposed OT extension protocols secure against semi-honest and malicious parties. Recently, more efficient OT extension secure against semi-honest [9] and malicious [10, 116, 160] adversaries have been proposed. There are also a number of 1-out-of-n OT (OT_n^1) protocols including [54, 154].

3.2.2 Oblivious RAM

Oblivious RAM (ORAM) was first introduced in the context of software protection against piracy [86, 91]. A trivial ORAM scheme is for the client to download the entire storage each time it wants to access one element. This results in a too large overhead and is therefore not acceptable. Efficient schemes replace each read/write

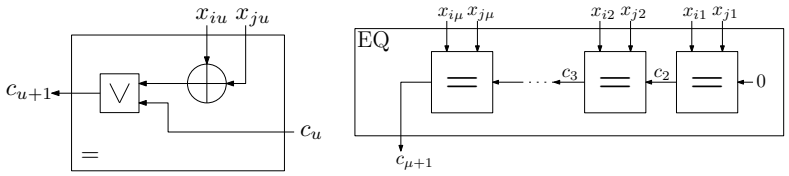
access to the original storage with a randomized series of read/write accesses to the remote storage. Many schemes have improved the performance of ORAM including tree-based ORAM schemes [171, 189] which have sublinear costs. The idea of using ORAM in secure computation was first mentioned in [94, 95]. In [190], a heuristic compact ORAM design has been proposed that is optimized for secure computation. In [72] Doerner and Shelat introduced FLORAM, an ORAM scheme based on function secret sharing, a primitive that allows sharing a function f to $p \geq 2$ parties such that on input x, each party learns a share $f_i(x)$ and $\sum f_i(x) = f(x)$. Despite its linear computation complexity, the access time is better than tree-based ORAM for trees up to 2^{30} large. The communication is $\mathcal{O}(log(n))$ for a database with n elements. Most importantly, the initialization does not require secure computation and is therefore very fast. For instance, an ORAM with 2^{20} 4-byte elements can be initialized in less than 200 milliseconds. However, the security model requires two non-colluding servers.

3.2.3 Oblivious Data Structures

Related to ORAM is the concept of Oblivious Data Structures (ODS) [191]. An ODS ensures that for any two sequences of k operations to the data structure, their resulting access patterns are indistinguishable. They apply to data structures such as binary trees or heaps with sparse access graphs. For such data structures, Wang et al. [191] introduce a pointer-based technique. Thereby, the key idea is to store each node with the indexes and labels of its child nodes. This eliminates the need to search through the position map (and hence the recursion step for tree based ORAM) for a child node label, achieving $\mathcal{O}\big(\log^2(n)\big)$ cost. Using similar techniques, Keller and Scholl [118] proposed schemes for ODS based on secret sharing. Their schemes are very efficient, translate to the multiparty setting, and provide active security as they fit naturally with the SPDZ framework [67, 69], which uses a pre-processing phase to generate secret random multiplication triples and random bits. The resulting online protocol is actively secure against a dishonest majority.

3.2.4 Private Information Retrieval

Similar to OT_n^1, Private Information Retrieval (PIR) allows a user to retrieve one out of n values from a database held by a server without revealing to the server which value was retrieved. A trivial way is to download the entire database and to dismiss the values the user is not interested in. Non-trivial PIR aims to provide the same user's privacy with efficient communication cost and was first investigated by Chor et al. [53]. They implemented PIR through replicated databases which provides information-theoretic security, if some database servers do not collude. In contrast to OT, the goal of PIR is not to provide privacy for non requested values. Nevertheless, some PIR schemes do ensure that the user can access only the requested value. Moreover, Kushilevitz and Ostrovsky [133] later discovered that replication is not needed and introduced computational PIR based on quadratic residuosity assumption. Computational PIR has been further investigated and improved by others including Lipmaa [141], which is computationally receiver-private, information-theoretically sender-private and has log-squared communication complexity. However, all computational PIR protocols have necessarily linear computation cost at the server which often makes them an unattractive choice for indexing [172].



(a) 1-bit Equality Tester (b) μ-bit Equality Tester

Figure 3.2: Equality Test Circuit

3.3 Secure Integer Comparison

In his seminal paper [195], Yao introduced the *millionaires' problem* where two millionaires are interested in knowing which of them is richer without revealing their actual wealth. The underlying functionality "greater-than" (GT) takes two integers and returns to the parties a comparison bit:

$$\mathcal{F}_{\mathsf{GT}}(x_i, x_j) \rightarrow (g_{ij}, g_{ij}),$$

where g_{ij} is a bit defined as $g_{ij} \coloneqq ([x_i > x_j]?1 : 0)$. A variant of the millionaires' problem is the *socialist millionaires' problem* [35, 110] whose underlying functionality "equality test" (EQ) takes two integers and returns to the parties an equality bit:

$$\mathcal{F}_{\mathsf{EQ}}(x_i, x_j) \rightarrow (e_{ij}, e_{ij}),$$

where e_{ij} is a bit defined as $e_{ij} \coloneqq ([x_i = x_j]?1 : 0)$. In this section, we review various protocols for computing theses functionalities. We start with comparison protocols that are closely related or have been used to implement protocols described in this thesis. Then, we briefly review other comparison protocols found in the literature.

3.3.1 Using Garbled Circuit

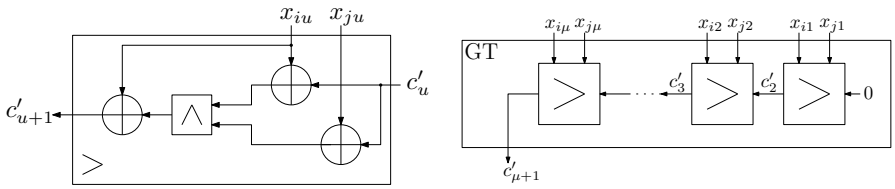
Using the garbled circuit protocol, the functionalities $\mathcal{F}_{\mathsf{GT}}$ and $\mathcal{F}_{\mathsf{EQ}}$ are represented as Boolean circuits that can be garbled by the generator and sent to the evaluator. The circuits described in this section have been introduced in [130, 131]. Let P_i, P_j be two parties having private inputs x_i, x_j, respectively and let $x_i^{\mathsf{b}} = x_{i\mu} \dots x_{i1}$, $x_j^{\mathsf{b}} = x_{j\mu} \dots x_{j1}$ be the corresponding bit representations.

Equality Test. The equality test is described by Equation 3.1. It consists of μ 1-bit equality testers for each bit position u from 1 to μ. The 1-bit equality tester at position u tests if the bits x_{iu} and x_{ju} are equal (i.e., their exclusive-or is 1). Then it combines the exclusive-or result in an OR-gate with the result of the 1-bit equality tester at position $u-1$, which is initially set to the bit 0 [131]:

$$\begin{cases} c_1 & = 0, \\ c_{u+1} & = (x_{iu} \oplus x_{ju}) \vee c_u, \ u = 1, \dots, \mu. \end{cases} \tag{3.1}$$

The output $c_{\mu+1}$ of the circuit is 1 if x_i and x_j are different and 0 otherwise (i.e. $e_{ij} = c_{\mu+1} = ([x_i \neq x_j]?1 : 0)$). The equality tester is illustrated in Figure 3.2.

Greater Than Comparison. The greater-than comparison is based on the fact that $[x_i > x_j] \Leftrightarrow [x_i - x_j - 1 \geq 0]$ and is described in Equation 3.2 [130]. Similarly to the equality tester, the circuit for the greater-than comparison (comparator) consists



(a) 1-bit Comparator (b) μ-bit Comparator

Figure 3.3: Greater-Than Comparison Circuit [130]

of μ 1-bit comparators for each bit position u from 1 to μ. The 1-bit comparator at position u depends on the result of the previous 1-bit comparator at position $u-1$, which is initially set to the bit 0.

$$\begin{cases} c'_1 & = 0, \\ c'_{u+1} & = (x_{iu} \oplus c'_u) \wedge (x_{ju} \oplus c'_u) \oplus x_{iu}, \ u = 1, \ldots, \mu. \end{cases} \tag{3.2}$$

The output $c'_{\mu+1}$ is 1 if $x_i > x_j$ and 0 otherwise (i.e., $g_{ij} = c'_{\mu+1} = ([x_i > x_j]?1:0)$). The comparator is illustrated in Figure 3.3.

While Equations 3.1 and 3.2 can be evaluated using FHE, the naive evaluation is inefficient. Moreover, they cannot be evaluated using only additively HE as they require homomorphic multiplications of ciphertexts.

3.3.2 Using Additive Homomorphic Encryption

Again let P_i, P_j be two parties having private inputs x_i, x_j. Moreover, party P_i holds a pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ of public and private key and P_j knows only the public key pk_i. Using AHE, party P_i encrypts its input under the public key pk_i and party P_j homomorphically evaluates a circuit on the resulting ciphertexts and sends back the result for decryption. It is well known that equality test can be implemented with zero testing [142]. Let $[\![x_i]\!]_i$ be a AHE ciphertext of x_i under pk_i. Party P_j receives $[\![x_i]\!]_i$ and sends back $\mathsf{c} = [\![r \cdot (x_i - x_j)]\!]_i$, where r is random, to party P_i. It easily follows that $x_i = x_j$ if and only if c decrypts to zero [138, 140]. We now describe two protocols for the GT comparison using AHE.

The Lin-Tzeng protocol. The main idea of the construction of Lin-Tzeng is to reduce the greater-than comparison to the set intersection problem [135].

Let x_i be an integer with binary representation $x_i^{\mathsf{b}} = x_{i\mu} \ldots x_{i1}$. We define the 0-encoding $S^0_{x_i}$ and 1-encoding $S^1_{x_i}$ of x_i to be the following sets:

$$S^0_{x_i} = \{x_{i\mu} x_{i(\mu-1)} \cdots x_{i(u+1)} 1 \mid x_{iu} = 0, 1 \le u \le \mu\}, \tag{3.3}$$

$$S^1_{x_i} = \{x_{i\mu} x_{i(\mu-1)} \cdots x_{iu} \mid x_{iu} = 1, 1 \le u \le \mu\}. \tag{3.4}$$

Basically, $S^0_{x_i}$ contains all prefixes of x_i that end with 0, where the last 0 is replaced by a 1, and $S^1_{x_i}$ contains all prefixes of x_i that end with a 1. Let x_i and x_j be two integers, then $x_i > x_j$ iff $|S^1_{x_i} \cap S^0_{x_j}| = 1$.

For example, if $\mu = 3$, $x_i = 6 = 110_2$ and $x_j = 2 = 010_2$ then $S^1_{x_i} = \{1, 11\}$, $S^0_{x_j} = \{1, 011\}$ and $S^1_{x_i} \cap S^0_{x_j} = \{1\}$. However, if $x_i = 2 = 010_2$ and $x_j = 6 = 110_2$ then $S^1_{x_i} = \{01\}$, $S^0_{x_j} = \{111\}$ and $S^1_{x_i} \cap S^0_{x_j} = \varnothing$.

Let $b = 0$. To homomorphically compare x_i and x_j, the protocol works as follows:

- P_i computes a $2 \times \mu$-table $T[i,j], i \in \{0,1\}, 1 \le j \le \mu$ defined as:

$$T[x_j, j] = [\![b]\!]_i \text{ and } T[1 - x_j, j] = [\![r]\!]_i \text{ for some random } r$$

  and sends it to party P_j.

- After receiving the table T, P_j does the following:
  - for each $u, 1 \le u \le \mu$, choose a random r and compute

$$\mathsf{c}_u = \begin{cases} (T[x_{j\mu}, \mu] \times \cdots \times T[x_{j(u+1)}, u+1] \times T[1,u])^r & \text{if } x_{j\mu} \cdots x_{j(u+1)} 1 \in S^0_{x_j} \\ [\![r]\!]_i & \text{if } x_{j\mu} \cdots x_{j(u+1)} 1 \notin S^0_{x_j}, \end{cases}$$

  - choose a random permutation π of $\{1, \ldots, \mu\}$,
  - send $(\mathsf{c}_1, \ldots, \mathsf{c}_\mu) \leftarrow \pi(\mathsf{c}_1, \ldots, \mathsf{c}_\mu)$ to P_i.
- If one c_u decrypts to b then $x_i > x_j$ and P_i outputs 1. Otherwise, it outputs 0.

The protocol can work with a multiplicatively homomorphic encryption such as ElGamal [76] by setting $b = 1$.

The DGK protocol. To determine whether $x_i < x_j$ or $x_i > x_j$, one computes for each $1 \le u \le \mu$ the following numbers z_u:

$$z_u = s + x_{iu} - x_{ju} + 3 \sum_{v=u+1}^{\mu} (x_{iv} \oplus x_{jv}). \tag{3.5}$$

The sum of exclusive-ors $\sum_{v=u+1}^{\mu}(x_{iv} \oplus x_{jv})$ will be zero exactly when $x_{iv} = x_{jv}$ for $u < v \le \mu$. The variable s can be set to either 1 (when checking $x_i \le x_j$) or -1 (when checking $x_i > x_j$) and allows secret-sharing the result of the comparison between two parties. To check whether $x_i \le x_j$ holds the protocol works as follows:

- Party P_i sends $[\![x_i^{\mathsf{b}}]\!]_i = ([\![x_{i\mu}]\!]_i, \ldots, [\![x_{i1}]\!]_i)$ to party P_j.
- P_j computes DGKEVAL() from Algorithm 3.4:

$$(\delta_{ji}, [\![z_\mu]\!]_i, \ldots, [\![z_1]\!]_i) \leftarrow \text{DGKEVAL}([\![x_i^{\mathsf{b}}]\!]_i, x_j^{\mathsf{b}}),$$

  sends back $([\![z_\mu]\!]_i, \ldots, [\![z_1]\!]_i)$ to P_i and outputs δ_{ji}.
- P_i computes $\delta_{ij} \leftarrow \text{DGKDECRYPT}([\![z_\mu]\!]_i, \ldots, [\![z_1]\!]_i)$ as defined in Algorithm 3.4 and outputs δ_{ij}.

After the computation the parties P_i and P_j hold random bits δ_{ij} and δ_{ji} such that $\delta_{ij} \oplus \delta_{ji} = (x \le y)$. In this protocol, parties P_i and P_j perform respectively $\mathcal{O}(\mu)$ and $\mathcal{O}(6\mu)$ asymmetric operations. The DGK protocol has been improved in [65, 187, 188].

3.3.3 Using Fully Homomorphic Encryption

To check whether $x_i \ge x_j$, one evaluates the following circuit:

$$c_u = \begin{cases} (1 \oplus x_{iu}) \cdot x_{ju} & \text{if } u = 1 \\ ((1 \oplus x_{iu}) \cdot x_{ju}) \oplus ((1 \oplus x_{iu} \oplus x_{ju}) \cdot c_{u-1}) & \text{if } u > 1, \end{cases} \tag{3.6}$$

function DGKEVAL($[\![x_i^{\mathsf{b}}]\!]_i, x_j^{\mathsf{b}}$)
  parse $[\![x_i^{\mathsf{b}}]\!]_i$ **as** $[\![x_{i\mu}]\!]_i, \ldots, [\![x_{i1}]\!]_i$
  parse x_j^{b} **as** $x_{j\mu} \ldots x_{j1}$
  for $u \coloneqq 1$ **to** μ **do**
    if $x_{ju} = 0$ **then**
      $[\![x_{iu} \oplus x_{ju}]\!]_i \leftarrow [\![x_{iu}]\!]_i$
    else
      $[\![x_{iu} \oplus x_{ju}]\!]_i \leftarrow [\![1]\!]_i [\![x_{iu}]\!]_i^{-1} = [\![1 - x_{iu}]\!]_i$
  choose a random bit δ_{ji} and **set** $s = 1 - 2 \cdot \delta_{ji}$
  for $u \coloneqq 1$ **to** μ **do**
    $[\![z_u]\!]_i \leftarrow [\![s]\!]_i [\![x_{iu}]\!]_i [\![x_{ju}]\!]_i^{-1} (\prod_{v=u+1}^{\mu} [\![x_{iv} \oplus x_{jv}]\!]_i)^3$
    $[\![z_u]\!]_i \leftarrow [\![z_u]\!]_i^{r_u}$, for a random r_u
  choose a random permutation π of $\{1, \cdots, \mu\}$
  return $(\delta_{ji}, \pi([\![z_\mu]\!]_i, \ldots, [\![z_1]\!]_i))$

function DGKDECRYPT($[\![z_\mu]\!]_i, \ldots, [\![z_1]\!]_i$)
  for $u \coloneqq 1$ **to** μ **do**
    if $\mathsf{Dec}(\mathsf{sk}_i, [\![z_u]\!]_i) = 0$ **then**
      return 1
  return 0

Algorithm 3.4: Algorithms of the DGK Comparison Protocol

where $1 \le u \le \mu$. Then, the result of the comparison is the bit c_μ which is 0 if $x_i \ge x_j$ and 1 otherwise. This can be expressed recursively as follows:

$$c_\mu = (1 \oplus x_{i\mu}) \cdot x_{j\mu} \oplus \left(\bigoplus_{v=1}^{\mu-1} (1 \oplus x_{iv}) \cdot x_{jv} \cdot b_{v+1} \cdot b_{v+2} \cdot \cdots \cdot b_\mu \right), \quad \text{where } b_v = (1 \oplus x_{iv} \oplus x_{jv}). \tag{3.7}$$

The above comparison circuit has multiplicative depth $\log(\mu - 1) + 1$ and can be homomorphically evaluated on two μ-bits integers in $\mathcal{O}(\mu \log \mu)$ homomorphic computations [45, 46, 47]. Given two encrypted bit representations $[\![x_i^{\mathsf{b}}]\!], [\![x_j^{\mathsf{b}}]\!]$, we will use SHECOMPARE($[\![x_i^{\mathsf{b}}]\!], [\![x_j^{\mathsf{b}}]\!]$) to denote a function that homomorphically evaluates the comparison circuit defined in Equation 3.7.

3.3.4 Other Integer Comparison Protocols

There are several other protocols for privately comparing integers. In this section, we briefly review a few of these protocols. Most of these protocols require access to the bit representation of the integers.

Fischlin [79] uses the fact that if $x_i > x_j$ then the bit representations of both integers have a common prefix which is followed by a bit 1 for x_i and a bit 0 for x_j. This is represented by the following Boolean formula:

$$\begin{aligned} [x_i > x_j] &\iff \bigvee_{u=1}^{\mu} (x_{iu} \wedge \neg x_{ju} \wedge \bigwedge_{v=u+1}^{\mu} (x_{iu} = x_{ju})) \\ &\iff \bigvee_{u=1}^{\mu} (x_{iu} \wedge \neg x_{ju} \wedge \bigwedge_{v=u+1}^{\mu} \neg(x_{iu} \oplus x_{ju})). \end{aligned} \tag{3.8}$$

Each disjunct in Equation 3.8 can be homomorphically evaluated using Goldwasser-Micali encryption [92] which is XOR-homomorphic (i.e., for two bits b and b' we have $\mathsf{Dec}(\mathsf{sk}, [\![b]\!] \cdot [\![b']\!]) = b \oplus b'$) and can be extended to an AND-homomorphism [79]. Hence, each term $x_{iu} \oplus x_{ju}$ is computed using the XOR-homomorphic property. Given the encryption $[\![b]\!]$ of a bit b, $[\![\neg b]\!]$ is computed as $[\![1]\!] \cdot [\![b]\!] = [\![1 \oplus b]\!]$. Finally, the conjunctions are computed using the AND-homomorphism.

In Blake and Kolesnikov's scheme [25], the input bits of party P_i are sent encrypted under its public key pk_i using Paillier encryption [159]. Party P_j homomorphically computes the following values in Equation 3.9 for u from μ down to 1:

$$\begin{cases} d_u &= x_{iu} - x_{ju} \\ f_u &= x_{iu} - 2x_{iu}x_{ju} + x_{ju} \\ y_u &= 2y_{u+1} + f_u, \text{ where } y_{\mu+1} = 0 \\ z_u &= d_u + r_u(y_u - 1), \text{ where } r_u \text{ is random.} \end{cases} \tag{3.9}$$

Party P_j then chooses a permutation $\pi \leftarrow_{\$} \mathfrak{S}_\mu$ and sends back $\pi([\![z_\mu]\!]_i, \ldots, [\![z_1]\!]_i)$ to party P_i. If exactly one ciphertext $[\![z_u]\!]_i$ decrypts to 1 (resp. -1) then we have $x_i > x_j$ (resp. $x_i < x_j$).

Garay et al. [83] use a circuit similar to Equation 3.6, however, representing it as an arithmetic circuit. The terms $(1 \oplus x_{iu}) \cdot x_{ju}$ and $(1 \oplus x_{iu} \oplus x_{ju}) \cdot c_{u-1}$ are replaced by $(1 - x_{iu}) \cdot x_{ju}$ and $(1 - (x_{iu} - x_{ju})^2) \cdot c_{u-1}$, respectively, resulting in the following arithmetic circuit:

$$c_u = \begin{cases} (1 - x_{iu}) \cdot x_{ju} & \text{if } u = 1 \\ ((1 - x_{iu}) \cdot x_{ju}) + ((1 - (x_{iu} - x_{ju})^2) \cdot c_{u-1}) & \text{if } 1 < u \leq \mu. \end{cases} \tag{3.10}$$

Using the circuit in Equation 3.10, they propose a protocol that runs in $\mathcal{O}(\log \mu)$ rounds while achieving simultaneously very low communication and computational complexities. Their schemes rely on the so-called *arithmetic black-box (ABB)* [57, 68]. The arithmetic black-box is an ideal functionality that performs basic arithmetic operations (i.e., addition and multiplication). While any party can in private specify input to the ABB, only a majority of parties can ask to perform any feasible computation and made (only) the result public. The ABB itself can be implemented using additive secret sharing or additively homomorphic encryption. Hence, it can perform the addition of two variables and the multiplication with a constant by itself, while the multiplication of variables requires interaction with the parties. Other protocols for computing the equality test and GT comparison using the ABB include [41, 62, 142, 158, 178].

Gentry et al. [85] reduce the GT comparison to the zero testing. Using AHE, party P_j gets the encrypted input bits $[\![x_i^{\mathsf{b}}]\!]_i = ([\![x_{i\mu}]\!]_i, \ldots, [\![x_{i1}]\!]_i)$ and first computes $[\![x_{i\mu} \oplus x_{j\mu}]\!]_i, \ldots, [\![x_{i1} \oplus x_{j1}]\!]_i$. Then it computes $[\![z_u]\!]_i \leftarrow [\![\sum_{v=u}^{\mu} x_{iv} \oplus x_{jv}]\!]_i$ for all $u = 1, \ldots, \mu$. Both parties run a zero testing for each $[\![z_u]\!]_i$ resulting in ciphertexts $[\![y_u]\!]_i$ encrypting a bit 0 if $[\![z_u]\!]_i = [\![0]\!]_i$ or a bit 1 otherwise. Let $[\![y_{\mu+1}]\!]_i = [\![0]\!]_i$, party P_j computes $[\![y'_u]\!]_i \leftarrow [\![(y_u - y_{u+1}) \cdot x_{ju}]\!]_i$ for all $u = 1, \ldots, \mu$. Finally, party P_j computes the ciphertext $[\![\sum_{u=1}^{\mu} y'_u]\!]_i$ which encrypts a bit 0 if $x_i \geq x_j$ and a bit 1 otherwise. The zero testing reduces to itself reducing logarithmically the bitlength of the input string. To check whether $[\![z_u]\!]_i$ encrypts 0, party P_j chooses a random value a_j with bit representation $a_{j\mu} \ldots a_{j1}$. Then P_j encrypts the bit representation of a_j under its own public key pk_j and sends $[\![z_u + a_j]\!]_i$ and $([\![a_{j\mu}]\!]_j, \ldots, [\![a_{j1}]\!]_j)$ to party P_i. Let $a_i = z_u + a_j$ with bit representation $a_{i\mu} \ldots a_{i1}$, party P_i computes $[\![a_{i\mu} \oplus a_{j\mu}]\!]_j, \ldots, [\![a_{i1} \oplus a_{j1}]\!]_j$ and $[\![z'_u]\!]_j \leftarrow [\![\sum_{v=1}^{\mu} a_{iv} \oplus a_{jv}]\!]_j$. Now the value z'_u is of bitlength $\log z_u$ and the protocol can

be applied recursively on $[\![z'_u]\!]_j$ with the parties switching their role in each recursive iteration.

Based on the observation of [142, 178] that the comparison of two strings can be reduced to comparing the first equal length sub-strings on which they differ, Couteau [56] recently propose new protocols for equality test and GT comparison. A protocol for obliviously finding this sub-string can, therefore, be used to reduce the comparison of larger strings to the comparison on smaller strings. The reduction can be applied recursively until the strings are small enough. However, while Toft and Lipmaa [142, 178] based their constructions on public-key primitive, Couteau [56] shows how to implement the reduction using exclusively oblivious transfer on short strings [129].

In summary, the schemes described above require access to the bit representation of the integers and perform operations on private data using additively homomorphic encryption or additive secret sharing. The scheme of Fischlin [79] and the scheme of Blake and Kolesnikov [25] have a constant number of rounds but require a complexity that is linear in the bitlength of the integers. The scheme of Garay et al. [83] and the scheme of Gentry et al. [85] have a logarithmic (in the input bitlength) number of rounds and a linear complexity, but in contrast to [25, 79], they output an encrypted comparison bit. The scheme of Couteau [56] has a log-logarithmic (in the input bitlength) number of rounds and a linear complexity, but outputs secret shares of the comparison bit.

3.4 Search over encrypted data

Cloud computing allows data owners to outsource their data to a database hosted by third party cloud server. To allow the data owners to maintain control over their outsourced data, one can use encryption to protect it against attackers inside the cloud. On the one hand, encryption makes the data unintelligible, such that executing complex search queries on it is no longer possible. On the other hand, downloading the complete data every time and search on it locally results in very high overhead. Therefore, specific cryptographic primitives have been designed to search on data in the encrypted form such that the data owner can delegate the search to a cloud server while maintaining control over the data. We review some of these primitives in this section.

3.4.1 Order-preserving Encryption

Order-preserving encryption (OPE) can be classified into stateless schemes and stateful schemes. The protocol in Chapter 4 is concerned with stateful schemes and hence we introduce some of their algorithms in this section. However, we review stateless schemes and their security definitions first in order to distinguish them from stateful schemes.

Stateless Order-preserving Encryption

Order-preserving encryption ensures that the order relation of the ciphertexts is the same as the order of the corresponding plaintexts. This allows to efficiently search on the ciphertexts using binary search or perform range queries without decrypting the ciphertexts. The concept of order-preserving encryption was introduced in the database community by Agrawal et al. [3]. The cryptographic study of Agrawal et al.'s scheme was first initiated by [30], which proposed an ideal security definition

IND-OCPA[1] for OPE. The authors proved that under certain implicit assumptions IND-OCPA is infeasible to achieve. Their proposed scheme was first implemented in the CryptDB tool of Popa et al. [164] and attacked by Naveed et al. [156]. In [31], Boldyreva et al. further improved the security and introduced modular order-preserving encryption (MOPE). MOPE adds a secret modular offset to each plaintext value before it is encrypted. It improves the security of OPE, as it does not leak any information about plaintext location, but still does not provide ideal IND-OCPA security. Moreover, Mavroforakis et al. [148] showed that executing range queries via MOPE in a naive way allows the adversary to learn the secret offset and so negating any potential security gains. They address this vulnerability by introducing query execution algorithms for MOPE. However, this algorithm assumes a uniform distribution of data and has already been attacked in [74]. In a different strand of work Teranishi et al. [177] improve the security of stateless order-preserving encryption by randomly introducing larger gaps in the ciphertexts. However, they also fail at providing ideal security.

Stateful Order-preserving Encryption

Popa et al. [163] were the first to observe that one can avoid the impossibility result of [30] by giving up certain restrictions of OPE. As a result of their observations, they introduced mutable OPE . Their first observation was that most OPE applications only require a less restrictive interface than that of encryption schemes. Their encryption scheme is, therefore, implemented as an interactive protocol running between a client that also owns the data to be encrypted and an honest-but-curious server that stores the data. Moreover, it is acceptable that a small number of ciphertexts of already-encrypted values change over time as new plaintexts are encrypted. With this relaxed definition, their scheme was the first OPE scheme to achieve ideal security.

Popa et al.'s scheme (mOPE_1) [163]. The basic idea of Popa et al.'s scheme is to have the encoded values organized at the server in a binary search tree called *OPE-tree*. Specifically, the server stores the state of the encryption scheme in a table (*OPE-table*). The state contains ciphertexts consisting of a deterministic AES ciphertext and the order (*OPE Encoding*) of the corresponding plaintext. To encrypt a new value x the server reconstructs the OPE-tree from the OPE-table and traverses it. In each step of the traversal, the client receives the current node v of the search tree, decrypts and compares it with x. If x is smaller (resp. larger), then the client recursively proceeds with the left (resp. right) child node of v. An edge to the left (resp. to the right) is encoded as 0 (resp. 1). The OPE encoding of x is then the path from the root of the tree to x padded with $10\cdots0$ to the same length l. To ensure that the length of OPE encodings does not exceed the defined length l, the server must occasionally perform balancing operations. This updates some order in the OPE table (i.e., the OPE encodings of some already encrypted values mutate to another encoding).

Kerschbaum and Schröpfer's scheme (mOPE_2) [124]. The insertion cost of Popa et al's scheme is high, because the tree traversal must be interactive between the client and the server. To tackle this problem Kerschbaum and Schröpfer [124] proposed another ideal secure, but significantly more efficient, OPE scheme. Both schemes use binary search and are mutable, but the main difference is that in the scheme of [124] the state is not stored on the server but on the client. Moreover,

[1]IND-OCPA means indistinguishability under ordered chosen plaintext attacks and requires that OPE schemes must reveal no additional information about the plaintext values besides their order.

the client chooses a range $\{0, \ldots, M\}$ for the order. For each plaintext x and the corresponding OPE encoding $y \in \{0, \ldots, M\}$ the client maintains a pair $\langle x, y \rangle$ in the state. To insert a new plaintext the client finds two pairs $\langle x_i, y_i \rangle$, $\langle x_{i+1}, y_{i+1} \rangle$ in the state such that $x_i \leq x < x_{i+1}$ and computes the OPE encoding as follows:

- if $x_i = x$ then the OPE encoding of x is $y = y_i$,
- else
  - if $y_{i+1} - y_i = 1$ then
    - update the state (Algorithm 2 in [124])[2].
  - the OPE encoding of x is $y = y_i + \lceil \frac{y_{i+1} - y_i}{2} \rceil$.

The encryption algorithm is keyless and the only secret information is the state which grows with the number of encryptions of distinct plaintexts. The client uses a dictionary to keep the state small and hence does not need to store a copy of the data.

Poddar et al.'s scheme (Arx) [162] Poddar et al. avoid the interactive tree traversal by replacing each node v of the OPE-tree with a prepared garbled circuit (GC). Let $\mathsf{AES.ENC}(k, \cdot)$, $\mathsf{AES.DEC}(k, \cdot)$ be encryption and decryption functions of AES with corresponding circuits denoted by $\mathsf{AES.ENC}_c(k, \cdot)$, $\mathsf{AES.DEC}_c(k, \cdot)$. Let $\mathsf{CMP}(\cdot, v)$ be a comparison circuit for computing $[x \leq v]$, for some input x. Let N be a tree node that encodes a plaintext v. The GC for N encodes $\mathsf{AES.DEC}_c(k, \cdot)$, $\mathsf{CMP}(\cdot, v)$, $\mathsf{AES.ENC}_c(k_l, \cdot)$ and $\mathsf{AES.ENC}_c(k_r, \cdot)$, where k, k_l, k_r are respectively AES keys for node N and its left and right child nodes. Our GC encodes only two comparison circuits and is relatively small. To traverse node N on input x, the GC takes the garbled input of $\mathsf{AES.ENC}(k, x)$, decrypts, compares, and returns a garbled input for either $\mathsf{AES.ENC}(k_l, x)$ or $\mathsf{AES.DEC}(k_r, x)$, and a bit left or right. To encode a new plaintext x the client sends $\mathsf{AES.ENC}(k_{root}, x)$ and must replace all GCs on the traversed path after the encryption.

Kerschbaum's scheme ($\mathsf{mOPE_3}$) [123]. Deterministic OPE schemes [3, 30, 124, 163, 177] are vulnerable to many attacks like frequency analysis, sorting attacks or cumulative attacks [98, 156]. To increase the security of OPE Kerschbaum first introduced in [123] a new security definition called *indistinguishability under frequency-analyzing ordered chosen plaintext attack* (IND-FAOCPA) that is strictly stronger than IND-OCPA. Second, he proposed a novel OPE scheme $\mathsf{mOPE_3}$ that is secure under this new security definition. The basic idea of this scheme is to randomize ciphertexts such that no frequency information from repeated ciphertexts leaks. It borrows the ideas of [124] with a modification that re-encrypts the same plaintext with a different ciphertext. First, the client's and server's states are as in $\mathsf{mOPE_2}$. The order ranges from 0 to M as in $\mathsf{mOPE_2}$ as well. The algorithm traverses the OPE-tree by going to the left or to the right depending on the comparison between the new plaintext and nodes of the tree. However, if the value being encrypted is equal to some value in the tree then the algorithm traverses the tree depending on the outcome of a random coin. Finally, if there is no more node to traverse the algorithm rebalances the tree if necessary and then computes the ciphertext similarly to $y = y_i + \lceil \frac{y_{i+1} - y_i}{2} \rceil$.

In subsequent independent analysis [98], $\mathsf{mOPE_3}$ [123] has been shown to be significantly more secure to the attacks against order-preserving encryption (albeit not perfectly secure).

[2] This potentially updates all OPE encoding y produced so far [124].

Kerschbaum and Tueno's scheme (ESEDS-OPE) [125]. This work first introduces a new security definition for encrypted data structure – called IND-CPA-DS – that is stronger than IND-CPA on each record (and hence stronger than IND-FAOCPA). IND-CPA-DS is a provable security model that encompasses plaintext guessing attacks on OPE where the attacker is able to break into a cloud system and steal the stored data. IND-CPA-DS ensures that such an attack and even frequency analysis, sorting and cumulative attacks by Naveed et al. [156] and Grubbs et al. [98] will reveal no additional information to the adversary. In [8], Amjad et al. address a similar security model for searchable encryption. Kerschbaum and Tueno [125] then propose an OPE scheme based on efficiently searchable encrypted data structure (ESEDS-OPE) satisfying IND-CPA-DS-security. The construction of ESEDS-OPE combines the benefits of three previous order-preserving encryption schemes: mOPE_1 [163], mOPE_3 [123] and MOPE [31]. First, ESEDS-OPE reuses the idea of managing the order of ciphertexts in a stateful and interactive manner as in mOPE_1 [163]. Second, it assigns a distinct ciphertext for each – even repeated – plaintext as mOPE_3 [123] does. But since mOPE_3 partially leaks the insertion order, ESEDS-OPE encrypts each plaintext – even repeated ones – using a probabilistic encryption algorithm before inserting the resulting ciphertext using Kerschbaum's random tree traversal. Third, ESEDS-OPE applies the idea of rotating around a modulus, however not on the plaintexts as in MOPE [31], but on the ciphertexts. This is done by updating the modulus after each encryption. Concretely, ESEDS-OPE maintains a list of ciphertexts for each plaintext on the server. This list is sorted by the plaintexts and rotated (on the ciphertexts) around a random modulus. For encryption and search, the client (which owns the decryption key) and the server run an interactive protocol to perform a binary search on the encrypted data structure.

3.4.2 Other Searchable Primitives

There exist other primitives such as searchable encryption [59, 99] and order-revealing encryption [44, 134] that allow searching on encrypted data. They have higher security than OPE, but they are less efficient and require a change to existing applications. We briefly review the two primitives below and refer to the respective literature for more details.

A searchable encryption basically consists of four algorithms. The probabilistic *key generation algorithm* takes a security parameter and outputs a master secret key. The probabilistic *encryption algorithm* takes the master secret key and a plaintext and outputs a ciphertext. The deterministic *token generation algorithm* takes the master secret key and a keyword and outputs a search token. The deterministic *match algorithm* takes a ciphertext and a search token and outputs 1 if the associated keyword to the search token matches the associated plaintext to the given ciphertext. A searchable encryption can be used for example by a data owner to outsource encrypted documents to the cloud. Later, the data owner might want to search for documents containing a specific keyword. Based on the keyword, it generates and sends a search token to the cloud server which can use the match algorithm to return all encrypted documents matching the keyword [59].

As OPE, order-revealing encryption (ORE) allows to efficiently perform range queries, sorting and filtering on encrypted data. An order-revealing encryption basically consists of three algorithms. The probabilistic *key generation algorithm* takes a security parameter and outputs a secret key. The probabilistic *encryption algorithm*

takes the secret key and a plaintext and outputs a ciphertext. The deterministic *compare algorithm* takes two ciphertexts and outputs a bit whose value reveals the order between the associated plaintexts [44, 134].

Similar to our work in Chapter 4, Ishai et al. [108] address a 3-party functionality consisting of a Data Owner (DO) that outsources encrypted data to the Cloud Service Provider (CSP), and a Data Analyst (DA) that wants to execute private range queries on this data. Ishai et al. solve this 3-party functionality by proposing an SSE-like scheme. Like us, they split the solution in two phases. In the first phase, the three parties jointly and privately traverse a search tree to reveal to the DA pointers to the data that match the query. In the second phase, the DA interacts with the CSP to get the matching data. While traversing the search tree, the parties combine SMC with 2-server private information retrieval (PIR) to compute and select the next node in the tree, hiding DA's access pattern to both DO and CSP. The SMC is implemented with a small garbled circuit whose size is similar to ours. Recall that in a 2-server PIR, both servers (CSP and DO) have shares of the complete database. In the query phase, the DA selects each matching data individually using PIR. This, however, hides the access pattern up to δ queries, where δ is a parameter defined by the DO and is used to add dummy entries to the database such that the CSP cannot decide if the DA is accessing a real data. After δ queries, the scheme must either reinitialize the PIR state (which is linear in the size of the database) or continue with access pattern leakage. The authors recommend the later option because keeping hiding the access is very expensive. Inserting new data requires a re-initialization of the PIR state as well.

3.5 Private Decision Tree Evaluation

Privacy-preserving machine learning takes advantage of SMC techniques to build classifiers on private databases [104, 136, 137, 138] or to classify private data with private models [14, 34, 38, 55, 96, 104, 151, 193]. Since our work described in Chapters 5 and 6 falls under the second category, we concentrate in this section only on privacy-preserving classifiers, particularly decision trees. In private decision tree evaluation, the server holds a private decision tree model and the client wants to classify its private attribute vector using the server's private model.

3.5.1 Using Program Transformation

Brikell et al. [38] combine homomorphic encryption (HE) and GCs in a novel way. In an initial phase, the server non-interactively transforms the plaintext decision tree in a secure program, by permuting the tree and replacing each decision node by a small GC implementing integer comparison, and each leaf node by an encryption of the corresponding classification label. The GC at a decision node will allow the client in the evaluation phase to learn the decryption key of one child node according to the result of the comparison. In the second phase, the parties execute an oblivious attribute selection protocol, where the client uses a homomorphic scheme to encrypt each element of the attribute vector under his public key. The server receives the encrypted vector, permutes it, homomorphically blinds each element, and sends it back to the client. The client decrypts the vector and the two parties execute oblivious transfers that allow the client to learn the garbling keys encoding its input. In the last phase, the client receives the secure program and evaluates it.

Although the evaluation time of Brikell et al.'s scheme is sublinear in the tree size, the secure program itself and hence the communication cost is linear and, therefore,

not efficient for large trees. Barni et al. [14] improve the previous scheme by not including the leaf node in the transformed secure program, thereby reducing the costs by a constant factor, however maintaining linear communication cost. Although more efficient, it is still suitable only for small trees.

3.5.2 Using Homomorphic Encryption

Using homomorphic encryption (HE) Bost et al. [34] propose a privacy-preserving protocol for different classifiers including decision trees. They represent the decision tree as a polynomial whose output is the result of the classification. The constant values of the polynomial are the classification labels and the variables represent the results of the Boolean conditions at the decision nodes. Then, the parties privately compute the inputs to this polynomial by comparing each threshold of the tree with the corresponding element of the attribute vector. Finally, the server privately evaluates the polynomial and returns the result to the client. The privacy of the tree is guaranteed by the fact that the server evaluates the polynomial non-interactively. The evaluation is done homomorphically on inputs encrypted under the client's public key. The number of invocations of the comparison protocol and the size of the polynomial are linear in the size of the tree. Moreover, the evaluation requires FHE or at least SHE.

Wu et al. [193] improve the protocol of [34] by using different techniques so that the protocol requires only AHE. Using the protocol from [63], they also perform as many comparisons as there are decision nodes. The server receives each comparison bit, encrypted under the client's public key pk, and uses them to evaluate the decision tree. The evaluation returns the index of the corresponding classification label to the client. Finally, the parties execute an Oblivious Transfer to allow the client to learn the classification label. Their protocol is more efficient than [34], because it relies on AHE, implemented using a variant of ElGamal based on elliptic curve cryptography.

Tai et al. [175] follow the same idea as in [193] by using the comparison protocol of [63] and AHE. Then, they mark the left and right edge of each node with the cost b and $1-b$ respectively, where b is the result of the comparison at that node. Finally, they sum for each path of the tree the cost along it. The label of the path whose costs sum to zero, is the classification label.

Joye et al. [112] follow as well the same idea as in [193] by using the comparison protocol of [63], by running the protocol encrypted under the client's public key using AHE, and by hiding the tree's structure using at each level of the tree a random permutation that is secret-shared between the parties. Their protocol works only for complete trees as in [193] and, therefore, requires adding dummy nodes if the tree is not complete. The tree is encrypted under the public key of the client. Let d be the depth of the tree, then the protocol runs in $\mathcal{O}(d)$ rounds performing one DGK comparison per level. For the comparison at level l the server's input is a random value r_l and the client's value is selected via OT on the encrypted nodes at level l. In the OT, the client uses its share of the random permutation at level l to compute an index j to the OT and get $[\![x_j - y_j + r_l]\!]$, where x_j and y_j are respectively the attribute and threshold values associated to the selected node. Finally, at level d the parties run again an OT on the classification labels where the client uses again its share of the permutation at that level to compute an index to OT.

Kiss et al. [126] propose a modular design consisting of the sub-functionalities: private selection of attribute values, private comparison, and private evaluation of the path corresponding to the given attribute vector. They then analyze the state-of-the-art of sub-protocols that make use of homomorphic encryption or garbled circuits

to privately compute the sub-functionalities. Finally, they explore the tradeoffs and performance of all possible combinations of the identified sub-protocols to privately evaluate decision trees.

3.5.3 Using Secret Sharing

De Cock et al. [55] follow the same blueprint as the two previous protocols by first comparing each threshold of the tree with the corresponding attribute. In contrast to all other protocols (ours included), which are secure in the computational setting, they operate in the information theoretic model using secret sharing based SMC and utilize commodity-based cryptography [16] to reduce the number of interactions. This results in a protocol that performs very well for small trees. However, their protocol is less efficient for large trees, since it is also linear in the size of the tree.

3.6 Computation of the k^{th}-Ranked Element

The secure computation of the k^{th}-ranked element can be addressed using generic secure multiparty computation [23, 87, 195] which guarantees that no protocol participant will learn more than what it can infer from its input and the output of the protocol, i.e., the other parties' inputs remain confidential. The first specialized protocol for computing the k^{th}-ranked element was introduced by Aggarwal et al. [1]. They presented protocols for the two-party and multiparty cases in the semi-honest and malicious model. This section briefly surveys these protocols.

3.6.1 Two-Party Case

Let there be two parties P_1 and P_2 with private sets S_1 and S_2 of integers. Without loss of generality, let assume that $k \leq |S_1| = |S_2| = 2^j = n$ and that the elements in $S_1 \cup S_2$ are all pairwise distinct. The two-party protocol of Aggarwal et al. [1] computes the k^{th}-ranked element by performing a binary search in $S_1 \cup S_2$ and works as follows:

1. Both parties sort their sets.
2. For $i = j - 1$ to $i = 0$
   (a) Each party computes the $(2^i)^{\text{th}}$-ranked element of its set denoted by m_1 and m_2.
   (b) Both parties engage in a generic secure computation which outputs 0 if $m_1 \geq m_2$, and 1 if $m_1 < m_2$.
   (c) If $m_1 < m_2$, then P_1 removes all elements ranked 2^i or less from S_1, while P_2 removes all elements ranked greater than 2^i from S_2.
   (d) If $m_1 \geq m_2$, then P_1 removes all elements ranked greater than 2^i from S_1, while P_2 removes all elements ranked 2^i or less from S_2.
3. The parties output the result of a generic secure computation of the minimum value of their respective elements.

Larger values of k (i.e., $k > |S_1|$ or $k > |S_2|$) and cardinality of the sets being non-power of 2 can be dealt with by padding the sets with $+\infty$ and $-\infty$. Duplicates can be handled by adding $\lceil \log n \rceil + 1$ bits to every input element, in the least significant positions. For every element in S_1, let these bits be a bit 0 followed by the rank of the element. Apply the same procedure to the element in S_2 using the bit 1 instead

of the bit 0. The protocol as described above is secure in the semi-honest and can be made secure in the malicious model. If μ is the bit length of each input element, then the scheme requires $\mathcal{O}(\log k)$ rounds and has a communication cost of $\mathcal{O}(\mu \cdot \log k)$.

3.6.2 Multiparty Case

The multi-party protocol of Aggarwal et al. [1] computes the k^{th}-ranked element by binary search in the domain of the input values. They assume that the elements in the datasets are not necessarily distinct integers of a fixed length. Let $[\alpha, \beta]$ be the publicly known range of the input values, $M = \beta - \alpha + 1$, N be the number of elements in all the datasets and n be the number of parties. Each party initializes two variables $a := \alpha$ and $b := \beta$, then the scheme runs in a series of $\mathcal{O}(\log M)$ rounds in which it suggests a value for the k^{th}-ranked element, performs a generic secure computation for computing the sum of all elements smaller (resp. greater) than the suggested k^{th}-ranked element, and updates the guess. Initially, the value $m = \left\lceil \frac{a+b}{2} \right\rceil$ is suggested as the k^{th}-ranked element. Each party P_i lets l_i be the number of its values strictly smaller than m and g_i be the number of its values strictly greater than m. Then the parties engage in a generic secure computation to compute the following:

- if $\sum l_i \leq k - 1 \wedge \sum g_i \leq N - k$ then return m and stop,
- if $\sum l_i \geq k$ then set $b = m - 1$,
- if $\sum g_i \geq N - k + 1$ then set $a = m + 1$.

The scheme as described above is semi-honest secure and can be made malicious secure by adding instructions that force the parties to behave semi-honestly. Each round requires a generic multiparty computation for computing two summations using a circuit of size $\mathcal{O}(n \log M)$ each and two comparisons using a circuit of size $\mathcal{O}(\log M)$ each.

3.6.3 Protocols Using Blockchain

Blass and Kerschbaum propose two protocols STRAIN [27] and SCIB [28] for computing the k^{th}-ranked element using Blockchain. While STRAIN relies on Fischlin's comparison protocol [79], SCIB relies on the DGK protocol [63]. Given n parties $P_1, \ldots, P_n$ with private inputs $x_1, \ldots, x_n$, the goal is to output the index of k^{th}-ranked element (but not the element itself). Each party encrypts its input bitwise and publishes the ciphertexts on the blockchain. Then each party evaluates the comparison of its input with all other parties' input and publishes the resulting ciphertexts on the blockchain as well. While each party must evaluate $n - 1$ comparisons, they can be done in parallel. Finally, each party decrypts comparison ciphertexts encrypted under its public key, and sums up the resulting comparison bits to get its rank. The party with rank k then announces the result of the protocol. The technical difficulty relies on the fact that parties must prove correct execution of the protocol which is done using zero-knowledge proofs [90] resulting in maliciously secure protocols. While they leak the order of inputs, similar to OPE, the core techniques of determining the index of the k^{th}-ranked element are efficient and asymptotically optimal, requiring only 3 rounds.

Chapter 4

Secure Computation of OPE

Order-preserving encryption (OPE) allows efficient range queries on encrypted data. Moreover, it does not require any change to the database management system and can be retrofitted to existing applications. However, all OPE schemes are necessarily symmetric, limiting the use case to one client and one server. This is due to the fact that a public-key encryption would allow a binary search on the ciphertext. Imagine a scenario where a Data Owner (DO) outsources encrypted data to the Cloud Service Provider (CSP) and a Data Analyst (DA) wants to execute private range queries on this data. Then either the DO must reveal its encryption key, since order-preserving encryption is symmetric, or the DA must reveal the private queries. In this chapter, we overcome the limitation of private range querying on order-preserving encrypted data by allowing the equivalent of a public-key encryption. Our idea is to replace public-key encryption with a secure, interactive protocol. Non-interactive binary search on the ciphertext is no longer feasible, since every encryption requires the participation of the DO who can rate limit (i.e., control the rate of query sent by) the DA. The chapter is structured as follows. We start with a motivation in Section 4.1 and introduce correctness and security definitions in Section 4.2. We describe our deterministic and non-deterministic construction in Section 4.3 and Section 4.4, respectively. We analyze correctness and security in Section 4.5 and present a complexity analysis in Section 4.6. We discuss evaluation results in Section 4.7 before summarizing the chapter in Section 4.8.

4.1 Problem Definition

In this section, we start by describing the problem, then we present a machine learning application and a related use case found in the literature. We conclude the section with an overview of our solution.

4.1.1 Description

Our work is motivated by the following scenario. Assume a data owner (DO) encrypts its data with an OPE, stores the encrypted data in a cloud database held by a cloud service provider (CSP), and retains the encryption key. We also assume that the underlying OPE scheme is stateful, the state of the encryption is stored in the same database with the data, however in separated tables, and is only used while encrypting new data. The data itself is stored in a relational database management system, where each plaintext is replaced by its OPE ciphertext. Then range queries are run by first computing the OPE ciphertext then running an SQL-query as usual, where the plaintexts in the SQL-query are replaced by their corresponding OPE ciphertext.

Later, assume a data analyst (DA) wants to query the encrypted data. Since OPE is necessarily symmetric, only the DO can encrypt and decrypt the data stored on

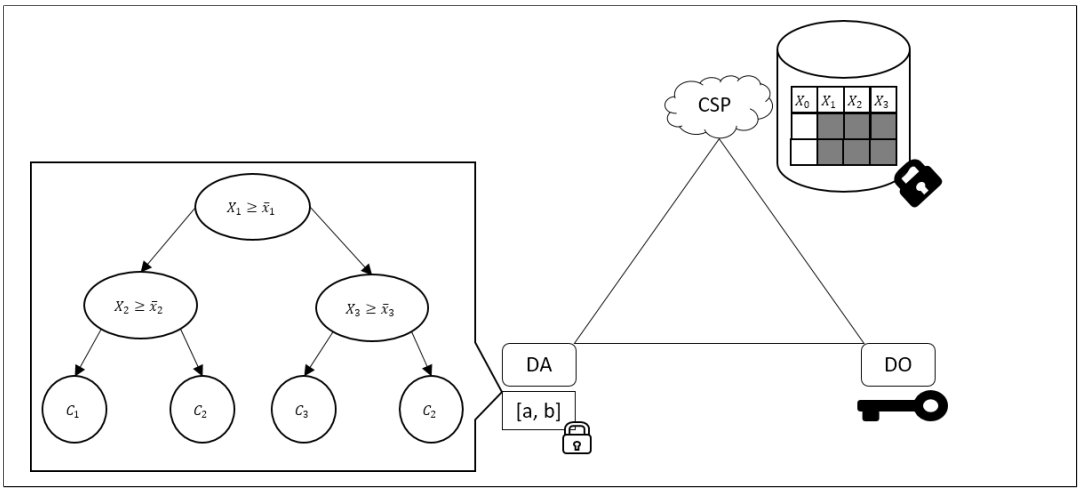


Figure 4.1: Illustration of the Problem: DO sends encrypted data to CSP and retains encryption keys. DA holds a private decision tree that can be represented as a set of range queries. DA wants to perform data analysis on DO's encrypted data without revealing any information on the queries. DO wants to maintain privacy of the data stored at CSP.

the cloud server. To query the data, the DA can send the plaintext query to the DO. However, if the query contains sensitive information which the DA wants to remain protected, then this free sharing of information is no longer possible. Our goal is to allow the DA to efficiently query the encrypted data without revealing any sensitive information on the query and without learning more than what the DO has allowed.

Since neither the DA wants to reveal his query value nor the DO his encryption state (key), this is clearly an instance of a SMC where two or more parties compute on their secret inputs without revealing anything but the result. In an ideal world the DA and DO would perform a two-party secure computation for the encryption of the query value and then the DA would send the encrypted value as part of an SQL query to the CSP. However, this two-party secure computation is necessarily linear in the encryption state (key) and hence the size of the database. The key insight of our solution is that we can construct an encryption with logarithmic complexity in the size of the database by involving the CSP in a three-party secure computation without sacrificing any security, since the CSP will learn the encrypted query value in any case. One may assume that in this construction the encryption key of the DO may be outsourced to secure hardware in the CSP simplifying the protocol to two parties, but that would prevent the DO from rate limiting the encryption and the binary search attack would be a threat again, even if the protocol were otherwise secure.

4.1.2 Application

This distinction between DO and DA occurs in many cases of collaborative data analysis, data mining, and machine learning. In such scenarios, multiple parties need to jointly conduct data analysis tasks based on their private inputs. As concrete examples from the literature consider, e.g., supply chain management, collaborative forecasting, benchmarking, criminal investigation, smart metering, etc. [12, 13, 73, 122]. Although in these scenarios plaintext information sharing would be a viable alternative, participants are reluctant to share their information with others. This reluctance is quite rational and commonly observed in practice. It is due to the fact

that the implications of the information disclosure are unknown or hard to assess. For example, sharing the information could weaken their negotiation position, impact customers' market information by revealing corporate performance and strategies or impact reputation [12, 13, 42].

As an example, we consider a private machine learning model involving comparisons. In a supply chain scenario, the DA could be a supplier (manufacturer) owning a machine learning model and wanting to optimize its manufacturing process based on data owned by its buyer (another supplier or distributor). For instance, we assume the model to be a decision tree as pictured in Figure 4.1, where the x_i are the thresholds and (X_1, X_2, X_3) is the input vector (that maps to corresponding columns in the DO's database) to be classified. In order to use the model for classification the DA transforms the decision tree into range queries, e.g., for class c_1 we have the query $(X_1 < x_1) \wedge (X_2 < x_2)$. More precisely, the DA wants to execute queries like in Equations 4.1 and 4.2, where we assume X_0 to be public and AggFunc to be an aggregate function such as Count, Sum, Average, Median etc.:

$$\text{Select AggFunc}(*) \text{ Where } X_1 < x_1 \text{ And } X_2 < x_2 \text{ OR } X_3 < x_3 \tag{4.1}$$

$$\text{Select } X_0 \text{ Where } X_1 < x_1 \text{ And } X_2 < x_2 \text{ OR } X_3 < x_3 \tag{4.2}$$

However, as the database is encrypted (i.e. columns X_1, X_2 and X_3 are OPE encrypted) the DA needs ciphertexts of the thresholds x_j.

4.1.3 Use case

In [176], Taigel et al. describe a specific use case that combines decision tree classification and OPE to enable privacy-preserving forecasting of maintenance demand based on distributed condition data. They consider the problem of a Maintenance, Repair, and Overhaul (MRO) provider from the aerospace industry that provides maintenance services to their customers' (e.g., commercial airlines or air forces) jet engines. The customers as Data Owners consider the real condition data of their airliners as very sensitive and therefore this data is stored encrypted in a cloud database using OPE. The MRO provider as Data Analyst holds a decision tree that can predict the probability of maintenance, repair, and overhaul of spare parts. However, the classification of an individual spare part is not necessary, but only aggregated numbers such as returned by Equation 4.1. The aggregated numbers then allow the MRO to compute the forecast without violating the privacy of the real condition data. The MRO would send a plaintext query as in Equation 4.1 to the customer who would compute the OPE ciphertext y_1, y_2 of x_1, x_2 and send instead the encrypted query as in Equation 4.3 to the database [176].

$$\text{Select AggF}(*) \text{ Where } X_1 < y_1 \text{ And } X_2 < y_2 \tag{4.3}$$

This, however, provides privacy only for the customer, while we want to allow privacy for both customer and MRO provider.

4.1.4 Overview of Our Construction

In theory, generic SMC allows computing any efficiently computable function. However, any generic SMC is at least linear in the input size, which in this case is the number of encrypted values in the database. The idea of our solution is to exploit the inherent leakage, i.e., implied by input and output, of stateful OPE schemes making

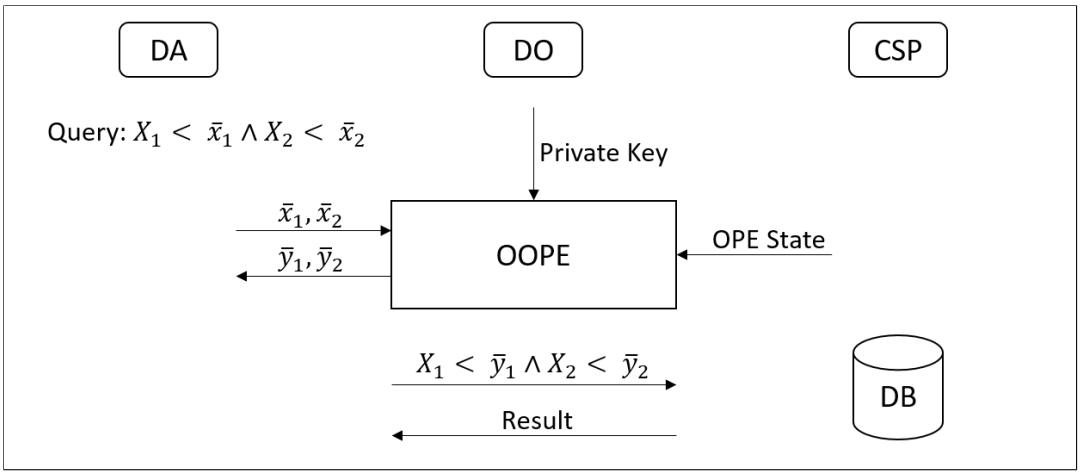


Figure 4.2: Illustration of Range Query Using OOPE

our oblivious OPE sublinear in the database size. Furthermore, we exploit the advantage of (homomorphic) encryption allowing a unique, persistent OPE state stored at the CSP while being able to generate secure inputs for the SMC protocol and the advantage of garbled circuits allowing efficient, yet provably secure comparison. Our oblivious OPE is, therefore, a mixed-technique, SMC protocol between the DO, the DA and the CSP in the semi-honest model.

In detail, our protocol proceeds as follows: The DO outsources its OPE state to the CSP. The state is an ordered list of pairs and each pair consists of a ciphertext and the order of the corresponding plaintext. However, in oblivious OPE the ciphertext is created using an additively homomorphic public-key encryption scheme instead of standard symmetric encryption. When the DA traverses the state in order to encrypt a query plaintext, the CSP creates secret shares using the homomorphic property. One secret share is sent to the DA and one to the DO. The DA and DO then engage in a secure two-party computation using Yao's Garbled Circuits in order to compare the reconstruction of the secret shares (done in the garbled circuit) to the query plaintext of the DA. The result of this comparison is again secret shared between DA and DO, i.e., neither will know whether the query plaintext is above or below the current node in the traversal. Both parties – DA and DO – send their secret shares of the comparison result to the CSP which then can determine the next node in the traversal. These steps continue until the query plaintext has been sorted into the OPE-table and the CSP has an order-preserving encoding that can be sent to the DA. A significant complication arises from this order-preserving encoding, since it must not reveal the result of the comparison protocols to the DA (although it may be correlated to the results).

We stress, that our scheme is totally transparent to the database and does not change the way SQL-queries are executed. Figure 4.2 illustrates how OOPE is used for private range query. The DA first computes an OPE encoding for each sensitive query values using OOPE which preserves DA's privacy against DO and CSP as well as DO's privacy against DA and CSP. Then the DA sends the query directly to the database hosted by the CSP. Note that the execution of the query does not depend on OOPE. The database executes the query and sends back the result to the DA.

	Number of Parties	Number of Rounds		Communication		DO Storage	DA Storage
		OPE Encoding	Query	OPE Encoding	Query		
Us	3	$\mathcal{O}(d)$	1	$\mathcal{O}(d \cdot \lvert C \rvert)$	$\mathcal{O}(\lvert R_q \rvert)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
[162]	2	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(d \cdot \lvert C \rvert + \lvert R_q \rvert)$	$\mathcal{O}(d)$	n/a
[108]	3	$\mathcal{O}(d)$	$\mathcal{O}(\lvert R_q \rvert)$	$\mathcal{O}(d \cdot \lvert C \rvert)$	$\mathcal{O}(\lvert R_q \rvert)$	$\mathcal{O}(2^d)$	$\mathcal{O}(2^d)$

Table 4.1: Comparison of Range Query Protocols: d is the depth of the search tree. $|R_q|$ is the size of the result set R_q. $|C|$ is the size of the used GC which is small for us and [108], but large for [162].

4.1.5 Comparison with Related Work

The related work to this chapter has been described in Section 3.4 and Section 3.4.2. In Table 4.1, we summarize the comparison between range query protocols, i.e., protocols that build on basic cryptographic building blocks (such as OPE, PIR, SMC) to solve privacy preserving range queries problems on encrypted outsourced data. Each protocol uses an encoding phase to compute an encoding of the endpoints of a query q and the query phase to actually run q on the database. While our query phase is actually independent of OOPE, Poddar et al. [162] replaces all GCs used during the query and Ishai et al. [108] selects each matching data individually using PIR. Finally, Poddar et al. is faster, but is 2-party and requires larger garbled circuits; Ishai et al. is more secure but requires a DO storage which depends on the database size.

In the next two sections we provide the detailed, step-by-step formalization of the construction.

4.2 Correctness and Security Definitions

In this section, we provide correctness and security definitions. Let $\mathbb{D} = \{x_1, \ldots, x_n\}$ be the finite data set of the DO, and $h = \log_2 n$. Let $[\![x_i]\!]$ denote the ciphertext of x_i under Paillier's scheme [159] with public key pk and corresponding private key sk that only the DO knows. Let $\preceq$ be the order relation on $[\![\mathbb{D}]\!] = \{[\![x_1]\!], \ldots, [\![x_n]\!]\}$ defined as: $[\![x_i]\!] \preceq [\![x_j]\!]$ if and only if $x_i \leq x_j$. The relations $\succeq, \prec, \succ$ are defined the same way with $\geq, <, >$ respectively. Let $\mathbb{P} = \{0, \ldots, 2^\mu - 1\}$ (e.g. $\mu = 32$) and $\mathbb{O} = \{0, \ldots, M\}$ (M positive integer) be plaintext and order[1] range resp., i.e.: $\mathbb{D} \subseteq \mathbb{P}$.

We begin by defining order-preserving encryption as used in this chapter.

Definition 4.2.1 (OPE)**.** *Let λ be the security parameter of the public-key scheme of Paillier. An* order-preserving encryption (OPE) *consists of the three following algorithms:*

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{OPE.KGen}(\lambda)$*: the probabilistic key generation algorithm* $\mathsf{OPE.KGen}$ *takes as input a security parameter λ and outputs a public key* pk *and a private key* sk*.*
- $S', \langle [\![x_i]\!], y_i \rangle \leftarrow \mathsf{OPE.Enc}(S, x_i, \mathsf{pk})$*: the encryption algorithm* $\mathsf{OPE.Enc}$ *takes the state S, a plaintext $x_i \in \mathbb{P}$ and the public key* pk*. It computes the ciphertext $\langle [\![x_i]\!], y_i \rangle$ and updates the state S to S', where $[\![x_i]\!] \leftarrow \mathsf{Paillier.Enc}(x_i, \mathsf{pk})$ is a Paillier ciphertext and $y_i \leftarrow \mathsf{OPE.Ord}(S, x_i)$ is the* order *of x_i with $y_i \in \mathbb{O}$, and $\mathsf{OPE.Ord}(S, x_i)$ is a function that computes the order of x_i from the state S.*

[1] We will use *order* and *OPE encoding* interchangeably.

- $x_i \leftarrow \mathsf{OPE.Dec}(\langle [\![x_i]\!], y_i \rangle, \mathsf{sk})$*: the decryption algorithm* OPE.Dec *takes a ciphertext* $\langle [\![x_i]\!], y_i \rangle$ *and the private key* sk*. Using the Paillier decryption algorithm, it outputs a plaintext* $x_i \leftarrow \mathsf{Paillier.Dec}([\![x_i]\!], \mathsf{sk})$.

The encryption scheme is correct *if:*

$$\mathsf{OPE.Dec}(\mathsf{OPE.Enc}(S, x_i, \mathsf{pk}), \mathsf{sk}) = x_i$$

for any valid state S *and plaintext* x_i*. It is* order-preserving *if the order is preserved, i.e.* $y_i < y_j \Rightarrow x_i \leq x_j$ *for any* i *and* j.

For a data set $\mathbb{D}$ the encryption scheme generates an ordered set of ciphertexts. We formalize it with the following definition.

Definition 4.2.2 (OPE-table)**.** *Let* $i_1, i_2, \ldots$ *be the ordering of the data set* $\mathbb{D}$ *(i.e.,* $x_{i_1} \leq x_{i_2} \leq \ldots$*), then the OPE scheme generates an* OPE-table *which is an ordered set* $\mathbb{T} = \langle \langle [\![x_{i_1}]\!], y_{i_1} \rangle, \langle [\![x_{i_2}]\!], y_{i_2} \rangle, \ldots$*, where* $y_{i_k} \in \mathbb{O}$ *is the order of* x_{i_k}*. For an OPE ciphertext* $\langle [\![x_i]\!], y_i \rangle \in \mathbb{T}$*, we use the function* $\mathsf{pred}(\langle [\![x_i]\!], y_i \rangle)$ *to denote its direct predecessor. Similarly, we use the function* $\mathsf{succ}(\langle [\![x_i]\!], y_i \rangle)$ *to denote its direct successor. If the predecessor or the successor of a ciphertext are not defined, then the functions return* null.

The OPE-table is sent to the server and used to generate the following search tree during the oblivious order-preserving encryption protocol.

Definition 4.2.3 (OPE-tree)**.** *An* OPE-tree *is a tree* $\mathcal{T} = (r, \mathcal{L}, \mathcal{R})$*, where* $r = [\![x_i]\!]$ *for some* x_i*,* $\mathcal{L}$ *and* $\mathcal{R}$ *are OPE-trees such that: If* r' *is a node in the left subtree* $\mathcal{L}$ *then* $r \geq r'$ *and if* r'' *is a node in the right subtree* $\mathcal{R}$*, then* $r \leq r''$*. For a node* $[\![x_i]\!]$ *of* $\mathcal{T}$ *, we use the function* $\mathsf{lnod}([\![x_i]\!])$ *to denote its left child node. Similarly, we use the function* $\mathsf{rnod}([\![x_i]\!])$ *to denote its right child node. If the left node or the right node of a node are not defined, then the functions return* null

Definition 4.2.4 (State)**.** *For a data set* $\mathbb{D}$, OPE.Enc *generates the* Data Owner state*, the set of all* $\langle x_i, y_i \rangle$ *such that* $x_i \in \mathbb{D}$ *and* y_i *is the order of* x_i*. The* server state *is the pair* $\mathbb{S} = \langle \mathcal{T}, \mathbb{T} \rangle$ *consisting of the OPE-tree* $\mathcal{T}$ *and the OPE-table* $\mathbb{T}$.

Definition 4.2.5 (Correctness)**.** *Let* $\mathbb{D}$ *be the data set and* $\mathbb{S} = \langle \mathcal{T}, \mathbb{T} \rangle$*,* $x_j \in \mathbb{P}$*,* sk *be the protocol's inputs of the CSP, DA, and DO respectively. At the end of the protocol, the Data Analyst obtains for its input* x_j *the output* y_j *such that* y_j *is the order of* x_j *in* $\mathbb{D} \cup \{x_j\}$*. The Cloud Provider obtains* $\langle [\![x_j]\!], y_j \rangle$ *that is added to the OPE-table. The Data Owner obtains nothing:*

- $\mathrm{OOPE}(\mathbb{S}, x_j, \mathsf{sk}) = (\langle [\![x_j]\!], y_j \rangle, y_j, \varnothing)$
- $\mathsf{OPE.Dec}(\langle [\![x_j]\!], y_j \rangle, \mathsf{sk}) = x_j$
- *For all* $\langle [\![x_{i_1}]\!], y_{i_1} \rangle, \langle [\![x_{i_2}]\!], y_{i_2} \rangle \in \mathbb{T} : y_{i_1} < y_j < y_{i_2} \Rightarrow [\![x_{i_1}]\!] \leq [\![x_j]\!] \leq [\![x_{i_2}]\!]$.

Remark 4.2.6. *Updating the server state, i.e., allowing the server to learn* $[\![x_j]\!]$*, is only for completeness with respect to the fact that the encryption depends on the state and the underlying OPE requires the state update after any insertion. However, as DA's data is only used in the queries and not inserted in the database, the update can be omitted without affecting the correctness of the DA's queries.*

Remark 4.2.7. *Allowing or preventing the DA to learn the order information y_j does not have any impact on the efficiency. In fact, the server can store a key-value pair, where the key is a random string and the value is y_j, and reveals only the key to the DA. The DA would still be able to submit its query to the server by replacing the sensitive plaintexts with their corresponding key.*

We refer to Chapter 2 for the definitions of *computational indistinguishability* (Definition 2.1.4) and *view* (Definition 2.1.11) of a party. We now formulate our security definition assuming a semi-honest adversary and honest majority.

Definition 4.2.8 (Semi-honest Security)**.** *Let $\mathbb{D}$ be the data set with cardinality n and the inputs and outputs be as previously defined. Then a protocol Π_{OOPE}* securely implements *the functionality OOPE in the* semi-honest model with honest majority *if the following conditions hold:*

- *there exists a probabilistic polynomial time algorithm $\mathsf{Sim}_{\mathsf{DO}}^{\mathsf{oope}}$ that simulates the DO's view $\mathsf{View}_{\mathsf{DO}}^{\Pi_{\mathsf{OOPE}}}$ of the protocol given n and the private key sk only,*
- *there exists a probabilistic polynomial time algorithm $\mathsf{Sim}_{\mathsf{DA}}^{\mathsf{oope}}$ that simulates the DA's view $\mathsf{View}_{\mathsf{DA}}^{\Pi_{\mathsf{OOPE}}}$ of the protocol given n, the input x_j and the output y_j only,*
- *there exists a probabilistic polynomial time algorithm $\mathsf{Sim}_{\mathsf{CSP}}^{\mathsf{oope}}$ that simulates the CSP's view $\mathsf{View}_{\mathsf{CSP}}^{\Pi_{\mathsf{OOPE}}}$ of the protocol given access to the server state $\mathbb{S}$ and the output $\langle [\![x_j]\!], y_j \rangle$[2] only.*

Formally:

$$\mathsf{Sim}_{\mathsf{DO}}^{\mathsf{oope}}(n, \mathsf{sk}, \varnothing) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{DO}}^{\Pi_{\mathsf{OOPE}}}(\mathbb{S}, x_j, \mathsf{sk}), \tag{4.4}$$

$$\mathsf{Sim}_{\mathsf{DA}}^{\mathsf{oope}}(n, x_j, y_j) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{DA}}^{\Pi_{\mathsf{OOPE}}}(\mathbb{S}, x_j, \mathsf{sk}), \tag{4.5}$$

$$\mathsf{Sim}_{\mathsf{CSP}}^{\mathsf{oope}}(\mathbb{S}, \langle [\![x_j]\!], y_j \rangle) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{CSP}}^{\Pi_{\mathsf{OOPE}}}(\mathbb{S}, x_j, \mathsf{sk}). \tag{4.6}$$

4.3 Deterministic Construction

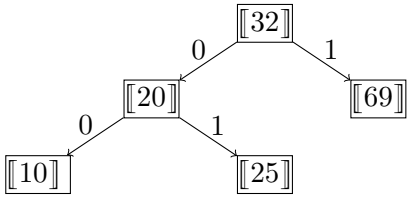
In this section, we present our scheme Π_{OOPE} that consists of an initialization step and a computation step. The initialization step generates the server state and is run completely by the Data Owner. The server state and the ciphertexts are sent to the CSP afterward.

4.3.1 Overview

Our starting points are the following OPE schemes which encrypt a plaintext using a two-party interactive protocol. These schemes have been briefly reviewed in Section 3.4 of Chapter 3.

- Popa et al. [163] introduced the idea of OPE based on a binary search tree and uses an interactive protocol for encryption. We will refer to this scheme as mOPE_1.
- Kerschbaum and Schröpfer [124] reduced the cost of encryption by storing the state at the client. We will refer to this scheme as mOPE_2. This scheme is not interactive and will be used by the DO to generate the initial OPE state in the initialization step.

[2] Recall from Remark 4.2.6 that $[\![x_j]\!]$ is used by the CSP to update the server state.



(a) OPE-tree $\mathcal{T}$

$[\![x_i]\!]$	y_i
$[\![10]\!]$	4
$[\![20]\!]$	7
$[\![25]\!]$	11
$[\![32]\!]$	14
$[\![69]\!]$	21

(b) OPE-table $\mathbb{T}$

$\langle x_i, y_i \rangle$
$\langle 32, 14 \rangle$
$\langle 20, 7 \rangle$
$\langle 25, 11 \rangle$
$\langle 69, 21 \rangle$
$\langle 10, 4 \rangle$

(c) Data Owner State

Figure 4.3: Example initialization

Our scheme Π_{OOPE} extends these two-party protocols to three-party protocols.

Remark 4.3.1. *When using* $\mathsf{mOPE_1.Enc}$*, the binary representation of the order reveals the corresponding path in the tree. In contrast,* $\mathsf{mOPE_2}$ *allows the secret key owner to choose not just the length of the OPE encoding, but also the order range like* $0, \ldots, M$*. If* $\log_2 M$ *is larger than the needed length of the OPE encoding and* M *is not a power of two, then for a ciphertext* $\langle [\![x_i]\!], y_i \rangle$*,* y_i *does not reveal the position of* $[\![x_i]\!]$ *in the tree. In Figure 4.3a for instance, when applying* $\mathsf{mOPE_1}$ *with* $h = 3$*, the order of 25 (i.e. 011 = 3) reveals the corresponding path in the tree. However, with* $\mathsf{mOPE_2}$ *and* $M = 28$*, 25 has order 11 = 1011.*

The initialization consists of generating the server state and can be done by the DO alone. Therefore, it will rely on $\mathsf{mOPE_2}$ only, which is non-interactive. For the online protocol, we will combine both $\mathsf{mOPE_1}$ and $\mathsf{mOPE_2}$, by traversing the search tree interactively as in $\mathsf{mOPE_1}$, and then encoding the order as in $\mathsf{mOPE_2}$. This is because of Remark 4.3.1 and the fact that the DA will receive the OPE encoding.

4.3.2 Initialization

Let $\mathbb{D} = \{x_1, \ldots, x_n\}$ be the unordered DO's dataset and $h = \log_2 n$. The DO chooses a range $0, \ldots, M$ such that $\log_2 M > h$ (Remark 4.3.1), runs $\mathsf{OPE.Enc}$ from Definition 4.2.1 (using $\mathsf{mOPE_2}$) and sends the generated OPE-table to the CSP.

For example, if $\mathbb{D} = \{10, 20, 25, 32, 69\}$ is the data set, $M = 28$ and the insertion order is $32, 20, 25, 69, 10$. Then the ciphertexts after executing algorithm $\mathsf{OPE.Enc}$ are $\langle [\![32]\!], 14 \rangle$, $\langle [\![20]\!], 7 \rangle$, $\langle [\![25]\!], 11 \rangle$, $\langle [\![69]\!], 21 \rangle$, $\langle [\![10]\!], 4 \rangle$. The OPE-tree, the OPE-table and the DO state are depicted in Figure 4.3. We have $\mathsf{pred}(\langle [\![32]\!], 14 \rangle) = \langle [\![25]\!], 11 \rangle, \mathsf{succ}(\langle [\![32]\!], 14 \rangle) = \langle [\![69]\!], 21 \rangle, \mathsf{lnod}([\![32]\!]) = [\![20]\!], \mathsf{rnod}([\![32]\!]) = [\![69]\!]$.

4.3.3 Algorithms

In the following, we present our main protocol that repeatedly makes calls to a subprotocol (Protocol 4.6). Both protocols run between the three parties. During the protocol's execution, the CSP runs Algorithm 4.7 to traverse the tree and Algorithm 4.8 to compute the order (as in $\mathsf{mOPE_2}$). We will deal with $\mathsf{mOPE_3}$ in Section 4.4. An overview of the protocol is illustrated in Figure 4.4.

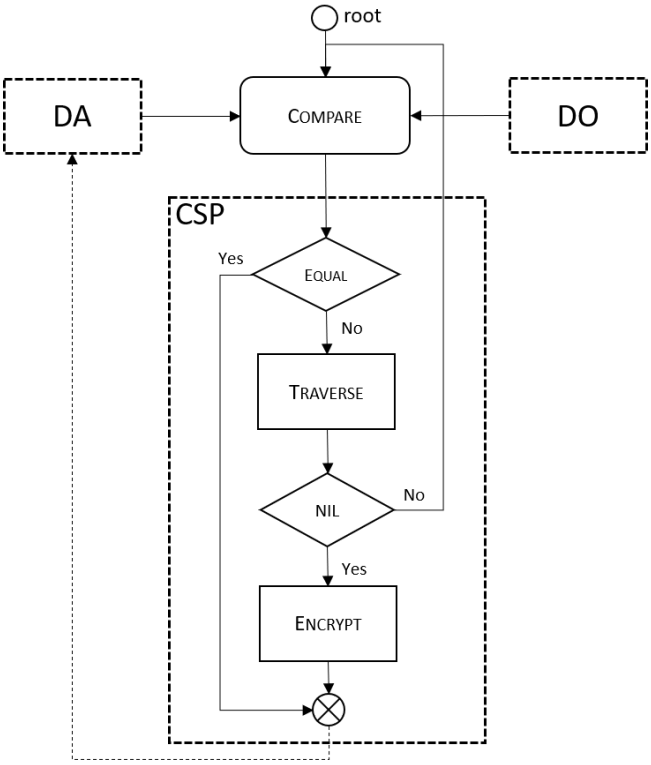


Figure 4.4: Overview of the protocol

Our OOPE Protocol. Protocol 4.5 is executed between the three parties. First, the CSP retrieves the root of the tree and sets it as the current node. Then the protocol loops h ($= \log_2 n$) times. In each step of the loop, the CSP increments the counter and the parties run an oblivious comparison protocol (Protocol 4.6) whose result enables the CSP to traverse the tree (Algorithm 4.7). If the inputs are equal or the next node is empty then the traversal stops. However, the CSP uses the current node as input to the next comparison until the counter reaches the value h. After the loop, the result is either the order of the current node in case of equality or it is computed by the CSP using Algorithm 4.8. In the last step, the DA computes $[\![x_j]\!]$ using DO's public key pk and sends it to the CSP as argued in Remark 4.2.6. Alternatively, the DA could generate a unique identifier (UID) for each element that is being inserted and send this UID instead. So if the corresponding node is later involved in a comparison step, the result is computed by the DA alone.

Oblivious Comparison Protocol. Protocol 4.6 runs between the three parties as well, with input $([\![x_i]\!], x_j, \mathsf{sk})$ for the CSP, the DA and the DO, respectively. It outputs two bits $b_g = (if\ \overline{x} > x\ then\ 1\ else\ 0)$ and $b_e = (if\ \overline{x} \neq x\ then\ 1\ else\ 0)$ to the server. As the actual comparison GC is run between the DA and DO and they are not allowed to learn the result, they use masking bits (b_a, b'_a), (b_o, b'_o) in the GC protocol.

First, the CSP randomizes its input, with a random integer $r \in \{0, \ldots, 2^{\mu+\sigma}\}$[3], to $[\![x_i + r]\!] \leftarrow [\![x_i]\!] \cdot [\![r]\!]$, by first computing $[\![r]\!]$ with DO's public key, such that the DO will not be able to identify the position in the tree, and it sends $[\![x_i + r]\!]$ to the DO and r to the DA. The next step is to compare $x_i + r$ and $x_j + r$ using a GC protocol between the DO and the DA, whereby the comparison results should be revealed only to the

[3]Where σ is the security parameter that determines the statistical leakage, e.g. $\sigma = 32$ [70].

Input $(In_{CSP}, In_{DA}, In_{DO})$: $(\mathbb{S}, x_j, \mathsf{sk})$
Output $(Out_{CSP}, Out_{DA}, Out_{DO})$: $(\langle [\![x_j]\!], y_j \rangle, y_j, \varnothing)$
Functionality : $\text{OOPE}(\mathbb{S}, x_j, \mathsf{sk})$

1: CSP : **retrieve** root $[\![x_{root}]\!]$ of $\mathcal{T}$
2: CSP : **let** $[\![x_i]\!] \leftarrow [\![x_{root}]\!]$
3: CSP : **let** $count \leftarrow 0$
4: **repeat**
5: $(\langle b_e, b_g \rangle, \varnothing, \varnothing) \leftarrow \textsc{Compare}([\![x_i]\!], x_j, \mathsf{sk})$
6: CSP : **if** $b_e \neq 0$ **then** ▷ meaning $x_j \neq x_i$
7: CSP : $[\![x_{next}]\!] \leftarrow \textsc{Traverse}(b_g, [\![x_i]\!])$
8: CSP : **if** $[\![x_{next}]\!] \neq$ **NIL then**
9: CSP : **let** $[\![x_i]\!] \leftarrow [\![x_{next}]\!]$
10: CSP : **end if**
11: CSP : **end if**
12: CSP : **let** $count \leftarrow count + 1$
13: **until** $count = h$
14: CSP : **if** $b_e = 0$ **then** ▷ meaning $x_j = x_i$
15: CSP : **retrieve** $\langle [\![x_i]\!], y_i \rangle$ **and let** $y_j \leftarrow y_i$
16: CSP : **else**
17: CSP : $y_j \leftarrow \textsc{Encrypt}(b_g, [\![x_i]\!])$
18: CSP : **end if**
19: CSP → DA: **send** y_j
20: DA → CSP: **send** $[\![x_j]\!]$

Protocol 4.5: Oblivious OPE Protocol Π_{OOPE}

CSP. Therefore, the DO and DA respectively choose masking bits b_o, b'_o, and b_a, b'_a to blind the comparison results. Then the DO with input $(b_o, b'_o, x_i + r)$ and the DA with input $(b_a, b'_a, x_j + r)$ engage in a garbled circuit protocol for comparison as described in Section 4.3.5. For simplicity, the garbled circuit is implemented in Protocol 4.6 as ideal functionality. In reality, the DO generates the garbled circuit and the DA evaluates it. The DA and the DO receive $(b_e \oplus b_a \oplus b_o, b_g \oplus b'_a \oplus b'_o)$ as output of this computation and resp. send $(b_a, b'_a, b_e \oplus b_o, b_g \oplus b'_o)$ and $(b_o, b'_o, b_e \oplus b_a, b_g \oplus b'_a)$ to the CSP. Finally, the CSP evaluates Equation 4.7 and outputs $\langle b_e, b_g \rangle$. This will be used to traverse the OPE-tree:

$$\begin{cases} b_e & = b_e \oplus b_o \oplus b_o = b_e \oplus b_a \oplus b_a \\ b_g & = b_g \oplus b'_o \oplus b'_o = b_g \oplus b'_a \oplus b'_a. \end{cases} \tag{4.7}$$

Tree Traversal Algorithm. The tree traversal (Algorithm 4.7) runs only at the CSP. Depending on the output of the oblivious comparison the CSP either goes to the left (line 2) or to the right (line 4). If the comparison step returns equality there is no need to traverse the current node and the protocol returns the corresponding ciphertext.

Encryption Algorithm. Algorithm 4.8 runs at the CSP as well and is called only if the tree traversal (Algorithm 4.7) has to stop. Then the compared values are strictly ordered and depending on that the algorithm finds the closest element to the current

Input $(In_{CSP}, In_{DA}, In_{DO})$: $(\llbracket x_i \rrbracket, x_j, \mathsf{sk})$
Output $(Out_{CSP}, Out_{DA}, Out_{DO})$: $(\langle b_e, b_g \rangle, \varnothing, \varnothing)$
Functionality : COMPARE$(\llbracket x_i \rrbracket, x_j, \mathsf{sk})$

1: CSP: choose $(\mu+\sigma)$-bits random r and compute $\llbracket x_i + r \rrbracket$
2: CSP $\rightarrow$ DO: send $\llbracket x_i + r \rrbracket$
3: CSP $\rightarrow$ DA: send r
4: DO: decrypt $\llbracket x_i + r \rrbracket$ and choose masking bits b_o, b'_o
5: DA: compute $x_j + r$ and choose masking bits b_a, b'_a
6: DO $\rightarrow$ GC: send $(b_o, b'_o, x_i + r)$
7: DA $\rightarrow$ GC: send $(b_a, b'_a, x_j + r)$
8: GC $\leftrightarrow$ DA: send $(b_e \oplus b_a \oplus b_o, b_g \oplus b'_a \oplus b'_o)$
9: GC $\leftrightarrow$ DO: send $(b_e \oplus b_a \oplus b_o, b_g \oplus b'_a \oplus b'_o)$
10: DA $\rightarrow$ CSP: send $(b_a, b'_a, b_e \oplus b_o, b_g \oplus b'_o)$
11: DO $\rightarrow$ CSP: send $(b_o, b'_o, b_e \oplus b_a, b_g \oplus b'_a)$
12: CSP: compute $b_e = b_e \oplus b_o \oplus b_o = b_e \oplus b_a \oplus b_a$
13: CSP: compute $b_g = b_g \oplus b'_o \oplus b'_o = b_g \oplus b'_a \oplus b'_a$
14: CSP: output $\langle b_e, b_g \rangle$

Protocol 4.6: Oblivious Comparison Protocol

function TRAVERSE$(b_g, \llbracket x_i \rrbracket)$
2: **if** $b_g = 0$ **then** ▷ traverse to left
$\llbracket x_{next} \rrbracket \leftarrow \mathsf{lnod}(\llbracket x_i \rrbracket)$
4: **else** ▷ traverse to right
$\llbracket x_{next} \rrbracket \leftarrow \mathsf{rnod}(\llbracket x_i \rrbracket)$
6: **return** $\llbracket x_{next} \rrbracket$

Algorithm 4.7: Tree Traversal at Node $\llbracket x_i \rrbracket$

node in the OPE-table. This element is either the predecessor – $\mathsf{pred}(\cdot)$ – if DA's input is smaller (line 4) or the successor – $\mathsf{succ}(\cdot)$ – if DA's input is larger (line 7). Then if necessary (line 9) rebalance the tree and compute the ciphertext as in line 11.

4.3.4 Optimization

One can reduce the number h of homomorphic decryptions by using plaintexts packing [166]. Let $\llbracket x_i \rrbracket$ be a node at the level $2l, 0 \leq l \leq \frac{h}{2}$, of the tree, and let

$$\llbracket x_i^{(0)} \rrbracket \leftarrow \mathsf{lnod}(\llbracket x_i \rrbracket) \text{ and } \llbracket x_i^{(1)} \rrbracket \leftarrow \mathsf{rnod}(\llbracket x_i \rrbracket)$$

be the left and right child nodes of $\llbracket x_i \rrbracket$. Then we modify Step 2 of Algorithm 4.6 as follows. The CSP chooses a random permutation $\pi : \{0,1\} \rightarrow \{0,1\}$ and random numbers r, r_0, r_1 and sends the ciphertext described in Equation 4.8 instead:

$$\begin{aligned} \mathsf{c} &= \llbracket 2^{2(\mu+\sigma)}(x_i + r) \rrbracket \cdot \llbracket 2^{\mu+\sigma}(x_i^{(\pi^{-1}(0))} + r_{\pi^{-1}(0)}) \rrbracket \cdot \llbracket x_i^{(\pi^{-1}(1))} + r_{\pi^{-1}(1)} \rrbracket \\ &= \llbracket x_i + r \| x_i^{(\pi^{-1}(0))} + r_{\pi^{-1}(0)} \| x_i^{(\pi^{-1}(1))} + r_{\pi^{-1}(1)} \rrbracket. \end{aligned} \tag{4.8}$$

```
function ENCRYPT(b_g, [[x_i]])
    retrieve ⟨[[x_i]], y_i⟩ from the OPE-table
    if b_g = 0 then                                   ▷ x_j < x_i
        ⟨[[x']], y'⟩ ← pred(⟨[[x_i]], y_i⟩)
        let y_l ← y' and y_r ← y
    else                                              ▷ x_i < x_j
        ⟨[[x'']], y''⟩ ← succ(⟨[[x_i]], y_i⟩)
        let y_l ← y_i and y_r ← y''
    if y_r − y_l = 1 then
        rebalance the OPE-tree
    y_j ← y_l + ⌈(y_r − y_l)/2⌉
    return y_j
```

Algorithm 4.8: Computing the order of x_j (by inserting at $[\![x_i]\!]$)

The DO decrypts c and uses $x_i + r$ in the comparison at level $2l$. In the next protocol iteration, the CSP sends $\pi(b_g)$ to the DO and $r_{\pi(b_g)}$ to the DA. The DO does not need to decrypt this time (i.e., at level $2l+1$) and just uses $x_i^{(\pi(b_g))} + r_{\pi(b_g)}$ in the next GC comparison. This results in $\frac{h}{2}$ homomorphic decryptions and can be extended to multiple levels (as proved in the following lemma). It can also be precomputed by the CSP.

Lemma 4.3.2. *Let $s = \log(\mathsf{pk})$ be the bitlength of the public key pk. Then the plaintext packing of Equation 4.8 can be extended to up to $\log(\lfloor \frac{s}{\mu+\sigma} \rfloor + 1)$ levels of a complete OPE-tree.*

Proof. A complete tree with d levels has $2^d - 1$ nodes (i.e., a tree with only 1 node has 1 level). If $s = \log(\mathsf{pk})$, then we can pack up to $\lfloor s/(\mu+\sigma) \rfloor$ plaintexts in one ciphertext. Finally, solving the following equation:

$$2^d - 1 = \lfloor s/(\mu + \sigma) \rfloor$$

in d returns $d = \log(\lfloor s/(\mu + \sigma) \rfloor + 1)$. □

Moreover, one can use JustGarble [19] for highly efficient circuit garbling. JustGarble is an optimized garbling scheme based on fix-key block cipher like AES instead of cryptographic hash function. It benefits from various implementation optimizations like AES-NI to speed up the garbling process and the evaluation of garbled circuits.

4.3.5 Oblivious Integer Comparison

In this section, we describe how the parties compare the inputs such that the result is revealed only to the server.

Using Garbled Circuit. To implement oblivious comparison, we adapted the GCs of [130, 131] to fit in our OOPE scheme. Firstly, instead of implementing one garbled circuit for comparison and another one for equality test, we combined both in the same circuit. This allows using the advantage that almost the entire cost of garbled circuit protocols can be shifted into the setup phase. In Yao's protocol, the setup phase contains all expensive operations (i.e., computationally expensive OT and creation

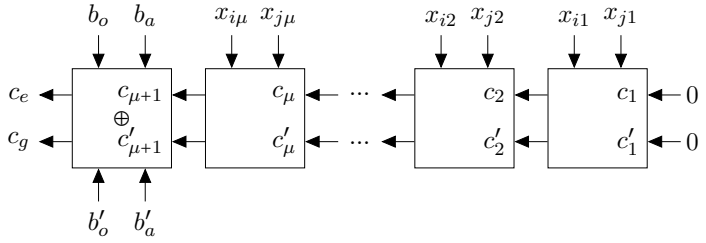


Figure 4.9: Overview of $GC_{=,>}$ for GT and EQ test: Each box for $u = 1, \ldots, \mu$ is a 1-bit circuit for EQ and GT test and outputs $c_{u+1} = (x_{iu} \oplus x_{ju}) \vee c_u$ (Equation 4.9) and $c'_{u+1} = (x_{iu} \oplus c'_u) \wedge (x_{ju} \oplus c'_u) \oplus x_{iu}$ (Equation 4.10) resp.. The last circuit implements the exclusive-or operation and outputs the bits $c_e = c_{\mu+1} \oplus b_o \oplus b_a$ and $c_g = c'_{\mu+1} \oplus b'_o \oplus b'_a$.

of GC, as well as the transfer of GC that dominates the communication complexity) [130]. Hence, by implementing both circuits in only one we reduce the two costly setup phases to one. Secondly, in our oblivious OPE protocol, integer comparison is an intermediate step, hence the output should not be revealed neither to the DA nor to the DO, since this will leak information. Thus the input of the circuit contains a masking bit for each party that is used to mask the actual output. Only the party that receives the masked output and both masking bits can recover the actual output.

Let $GC_{=,>}$ denote this circuit. Let $x_i^{\mathsf{b}} = x_{i\mu}, \ldots, x_{i1}$, $x_j^{\mathsf{b}} = x_{j\mu}, \ldots, x_{j1}$ be the i-th and j-th inputs of the DO and the DA in binary representation. The parties choose masking bits b_o, b'_o, b_a, b'_a and extend their input to $(b_o, b'_o, x_{i\mu}, \ldots, x_{i1})$, $(b_a, b'_a, x_{j\mu}, \ldots, x_{j1})$.

For the equality test, we extend Equation 3.1 to Equation 4.9:

$$\begin{cases} c_1 & = 0 \\ c_{u+1} & = (x_{iu} \oplus x_{ju}) \vee c_u, \ u = 1, \ldots, \mu \\ c_e & = c_{\mu+1} \oplus b_o \oplus b_a. \end{cases} \tag{4.9}$$

The actual output $c_{\mu+1}$ of the circuit is blinded by applying exclusive-or operations with the masking bits b_o and b_a resulting in the bit c_e, where e stands for *equality test.*

For the greater-than comparison, we extend Equation 3.2 to Equation 4.10.

$$\begin{cases} c'_1 & = 0 \\ c'_{u+1} & = (x_{iu} \oplus c'_u) \wedge (x_{ju} \oplus c'_u) \oplus x_{iu}, \ u = 1, \ldots, \mu \\ c_g & = c'_{\mu+1} \oplus b'_o \oplus b'_a. \end{cases} \tag{4.10}$$

Again the actual output $c'_{\mu+1}$ is blinded by applying exclusive-or operations with the masking bits b'_o and b'_a resulting in the bit c_g, where g stands for *greater-than.* An overview of the comparison circuit is illustrated in Figure 4.9.

The performance metric for a garbled circuit is the number of non XOR-gates [130, 131]. Hence, assuming the inputs are μ-bit long, both circuits for equality test and greater-than comparison contain each μ AND-gates. Each AND-gate is garbled with 4μ ciphertexts. However, the *halfGate optimization* [197] reduces the number of ciphertexts per AND-gate by a factor of 2 at the cost of the evaluator to perform two cheap symmetric operations, rather than one. As a result, the garbled circuit $GC_{=,>}$ contains 4μ symmetric ciphertexts.

Using Other Comparison Protocols. Our scheme works with any 2-PC for comparison. We describe how oblivious comparison can be implemented using DGK comparison protocol [63].

The DGK comparison protocol can be used for oblivious comparison as follows. For each node x_i in the OPE-tree, the DO stores both $[\![x_i]\!], [\![x_i^{\mathsf{b}}]\!]$ during the initialization. During the oblivious comparison step, the CSP sends $[\![x_i]\!], [\![x_i^{\mathsf{b}}]\!]$ to the DA and nothing to the DO.

For equality testing, the DA uses $[\![x_i]\!]$ to evaluate a zero testing by computing $[\![(x_i - x_j) \cdot r_1 + r_2]\!]$ where r_1, r_2 are two numbers chosen randomly in the plaintext space.

For the greater-than comparison, the DA evaluates

$$(b_a, ([\![z'_\mu]\!], \ldots, [\![z'_1]\!])) \leftarrow \textsc{DgkEval}([\![x_i^{\mathsf{b}}]\!], x_j^{\mathsf{b}}).$$

Then the DA sends $[\![(x_i - x_j) \cdot r_1 + r_2]\!]$ and $([\![z'_\mu]\!], \ldots, [\![z'_1]\!])$ to the DO. The DA also sends r_2 and b_a to the CSP. The DO evaluates $b_o \leftarrow \textsc{DgkDecrypt}([\![z'_\mu]\!]_i, \ldots, [\![z'_1]\!]_i)$ and sends $(x_i - x_j) \cdot r_1 + r_2$ and b_o to the CSP. Finally, the CSP sets $b_e \leftarrow 1$ if $(x_i - x_j) \cdot r_1 + r_2 \neq r_2$ and $b_e \leftarrow 0$ otherwise, and $b_g = b_a \oplus b_o$.

4.4 Non-deterministic Construction

In this section, we consider the case where the underlying OPE is non-deterministic as in [123, 125]. Our starting point is the scheme of Kerschbaum [123] that strengthens the security of OPE by allowing the same plaintext to be encrypted with different ciphertexts. We will refer to this scheme as $\mathsf{mOPE_3}$. As above, the first step is the initialization procedure (Section 4.3.2). It remains the same with the difference that the tree traversal and the encryption algorithms work as in $\mathsf{mOPE_3}$ [123]. In the online protocol, we will traverse the tree interactively as in $\mathsf{mOPE_1}$ but randomly as in $\mathsf{mOPE_3}$. Hence, if the equality test returns true (line 14 of Protocol 4.5), then the CSP traverses the tree to the left or to the right depending on the outcome of a random coin. The order y_j of x_j is computed as $y_j = y_{i-1} + \lceil \frac{y_i - y_{i-1}}{2} \rceil$ resp. $y_j = y_i + \lceil \frac{y_{i+1} - y_i}{2} \rceil$ if the algorithm is inserting x_j left resp. right to a node $[\![x_i]\!]$ with corresponding order y_i. However, the result of the equality test leaks the frequency of plaintexts, as it allows the CSP to deduce from the OPE-table that certain nodes have the same plaintext. Therefore it would be preferable to implement the random coin in the secure computation.

4.4.1 Implementing the Random Coin Securely

In the following, x_j and x_i represent as before the inputs of the DA and the DO in the oblivious comparison respectively, and $\mathrm{GC}^u_{=,>}$ represents the unmasked comparison circuit[4] that outputs the bits $b_e = c_{\mu+1}$ as result of the equality test and $b_g = c'_{\mu+1}$ as result of the greater-than comparison. The idea is to adapt the garbled circuit for integer comparison (Section 4.3.5) such that its output allows traversing the tree randomly as in [123], but without revealing the result of the equality test to the CSP.

Lemma 4.4.1. *Let r_j and r_i be any DA's and DO's random bits and $b_r = r_j \oplus r_i$. Then extending the circuit $GC^u_{=,>}$ to the circuit GC^u_b with additional input bits r_j, r_i and with output $b = (b_e \wedge b_g) \vee (\neg b_e \wedge b_r)$ traverses the tree as required.*

[4] This is the sub-circuit that operates on the real input bits (from 1 to μ) without the masking bits.

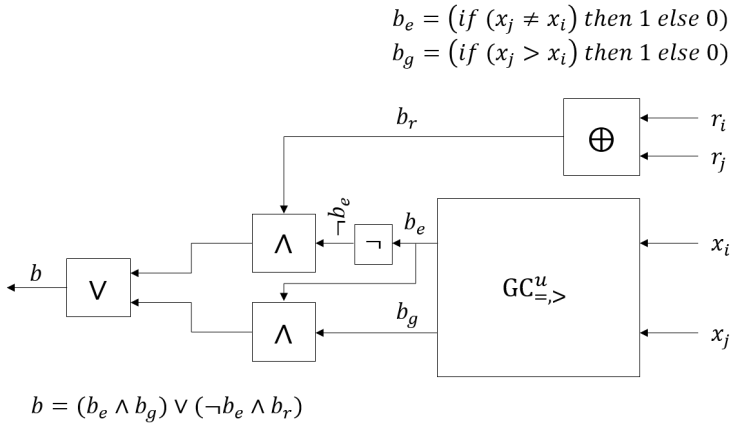


Figure 4.10: Overview of the comparison circuit from Lemma 4.4.1

Proof. If $x_j \neq x_i$ then $b_e = 1$ and $b = b_g$, hence the algorithm traverses the tree depending on the greater-than comparison. Otherwise $\neg b_e = 1$, hence b is the random bit b_r and the tree traversal depends on a random coin. In each case the circuit returns either 0 or 1, and does not reveal if the inputs are equal. □

Now the circuit GC^u_b, which is illustrated in Figure 4.10, can also be extended to the circuit GC_b by using the masking bits b_a and b_o for the DA and the DO respectively as described in Section 4.3.5. The output is then $((b_g \wedge b_e) \vee (\neg b_e \wedge b_r)) \oplus b_a \oplus b_o$.

However, care has to be taken when returning the random bit to the CSP. Recall that the protocol loops h times to prevent the DA and DO from learning the right number of comparisons. This means, if after l iterations we have reached a node $[\![x_i]\!]$ and $l < h$ (i.e., node $[\![x_i]\!]$ is a leaf node), then we perform a "real" comparison $x_j \geq x_i$ (i.e., node $[\![x_i]\!]$ exists in the tree) in that iteration and "dummy" comparisons in the remaining iterations $l' > l$. The "dummy" comparisons consist of comparing $x_j \geq x_i$ again. Therefore, if $x_j = x_i$ and the garbled circuit returns a random bit b in the "real" comparison, then the same random bit b must be returned to the CSP in the garbled circuits of the "dummy" comparison to prevent leaking to the CSP that x_j and x_i are equal. To solve this, the DA and DO must keep track on shares of b_e and b_r which must be extra inputs to the circuit. Let $\hat{b}_e, \hat{b}_r$ be the previous equality bit (initially 1, e.g. 0 for DO and 1 for DA) and random bit (initially 0), then the garbled circuit must execute the following procedure: If $b_e = 0$ then check if $\hat{b}_e = 0$ and return $\hat{b}_r$ otherwise return b_r. If $b_e \neq 0$ then return b_g.

```
    function MUX(sk, [[d]], [[c1]]_a, [[c2]])
2:      b ← [Paillier.Dec([[d]], sk) ?= 0]
        if b = TRUE then
4:          [[c]]_a ← [[c1]]_a
        else
6:          c2 ← Paillier.Dec([[c2]], sk)
            [[c]]_a ← Paillier.Enc(c2, pk_a)
8:      return [[c]]_a
```

Algorithm 4.11: Homomorphic Multiplexer (run by DO)

Input (CSP, DA, DO): $(\langle \mathbb{S}, \langle [\![x_j]\!], y_j \rangle \rangle, \varnothing, \mathsf{sk})$
Output (CSP, DA, DO): $(\varnothing, \langle c_{min}(x_j), c_{max}(x_j) \rangle, \varnothing)$
Functionality : MinMaxOrder$(\mathbb{S}, \langle [\![x_j]\!], y_j \rangle, \mathsf{sk})$

1: CSP: retrieve $\langle [\![x_i]\!], y_i \rangle, \langle [\![x_{i+1}]\!], y_{i+1} \rangle$
with $y_i < y_j < y_{i+1}$
2: CSP: choose random integers s_1, s_2, r_1, r_2
3: CSP: compute
$[\![d_1]\!] \leftarrow [\![(x_i - x_j) \cdot s_1]\!]$
$[\![d_2]\!] \leftarrow [\![(x_{i+1} - x_j) \cdot s_2]\!]$
4: CSP $\rightarrow$ DO:
$\langle [\![d_1]\!], [\![y_j \cdot r_1]\!]_a, [\![c_{min}(x_i) \cdot r_1]\!] \rangle$
$\langle [\![d_2]\!], [\![y_j \cdot r_2]\!]_a, [\![c_{max}(x_{i+1}) \cdot r_2]\!] \rangle$
5: CSP $\rightarrow$ DA: r_1 and r_2
6: DO:
Let $[\![c_{11}]\!]_a \leftarrow [\![y_j \cdot r_1]\!]_a, [\![c_{12}]\!] \leftarrow [\![c_{min}(x_i) \cdot r_1]\!]$
Let $[\![c_{21}]\!]_a \leftarrow [\![y_j \cdot r_2]\!]_a, [\![c_{22}]\!] \leftarrow [\![c_{max}(x_{i+1}) \cdot r_2]\!]$
$[\![c_{min}(x_j)]\!]_a \leftarrow \text{Mux}(\mathsf{sk}, [\![d_1]\!], [\![c_{11}]\!]_a, [\![c_{12}]\!])$
$[\![c_{max}(x_j)]\!]_a \leftarrow \text{Mux}(\mathsf{sk}, [\![d_2]\!], [\![c_{21}]\!]_a, [\![c_{22}]\!])$
7: DO $\rightarrow$ DA: $[\![c_{min}(x_j)]\!]_a, [\![c_{max}(x_j)]\!]_a$
8: DA: decrypt and output
$c_{min}(x_j) \leftarrow \mathsf{Paillier.Dec}([\![c_{min}(x_j)]\!]_a, \mathsf{sk}_a)$
$c_{max}(x_j) \leftarrow \mathsf{Paillier.Dec}([\![c_{max}(x_j)]\!]_a, \mathsf{sk}_a)$

Protocol 4.12: Computing $c_{min}(x_j)$ and $c_{max}(x_j)$ securely

4.4.2 Dealing with Queries

So far we have computed the ciphertext in the non-deterministic case. However, as Kerschbaum pointed out [123] this ciphertext cannot be directly used to query the database. As in the deterministic case let x_i and y_i be symbols for plaintext and order respectively. Since a plaintext x_i might have many ciphertexts let c_{min} and c_{max} be respectively the minimum and maximum order of x_i, hence:

$$\begin{cases} c_{min}(x_i) = \min(\{y_i : \mathsf{OPE.Dec}(\langle [\![x_i]\!], y_i \rangle, \mathsf{sk}) = x_i\}) \\ c_{max}(x_i) = \max(\{y_i : \mathsf{OPE.Dec}(\langle [\![x_i]\!], y_i \rangle, \mathsf{sk}) = x_i\}). \end{cases} \tag{4.11}$$

Thus, a query $[a, b]$ must be rewritten in $[c_{min}(a), c_{max}(b)]$. Unfortunately, in Kerschbaum's scheme the $c_{min}(x_i), c_{max}(x_i)$ are only known to the DO, because they reveal to the server the frequency of plaintexts. Recall that the goal of [123] was precisely to hide this frequency from the CSP.

Instead of returning y_j to the DA, which is useless for queries, our goal is to allow the DA to learn $c_{min}(x_j)$ and $c_{max}(x_j)$ and nothing else. The CSP learns only $\langle [\![x_j]\!], y_j \rangle$ as before and the DO learns nothing besides the intermediate messages of the protocol. We begin by proving the following lemma.

Lemma 4.4.2. *Let $\langle [\![x_i]\!], y_i \rangle, \langle [\![x_{i+1}]\!], y_{i+1} \rangle \in \mathbb{T}$ be elements of the OPE-table such that $x_i \leq x_{i+1}$ and $y_i < y_{i+1}$. Let x_j be a new inserted plaintext with corresponding order y_j such that $x_i \leq x_j \leq x_{i+1}$. Then it holds: $c_{min}(x_j) \in \{c_{min}(x_i), y_j\}$ and $c_{max}(x_j) \in \{c_{max}(x_{i+1}), y_j\}$.*

Proof. If $x_i = x_j$ then by definition of c_{min}, we have $c_{min}(x_j) = c_{min}(x_i)$. If $x_i < x_j$ and $x_j < x_{i+1}$ then x_j occurs only once in the OPE-table $\mathbb{T}$ and it holds $c_{min}(x_j) = c_{max}(x_j) = y_j$. Otherwise x_j is equal to x_{i+1}, but since x_j is new and by assumption $x_j \leq x_{i+1}$ the algorithm is inserting x_j right to x_i and left to x_{i+1} hence $y_i < y_j < y_{i+1}$ must hold. Then by definition again $c_{min}(x_j) = y_j$. For the case of max the proof is similar. □

Corollary 4.4.3. *Let $x_j, x_i, x_{i+1}, y_j, y_i, y_{i+1}$ be as above and let $b_i = [x_i = x_j]?1 : 0$ resp. $b_{i+1} = [x_j = x_{i+1}]?1 : 0$ then it holds: $c_{min}(x_j) = b_i \cdot c_{min}(x_i) + (1 - b_i) \cdot y_j$, resp. $c_{max}(x_j) = b_{i+1} \cdot c_{max}(x_{i+1}) + (1 - b_{i+1}) \cdot y_j$.*

Now we are ready to describe the solution. The first step is to store besides each ciphertext $\langle [\![x_i]\!], y_i \rangle$ two ciphertexts $[\![c_{min}(x_i)]\!]$ and $[\![c_{max}(x_i)]\!]$ in the OPE-table. This is done by the DO during the initialization. Recall that during insertion of a plaintext, rebalancing the tree might become necessary and this affects the c_{min} and c_{max} values. The CSP cannot update c_{min} and c_{max} without knowing the frequency. Hence, if rebalancing happens, the DO must interactively update the ciphertexts $[\![c_{min}(x_i)]\!]$ and $[\![c_{max}(x_i)]\!]$ with the CSP without leaking the frequency. This is done by just scanning the sorted OPE-table. According to [123] the probability of rebalancing is negligible in n for uniform inputs if the maximum order M is larger than $2^{6.4 \cdot \log_2 n}$. For non-uniform input, smaller values of M are likely.

Let $(\mathsf{pk}_a, \mathsf{sk}_a)$ be a Paillier public/private key pair such that sk_a is known only to the DA and let $[\![x_j]\!]_a \leftarrow \mathsf{Paillier.Enc}(x_j, \mathsf{pk}_a)$ be a ciphertext of x_j encrypted with Paillier's encryption under the public key pk_a. After the computation of y_j using Protocol 4.5 with the random traversal as discussed in Section 4.4.1, the CSP learns $\langle [\![x_j]\!], y_j \rangle$. Then the parties execute Protocol 4.12 with $\langle \mathbb{S}, \langle [\![x_j]\!], y_j \rangle \rangle$ and sk as input for the CSP and the DO respectively. The DA does not have any input, but is the only one to receive the output of Protocol 4.12 which works as follows. The CSP retrieve the two closest elements $\langle [\![x_i]\!], y_i \rangle, \langle [\![x_{i+1}]\!], y_{i+1} \rangle$ such that $y_i < y_j < y_{i+1}$ (Step 1). Then it homomorphically computes the differences $x_i - x_j$ and $x_{i+1} - x_j$ and randomizes them resulting in the ciphertexts $[\![d_1]\!]$, $[\![d_2]\!]$ (Step 3). Recall that the ciphertext $[\![c_{min}(x_i)]\!]$ and $[\![c_{max}(x_{i+1})]\!]$ are already in the OPE-table and the CSP knows y_j. The CSP then sends the ciphertexts

$$\langle [\![d_1]\!], [\![y_j \cdot r_1]\!]_a, [\![c_{min}(x_i) \cdot r_1]\!] \rangle \text{ and } \langle [\![d_2]\!], [\![y_j \cdot r_2]\!]_a, [\![c_{max}(x_{i+1}) \cdot r_2]\!] \rangle$$

to the DO (Step 4) and the random numbers r_1 and r_2 to the DA (Step 5). The DO uses Algorithm 4.11 to compute (Step 6) and send the following ciphertexts

$$[d_1 = 0]?[\![y_j \cdot r_1]\!]_a : [\![c_{min}(x_i) \cdot r_1]\!]_a \text{ and } [d_2 = 0]?[\![y_j \cdot r_2]\!]_a : [\![c_{max}(x_i) \cdot r_2]\!]_a$$

to the DA (Step 7). The DA finally decrypts and outputs the values $c_{min}(x_j)$ and $c_{max}(x_j)$ (Step 8), which can then be used to query the encrypted database.

Notice that for an input x_j of the DA the ciphertext $\langle [\![x_j]\!], y_j \rangle$ is not inserted in the database, but only in the OPE-table, because it cannot be included in the result of a query. Particularly, if $\langle [\![x_j]\!], y_j \rangle$ is no longer needed (e.g., after the data analysis) it must be removed from the OPE-table. As stated in Lemma 4.4.2, if it happens that the new x_j with corresponding order y_j is inserted between x_i and x_{i+1} such that $x_i < x_j = x_{i+1}$ then $c_{min}(x_j) = y_j$ implies that the previous $c_{min}(x_{i+1})$ should be updated to y_j. However, as explained before this update is not necessary.

In Protocol 4.12, the DO sees two semantically secure ciphertexts $[\![y_j \cdot r_1]\!]_a$ and $[\![y_j \cdot r_2]\!]_a$, which it cannot decrypt, and four randomized plaintexts $d_1, c_{min}(x_i) \cdot$

$r_1, d_2, c_{max}(x_{i+1}) \cdot r_2$. The DA sees two random integers r_1, r_2 and the output of the protocol. The CSP receives no new message. Hence, the simulation is straightforward. Finally, correctness and security proof of the overall protocol when OOPE is instantiated with $\mathsf{mOPE_3}$ is similar to the case of $\mathsf{mOPE_2}$.

4.4.3 Extending to Data Structures based OPE

We now briefly describe how OOPE can be realized with ESEDS-OPE [125] that is IND-CPA-DS-secure. As explained before, ESEDS-OPE combines the benefits of three previous order-preserving encryption schemes: $\mathsf{mOPE_1}$ [163], $\mathsf{mOPE_3}$ [123] and MOPE [31]. ESEDS-OPE consists of four algorithms: key generation, encryption, decryption and search. The key generation algorithm takes a security parameter and outputs a secret key. The decryption algorithm takes the secret key and a ciphertext and outputs the corresponding plaintext. Key generation and decryption algorithms are run as before by the data owner alone.

The encryption algorithm takes the secret key, a plaintext and the state of the encryption stored on the server. Then it requires the data owner and the server to interactively perform a binary search using Kerschbaum's random tree traversal as in $\mathsf{mOPE_3}$. Finally, the server inserts the new plaintext and rotates the resulting data structure around a new random modulus. The extension to OOPE will however concern only the tree traversal. In the OOPE protocol, the three parties just have to traverse the tree with the extension discussed in Section 4.4.1 by implementing the comparison and the random coin in a garbled circuit.

The search algorithm takes the secret key, the state of the encryption and a range $[a, b]$. Let c_α (resp. c_β) be the ciphertext with the smallest (resp. largest) order such that for the associated plaintext α (resp. β) it holds $\alpha \geq a$ (resp. $\beta \leq b$). Then the search algorithm requires the data owner and the server to interactively perform two binary searches (using a non random tree traversal) to get c_α and c_β and then returns all ciphertexts between c_α and c_β. Here as well, the extension to OOPE only concerns the tree traversal. The parties will just have to traverse the tree as in the original OOPE protocol as described in Section 4.3.3 with the only difference that they will not have to check for equality as the traversal here is no longer random.

4.5 Correctness and Security Analysis

In this section, we provide correctness and security analysis for the deterministic protocol described in Section 4.3. The non-deterministic case is similar. The proofs rely on the correctness and security of the Paillier scheme and Yao's protocol. The correctness and semantic security proofs of Paillier's scheme are provided in [159]. The proofs for Yao's protocol are provided in [139]. We will use the resulting simulators for Yao's protocol to construct simulators for DO and DA in the GC evaluation.

Theorem 4.5.1 (Correctness)**.** *The protocol* Π_{OOPE} *correctly implements the OOPE functionality* $\mathcal{F}_{\mathsf{OOPE}}$.

Proof. The correctness depends on the oblivious comparison (Protocol 4.6). Since Paillier is correct, the DO can correctly get $x + r$ from $[\![x + r]\!]$. Let $b_g = (if\ x_j > x_i\ then\ 1\ else\ 0)$ and $b_e = (if\ x_j \neq x_i\ then\ 1\ else\ 0)$. From inputs $(b_a, b'_a, x_j + r)$ of the DA and $(b_o, b'_o, x_i + r)$ of the DO the garbled circuit $\mathrm{GC}_{=,>}$ (Figure 4.9) correctly returns $(b_e \oplus b_a \oplus b_o, b_g \oplus b'_a \oplus b'_o)$ to the DA and $(b_e \oplus b_a \oplus b_o, b_g \oplus b'_a \oplus b'_o)$ to the DO. Then the DA resp. the DO sends $(b_a, b'_a, b_e \oplus b_o, b_g \oplus b'_o)$ resp. $(b_o, b'_o, b_e \oplus b_a, b_g \oplus b'_a)$ to

$$[\![x_i^{(l)} + r^{(l)}]\!], x_i^{(l)} + r^{(l)}, b_o^{(l)}, b_o'^{(l)},$$
$$\mathsf{View}_{\mathsf{DO}}^{\Pi_{\mathsf{GC}}}(\langle b_o^{(l)}, b_o'^{(l)}, x_i^{(l)} + r^{(l)}\rangle, \langle b_a^{(l)}, b_a'^{(l)}, x_j^{(l)} + r^{(l)}\rangle),$$
$$\langle b_e^{(l)} \oplus b_a^{(l)} \oplus b_o^{(l)}, b_g^{(l)} \oplus b_a'^{(l)} \oplus b_o'^{(l)}\rangle,$$
$$\langle b_e^{(l)} \oplus b_a^{(l)}, b_g^{(l)} \oplus b_a'^{(l)}\rangle$$

(a) DO View: for $l = 0, \cdots, d-1$

$$[\![x'^{(l)}]\!], x'^{(l)}, b^{(l)}, b'^{(l)},$$
$$\mathsf{Sim}_{\mathsf{DO}}^{\mathsf{gc}}(\langle b^{(l)}, b'^{(l)}, x'^{(l)}\rangle, \langle b_1^{(l)} \oplus b^{(l)}, b_2^{(l)} \oplus b'^{(l)}\rangle),$$
$$\langle b_1^{(l)} \oplus b^{(l)}, b_2^{(l)} \oplus b'^{(l)}\rangle,$$
$$\langle b_1^{(l)}, b_2^{(l)}\rangle$$

(b) DO Simulator: for $l = 0, \cdots, d-1$

Figure 4.13: DO Simulation

the CSP. With Equation 4.7 the CSP can correctly deduce b_e and b_g. The correctness of binary search concludes the proof. □

Theorem 4.5.2 (Security)**.** *The protocol* Π_{OOPE} *securely implements the OOPE functionality* $\mathcal{F}_{\mathsf{OOPE}}$ *in the semi-honest model with honest majority.*

Proof. Since the protocol makes a call to the comparison functionality involving the DO and the DA, the proof will use the simulators provided by [139] to generate their views. Let $\mathsf{Sim}_{\mathsf{DO}}^{\mathsf{gc}}$ and $\mathsf{Sim}_{\mathsf{DA}}^{\mathsf{gc}}$ (resp. $\mathsf{View}_{\mathsf{DO}}^{\Pi_{\mathsf{GC}}}$ and $\mathsf{View}_{\mathsf{DA}}^{\Pi_{\mathsf{GC}}}$) be the simulators (resp. the views) of the DO and the DA in the comparison protocol. We follow the idea of [139] by proving the cases separately, when the DO is corrupted, the DA is corrupted and the CSP is corrupted.

Case 1 - DO is corrupted. The view of the DO consists of randomized inputs and its view in the comparison steps. Let d be the number of comparisons required to encrypt x_j, then $\mathsf{View}_{\mathsf{DO}}^{\Pi_{\mathsf{OOPE}}}(\mathbb{S}, x_j, \mathsf{sk})$ is illustrated in Figure 4.13a.

Upon input $(n, \mathsf{sk}, \varnothing)$, $\mathsf{Sim}_{\mathsf{DO}}^{\mathsf{oope}}$ generates the output as illustrated in Figure 4.13b, where $x'^{(l)}$ is a random integer and $b^{(l)}, b'^{(l)}, b_1^{(l)}, b_2^{(l)}$ are random bits.

Clearly, the outputs of Figures 4.13a and 4.13b are indistinguishable from each other (Equation 4.4). This is because $x_i^{(l)} + r^{(l)}, b_o^{(l)}, b_o'^{(l)}$ are just as random as $x'^{(l)}, b^{(l)}, b'^{(l)}$ respectively. Furthermore, since $b_a^{(l)}$ and $b_a'^{(l)}$ are randomly chosen by the DA, $b_e^{(l)} \oplus b_a^{(l)}, b_g^{(l)} \oplus b_a'^{(l)}$ are also just as random as $b_1^{(l)}, b_2^{(l)}$ respectively. The security of Yao's protocol [139] finishes the proof.

Case 2 - DA is corrupted. This case is similar to the DO's case with the only difference that the DA knows x_j which is the same in each protocol round. The view $\mathsf{View}_{\mathsf{DA}}^{\Pi_{\mathsf{OOPE}}}(\mathbb{S}, x, \mathsf{sk})$ is illustrated in Figure 4.14a.

Notice that also the DA is unaware of the result of the comparison, because the output is randomized by a bit of the DO. The simulator for the DA works in the same way as $\mathsf{Sim}_{\mathsf{DO}}^{\mathsf{oope}}$. On input (n, x_j, y_j) $\mathsf{Sim}_{\mathsf{DA}}^{\mathsf{oope}}$ generates the output illustrated in Figure 4.14b, where $b^{(l)}, b'^{(l)}, b_1^{(l)}, b_2^{(l)}$ are random bits.

$$r^{(l)}, x_j + r^{(l)}, b_a^{(l)}, b_a'^{(l)},$$
$$\mathsf{View}_{\mathsf{DA}}^{\Pi_{\mathsf{GC}}}(\langle b_o^{(l)}, b_o'^{(l)}, x_i^{(l)} + r^{(l)}\rangle, \langle b_a^{(l)}, b_a'^{(l)}, x_j + r^{(l)}\rangle),$$
$$\langle b_e^{(l)} \oplus b_a^{(l)} \oplus b_o^{(l)}, b_g^{(l)} \oplus b_a'^{(l)} \oplus b_o'^{(l)}\rangle,$$
$$\langle b_e^{(l)} \oplus b_o^{(l)}, b_g^{(l)} \oplus b_o'^{(l)}\rangle$$

(a) DA View: for $l = 0, \cdots, d-1$

$$r'^{(l)}, x_j + r'^{(l)}, b^{(l)}, b'^{(l)},$$
$$\mathsf{Sim}_{\mathsf{DA}}^{\mathsf{oope}}(\langle b^{(l)}, b'^{(l)}, x_j + r'^{(l)}\rangle, \langle b_1^{(l)} \oplus b^{(l)}, b_2^{(l)} \oplus b'^{(l)}\rangle),$$
$$\langle b_1^{(l)} \oplus b^{(l)}, b_2^{(l)} \oplus b'^{(l)}\rangle,$$
$$\langle b_1^{(l)}, b_2^{(l)}\rangle$$

(b) DA Simulator: for $l = 0, \cdots, d-1$

Figure 4.14: DA Simulation

Case 3 - CSP is corrupted. The CSP's view $\mathsf{View}_{\mathsf{CSP}}^{\Pi_{\mathsf{OOPE}}}(\mathbb{S}, x, \mathsf{sk})$ consists of random integers and GC's outputs from the DA and the DO. It is illustrated in Figure 4.15a.

$\mathsf{Sim}_{\mathsf{CSP}}^{\mathsf{oope}}$ is given the server state $\mathbb{S}$ and a valid ciphertext $\langle [\![x_j]\!], y_j\rangle$. Then it chooses two elements $\langle [\![x_i]\!], y_i\rangle$, $\langle [\![x_{i+1}]\!], y_{i+1}\rangle$ from the OPE-table, such that $y_i \le y_j < y_{i+1}$. Finally, it inserts $[\![x_j]\!]$ in the tree and simulates the path P from the root to $[\![x_j]\!]$. Let $\mathsf{depth}([\![x_i]\!])$ denotes the number of edges from the root to $[\![x_i]\!]$. There are three possible cases:

- if $y_i = y_j$ then $[\![x_i]\!]$ and $[\![x_j]\!]$ encrypt the same plaintext (i.e., $x_i = x_j$),
- else if $\mathsf{depth}([\![y_i]\!]) > \mathsf{depth}([\![y_{i+1}]\!])$ then insert $[\![x_j]\!]$ right to $[\![x_i]\!]$,
- else $\mathsf{depth}([\![x_{i+1}]\!]) > \mathsf{depth}([\![x_i]\!])$, insert $[\![x_j]\!]$ left to $[\![x_{i+1}]\!]$.

For all ancestors of $[\![x_j]\!]$, $b_g^{(l)}$ is 0 (resp. 1) if the path P goes to the left (resp. to the right). The value of $b_e^{(l)}$ is 1 for all ancestors of $[\![x]\!]$ (the equality test from Equation 3.1 returns 1 if the inputs are different and 0 otherwise). For the node $[\![x_j]\!]$ itself, there are two possible cases:

- $[\![x_j]\!]$ is not a leaf: this occurs if one is trying to insert a value, that was already in the tree. It holds $b_g^{(l)} = b_e^{(l)} = 0$ because y_i is equal to y_j.
- $[\![x_j]\!]$ is a leaf: this occurs either because $y_i = y_j$ holds as above or $[\![x_j]\!]$ is inserted at a leaf node. If $y_i = y_j$ holds, then $b_e^{(l)} = 0$ which also implies $b_g^{(l)} = 0$. If $[\![x_j]\!]$ is inserted at a leaf node, then $b_g^{(l)}$ and $b_e^{(l)}$ are undefined because no comparison was done. Hence the simulator chooses $b_g^{(l)} = b_e^{(l)}$ randomly between 0 and undefined.

To simulate the CSP's view, $\mathsf{Sim}_{\mathsf{CSP}}^{\mathsf{oope}}$ chooses a random integer $r'^{(l)}$ and random bits $b_\alpha^{(l)}, b_\alpha'^{(l)}$ and $b_\omega^{(l)}, b_\omega'^{(l)}$ and generates the output illustrated in Figure 4.15b.

Since $[\![x^{(l)}]\!], b_e^{(l)}, b_g^{(l)}$ depend on the path, they are the same in Figures 4.15a and 4.15b, and $r^{(l)}, b_a^{(l)}, b_a'^{(l)}, b_o^{(l)}, b_o'^{(l)}$ are indistinguishable from $r'^{(l)}, b_\alpha^{(l)}, b_\alpha'^{(l)}, b_\omega^{(l)}, b_\omega'^{(l)}$. □

$$[\![x_i^{(l)}]\!], r^{(l)}, [\![x_i^{(l)} + r^{(l)}]\!], \langle b_a^{(l)}, b_a^{\prime(l)}, b_e^{(l)} \oplus b_o^{(l)}, b_g^{(l)} \oplus b_o^{\prime(l)} \rangle, \langle b_o^{(l)}, b_o^{\prime(l)}, b_e^{(l)} \oplus b_a^{(l)}, b_g^{(l)} \oplus b_a^{\prime(l)} \rangle$$

(a) CSP View: for $l = 0, \cdots, d-1$

$$[\![x^{(l)}]\!], r^{\prime(l)}, [\![x^{(l)} + r^{\prime(l)}]\!], \langle b_\alpha^{(l)}, b_\alpha^{\prime(l)}, b_e^{(l)} \oplus b_\omega^{(l)}, b_g^{(l)} \oplus b_\omega^{\prime(l)} \rangle, \langle b_\omega^{(l)}, b_\omega^{\prime(l)}, b_e^{(l)} \oplus b_\alpha^{(l)}, b_g^{(l)} \oplus b_\alpha^{\prime(l)} \rangle$$

(b) CSP Simulator: for $l = 0, \cdots, d-1$

Figure 4.15: CSP Simulation

4.6 Complexity Analysis

This section presents the complexity analysis of our OOPE scheme. We focus on the main scheme as described in Section 4.3.3. The analysis for the extensions of Sections 4.4 and 4.4.3 is similar. Let λ be the security parameter, κ be the bitlength of an asymmetric ciphertext, μ be the bitlength of each plaintext and d be the depth of the OPE-tree.

4.6.1 Computation Complexity

The CSP performs one asymmetric operation per comparison resulting in a total $\mathcal{O}(d)$ asymmetric operations.

The DO performs one asymmetric operation and $\mathcal{O}(\mu)$ symmetric operations per comparison resulting in a total $\mathcal{O}(d)$ asymmetric operations and $\mathcal{O}(d\mu)$ symmetric operations.

The DA performs $\mathcal{O}(\mu)$ symmetric operations per comparison resulting in a total of $\mathcal{O}(d\mu)$ symmetric operations.

4.6.2 Communication Complexity

For each comparison, the CSP sends one asymmetric ciphertext to the DO and one random μ-bit integer to the DA resulting in $\kappa+\mu$ bits. In total, the CSP sends $(\kappa+\mu)d$ bits.

In a GC protocol, the communication cost of the generator consists of:

- its cost in the OT extension protocol which is $\mu\lambda$ [9] for the OT sender, where μ is the bitlength of the evaluator's input,
- the cost of sending the garbled circuit which is the number of ciphertexts per AND-gate multiplied by the security parameter and
- the cost of sending the garbled input of the generator which is the bitlength multiplied by the security parameter.

As GC generator, the DO's communication cost in each comparison consists of $(\mu+2)\lambda$ bits (our GC for comparison has additionally two blinding bits per party) for OT, $4\mu\lambda$ for the GC itself and $(\mu+2)\lambda$ for the garbled input. After the GC protocol with the DA, the DO sends 2 bits to the CSP. In total, the DO sends

$$((\mu+2)\lambda + 4\mu\lambda + (\mu+2)\lambda + 2)d = ((6\mu+4)\lambda + 2)d \text{ bits.}$$

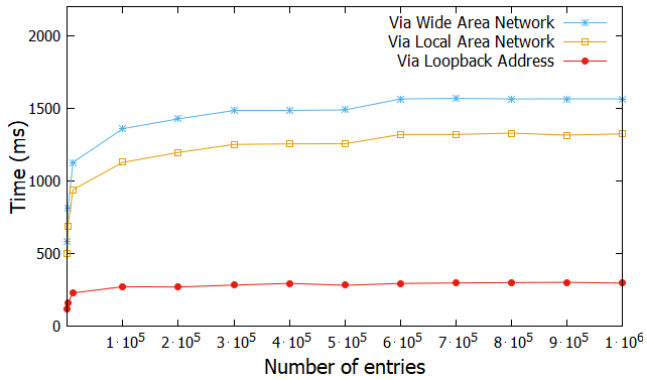


Figure 4.16: Encryption time: The time for WAN is divided by 10.

	Loopback	LAN	WAN
Time (ms)	15	63	755

Table 4.2: Execution Time of the Oblivious Comparison Protocol

The communication cost of the GC evaluator consists only of its cost of the OT extension protocol which is $\mu\lambda$ [9] for the OT receiver, where μ is the bitlength of the evaluator's input. Hence, in the OOPE protocol the DA sends $((\mu+2)\lambda+2)d$ bits.

At security level λ = 128 bits, κ is at least 4096 bits. Assuming μ = 32 bits and d = 20 (i.e., the OPE-tree contains about one million entries), this results in communication costs of 10.07 kB, 61.25 kB, and 10.62 kB for the CSP, the DO and the DA, respectively.

4.7 Evaluation

We have implemented our scheme using SCAPI (Secure Computation API) [75]. SCAPI is an open-source Java library for implementing secure two-party and multiparty computation protocols. It provides a reliable, efficient, and highly flexible cryptographic infrastructure. It also provides many optimizations of GC such as OT extensions, free-XOR, garbled row reduction [75]. Furthermore, there is a built-in communication layer that provides communication services for any interactive cryptographic protocol.

4.7.1 Parameters

The first parameter is the security parameter (i.e., the bit length of the public key) of Paillier's scheme (e.g. 2048 or 4096). Paillier's scheme requires to choose two large prime numbers P and Q of equal length and to compute a modulus $N = PQ$ and the private key $\phi = lcm(P-1, Q-1)$. Then select a random $g \in \mathbb{Z}^*_{N^2}$ such that if e is the smallest integer with $g^e = 1 \bmod N^2$, then N divides e. The public key is (g, N). To encrypt a plaintext m select a random $r \in \mathbb{Z}^*_N$ and compute $\mathsf{c} \leftarrow g^{\mathsf{m}} r^N \bmod N^2$. To decrypt a ciphertext c compute $\mathsf{m} \leftarrow L(\mathsf{c}^\phi \bmod N^2) \cdot \psi \bmod N$, where $L(u) = \frac{u-1}{N}$ and $\psi = (L(g^\phi \bmod N^2))^{-1} \bmod N$. Thus,

$$\mathsf{c} \leftarrow g^{\mathsf{m}} r^N \bmod N^2, \tag{4.12}$$

$$\mathsf{m} \leftarrow L(\mathsf{c}^\phi \bmod N^2) \cdot \psi \bmod N. \tag{4.13}$$

We applied optimizations as recommended in [159]. We implemented our scheme with $g = 1+N$. This transforms the modular exponentiation $g^{\mathsf{m}} \bmod N^2$ to a multiplication, since $(1+N)^{\mathsf{m}} \bmod N^2 = 1+\mathsf{m}N \bmod N^2$. Moreover, we precomputed ψ in Equation 4.13, used Chinese remaindering for decryption and pre-generated randomness for encryption and homomorphic plaintext randomization (Protocol of Figure 4.6). As a result, using the experimental setup described in Section 4.7.3, encryption, decryption, and homomorphic addition take respectively 52 μs, 12 ms, and 67 μs when the key length is 2048 bits.

The other parameters of the OOPE protocol are the length of the inputs (e.g. 32, 64, 128, 256 bits integer), the length of the order $\log_2 M$ – with M the maximal order – (e.g., 32, 64, 128 bits), and the size of the OPE tree (e.g. 10^3, 10^4, 10^5, 10^6 entries).

4.7.2 Evaluation Goals

To evaluate the performance of our scheme, we answer the following questions:

- What time does the scheme take to encrypt an input of the DA?
- How does the network communication influence the protocol?
- What are the average generation time and the storage cost of the OPE-tree?

4.7.3 Experimental Setup

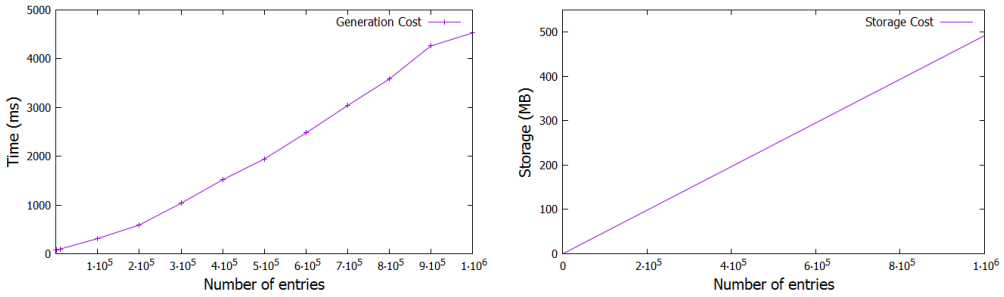
We chose 2048 bits as the bitlength of the public key for Paillier's scheme and ran experiments via loopback address, LAN, and WAN. For LAN, we used 3 machines with Intel(R) Xeon(R) CPU E7-4880 v2 at 2.50GHz: 4 CPUs and 8 GB RAM, 4 CPUs and 4 GB RAM, 2 CPUs and 2 GB RAM. For the loopback, we used the first LAN machine with 4 CPUs and 8 GB RAM. For WAN, we used three machines on amazon web services (Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz 2.30GHz - 244 GB RAM - 64-bit Windows) distributed in three regions (Northern California, Frankfurt, Tokyo).

We generated the OPE-tree with random inputs, balanced it, and encrypted the plaintexts with Paillier encryption. For the DA, we generated 100 random inputs. Then we executed the OOPE protocol 100 times and computed the average time spent in Protocols 4.5 and 4.6, in the GC step, in Paillier's decryption. The optimization introduced in Section 4.3.4 is not part of the following evaluation.

4.7.4 Encryption Costs

Figure 4.16 shows the average cost (y-axis) needed to encrypt a value with the OOPE protocol for OPE-tree with size (x-axis) between 100 and 1,000,000. Overall, the cost of OOPE goes up as the size of the OPE-tree increases. This is because the depth of the tree increases with its size. Hence, this implies a larger number of oblivious comparisons for larger trees. The average encryption time of OOPE for a database with one million entries is about 0.3 s via loopback (1.3 s via LAN, 15.6 s via WAN). This cost corresponds to the cost of comparison multiply by the number of comparisons (e.g. 20 for 1000000 entries) which can be reduced by the optimization of Section 4.3.4.

The inherent sub-protocol for oblivious comparison does not depend on the OPE-tree size but on the input length and the security parameter $\log_2 N$. The time for comparison is illustrated in Table 4.2. Via loopback, the comparison costs about 15 ms which is dominated by the time (about 12 ms to the DO) to decrypt $[\![x+r]\!]$ in Step 4 of Protocol 4.6. The remaining 3 ms are due to the garbled circuit execution, since the



(a) Average generation time

(b) Storage cost for $\log_2 N = 2048$

Figure 4.17: OPE-tree costs

overhead due to network communication is negligible. Via LAN, the comparison costs about 63 ms where the computation is still dominated by the 12 ms for decryption. However, the network traffic causes an overhead of about 49 ms. Via WAN, the time is dominated by the network round-trip time which is about 190 ms resulting in 755 ms for the oblivious comparison.

4.7.5 State's Generation and Storage Costs

The time to generate the OPE-tree also increases with the number of entries in the database and it is dominated by the time needed to encrypt the input data with Paillier's scheme. However, the above optimizations (i.e., choice of $g = 1 + N$ and pre-generated randomness) enable a fast generation of the OPE-tree. Figure 4.17a illustrates the generation time on the y-axis for OPE-tree with size between 100 and 1,000,000 on the x-axis. For 1 million entries, the generation costs on average only about 4.5 seconds.

The storage cost of the tree depends on $\log_2 N$, the bit length of the order, and the tree size. Since Paillier ciphertexts are twice longer than $\log_2 N$, each OPE ciphertext $\langle [\![x]\!], y \rangle$ needs $2 \cdot \log_2 N + \log_2 M$ bits storage. This is illustrated in Figure 4.17b, with the x-axis representing the tree size. The scheme needs 492.1 MB to store 1 million OPE ciphertexts, when the security parameter is 2048 and the order is 32-bit long.

4.7.6 Performance Comparison

Ishai et al. [108] evaluated their scheme on a machine with 8GB of RAM and 4 cores of an Intel i7-2600K 3.4GHz CPU. Our loopback experiment was done on a machine with 8 GB of RAM and 4 cores of an Intel(R) Xeon(R) CPU E7-4880 v2 at 2.50GHz. In their experiment, Ishai et al. [108] used a database with 10 million entries and performed ranges queries with result set size of 1000, 10000, 50000, 100000, 250000, 500000, 750000, and 1 million records. Then they performed the same experiment on plaintext data using MySQL and computed the overhead of their scheme. For result sets of 1000 and 1 million, their overhead against plaintext MySQL is ≈100 and ≈10 seconds, respectively. They attribute the reduced overhead to the fact their construction has additional fixed cost time that dominates small queries. As illustrated in Figure 4.2, our scheme adds to MySQL only the overhead for computing the OPE encodings of the query endpoints, which does not depend on the size of the result set. For a database with 10^7 entries, our overhead is only about 360 milliseconds, i.e., the time for one oblivious comparison (15 ms) multiplies by the depth of the OPE-tree ($\log_2(10^7) = 24$).

4.8 Summary

Since OPE schemes are limited to the use case to one server and one client, we introduced a novel notion of oblivious OPE (OOPE) as an equivalent of a public-key OPE. Then we presented a protocol for OOPE that combines deterministic OPE schemes based on binary tree search with Paillier's HE scheme and GC. We also applied our technique to the case where the underlying OPE scheme is probabilistic. Finally, we implemented our scheme with SCAPI and an optimized Paillier's scheme and showed that it achieves acceptable performance for interactive use.

Chapter 5

Classification with Sublinear Costs

Decision trees are common and very popular classifiers because they are explainable. A decision tree consists of two types of nodes. Internal nodes are *decision nodes* that are used to compare an attribute to a constant. *Leaf nodes* give a classification that applies to all instances that reach the leaf. To classify an unknown instance, the tree is traversed according to the values of the attributes tested in successive nodes, and when a leaf is reached the instance is classified according to the class assigned to that leaf [192]. In this chapter, we address the problem of evaluating a private decision tree on private data. The chapter is structured as follows. We start by defining the problem in Section 5.1 and then we introduce correctness and security definitions in Section 5.2. We describe the modular design of our main construction in Section 5.3. We discuss implementation of array indexing in Section 5.4 and optimizations in Section 5.5. We analyse correctness and security in Section 5.6 and present a complexity analysis in Section 5.7. We discuss evaluation results in Section 5.8 before summarizing the chapter in Section 5.9.

5.1 Problem Definition

We start in this section by describing the problem and the applications. Then we describe the generic solution, our own solution approach before concluding the section with a brief comparison to the related work.

5.1.1 Description

The problem consists of a server holding a private decision tree and a client holding a private attribute vector. The client wants to classify its private attribute vector using a server's private decision tree model. The goal is to obtain the classification, while preserving privacy of both the decision tree and the client input. After the computation, the result of the classification is revealed only to the client and nothing else is revealed, neither to the client nor the server. This problem is related to privacy-preserving machine learning which uses cryptographic primitives to build either classifiers on private data [104, 136, 137, 138] or to classify private data with a private model [14, 34, 38, 55, 96, 104, 151, 193]. Our work falls under the second category, namely privacy-preserving classification using decision trees.

5.1.2 Use Cases

Decision tree classifiers are a special type of machine learning classifier. Machine learning classifiers are valuable tools with applications in many areas. We surveyed machine learning in Section 2.2.1. We focus in this section on describing use cases.

As concrete motivation for the usefulness of privately evaluating a decision tree on private data, consider remote diagnostic services [38] or healthcare [14]. Many cloud providers are already proposing platforms that allow users to build machine learning applications [7, 24, 93, 150, 165]. A hospital may want to use such a platform to offer a medical expert system as an ML-as-a-service application to other doctors or even its patients. A software provider may leverage ML-as-a-service to allow its customers to detect the cause of a software error. Software systems use log files to collect information about the system behavior. In case of an error, these log files can be used to find the cause of the crash. Both examples (medical data and log files) contain sensitive information that must be protected.

5.1.3 Generic Solution

While generic secure multiparty computation [57, 90, 195] can implement a decision tree classifier, they are not efficient, in particular when the size of the tree is large. For example, frameworks such as ObliVM [144] or CBMC-GC [80] are able to transform plaintext programs into oblivious programs suitable for secure computation. Their straightforward application to decision tree programs does certainly improve performance over a hand-crafted construction. However, the size of the resulting oblivious program is proportional to the size of the tree. Decision programs can be seen as nested *if*-instructions; the ObliVM compiler transforms the exemplary program

$$\textsc{If}\ (s)\ \textsc{Then}\ x = 1;\ \textsc{Else}\ x = 2;$$

where the Boolean expression s involves secret variables, into

$$x_1 = 1;\ x_2 = 2;\ x = \textsc{Mux}(s, x_1, x_2),$$

where Mux is a multiplexer that returns either x_1 or x_2, depending on s being true or false. Any framework implementing the whole program will generate for each condition a corresponding oblivious computation. This results in an oblivious program whose size is linear in the number of decision nodes, which might be too large to be practical for some applications (e.g., Spam filtering). Therefore, optimized protocols, which exploit domain knowledge of the problem at hand and make use of generic techniques only where it is necessary, yield more efficient solutions [6, 14, 26, 34, 38]. Many proposed protocols for privately evaluating decision trees have a constant number of rounds at the cost of performing as many comparisons as there are decision nodes or transforming the whole plaintext decision tree into an oblivious program.

5.1.4 Our Solution Approach

The main idea of our novel solution is to represent the tree as an array. Then we execute only d comparisons, where d denotes the depth of the tree. The result of each comparison allows to obliviously select the index of the next node which is never revealed to any party in clear. This selection of the next node is computed using a small garbled circuit (GC), which is independent of the position in the tree and its size. This GC is handcrafted and executed using the ObliVM GC runtime. However, it is also possible to use other GCs runtime environments like SCAPI [75]. The framework ObliVM was chosen because of its implementation of the state-of-the-art ORAM [189]. We get the inputs to the comparison by obliviously indexing the tree and the attribute vector. We construct oblivious array indexing using either garbled circuits (GC), Oblivious Transfer (OT), or Oblivious RAM (ORAM).

Scheme	Rounds	Tools	Communication	Comparisons
[38]	≈5	HE+GC	$\mathcal{O}(M)$	d
[14]	≈4	HE+GC	$\mathcal{O}(m)$	d
[34]	≥6	FHE/SHE	$\mathcal{O}(m)$	m
[193]	6	HE+OT	$\mathcal{O}(m)$	m
[175]	4	HE	$\mathcal{O}(m)$	m
[55]	≈9	SS	$\mathcal{O}(M)$	m
Ours	$4d$	GC,OT	$\mathcal{O}(M)$	d
	d^2+3d	ORAM [189]	$\mathcal{O}(d^4)$	
	$4d$	ODS [191]	$\mathcal{O}(d^3)$	
	$4d$	FLORAM [72]	$\mathcal{O}(d^2)$	

Table 5.1: Summary of private decision tree evaluation protocols.

Using ORAM our protocol is the first that achieves sub-linear communication and computation cost in the size of the tree, and hence likely the first able to evaluate very large trees with thousands to millions of nodes, which will happen with the future growth of big data [40]. Currently, communication cost is an important asset in mobile settings. Remote clients using smartphones often do not have the bandwidth to run heavy protocols. However, ORAM has large constants hidden in its asymptotic complexity and a significant setup cost that needs to be amortized over many invocations of the protocol. Hence, we aim not only at improving asymptotic communication cost, but also at the practical cost on real-world data sets. Other alternative indexing protocols, particularly OT, have much smaller constants and help improve the practical communication cost for smaller trees. By using OT instead of ORAM, we reduce the cost for the large real-world data set "Spambase" in the UCI repository from 18 MB to 1.2 MB and the computation time from 17 seconds to less than 1 second in a LAN setting, compared to the best related work.

5.1.5 Comparison with Related Work

The related work to this chapter has been described in Section 3.5. We summarize the properties of private decision trees protocols in Table 5.1. Already from the table, we can conclude that our protocol has the best asymptotic communication complexity. Since the size of the tree is in the worst case exponential in the depth of the tree d, $M = \mathcal{O}(2^d)$, and $m = \mathcal{O}(2^d)$, we can expect our protocol to outperform previous work for large trees. However, we also aim to improve practical communication cost and computation time. Hence, we compare our implementation to the protocol of Wu et al. [193] on relevant, real-world data sets from the UCI repository. We chose Wu et al. because they perform an extensive comparison to the other protocols and have the best performance in the computational, two-party setting. The results will be discussed in the evaluation section.

5.2 Correctness and Security Definitions

In this section, we introduce relevant definitions and notations for our scheme. We use n to denote the number of attributes, m to denote the number of decision nodes, M to denote the number of nodes and d to denote the depth of the tree. Let $c_0, \ldots, c_{k-1}$ be the classification labels, $k \in \mathbb{N}_{>0}$.

5.2.1 The Model

Definition 5.2.1 (Decision Tree)**.** *A* decision tree *(DT) is a function*

$$\mathcal{T} : \mathbb{Z}^n \to \{c_0, \ldots, c_{k-1}\}$$

that maps an n-dimensional attribute vector $x = (x_0, \ldots, x_{n-1})$ *to a finite set of* classification labels*. The tree consists of:*

- *internal nodes (*decision nodes*) containing a test condition and*
- leaf nodes *containing a classification label.*

A decision tree model *consists of a decision tree and the following functions:*

- *a function* thr *that assigns to each decision node a* threshold *value,*

$$\mathsf{thr} : [0, m-1] \mapsto \mathbb{Z},$$

- *a function* att *that assigns to each decision node an* attribute index*,*

$$\mathsf{att} : [0, m-1] \mapsto [0, n-1], \text{ and}$$

- *a labeling function* lab *that assigns to each leaf node a label,*

$$\mathsf{lab} : [m, M-1] \mapsto \{c_0, \ldots, c_{k-1}\}.$$

The decision at each decision node is a "greater-than" comparison between the assigned threshold and attribute values, i.e., the decision at node v is $[x_{\mathsf{att}(v)} \geq \mathsf{thr}(v)]$.

Definition 5.2.2 (Node Indices)**.** *Given a decision tree, the* index *of a node is its order as computed by* breadth-first search (BFS) *traversal, starting at the root with index 0. If the tree is complete, then a node with index v has left child $2v+1$ and right child $2v+2$.*

We will also refer to the node with index v as the node v. W.l.o.g, we will use $[0, k-1]$ as classification labels (i.e., $c_j = j$ for $0 \leq j \leq k-1$) and we will label the first (second, third, ...) leaf in BFS traversal with classification label 0 (1, 2, ...). For a complete decision tree with depth d, the leaves have indices ranging from $2^d, 2^d+1, \ldots 2^{d+1}-2$ and classification labels ranging from $0, \ldots, 2^d-1$ respectively. Since the classification labeling is now independent of the tree, we use $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$ to denote a *decision tree model* consisting of a tree $\mathcal{T}$ and the labeling functions $\mathsf{thr}, \mathsf{att}$ as defined above. We also assume that the tree parameters d, m, M can be derived from $\mathcal{T}$.

Definition 5.2.3 (Decision Tree Evaluation)**.** *Given $x = (x_0, \ldots, x_{n-1})$ and $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$, then starting at the root, the* Decision Tree Evaluation *(DTE) evaluates at each reached node v the decision $b \leftarrow [x_{\mathsf{att}(v)} \geq \mathsf{thr}(v)]$ and moves either to the left*

(if $b = 0$) or right (if $b = 1$) subsequent node. The evaluation returns the label of the reached leaf as result of the computation. We denote this by $\mathcal{T}(x)$.

Definition 5.2.4 (Private DTE)**.** *Given a* client *with a* private $x = (x_0, \ldots, x_{n-1})$ *and a* server *with a* private $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$*, a* private DTE (PDTE) *functionality evaluates the model $\mathcal{M}$ on input x, then reveals to the client the* classification label $\mathcal{T}(x)$ *and nothing else, while the server learns* nothing*, i.e.,*

$$\mathcal{F}_{\mathsf{PDTE}}(\mathcal{M}, x) \rightarrow (\varepsilon, \mathcal{T}(x)).$$

Definition 5.2.5 (Correctness)**.** *Given a* client *with a* private $x = (x_0, \ldots, x_{n-1})$ *and a* server *with a* private $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$*, a protocol* Π correctly implements *a PDTE functionality if after the computation it holds for the result c obtained by the client that $c = \mathcal{T}(x)$.*

Recall that, two distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ are *computationally indistinguishable* (denoted $\mathcal{D}_1 \overset{c}{\equiv} \mathcal{D}_1$) if no probabilistic polynomial time (PPT) algorithm can distinguish them except with negligible probability.

In SMC protocols the *view* of a party consists of its input and the sequence of messages that it has received during the protocol execution [90]. The protocol is said to be secure if, for each party, one can construct a simulator that, given only the input of that party and the output, can generate a distribution that is computationally indistinguishable to the party's view.

Definition 5.2.6 (PDTE Semi-Honest Security)**.** *Given a* client *with a* private $x = (x_0, \ldots, x_{n-1})$ *and a* server *with a* private $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$*, a protocol* Π_{PDTE} securely implements *the PDTE functionality in the* semi-honest model *if the following conditions hold:*

- *there exists a PPT algorithm* $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{pdte}}$ *that simulates the server's view* $\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{PDTE}}}$ *given only the private decision tree model* $(\mathcal{T}, \mathsf{thr}, \mathsf{att})$ *such that:*

$$\mathsf{Sim}_{\mathsf{S}}^{\mathsf{pdte}}(\mathcal{M}, \varepsilon) \quad \overset{c}{\equiv} \quad \mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{PDTE}}}(\mathcal{M}, x), \tag{5.1}$$

- *there exists a PPT algorithm* $\mathsf{Sim}_{\mathsf{C}}^{\mathsf{pdte}}$ *that simulates the client's view* $\mathsf{View}_{\mathsf{C}}^{\Pi_{\mathsf{PDTE}}}$ *given only the depth d of the tree, $x = (x_0, \ldots, x_{n-1})$ and a classification label $\mathcal{T}(x) \in \{0, \ldots, k-1\}$ such that:*

$$\mathsf{Sim}_{\mathsf{C}}^{\mathsf{pdte}}((d, x), \mathcal{T}(x)) \quad \overset{c}{\equiv} \quad \mathsf{View}_{\mathsf{C}}^{\Pi_{\mathsf{PDTE}}}(\mathcal{M}, x). \tag{5.2}$$

5.2.2 Oblivious Array Indexing

Before moving to our main construction, we describe a primitive called *oblivious array indexing (OAI)*, which will serve as a sub-protocol. As illustrated in Figure 5.1, *secret array indexing (SAI)* allows a receiver C holding a secret index i to privately access the i-th element of an array held by a sender S. We will use array indexing as an intermediate step, and therefore execute it obliviously such that the index and the indexed element are secret-shared to the parties. We will refer to it as *oblivious array indexing*. This is illustrated in Figure 5.2.

Definition 5.2.7 (Oblivious Array Indexing)**.** *Let $\boldsymbol{A} = [a_0, \ldots, a_{n-1}]$ be an array and $i \in [0, n-1]$ be an index. An* Oblivious Array Indexing (OAI) *functionality consists of:*

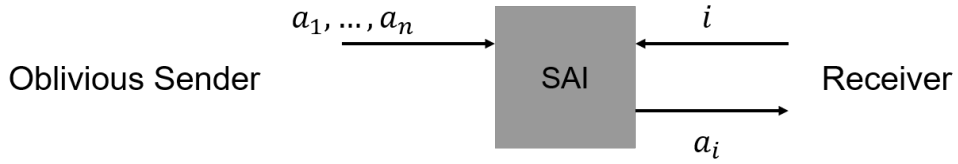


Figure 5.1: Secret Array Indexing

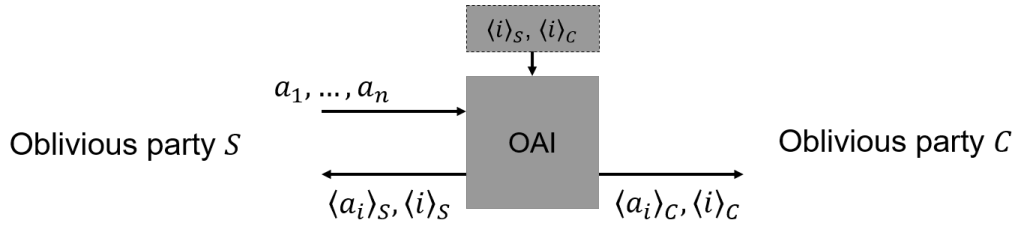


Figure 5.2: Oblivious Array Indexing

- *a party S holding privately $\boldsymbol{A}$,*
- *a party C.*

The functionality receives the array $\boldsymbol{A}$ and some (secret) program state containing shares of an index i. Then it computes the secret index i, selects the i-th element a_i, computes two shares $\langle a_i \rangle_S$ and $\langle a_i \rangle_C$ of A_i, and two shares $\langle i \rangle_S$ and $\langle i \rangle_C$ of i. It finally returns $\langle a_i, i \rangle_S = (\langle a_i \rangle_S, \langle i \rangle_S)$ to party S and $\langle a_i, i \rangle_C) = (\langle a_i \rangle_C, \langle i \rangle_C)$ to party C, i.e,

$$\mathcal{F}_{\mathsf{OAI}}(\boldsymbol{A}, \varepsilon) \rightarrow (\langle a_i, i \rangle_S, \langle a_i, i \rangle_C),$$

such that if $i = \langle i \rangle_S \odot \langle i \rangle_C$ then $a_i = \langle a_i \rangle_S \odot \langle a_i \rangle_C$, where $\odot \in \{\oplus, +\}$.

Definition 5.2.8 (OAI Semi-Honest Security)**.** *Given a party S with input $\boldsymbol{A} = [a_0, \ldots, a_{n-1}]$ and a party C, a protocol Π_{OAI}* securely implements *the OAI functionality in the* semi-honest model *if the following conditions hold:*

- *there exists a PPT algorithm $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{oai}}$ that simulates party S's view $\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{OAI}}}$ given only the array $\boldsymbol{A} = [a_0, \ldots, a_{n-1}]$ such that:*

$$\mathsf{Sim}_{\mathsf{S}}^{\mathsf{oai}}(\boldsymbol{A}, \langle a_i, i \rangle_S) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{OAI}}}(\boldsymbol{A}, \varepsilon), \tag{5.3}$$

- *there exists a PPT algorithm $\mathsf{Sim}_{\mathsf{C}}^{\mathsf{oai}}$ that simulates party C's view $\mathsf{View}_{\mathsf{C}}^{\Pi_{\mathsf{OAI}}}$ given only the index share $\langle i \rangle_C$ such that:*

$$\mathsf{Sim}_{\mathsf{C}}^{\mathsf{oai}}(\varepsilon, \langle a_i, i \rangle_C) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{C}}^{\Pi_{\mathsf{OAI}}}(\boldsymbol{A}, \varepsilon). \tag{5.4}$$

5.3 Main Construction

In this section, we describe the modular design of our scheme which consists of four sub-protocols. We first describe the relevant data structure and an overview of the main construction. Then we present each sub-protocol.

5.3.1 Intuition

At first, the server transforms the decision tree into an array which is formalized in the following definition.

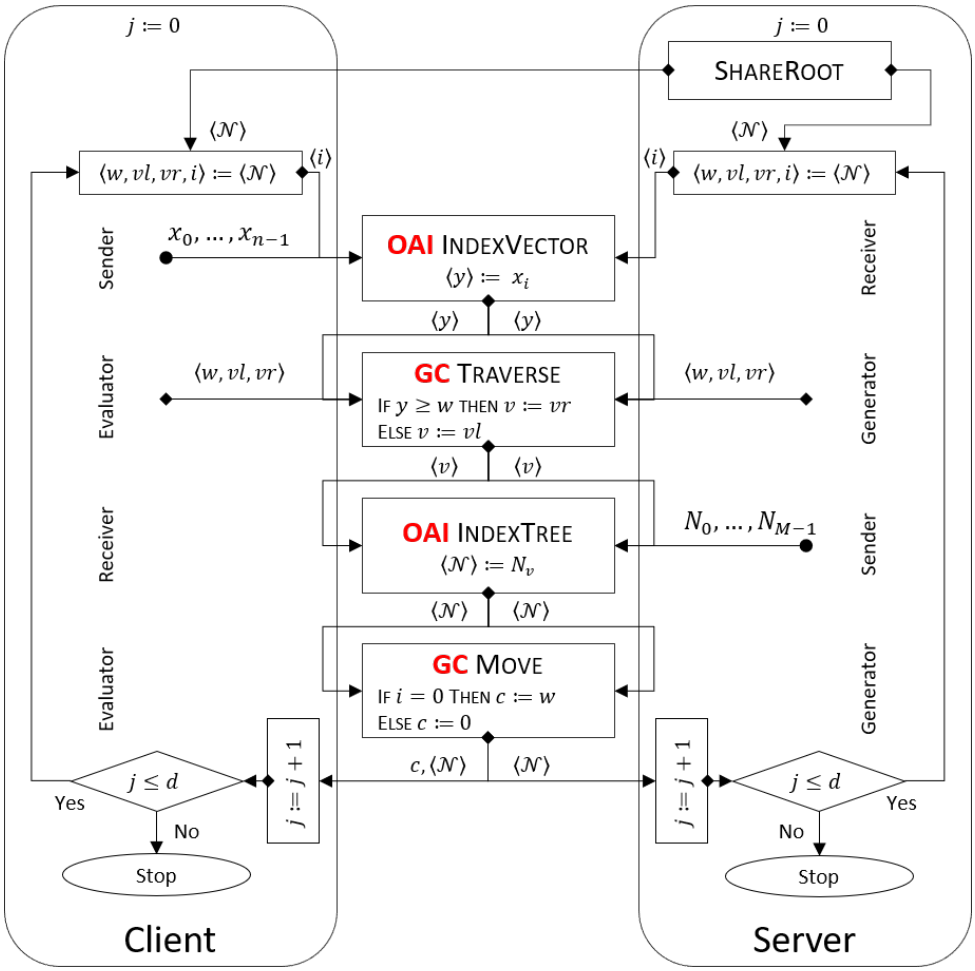


Figure 5.3: Illustration of the Protocol: The attribute vector x is indexed with i (i.e., $y = x_i$). The array of tree nodes N is indexed with v (i.e., $\mathcal{N} = N_v$, N_0 is the root node).

Definition 5.3.1 (Data Structure)**.** *Let* $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$ *with* M *and* m *as above. A* node data structure (DS) *of a decision node* $v \in [0, m-1]$ *consists of the tuple* $(\mathsf{thr}(v), \mathsf{lnod}(v), \mathsf{rnod}(v), \mathsf{att}(v))$*, where* $\mathsf{lnod}(v)$ *and* $\mathsf{rnod}(v)$ *are functions that return the indices of the left and right child node of* v *in BFS traversal. A* node DS *of a leaf node* $v \in [m, M-1]$ *consists of the tuple* $(\mathsf{lab}(v), \textsc{Null}, \textsc{Null}, \textsc{Null})$*. A* tree DS *consists of the array* $\boldsymbol{N} = N_0, \ldots, N_M$ *such that* N_v *is the node DS of* $v \in [0, M-1]$*.*

Hence, besides a threshold and an index to x, each node stores the index to its two child nodes, similar to the pointer-based technique of [191]. The tree DS $N_0, \ldots, N_M$ is equivalent to $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$ and will be used instead.

The server secret-shares the root node with the client and the protocol loops d times. In each iteration, the parties select the attribute value that corresponds to the current node (initially the root). Then, they execute a GC to compute an index which they use to obliviously select one child node of the current node. Finally, they execute another GC to check if the selected child node is a leaf. Otherwise, this node is evaluated in the next iteration. The protocol is illustrated in Figure 5.3, where each OAI block can be instantiated with any protocol that allows secret indexing, such as GC, OT, ORAM. Each GC block can be implemented with any generic secure 2-party computation approach.

5.3.2 Algorithms

Notations. In our algorithms, we use v to denote the index of a node, i.e., T_v. Moreover, we use $\mathcal{N} = (w, vl, vr, i)$ to denote the content of nodes. If it is a decision node then $w = \mathsf{thr}(v), vl = \mathsf{lnod}(v), vr = \mathsf{rnod}(i)$ and $i = \mathsf{att}(v)$. Otherwise $w = \mathsf{lab}(v)$ and $vl = vr = i = \textsc{Null}$. For a variable V that is secret-shared between server and client, let $\langle V \rangle_S$ and $\langle V \rangle_C$ denote the respective shares, i.e., $\langle \mathcal{N} \rangle_S = \langle w, vl, vr, i \rangle_S$ denotes the server's share of $\mathcal{N}$. In our GCs, we use the following basic operations:

- $\textsc{And}(b_1, b_2) = (b_1 \wedge b_2)$,
- $\textsc{Eq}(a_1, a_2) = (\textsc{If}\ a_1 = a_2\ \textsc{Then}\ 1\ \textsc{Else}\ 0)$,
- $\textsc{Geq}(a_1, a_2) = (\textsc{If}\ a_1 \geq a_2\ \textsc{Then}\ 1\ \textsc{Else}\ 0)$,
- $\textsc{Mux}(b, a_1, a_2) = (\textsc{If}\ b\ \textsc{Then}\ a_2\ \textsc{Else}\ a_1)$,
- $\textsc{Not}(b) = (\neg b)$, and
- $\textsc{Xor}(a_1, a_2) = (a_1 \oplus a_2)$.

Additionally, our algorithms use a combination of GC and simple secret sharing. We use the following operations to denote re-sharing of a value in a GC:

- $\textsc{Reshare}(r, a) = (r, a \odot r)$, where $\odot \in \{\oplus, +\}$, and
- $\textsc{Reshare}(\vec{r}, \vec{a}) = (\vec{r}, \vec{a} \odot \vec{r})$, where $\vec{a}, \vec{r}$ are vectors and $\odot$ is applied component-wise.

The value to be re-shared (a or $\vec{a}$) is an intermediate result (e.g., the index of the subsequent node), which should not be revealed in clear to the parties. The value r or $\vec{r}$ is randomly chosen by the GC generator and used to blind an intermediate result.

The Evaluation Algorithm. Protocol 5.4 shows our PDTE scheme. It always starts at the root, i.e., the first element of the tree DS. Let the initial current node $\mathcal{N} = (w, vl, vr, i)$ be the node DS of the root. With the procedure ShareRoot, the server first shares the root node by choosing a random numbers r_w, r_{vl}, r_{vr}, r_i, setting $\langle \mathcal{N} \rangle_S = \langle w, vl, vr, i \rangle_S = (r_w, r_{vl}, r_{vr}, r_i)$ and sending $\langle \mathcal{N} \rangle_C = \mathcal{N} \oplus \langle \mathcal{N} \rangle_S = (w \oplus r_w, vl \oplus r_{vl}, vr \oplus r_{vr}, i \oplus r_i)$ to the client. The values $\langle \mathcal{N} \rangle_S, \langle \mathcal{N} \rangle_C$ are used as shares of the current node in the first iteration of the following loop. In each iteration the parties successively perform the following operation:

- Select the corresponding attribute value (Step 5): Using the procedure IndexVector, the parties then execute an oblivious array indexing (see Section 5.4) to select and secret-share an array element. The server's input is its share of i and the client's input consists of its share of i and the attribute vector x. The result of Step 5 consists of two shares of the attribute value corresponding to the current decision node.
- Compute the index of the next node (Step 6): The parties execute a GC which evaluates the current node (i.e., compares the current attribute value with the current node's threshold) and uses the comparison result to select the index of the next node in the tree. The resulted index is secret-shared to the parties. This step is fully specified in Algorithm 5.5.

Input : $(\boldsymbol{N}, x)$
Output : (ε, c)

1: $(\langle \mathcal{N} \rangle_S, \langle \mathcal{N} \rangle_C) \leftarrow$ ShareRoot$(root, \varepsilon)$
2: $j \leftarrow 1$
3: **while** $j \leq d$ **do**
4: parse $\langle \mathcal{N} \rangle_p$ to $\langle w, vl, vr, i \rangle_p$ ▷ $p = S, C$
5: $(\langle x_i \rangle_S, \langle x_i \rangle_C) \leftarrow$ IndexVector$(\langle i \rangle_S, (x, \langle i \rangle_C))$
6: $(\langle v \rangle_S, \langle v \rangle_C) \leftarrow$ Traverse$((\langle w, vl, vr, x_i \rangle_S, r_S), \langle w, vl, vr, x_i \rangle_C)$
7: $(\langle \mathcal{N} \rangle_S, \langle \mathcal{N} \rangle_C) \leftarrow$ IndexTree$((\boldsymbol{N}, \langle v \rangle_S), \langle v \rangle_C)$
8: parse $\langle \mathcal{N} \rangle_p$ to $\langle w, vl, vr, i \rangle_p)$ ▷ $p = S, C$
9: $((0, \langle i \rangle_S), (c, \langle i \rangle_C)) \leftarrow$Move$(\langle w, i \rangle_S, (\langle w, i \rangle_C, r_C)))$
10: $j \leftarrow j + 1$
11: **return** (ε, c)

Protocol 5.4: Our Private Decision Tree Evaluation Protocol

- Select the next node in the tree (Step 7): The parties use their secret shares from the previous step as input to the procedure IndexTree to select and secret-share the next node in the tree. The additional input to the array indexing operation is the array. The result is used to update the current node.
- Move (Step 9): Finally, another GC, which is fully specified in Algorithm 5.6, checks if the current node $\mathcal{N}$ is a leaf and computes the classification label and re-shares the index i. Note that, as we assumed a complete tree, it is enough to perform this check only for the last iteration. In this case, the Move algorithm only needs to re-share the index i for all iterations but the last one.

The algorithm loops d times and $\mathcal{N}$ is evaluated in the next iteration. Steps 4, 8, and 10 are local steps and do not require any secure computation.

We now turn to the description of the algorithms Traverse and Move, which are implemented using Yao's garbled circuits approach.

The Traverse Algorithm. Algorithm 5.5 describes the GC that is used in Protocol 5.4 (Step 6) to compute the index of the next node. The GC is generated by the server and evaluated by the client. Each party inputs its share of (w, vl, vr, y) which is then recovered (Step 1) in the execution of the GC. Then the result of the comparison $y \geq w$ (Step 2) is used to select (Step 3) the index of the next node between vl and vr. The selection is implemented using an l–bit multiplexer Mux(b, a_1, a_2). Finally, this index is secret-shared (Step 4) and returned to the parties. Note that, the output $\langle v \rangle_S$ is identical to the random string r_S that is also part of the server's input, such that in Step 4, $\langle v \rangle_C \leftarrow v \odot r_S$ is computed and returned only to the client.

The Move Algorithm. At the end of each iteration in Protocol 5.4, the GC of Algorithm 5.6 receives shares of (w, i) and recombines them (Step 1) in the secure computation. Then it checks (Step 2) if i equals Null and returns either 0 or w. Again, as we assumed a complete tree, it is enough to perform this check only for the last iteration. The selection (Step 3) is also implemented as an l–bit multiplexer. The final step re-shares the index i to both parties. For this GC as well, the server and the client act as generator and evaluator respectively.

Input : $((\langle w, vl, vr, x_i\rangle_S, r_S), \langle w, vl, vr, x_i\rangle_C)$
Output : $(\langle v\rangle_S, \langle v\rangle_C)$

1: $(w, vl, vr, x_i) \leftarrow \textsc{Xor}(\langle w, vl, vr, x_i\rangle_S, \langle w, vl, vr, x_i\rangle_C)$
2: $b \leftarrow \textsc{Geq}(x_i, w)$
3: $v \leftarrow \textsc{Mux}(b, vl, vr)$
4: $(\langle v\rangle_S, \langle v\rangle_C) \leftarrow \textsc{Reshare}(r_S, v)$

Algorithm 5.5: Traverse Algorithm

Input : $((\langle w, i\rangle_S), (\langle w, i\rangle_C, r_C))$
Output : $((\textsc{Null}, \langle i\rangle_S), (R, \langle i\rangle_C))$

1: $(w, i) \leftarrow \textsc{Xor}(\langle w, i\rangle_S, \langle w, i\rangle_C)$
2: $b \leftarrow \textsc{Eq}(i, \textsc{Null})$
3: $R \leftarrow \textsc{Mux}(b, 0, w)$
4: $(\langle i\rangle_C, \langle i\rangle_S) \leftarrow \textsc{Reshare}(r_C, i)$

Protocol 5.6: Move Algorithm

Brief Analysis. We stress that Algorithms 5.5 and 5.6 represent the full specification of the underlying GCs that are neither dependent on the position in the tree nor on its size. It is straightforward to see that the resulting GCs are indeed small. Their cost depends on the operations Eq, Geq, Mux, and Reshare, as the performance metric for GC is the number of AND-gates (XOR being free). Assuming inputs are μ-bit integers, operations Eq, Geq, Mux contain each exactly μ AND-gates [130, 131], which results in 4μ ciphertexts each. The Reshare operation contains AND-gates only when the sharing is additive. In this case, re-sharing is implemented as full-adder with μ AND-gates [130]. The *halfGate* optimization [197] reduces the number of ciphertexts per AND-gate by a factor of 2 at the cost for the evaluator to perform two cheap symmetric operations, rather than one. Also, notice that the size of Eq and Geq depends on the size of the attribute vector n and the tree M respectively, whose bit-length can be much more smaller than the input's bit-length in practice (i.e., $\log(n)$ and $\log(M)$ are smaller than μ). In the worst case, the communication cost of our scheme (Protocol 5.4) is dominated by the cost of the array indexing on the tree (in general the tree is larger than the attribute vector) which is $\mathcal{O}(M), \mathcal{O}(M), \mathcal{O}(d^3), \mathcal{O}(d^2), \mathcal{O}(d)$ for GC, OT, Circuit ORAM, ODS and FLORAM respectively.

5.4 Implementing Oblivious Array Indexing (OAI)

OAI can be instantiated with any protocol that allows secret indexing, such as GC, OT, ORAM. Notice that, OAI only makes sense when used as a sub-protocol. In the overall protocol the array indexing is always preceded by a GC step that computes and returns the shares $\langle i\rangle_S$ and $\langle i\rangle_C$ to server and client (i.e., in Step 4 of Traverse and in Step 4 of Move). To initially index the attribute vector the shares are computed by the server in Step 1 of Protocol 5.4. This is not a security problem as the classification always starts at the root.

Input : $((a_0, \ldots, a_{n-1}, \langle i \rangle_S, r_S), \langle i \rangle_C)$
Output : $(\langle a_i \rangle_S, \langle a_i \rangle_C)$ s.t. if $i = \langle i \rangle_S \oplus \langle i \rangle_C$ then $a_i = \langle a_i \rangle_S \oplus \langle a_i \rangle_C$

1: $i \leftarrow$XOR$(\langle i \rangle_S, \langle i \rangle_C)$
2: $a_i \leftarrow 0$
3: **for** $j = 0$ **to** $n - 1$ **do**
4: $\quad b \leftarrow$EQ(i, j)
5: $\quad a_i \leftarrow$MUX(b, a_i, a_j)
6: $(\langle a_i \rangle_S, \langle a_i \rangle_C) \leftarrow$RESHARE$(r_S, a_i)$

Algorithm 5.7: OAI with Garbled Circuit

Input : $((A = [a_0, \ldots, a_{n-1}], r, s), k = i + r \bmod n)$
Output : $(\langle a_i \rangle_S, \langle a_i \rangle_C)$ s.t. $a_i = \langle a_i \rangle_S \oplus \langle a_i \rangle_C$

1: For all $0 \le j < n$, S sets $a'_{j+r \bmod n} \leftarrow a_j \oplus s$
2: S and C execute $(\varepsilon, a'_k) \leftarrow \mathrm{OT}^1_n(A' = [a'_0, \ldots, a'_{n-1}], k)$
3: $\langle a_i \rangle_S \leftarrow s$ and $\langle a_i \rangle_C \leftarrow a'_k = a'_{i+r \bmod n} = a_i \oplus s$

Protocol 5.8: OAI with Oblivious Transfer

As already mentioned, we assume the array elements and the indices to be μ-bit integers. In this section, we refer to party S and party C as *Sender* and *Receiver* respectively.

5.4.1 OAI with Garbled Circuits

The algorithm for OAI using GC is described in Algorithm 5.7. The circuit uses three sub-circuits: XOR, EQ, MUX. The input of the sender also contains a random string r_S that is used in line 6 to randomize the output. The algorithm scans the array and uses MUX to select the indexed element. Since the sender is also the generator of the GC, the evaluator does not have to send back the result of the evaluation. Using the same metric explained in Section 5.3.2, the GC of Algorithm 5.7 contains $\mu(n + 1)$ AND-gates. The communication cost is clearly linear in the array size, i.e., $\mathcal{O}(n)$.

Alternatively, we can also represent the index i as a vector of bits with 0 everywhere except at position i. This results in a more efficient GC by getting rid of the equality check in line 4, which requires μ AND-gates. However, since we use μ bits to represent the index, we run this more efficient alternative only when the size of the array is smaller or equal to μ.

5.4.2 OAI with Oblivious Transfer

For the oblivious indexing with OT, we use the OT^1_n protocol by Naor and Pinkas [154], which is described in Protocol 2.2. The idea of OAI with OT is to share the array index i with *additive sharing* by choosing a random r as sender's share and $i + r$ as receiver's share. Then the sender rotates the array by r and the parties execute OT^1_n on the new array and $i + r$. Hence, let $r \in \{0, 1\}^l$ be a random l–bit integer and $\langle i \rangle_S = r$ and $\langle i \rangle_C = i + r$. The protocol is described in Protocol 5.8.

In [111], Jarrous and Pinkas used a similar idea in a protocol called HDOT (Hamming Distance based OT) in which parties C and S have private binary strings

$\alpha = \alpha_0 \dots \alpha_t$ and $\beta = \beta_0 \dots \beta_t$ of length $t = \log(n)$. The sender S additionally has a private dataset $\boldsymbol{A} = [a_0, \dots, a_{n-1}]$. The parties then run a protocol that privately computes the Hamming distance $i = d_H$ and uses it to reveal a_i to C using OT_n^1. To select a_i without revealing d_H to the parties, C first computes $[\![\alpha]\!] = ([\![\alpha_0]\!], \dots, [\![\alpha_t]\!])$ under its public key using an additively HE. Then S receives $[\![\alpha]\!]$ and computes $[\![i]\!] = \prod [\![\alpha_j \oplus \beta_j]\!] = [\![\sum \alpha_j \oplus \beta_j]\!]$. Finally, S chooses a random r and sends $[\![i + r]\!]$ to C.

Alternatively, we can use the following proposition to share the index with *XOR (exclusive-OR) sharing* instead. The benefit is that the re-sharing of the index in the GC (e.g., in Step 4 of Algorithm 5.5) is more efficient as it requires only exclusive-OR operations. In our implementation, we use this alternative, whenever it is possible.

Proposition 5.4.1. *Let $B = \{0,1\}^l$ and $x \in B$ then we have $x \oplus B = B$.*

Proof. $\forall y \in B, x \oplus y$ is clearly in B, since B contains alls l–bit strings. Moreover, $\forall y_1, y_2 \in B$ with $y_1 \neq y_2$ it holds $x \oplus y_1 \neq x \oplus y_2$, because otherwise $x \oplus y_1 = x \oplus y_2$ implies $y_1 = y_2$. □

From the above proposition, it follows that if the length n of the array is a power of 2 with $\log(n) = h$, then $[0, n-1]$ is isomorphic to $\{0,1\}^h$. As a result, we can modify the algorithm in Protocol 5.8 as follows: we set $\langle i \rangle_S = r$ and $\langle i \rangle_C = i \oplus r$, and replace Step 1 by $a'_{j \oplus r} \leftarrow a_j \oplus s$, $0 \le i < n$. If n is not a power of 2, one may consider padding the array with 0s to an array of length $n' = 2^{h+1}$. However, since the OT_1^n requires sending n ciphertexts (resulting in $\mathcal{O}(n)$ communication cost) we will use the alternative protocol only if n is a power of 2.

Indexing can be done with PIR as well, similar to the OT case as described in Protocol 5.8, where the OT_1^n is replaced by a receiver-private and sender-private PIR scheme. Using Lipmaa's PIR scheme [141] that has log-squared communication complexity results in a sublinear communication complexity. However, it relies on public key primitive and requires high computational cost.

5.4.3 OAI with Oblivious RAM

For an array of size N, tree-based ORAM organizes the data into blocks stored in a the ORAM state State, which is a binary tree of height $\log(N)$, at the server. Each node of the tree is a bucket of $\log(N)$ blocks. Each block has the form $\{\mathsf{idx} \| \mathsf{label} \| \mathsf{data}\}$, with idx a block index, label a leaf identifier specifying the path on which the block resides and data the actual data. The client stores a stash Stash for buffering overflowing blocks and a position map PosMap mapping idx to label. The position map can be reduced to $\mathcal{O}(1)$ by recursively storing it in smaller ORAMs at the server. There are two basic operations. The first one, ReadAndRemove, reads and removes a block from its current position and the second one, Evict, randomly pushes blocks down to their path. They are used to implement the ACCESS operation (Algorithm 5.9) which allows two functionalities [171, 189]:

- Oram.Read(idx) := Oram.Access(idx, ε): read block idx from the State
- Oram.Write(idx, data) := Oram.Access(idx, data): write block idx with data.

Similarly to ObliVM [144], the GC framework used in our evaluation, we use read and write interface for convenience. OAI with ORAM requires only Read(). In an initial step, the server places the array in the ORAM state State, and secret-shares the resulting State with the client. OAI with ORAM (Algorithm 5.10) is similar to the case of GC, with the only difference that the array is stored in the shared ORAM state

Input: (idx, data)
Output: out

1: {idx‖lbl‖out} ← ReadAndRemove(idx, PosMap[idx])
2: PosMap[idx] ←$ {0, . . . , N − 1} ▷ Uniformly random choice
3: **if** data = ε **then**
4: data ← out
5: Stash.Add({idx‖PosMap[idx]‖data})
6: Evict()
7: **return** out

Algorithm 5.9: Algorithm Oram.Access [189]

Input: $((\langle \mathsf{State}, i\rangle_S, r_S), \langle \mathsf{State}, i\rangle_C)$
Output: $(\langle a_i\rangle_S, \langle a_i\rangle_C)$ s.t. if $i = \langle i\rangle_S \oplus \langle i\rangle_C$ then $a_i = \langle a_i\rangle_S \oplus \langle a_i\rangle_C$

1: $i \leftarrow \textsc{Xor}(\langle i\rangle_S, \langle i\rangle_C)$
2: $a_i \leftarrow$ Oram.Read(i)
3: $\langle a_i\rangle_C \leftarrow \textsc{Xor}(a_i, r_S)$
4: $\langle a\rangle_S \leftarrow r_S$

Algorithm 5.10: OAI with ORAM

and the computation of the indexed element is replaced by Oram.Read(idx). In fact, the read and write operations are implemented using GCs. In our implementation, we used the recursive circuit ORAM [189]. However, OAI with ORAM can be optimized by using oblivious data structures [118, 191] or FLORAM [72] instead. Using ODS is straightforward since ODS schemes are basically optimized tree-based ORAM like circuit ORAM. OAI with FLORAM requires to implement the PDTE as a 3-party protocol (one client and two servers). Because of our 2-party security model, we see the integration of FLORAM to our scheme as a future work.

5.5 Optimizations

We now discuss three optimizations of our scheme. The first one consists of indexing the tree by level. The second optimization performs a pre-processing which allows avoiding the indexing of the attribute vector in the online computation. Finally, we show how to handle sparse trees efficiently.

5.5.1 Level Indexing

In each iteration, our protocol executes array indexing on the tree to select the index of the next node. Since this node is always a child node of the current node, we do not have to use the whole tree during array indexing. It is sufficient to use only the nodes of the next level. Therefore, before invoking IndexTree (Step 7 of Protocol 5.4) at level d', we construct an array containing only the nodes of level $d' + 1$, where for each node vl and vr are recomputed according to the number of nodes at level $d' + 2$. For ORAM, each level of the tree is stored in its own ORAM during initialization. Of course, this reveals the number of nodes per level to the client. However, if we assume that the tree is complete, then it is not a new leakage. Otherwise, we can still extend

all levels to the size of the longest level, which is smaller than 2^d, leaking only this longest size. To be secure for sparse trees, this optimization has to be combined with the *Handling Sparse Trees* optimization below. We note that for efficiency reasons, previous schemes also leak either the number of decision nodes, the number of nodes, the number of paths, or the depth of the tree to the client.

5.5.2 Pre-processing the Vector Indexing

One can also avoid indexing the attribute vector x in each step. Let $x = (x_0, \ldots, x_{n-1})$, then in an initial step, the client computes $[\![x]\!] = ([\![x_0]\!], \ldots, [\![x_{n-1}]\!])$ under its public key using additively HE. The server receives $[\![x]\!]$, chooses n random numbers $r_0, \ldots, r_{n-1} \in \{0, \ldots, 2^{\mu+\sigma}\}$ (σ is a security parameter that determines the statistical leakage [70], e.g., $\sigma = 32$) and computes $([\![x_0 + r_0]\!], \ldots, [\![x_{n-1} + r_{n-1}]\!])$. Then the server chooses a random permutation π and sends back $\pi([\![x_0 + r_0]\!], \ldots, [\![x_{n-1} + r_{n-1}]\!])$ to the client. Finally, the server replaces each decision node $\mathcal{N} = (w, vl, vr, i)$ of the tree by $\mathcal{N} = (w + r_{\pi(i)}, vl, vr, \pi(i))$. Note that, we can reduce the number of ciphertexts sent, by packing many plaintexts in one ciphertext [166]. During the evaluation, the client learns $\pi(i)$ in each iteration, selects the attribute value $x_{\pi(i)} + r_{\pi(i)}$ locally and uses it in the TRAVERSE algorithm which evaluates $\text{GEQ}(x_{\pi(i)} + r_{\pi(i)}, w + r_{\pi(i)})$ instead of $\text{GEQ}(x_i, w)$. Hence, with this optimization the server's share $\langle x_i \rangle_S$ in the TRAVERSE algorithm is empty, while $\langle x_i \rangle_C = x_{\pi(i)} + r_{\pi(i)}$. To preserve privacy, this pre-processing step must be recomputed for each tree evaluation. We have not yet implemented and evaluated this optimization and intend to do it in future work.

5.5.3 Handling Sparse Trees

We assumed in our protocol that the tree is complete. However, this is inefficient for sparse trees. We, therefore, optimize our protocol to handle sparse trees efficiently. This optimization requires only small changes to Algorithm 5.5 and 5.6.

Let M_j denotes the number of nodes at level j. Then, for a sparse tree M_j is much more smaller than 2^j. However, a path may end at a level $j < d$. The idea is to store each level j in an array of size at least M_j, instead of 2^j. When we are traversing a path that ends at level $j < d$, we secret-share the corresponding classification label to the parties in iteration $j - 1$. Then we simulate the remaining $d - j$ iterations and refresh the shares of the classification label each time.

We, therefore, use two additional variables u and e to assign the classification label and a bit respectively. During the tree evaluation, if we reached a leaf at level $j < d$, we assign the corresponding classification label to u and set e to 1. Then, we re-share both variables to the parties in each iteration.

For Algorithm 5.5, parties additionally receive shares of variables u and e. Then, after the comparison (Step 2), we check if $e = 1$ (i.e., a leaf node was reached at level $j < d$) and choose v' randomly. This check is however only necessary for levels $j \geq 1$ (i.e., not at the root).

The modification for Algorithm 5.6 is described in Algorithm 5.11, where FSTCALL is used only for the first iteration (i.e., at the root node) and ITHCALL for all other iterations. The inputs are shares of w (either a threshold or a classification label), i, u, and e for each party. The server has a random vector $\vec{r}_S = (r_u, r_e)$ as additional input, that is used to re-share the values u, e. The client has a random r_C as additional input, that is used to re-share the index i. For the first iteration, FSTCALL checks if the next node is a leaf (Step 3), sets u and e appropriately (Step 4), and re-shares i, u, e (Step 5 and 6). After the first iteration, ITHCALL checks if e is still 0 and if the

Input : $((\langle w, i, u, e\rangle_S, \vec{r}_S), (\langle w, i, u, e\rangle_C, r_C))$
Output : $(\langle i, u, e\rangle_S, \langle i, u, e\rangle_C))$ ▷ fresh shares

1: **function** FSTCALL
2: $(w, i) \leftarrow \text{XOR}(\langle w, i\rangle_S, \langle w, i\rangle_C)$
3: $b \leftarrow \text{EQ}(i, \text{NULL})$
4: $(u, e) \leftarrow \text{MUX}(b, (\text{NULL}, 0), (w, 1))$
5: $(\langle i\rangle_C, \langle i\rangle_S) \leftarrow \text{RESHARE}(r_C, i)$
6: $(\langle u, e\rangle_S, \langle u, e\rangle_C) \leftarrow \text{RESHARE}(\vec{r}_S, (u, e))$
7: **function** ITHCALL
8: $(w, i, u, e) \leftarrow \text{XOR}(\langle w, i, u, e\rangle_S, \langle w, i, u, e\rangle_C)$
9: $b_1 \leftarrow \text{EQ}(e, 1)$, $b_2 \leftarrow \text{EQ}(i, \text{NULL})$
10: $b \leftarrow \text{AND}(\text{NOT}(b_1), b_2)$
11: $(u, e) \leftarrow \text{MUX}(b, (u, e), (w, 1))$
12: $(\langle i\rangle_C, \langle i\rangle_S) \leftarrow \text{RESHARE}(r_C, i)$
13: $(\langle u, e\rangle_S, \langle u, e\rangle_C) \leftarrow \text{RESHARE}(\vec{r}_S, (u, e))$

Algorithm 5.11: Garbled Circuit MOVE for Sparse Trees

next node is a leaf (Step 9 and 10). If this is true, then we set u, e to $w, 1$. Otherwise, we maintain their previous values (Step 11). Finally, we re-share i, u, e (Step 12 and 13).

5.6 Correctness and Security Analysis

This section discusses the correctness and security of our scheme. Our security proofs follow the idea of [139]. We construct simulators as in [101, 139] and refer to the same references for the indistinguishability part.

5.6.1 Sub-protocols

Lemma 5.6.1. *The GC protocol in Algorithm 5.5 is correct and secure in the semi-honest model.*

Proof. Depending on the comparison result between the attribute value xi and the threshold, Algorithm 5.5 returns the correct index of the next node. Security follows from Yao's garbled circuit protocol. □

Lemma 5.6.2. *The GC protocol in Algorithm 5.6 is correct and secure in the semi-honest model.*

Proof. If the execution of the protocol reaches a leaf, then i must be NULL which is correctly checked in Step 2. As a result, the correct classification label is computed in Step 3. Security follows from Yao's garbled circuit protocol. □

Lemma 5.6.3. *The OAI using the GC in Algorithm 5.7 is correct and secure in the semi-honest model.*

Proof. The equality check in step 4 returns 1 for exactly one index between $0, \ldots, n-1$. The selection step 5 returns 0 for all $j < i$ and a_i for all $j \geq i$. The output a_i is secret-shared in step 6. Security follows from Yao's garbled circuit protocol. □

Lemma 5.6.4. *The OAI as described in Algorithm 5.8 is correct and secure in the semi-honest model.*

Proof. Let $A = [a_0, \ldots, a_{n-1}]$ be the array of the sender, $\langle i \rangle_S = r$ and $\langle i \rangle_C = k = i + r \bmod n$. Moreover, let $A' = [a'_0, \ldots, a'_{n-1}]$ where $a'_{j+r \bmod n} = a_j \oplus s$ for a random s. From the correctness of OT_n^1 protocol with input (A', k), C receives the keys corresponding to $k = i + r \bmod n$ and can decrypt $a'_{i+r \bmod n} = a_i \oplus s$. Security also follows from the security of $\mathrm{OT}_n^1(A', k)$, which guarantees that C can decrypt only one message. Finally, this message is blinded by a random value s known only to the server.

Let $\mathsf{View}_S^{\Pi_{\mathrm{OT}}}, \mathsf{View}_C^{\Pi_{\mathrm{OT}}}$ be the view of the sender and the receiver in the OT_n^1 protocol. Then the respective views of sender and receiver in the OT indexing are $\mathsf{View}_S^{\Pi_{\mathrm{OTI}}}(A, \varepsilon) = (A', \mathsf{View}_S^{\Pi_{\mathrm{OT}}}(A', k), \langle a_i \rangle_S)$ and $\mathsf{View}_C^{\Pi_{\mathrm{OTI}}}(A, \varepsilon) = \mathsf{View}_C^{\Pi_{\mathrm{OT}}}(A', k)$. Moreover, let $\mathsf{Sim}_S^{\mathsf{ot}}, \mathsf{Sim}_C^{\mathsf{ot}}$ be the simulators of sender and receiver the OT_n^1 protocol. We construct the simulators of the OT indexing as follows:

- Sender: the simulator $\mathsf{Sim}_S^{\mathsf{oti}}(A, \langle a_i, i \rangle_S)$ receives as input the array A, a share $\langle i \rangle_S$ of the index and a share $\langle a_i \rangle_S$ of the indexed element. The simulator computes $A' = [a'_0, \ldots, a'_{n-1}]$ where $a'_{j+r \bmod n} \leftarrow a_j \oplus \langle a_i \rangle_S, j = 0, \cdots, n-1$ and outputs $(A', \mathsf{Sim}_S^{\mathsf{ot}}(A', \varepsilon), \langle a_i \rangle_S)$.
- Receiver: the simulator $\mathsf{Sim}_C^{\mathsf{oti}}(\varepsilon, \langle a_i, i \rangle_C)$ receives as input a share $\langle i \rangle_C$ of the index and a share $\langle a_i \rangle_C$ of the indexed element. It just outputs $\mathsf{Sim}_C^{\mathsf{ot}}(\langle i \rangle_C, \langle a_i \rangle_C)$.

The output of both simulators $\mathsf{Sim}_S^{\mathsf{oti}}$ and $\mathsf{Sim}_C^{\mathsf{oti}}$ are clearly indistinguishable from the corresponding party's view in the real protocol. □

5.6.2 Main Protocol

Theorem 5.6.5 (Correctness)**.** *The protocol described in Protocol 5.4 is correct.*

Proof. We prove by induction on the level of the tree that the protocol correctly traverses the tree. At level 0 there is only the root node. At level $d' < d$ the correctness of the sub-protocols guarantees the computation of the correct attribute value x_i at Step 5, the correct node index v' at Step 6 and the correct node $\mathcal{N}'$ at Step 7. Finally, at level d Algorithm 5.6 returns the correct classification label. □

Theorem 5.6.6 (Security)**.** *The protocol described in Protocol 5.4 is secure in the semi-honest model.*

Proof. Given a DT model $\mathcal{M}$, the simulator $\mathsf{Sim}_S^{\mathsf{pdte}}$ generates random elements to simulate the sharing of the root (Step 1). Then it generates a random attribute vector $\tilde{x}$ and invokes d times the simulators of the sub-protocols for the server. Analogously, given d and x the simulator $\mathsf{Sim}_C^{\mathsf{pdte}}$ generates a random model $\mathcal{M}'$, simulates the sharing of the root as above and invokes d times the simulators of the sub-protocols for the client.

Let the input of the client be the attribute vector $x = [x_0, \ldots, x_{n-1}]$ and the input of the server be the node data structure $\mathbf{N} = [N_0, \ldots, N_M]$ as defined in Definition 5.3.1.

The view $\mathsf{View}_S^{\Pi_{\mathrm{PDTE}}}(\mathcal{M}, x)$ of the server consists of its views in the sub-protocols which are denoted as follows:

- $\mathsf{View}_S^{\Pi_{\mathrm{IV}}}(\varepsilon, x)$ for the sub-protocol IndexVector,

- $\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{TR}}}(\langle w, vl, vr, x_i\rangle_S, \langle w, vl, vr, x_i\rangle_C)$ for the sub-protocol TRAVERSE,
- $\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{IT}}}(\mathbf{N}, \varepsilon)$ for the sub-protocol INDEXTREE,
- $\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{MV}}}((\langle w, i\rangle_S, \mathsf{in}_S), (\langle w, i\rangle_C, \mathsf{in}_C))$ for the sub-protocol MOVE.

Moreover, let the simulators of the server in the sub-protocols be denoted as follows:

- $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{iv}}(\varepsilon, \langle x_i, i\rangle_S)$ for the sub-protocol INDEXVECTOR,
- $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{tr}}(\langle w, vl, vr, x_i\rangle_S, \langle v\rangle_S)$ for the sub-protocol TRAVERSE,
- $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{it}}(\mathbf{N}, \langle w, vl, vr, i\rangle_S)$ for the sub-protocol INDEXTREE,
- $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{mv}}((\langle w, i\rangle_S, \mathsf{in}_S), (\langle i\rangle_S, \mathsf{out}_S))$ for the sub-protocol MOVE.

The variables $\mathsf{in}_S, \mathsf{in}_C$ and $\mathsf{out}_S, \mathsf{out}_C$ denote respectively additional inputs and outputs as described in Algorithms 5.6 and 5.11. Then the view $\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{PDTE}}}$ of the server in Protocol 5.4 is:

$$\langle \mathcal{N}\rangle_S, \underbrace{\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{IV}}}(\cdot), \mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{TR}}}(\cdot), \mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{IT}}}(\cdot), \mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{MV}}}(\cdot), \cdots}_{d \text{ times}}.$$

The simulator for the server $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{pdte}}(\mathcal{M}, \varepsilon)$ receives as input the decision tree, generates a random $\widetilde{\langle \mathcal{N}\rangle_S}$, invokes d times the simulators of the sub-protocols and outputs:

$$\widetilde{\langle \mathcal{N}\rangle_S}, \underbrace{\mathsf{Sim}_{\mathsf{S}}^{\mathsf{iv}}(\cdot), \mathsf{Sim}_{\mathsf{S}}^{\mathsf{tr}}(\cdot), \mathsf{Sim}_{\mathsf{S}}^{\mathsf{it}}(\cdot), \mathsf{Sim}_{\mathsf{S}}^{\mathsf{mv}}(\cdot), \cdots}_{d \text{ times}}.$$

The simulation for the client is similar. □

5.6.3 One-Sided Simulatability

Since their original schemes are secure against passive adversaries, [175, 193] also consider the one-sided simulatability. When in a 2-party protocol between parties P_1 and P_2, only P_2 receives an output, the *one-sided simulatability* requires that the protocol is private against a corrupt P_1 and fully simulatable against P_2 [101]. For the private decision tree functionality, P_1 is the server, while P_2 is the client. We now discuss how our scheme can be made one-side simulatable when the indexing is performed with OT. First note that, if OT extension is actively secure and the GC output is not revealed to the GC generator, then the GC protocol is one-sided simulatable. In the GC TRAVERSE (Algorithm 5.5), the client does not reveal the final output to the server. We apply a small change to the GC MOVE (Algorithm 5.6) such that it reveals $i+r_S$ to the client, where r_S is randomly chosen by the server. After the GC execution the client chooses $\langle i\rangle_C \leftarrow r_C$ and sends $i+r_S+r_C$ to the server. Finally, the server computes $\langle i\rangle_S \leftarrow i + r_C$. We, therefore, expect that, by using the actively secure OT extension of [116] which adds a negligible overhead on top of the passive one, our scheme can be made one-sided simulatable with negligible extra costs. On the other hand, to have one-sided simulatability [175, 193] use zero-knowledge proof and have at least double costs.

5.7 Complexity Analysis

This section presents the complexity analysis of our scheme and compares it to previous work [175, 193]. We focus on the level indexing and sparse trees optimizations of Sections 5.5.1 and 5.5.3.

5.7.1 Asymptotic Analysis

Asymptotic Analysis of [175, 193]. We begin by recalling the cost provided by [175, 193]. In [193] the client performs $\mathcal{O}((n+m)\mu)$ asymmetric and $\mathcal{O}(d)$ symmetric operations, while the server performs $\mathcal{O}(m\mu)$ asymmetric and $\mathcal{O}(2^d)$ symmetric operations. In [175] the client performs $\mathcal{O}((n+m)\mu)$ asymmetric operations, and the server $\mathcal{O}(m\mu)$ asymmetric operations.

Asymptotic Analysis of our Scheme. Our cost consists of the cost for GCs Traverse and Move (C_{GC}), the cost for IndexVector (C_{IV}) and the cost for IndexTree (C_{IT}). If $C_{IT}^{(j)}$ denotes the cost for IndexTree at level j (Section 5.5.1), then the total cost is $d \cdot C_{GC} + d \cdot C_{IV} + \sum_{j=1}^{d} C_{IT}^{(j)}$. All computations require only symmetric operations. Each GC consists of 6μ ciphertexts resulting in a total of $\mathcal{O}(6\mu d) = \mathcal{O}(\mu d)$ operations for each party. For IndexVector, we consider the cases, where it is implemented either with OT or ORAM:

- **OAI With OT.** In this case, the cost of IndexVector is $\mathcal{O}(dn)$ for the client, and $\mathcal{O}(d\log(n))$ for the server. Each level j of the tree has 2^j elements, resulting in $\mathcal{O}(d^2)$ and $\mathcal{O}(2^d)$ operations in IndexTree for the client and the server respectively. Overall, the client and server perform $\mathcal{O}(\mu d + dn + d^2) = \mathcal{O}(d^2)$ and $\mathcal{O}(\mu d + d\log(n) + 2^d) = \mathcal{O}(2^d)$ symmetric operations.
- **OAI With ORAM.** The asymptotic cost of recursive circuit ORAM for an array of size $\mathcal{O}(2^d)$ is $\mathcal{O}(d^3)$. Hence with ORAM, the costs of the client and server are respectively $\mathcal{O}(\mu d + dn + d^4)$ and $\mathcal{O}(\mu d + d\log(n) + d^4)$. For ODS and FLORAM, it suffices to replace $\mathcal{O}(d^3)$ by $\mathcal{O}(d^2)$ and $\mathcal{O}(d)$ respectively. Assuming constant μ and n, the complexity is indeed sublinear in the size $\mathcal{O}(2^d)$ of the tree.

Asymptotic Analysis Summary. The above cryptographic operations are in fact encryption/decryption operations such that we can assimilate their number to the number of ciphertexts sent. In the worst case (complete tree), $m \approx 2^d$ is the dominant factor. This results in an overall asymptotic cost of $\mathcal{O}(m)$ as claimed in Table 5.1 for [175, 193] and our scheme.

5.7.2 Precise Analysis

We now compute the estimated actual communication cost depending on the protocols parameters.

Precise Analysis of [175, 193]. We first provide our own analysis of the communication of previous work. Let κ denote the bit length of the asymmetric ciphertext. For ElGamal using 256-bit elliptic curve, $\kappa = 1024$ (a ciphertext consists of two group elements, which on the elliptic curve are encoded as points with coordinates 256-bit long). We use λ to denote the symmetric security parameter, hence $\lambda = 128$. Wu et

al.'s scheme uses the OT extension protocol of [9] in which for sending k strings of length λ in parallel, the receiver sends $k\lambda$ and the sender $2k\lambda$ bits.

In [193] each party sends two messages before engaging in a 1-out-of-2^d-OT (with the server as sender) which is implemented using Naor and Pinkas OT [154] by running d times the OT extensions of [9] and transferring 2^d symmetric ciphertexts of length 2λ each. This results in the client sending $\kappa\mu n + \kappa m + 2d\lambda$ and the server sending $\kappa\mu m + \kappa 2^d + 4d\lambda + 2^d 2\lambda$ bits.

The analysis of [175] is much simpler. Also in this case, each party sends two messages, which results in the client sending $\kappa\mu n + \kappa m$ and the server sending $\kappa\mu m + 2\kappa(m+1)$ bits.

Precise Analysis of our Scheme. We implemented our scheme with the OT extensions of [107] which has a communication cost of $2\lambda(\lambda + k)$ bits for both sender and receiver to transfer k strings of length λ. In each GC the cost of the client (as evaluator) consists of its cost in OT, while the cost of the server (as generator) consists of the ciphertexts of the garbled tables (GT), the garbled input (GI) of the server and the OT cost. For each node DS (w, vr, vl, i), the threshold w has the same bit length μ as the attribute values, while the length of vl, vr, i might be much smaller and will be denoted by l. The length of i depends on the size of the attribute vector, while vl, vr depend on the number of nodes at the corresponding level in the tree (Section 5.5.1). In our evaluation for UCI Datasets, l = 20 for Spambase and l = 16 for the other datasets.

The input length of the client in GC Traverse is $2(\mu + l)$ hence the client sends $2\lambda(\lambda+2\mu+2l)$ bits. Similarly, in GC Move the client sends $2\lambda(\lambda+\mu+2l)$ bits resulting in a total of $2\lambda d(2\lambda + 3\mu + 4l)$. In GC Traverse, the server sends $2\lambda(\mu + 2l)$ bits as GT cost, $\lambda(2\mu + 3l)$ bits as GI cost and $2\lambda(\lambda + 2\mu + 2l)$ bits in OT. In GC Move, the server sends $2\lambda(\mu + 2l)$ bits as GT cost, $\lambda(\mu + l)$ bits as GI cost and $2\lambda(\lambda + \mu + 2l)$ bits in OT. Overall the server sends $\lambda d(4\lambda + 13\mu + 20\lambda l)$ bits as GC cost.

Recall that the array indexing with OT is implemented with Naor and Pinkas OT^1_n that requires running $\log(n)$ times OT^1_2 and transferring n symmetric ciphertexts. In IndexVector, the client as sender sends $\lambda d(2\lambda + 2\log(n) + n)$ and the server sends $2\lambda d(\lambda + \log(n))$ bits.

In IndexTree, the parties run $\text{OT}^1_{2^j}$ at each level j in $[1, d]$, where the client and the server send $2\lambda(\lambda + j)$ and $2\lambda(\lambda + j) + \kappa 2^i$ bits respectively. As a result, the client sends $2\lambda(\lambda d + \frac{(1+d)d}{2})$ and the server $2\lambda(\lambda d + \frac{(1+d)d}{2}) + 2\lambda(2m + 1)$ bits (a tree with m decision nodes, has $2m + 1$ nodes in total).

In summary, the client sends

$$2\lambda d(2\lambda + 3\mu + 4l) + \lambda d(4\lambda + 2\log n + n + d + 1)$$

bits and the server sends

$$\lambda d(4\lambda + 13\mu + 20l) + \lambda d(4\lambda + 2\log(n) + d + 1) + 2\lambda(2m + 1)$$

bits in our protocol.

Precise Analysis Summary. To summarize this section (see Table 5.5), the protocols in [175, 193] perform m comparisons using asymmetric operations, while we perform only $d = \log(m)$ comparisons using symmetric operations. Using GC or OT, we still have linear cost as previous work, while requiring only symmetric operations with small constants. Using ORAM (or ODS, FLORAM), C_{IT} is computed

accordingly and OAI is sublinear while requiring only symmetric operations. Our implementation uses a concurrent queue implementation offered by ObliVM, which uses only 2 threads to manage network input/output. While running the protocol, the main thread inserts new messages in the queue (it waits until there is enough place). The second thread is only responsible for sending the content of the queue over the network. Previous work can also use multithreading to improve the execution time, but not the communication. Their major advantage is the use of ECC which allows smaller ciphertexts than when using a group $\mathbb{Z}_p$ with prime p. Our scheme, however, can still be optimized by using more efficient garbling [19], OT extensions [9, 11, 116] and ORAM (ODS [118, 191], FLORAM [72]). Using more efficient OT extensions [9, 11] that reduces the runtime and bandwidth to 41% and 50% respectively, will significantly improve the performance of our scheme.

5.7.3 Round Complexity

The main cryptographic primitives used in our scheme are OT extension, OT_N^1 and GC, which are all one round protocol. The scheme itself consists of d iterations. Each iteration consists of an OAI on x, a GC Traverse, an OAI on the decision tree and a GC Move. When instantiated with OT or GC, OAI has one round. Hence, our scheme has $4d$ rounds. Recursive Circuit ORAM with size N has $log(N)$ rounds, resulting in $(d+3)d = d^2 + 3d$ rounds for our scheme, when instantiated with ORAM. ODS and FLORAM have one round, resulting in $4d$ rounds for our scheme.

5.8 Evaluation

We have implemented our scheme with the *level indexing* optimization and performed some experiments which will be discussed in this section.

5.8.1 Experimental Setup

We evaluated our scheme with the *ObliVM* [144] which is a Java framework for secure computation. It offers a compiler for a domain-specific language *ObliVM-lang* and a GC backend *ObliVM-GC*, which primarily supports semi-honest GC-based protocols. We implemented our scheme in Java (1.8) using only the GC backend.

ObliVM-GC supports a standard garbling scheme with garbled row reduction, *FreeXOR* and *HalfGate*. It also implements the OT extension protocol proposed by [107] and a basic OT protocol by [153] based on the decisional Diffie-Hellman assumption in $\mathbb{Z}_p$. In our experiments, we use a 2048-bit key length for $\mathbb{Z}_p$ and SHA-256 as random oracle for the OT extension. For our OT_n^1 implementation, we instantiated the PRF with AES-128. Finally, *ObliVM-GC* provides a large set of built-in Boolean circuits and a GC implementation of Circuit ORAM [189]. The ORAM is secret-shared between the two parties and for each read and write operation the client and the server execute a GC protocol to scan the corresponding path of the ORAM, then they execute another GC protocol to perform the eviction operation.

We stress that we did not use the ObliVM compiler. Using the compiler to transform the plaintext tree evaluation program in a secure one results in a program whose size is proportional to the tree size. A similar idea was already considered in the related work [14, 38] and was outperformed by Wu et al.'s paper [193]. Our memory accesses are not only to variable locations but they also depend on conditions involving secret variables. The ObliVM Compiler or any framework implementing the whole program will generate for each condition a corresponding oblivious computation whose number

DSet	n	d	m	Time (s)				Bandwidth (KB)		
				[193]	F	M	S	Upload	Download	Total
ECG	6	4	6	0.344	0.154	0.236	0.497	- (116.4)	- **(164.7)**	101.9 (281.0)
NUS	8	4	4	0.269	0.127	0.273	0.479	- (116.4)	- **(162.6)**	101.7 (279.0)
BCA	9	8	12	0.545	0.256	0.376	0.927	73.7 (233)	132.0 **(325.5)**	205.7 (558.5)
HDI	13	3	5	0.370	0.118	0.137	0.471	73.3 (87.4)	43.9 **(124.5)**	117.2 (211.8)
HOU	13	13	92	4.081	0.445	0.548	1.480	115.7 (378.7)	1795.2 **(531.8)**	1910.9 (910.5)
CSC	15	4	5	0.551	0.164	0.306	0.474	49.9 (116.5)	45.0 **(164.6)**	94.9 (281.1)
SPA	57	17	58	16.595	0.562	0.767	1.969	463.4 (490.5)	17363.3 **(684.5)**	17826.7 (1174.9)

Table 5.2: Performance on UCI datasets: The numbers on the left are taken from [193]. The numbers on the right and bold are our costs using OT/OT. The columns F (LAN 10Gbps for Server and Client), M (LAN 1Gbps for Server and Wifi 72Mbps for Client), S (Wifi 144Mbps for Server and Wifi 72Mbps for Client) represent the type of network which has no impact on the bandwidth.

is proportional to the number of decision nodes. We then use only ObliVM-GC to run our manually created GCs which are independent of the branching result.

As mentioned above, we compare our protocol to the scheme of Wu et al. [193] in the semi-honest model at the same security level 128, because they perform an extensive comparison to the other protocols and have the best performance in the computational two-party setting. Hence, we choose a similar test environment. We run both parties on two machines with Intel(R) Xeon(R) CPU E7-4880 v2 at 2.50GHz connected via a shared LAN and running Windows 10. The server machine has 4 CPUs and 4GB of RAM. The client machine has 4 CPUs and 8GB RAM. However, Wu et al. implemented their protocol in C++. As HE they implemented exponential ElGamal using the 256-bit elliptic curve *numsp256d1*, which is one of the main source of their performance improvement comparing to previous protocols. Our implementation uses Oracle's Java 1.8 and all experiments were run on the Java SE 64-Bit Server virtual machine. We also compare our scheme to the scheme of Tai et al. [175]. They implemented their protocol using ElGamal over elliptic curve *secp256k1*. However, we notice that, in their experiments [175], both the client and the server are run on one desktop computer equipped with Intel Core i7-6700 CPU (3.40 GHz).

Our scheme uses the OT extension protocol of [107] and requires therefore a setup phase which consists of an initialization of the OT extension protocol by running the basic OT ([153] in our implementation). The setup phase is executed only once to exchange symmetric keys which will be used in the OT extension [9]. It takes about 1 second and consumes about 20.31 KB (from sender to receiver) and 3.96 KB (from receiver to sender) communication. We note that [193] uses the very efficient OT extension of [9], which is not implemented in ObliVM yet. Additionally, if we want to index the tree with ORAM, we populate it in this setup phase. However, this is executed only once for each client. In the experiments below, we evaluate and report the costs for the online tree evaluation. In the following figures, we use *Server* (*Client*) *Time* to denotes the running time of the server (client) and *Server* (*Client*) *upload* to denote the number of bits sent by the server (client).

5.8.2 Performance on Real Datasets

As Wu et al., we evaluate our protocol on the following seven real datasets from the UCI repository [185] in the semi-honest model and at the security level 128: ECG (ECG), Nursery (NUS), Breast-Cancer (BCA), Heart-Disease (HDI), Housing (HOU), Credit-Screening (CSC), and Spambase (SPA). We first transform each tree in

DSet	Bandwidth (KB) IndexVector			Bandwidth (KB) IndexTree			Bandwidth (KB) GC		
	↑	↓	Total	↑	↓	Total	↑	↓	Total
ECG	35.80	22.83	61.63	14.76	30.94	45.7	65.68	110.76	176.44
NUS	36.00	22.18	58.18	13.84	31.68	45.52	66.47	108.63	175.10
BCA	71.76	45.05	116.81	33.17	64.19	95.36	127.90	216.15	344.05
HDI	26.88	16.57	43.45	11.74	23.48	35.22	48.65	84.31	132.96
HOU	116.57	72.53	189.10	54.51	108.80	163.31	207.50	350.33	557.83
CSC	36.14	26.74	62.88	11.49	24.65	36.14	68.67	113.13	181.80
SPA	152.53	92.32	244.83	71.04	144.56	215.60	266.78	447.48	714.26

Table 5.3: Detailed Bandwidth Costs on UCI datasets using OT/OT. The symbols ↑ and ↓ stand for upload and download.

DSet	Time (ms)								
	IndexVector			IndexTree			GC		
	F	M	S	F	M	S	F	M	S
ECG	16.52	56.95	139.15	11.92	34.36	59.11	108.76	136.35	314.99
NUS	15.31	67.12	126.31	12.05	39.12	46.70	85.40	160.51	323.17
BCA	33.51	104.31	211.95	21.25	54.10	108.11	179.10	210.65	626.59
HDI	9.95	37.25	128.21	7.84	18.13	69.12	73.52	75.83	291.33
HOU	50.92	156.0	265.88	37.56	76.86	165.04	334.80	310.52	1067.25
CSC	17.53	65.87	127.53	10.14	46.24	44.90	127.88	187.80	321.87
SPA	71.83	213.33	501.18	44.64	104.61	220.44	427.85	443.98	1266.41

Table 5.4: Detailed Time Costs on UCI datasets using OT/OT. The columns F, M, S have the same meaning as in Table 5.2.

DSet	[193]			[175]			Ours			Ours
	↑	↓	Total	↑	↓	Total	↑	↓	Total	With [9]
ECG	48.88	50.75	99.63	48.75	49.75	98.5	97.06	137.09	234.16	105.78
NUS	64.62	34.75	99.37	64.5	33.25	97.75	97.31	137.09	234.40	105.90
BCA	73.75	136.5	210.25	73.5	99.25	172.75	195.25	274.90	470.15	213.15
HDI	104.71	41.43	146.15	104.62	41.5	146.12	73.17	102.90	176.07	79.70
HOU	115.90	2016.81	2132.71	115.5	759.25	874.75	319.10	452.25	771.35	353.73
CSC	120.75	42.75	163.5	120.62	41.5	162.12	97.75	137.15	234.90	106.40
SPA	463.78	20945.06	21408.84	463.25	478.75	942.0	439.60	610.87	1050.48	494.79

Table 5.5: Comparison of estimated bandwidth costs (in KB) on UCI datasets. The symbols ↑ and ↓ stand for upload to server and download to client. Columns **Total** are the total costs. The last column is our estimated total cost when using OT extension of [9].

an array as explained above. Then, we perform 100 tree evaluations, measure runtime and bandwidth costs and compute the mean.

Our results are summarized in Table 5.2, where n denotes the number of attributes, m denotes the number of decision nodes, d denotes the depth of the tree and μ denotes the bitlength of attribute and threshold values ($\mu = 64$ as in [193]). Moreover, Tables 5.3 and 5.4 show how each sub-protocol contributes to the costs of Table 5.2. Recall that we can instantiate our scheme with three different array indexing methods. The results in Tables 5.2, 5.3, and 5.4 were achieved using OT indexing on both sides.

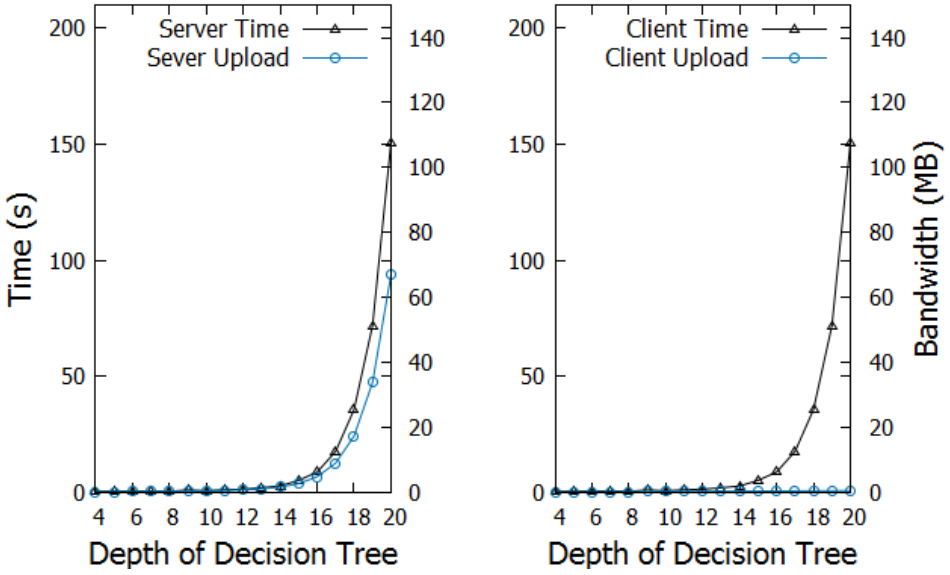
For small size trees, [193] has better bandwidth costs compared to the current implementation of our scheme. However, for applications that are willing to compromise on bandwidth, our scheme is more suitable as it is faster. Moreover, it has other advantages. First, it can be further optimized, e.g., by using the efficient OT extension [9, 11, 129] and fast garbling (e.g., JustGarble [19]), which are not yet implemented in ObliVM. In [144], it is estimated that combining ObliVM with JustGarble may reduce the time to compare 16384 bit Integers from 26 ms (using the original ObliVM as in our experiments) to 1.96 ms. The scheme of [9, 11] significantly reduces the runtime and bandwidth costs of the OT extension protocol to 41% and 50% respectively. An estimate of the communication cost using [9] is depicted in Table 5.5. OAI with OT can be implemented with the OT_n^1 of [129] which improves upon [154] by a factor ≈ 5.39. It is therefore clear that these optimizations will significantly improve the performance of our scheme in both runtime and bandwidth. Additionally, while [193] is based on public-key primitives (i.e., discrete logarithm on elliptic curve), our scheme relies only on symmetric cryptography. We require public-key primitives only for a one time initialization of the OT extension protocol. Since our scheme is based on secret array indexing, it naturally benefits from optimizations on oblivious data structures [118, 191]. Finally, the optimization described in Section 5.5.2 reduces the bandwidth costs of IndexVector to a few kilobytes.

For mid-size to large trees (e.g., "housing" and "Spambase" datasets) our protocol outperforms previous protocols. Our protocol reduces the runtime of the "housing" dataset from 4 seconds (by Wu et al.) to 0.5 second, while the communication cost is reduced from 2 MB to 1 MB. For the "Spambase" dataset our protocol is even 17 times better. Theirs takes 17 seconds consuming 18 MB bandwidth, while we run within less than 1 second and require only 1.2 MB communication.

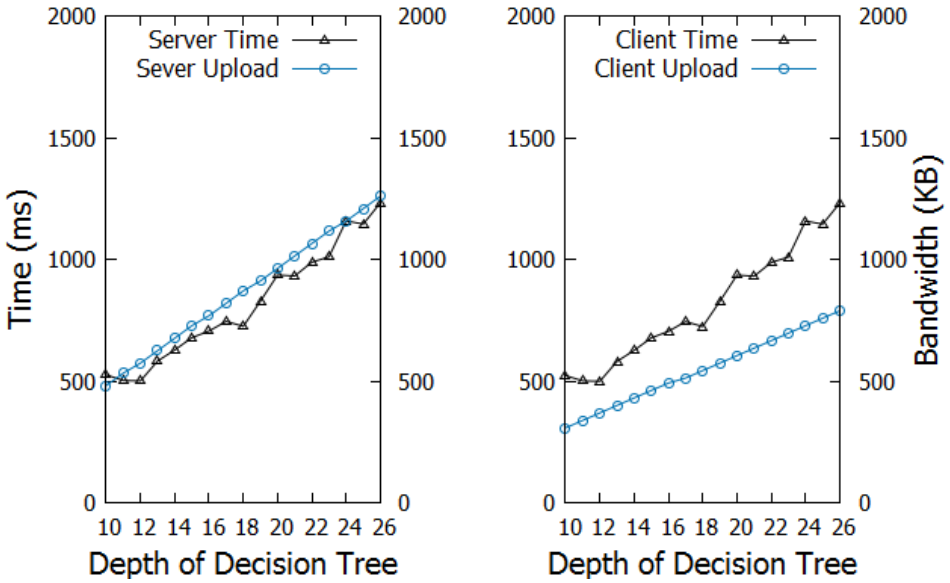
Tai et al. [175] also compare their work to [193], however, running both the client and the server on one desktop computer equipped with Intel Core i7-6700 CPU (3.40 GHz). Their reported time consists therefore only of the local computation time without network traffic. For small trees, they reported similar performances to [193] in both bandwidth and runtime. For large trees such as "housing" and "Spambase", their protocols run in about 2s (1.984 and 1.804 resp.) on one machine consuming slightly less than 1MB (0.854 and 0.920 resp.). As a result, our scheme is already faster and can be further optimized as explained above. This shows that flattening the tree increases performance.

5.8.3 Scalability

We also evaluate the scalability of our scheme by experimenting with synthetic trees of different depths and densities and using similar parameters as previous work. We vary the depth of the tree between 4 and 26 and use 64-bit precision and $n = 16$. We also ran both experiments of this section on a LAN, while Wu et al. reported costs excluding network traffic.



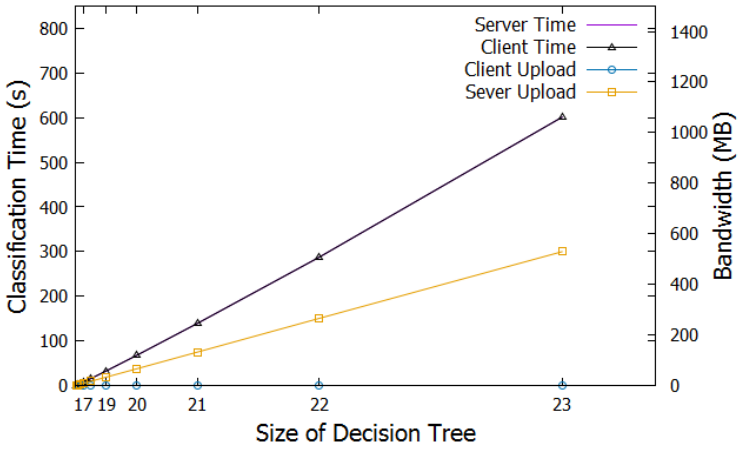
(a) Complete Trees



(b) Sparse Trees

Figure 5.12: Scalability experiment with OT/OT indexing

(a) Indexing with GC/OT



(b) Indexing with OT/OT

(c) Indexing with ORAM/OT

Figure 5.13: Costs for very large trees: For readability only the depth (i.e., $\log(M) - 1$) is displayed on the x-axis, which ends at 18 for GC and 23 for OT and ORAM because we ran out of memory.

First, we consider complete decision trees, which are the worst-case. We evaluate the scheme with OT indexing for the tree and the attribute vector and measure computation and communication costs of both parties. The result is depicted in Figure 5.12a. Our results show that even for deeper trees with a depth around 20, the scheme takes less than 3 minutes and less than 70 MB bandwidth. For complete trees of depth 14, the maximal depth reported by [193], their protocol runs in about 5 minutes, excluding network communication, and requires about 120 MB bandwidth. Our protocol evaluates a complete tree of depth 14 in less than 3 seconds via LAN, while consuming only about 2 MB of the bandwidth, which is 60 times better.

Using a complete tree even in the case where the decision tree has only a few nodes is totally inefficient, because for deeper trees the difference between $2^{d+1} - 1$ and the real size of the tree can be very huge. For their experiments on sparse trees, Wu et al. assume that the number of decision nodes is linear in the depth of the tree, e.g., $m = 25d$. We experiment with sparse trees using the optimization mentioned in Section 5.5.3. We vary the depth of the tree between 10 and 26 and set the number of decision nodes to $m = 25d$. Then we generate random trees with defined parameters d and $m = 25d$, run our protocol, and measure the results. Figure 5.12b shows that our costs grow linearly, rather than exponentially in the depth of the tree as in [193]. For sparse trees of depth up to 26, our protocol takes less than 1.5 seconds and less than 1.5 MB bandwidth. For sparse trees of depth 20, the maximal depth reported by [193], their protocol runs in about 2 minutes (excluding network communication) and requires about 140 MB bandwidth. The scheme of [175] runs in about 10 seconds (excluding network communication) and requires about 4.1 MB bandwidth. Our protocol evaluates a sparse tree of depth 20 with 500 decision nodes in less than 1 second via LAN, while consuming only about 1.5 MB bandwidth.

5.8.4 Very Large Trees

In our last experiment, we consider very large complete trees with a depth larger than 20, containing millions of decision nodes. In this setting, we expect the ORAM solution to outperform the other approaches, since it yields to a sub-linear complexity. We ran this experiment on a single machine (via a loopback interface) with Intel(R) Core(TM) i7-4770 CPU at 3.40GHz, 16GB of RAM, and Windows 10. We then run three experiments using either GC, OT, or ORAM to index the decision tree and OT indexing for the attribute vector. For each experiment, we set $n = 64$ and vary the depth from 10 to 24. As mentioned above, indexing with ORAM requires populating the ORAM in the setup phase, which is very tedious for large trees and may take up to 20 days to compute [94]. However, notice that the costs for an ORAM access depend on the capacity of the ORAM, but not on the actual number of elements stored in it. For this reason, we avoid a long ORAM initialization by populating the ORAM for larger trees with just enough elements to evaluate the decision tree. The results of the experiment are summarized in Figure 5.13. For small trees with depth smaller than 12, GC and OT indexing are better than ORAM. The computation cost for OT remains better than ORAM up to depth 16. However, the costs for GC and OT double with the depth and are linear in the size of the tree, while the costs for ORAM are sublinear in the size of the tree as shown in Figure 5.13. For trees of depth larger than 21, ORAM outperforms both GC and OT in computation and communication costs. For example, for depth 22, ORAM takes about 13 seconds and 175 MB of Bandwidth, while OT takes 287 seconds and 266 MB.

5.9 Summary

In this chapter, we presented a protocol for evaluating private decision trees using sublinear communication. The idea of our novel solution is to represent the tree as an array. Then we execute a number of comparisons that is equal to the depth of the tree. We get the inputs to the comparison by obliviously indexing the tree and the attribute vector. Each comparison outputs secret shares of the index of the next node to be evaluated. We implement oblivious array indexing using either GC, OT, or ORAM. Using ORAM this results in the first protocol with sub-linear communication cost in the size of the tree. We implemented and evaluated our scheme on real datasets and synthetic decision trees. Our results show that, we are not only able to provide the first sublinear communication cost for large trees, but also reduce the computation and communication costs for mid-size to large real-world data set compared to the best related work.

Chapter 6

Non-Interactive Classification

In the previous chapter, we propose a protocol for private decision tree evaluation (PDTE) with sublinear cost. Almost all existing PDTE protocols (including those of the previous chapter) have several rounds requiring several interactions between the client and the server. Moreover, the communication cost depends on the size of the decision tree, while only a single classification is required by the client. Finally, they also require a computation complexity from the client that depends on the size of the tree. The goal of this chapter is to design and implement a novel client-server protocol that delegates the complete tree evaluation to the server while preserving privacy and keeping the overhead low. The chapter is structured as follows. We start by describing the problem with interactive solutions in Section 6.1 and subsequently give an overview of our non-interactive PDTE, a comparison with related work, and the main building block. Correctness and security definitions are similar to Section 5.2 of the previous chapter. We describe the data structure and the algorithms of our basic construction in Section 6.2. We discuss a binary implementation in Section 6.3 and an integer implementation in Section 6.4. We analyze correctness and security in Section 6.5 and present a complexity analysis in Section 6.6. We discuss evaluation results in Section 6.7 before summarizing the chapter in Section 6.9.

6.1 Problem Definition

In this section, we briefly review the problem of private decision tree evaluation (PDTE). We refer to Section 5.1 for more details and focus on describing the weaknesses of interactive solutions that we want to address in the current chapter.

6.1.1 Description

Recall that the PDTE problem consists of a server holding a private decision tree model and a client interested in classifying its private attribute vector using the server's private model. The goal of the computation is to obtain the classification while preserving the privacy of both – the decision tree and the client input. After the computation, the classification result is revealed only to the client, and nothing else is revealed neither to the client nor to the server. For applications and use cases, we refer to Section 5.1. Existing privacy-preserving protocols that address this problem use or combine different generic secure multiparty computation approaches, resulting in several interactions between the client and the server. Moreover, the communication cost depends on the size of the decision tree, while only a single classification is required by the client. Finally, they also require a computational power from the client that depends on the size of the tree. As a result, interactive protocols might be problematic in settings where the client has poor network communication and limited computation capability.

6.1.2 Solution Approach

While the bottleneck of interactive solutions is the network communication, in this chapter, we shift the bottleneck to the computation power as it is easier to manage by using parallelization. Our goal is to design and implement a novel client-server protocol that delegates the complete tree evaluation to the server while preserving privacy and keeping the performance acceptable. The idea is to use fully or somewhat homomorphic encryption (FHE/SHE) and evaluate the tree on ciphertexts encrypted under the client's public key. As a result, no intermediate or final computational result is revealed to the evaluating server. However, since current somewhat homomorphic encryption schemes have high overhead, we combine efficient data representations with different algorithmic optimizations to keep the computational overhead and the communication cost low. At the end, the computational overhead might still be higher than in existing protocols, however, the computation task can be parallelized resulting in a reduced computation time. As a result, we are able to provide the first non-interactive protocol, that allows the client to delegate the evaluation to the server by sending an encrypted input and receiving only the encryption of the result. Finally, existing approaches are secure in the semi-honest model and can be made *one-sided simulatable*[1] using techniques that may double the computation and communication costs. Our approach is one-sided simulatable by default, as the client does no more than encrypting its input and decrypting the final result of the computation (simulating the client is straightforward), while the server performs computation on ciphertexts encrypted with a semantically secure encryption under the client's public key.

In summary, we propose a non-interactive protocol for PDTE. Our scheme allows the client to delegate the evaluation to the server by sending an encrypted input and receiving only the encryption of the result. We propose PDT-BIN which is an instantiation of the main protocol with the binary representation of the input. Then we combine efficient data representations with different algorithmic optimizations to keep the computational overhead and the communication cost low. We also propose PDT-INT which is an instantiation of the main protocol using an arithmetic circuit, where the values are compared using a modified variant of the Lin-Tzeng comparison protocol [135]. Finally, we provide correctness and security proofs of our scheme, and implement and benchmark both instantiations using HElib [100] and TFHE [52].

6.1.3 Comparison with Related Work

The related work to interactive PDTE protocols has been described in Section 3.5. We briefly describe another related work by Lu et al. [145] that is non-interactive, then we compare our non-interactive PDTE to existing protocols, focusing on Lu et al. [145]. Using a polynomial encoding of the inputs and BGV homomorphic scheme [36], Lu et al. [145] propose a non-interactive comparison protocol called XCMP which is *output expressive* (i.e., it preserves additive homomorphism). They then implement the private decision tree protocol of Tai et al. [175] using XCMP. The resulting decision tree protocol is non-interactive and efficient because of the small *multiplicative depth*. However, it is not *generic*, that is, it primarily works for small inputs and depends explicitly on BGV-type HE scheme. Moreover, it does not support *SIMD* operations and is no longer output expressive as XCMP. Hence, it cannot be extended

[1] A 2-party protocol between parties P_1 and P_2 in which only P_2 receives an output, is *one-sided simulatable* if it is private (via indistinguishability) against a corrupt P_1 and fully simulatable against a corrupt P_2 [101].

Scheme	Rounds	Tools	Communication	Comparisons	Leakage
[38]	≈5	HE+GC	$\mathcal{O}(2^d)$	d	m, d
[14]	≈4	HE+GC	$\mathcal{O}(2^d)$	d	m, d
[34]	≥6	FHE/SHE	$\mathcal{O}(2^d)$	m	m
[193]	6	HE+OT	$\mathcal{O}(2^d)$	m	m
[175]	4	HE	$\mathcal{O}(2^d)$	m	m
[55]	≈9	SS	$\mathcal{O}(2^d)$	m	m, d
[183]	$\mathcal{O}(d)$	GC,OT ORAM	$\mathcal{O}(2^d)$ $\mathcal{O}(d^2)$	d	m, d
[145]	1	FHE/SHE	$\mathcal{O}(2^d)$	m	m
Pdt-Bin	1	FHE/SHE	$\mathcal{O}(1)$ or $\mathcal{O}(d)$	m	-
Pdt-Int	1		$\mathcal{O}(2^d/s)$		m

Table 6.1: Comparison of PDTE protocols.

Scheme	SIMD	Generic	Output-expressive	Multiplicative Depth	Output Length
[145]	no	no	no	3	2^{d+1}
Pdt-Bin	yes	yes	yes	$\lvert\mu\rvert + \lvert d\rvert + 2$	1 or d
Pdt-Int	yes	yes	no	$\lvert\mu\rvert + 1$	$\lceil 2^d/s \rceil$

Table 6.2: Comparison of 1-round PDTE protocols.

to a larger protocol (e.g., random forests [37]) while preserving the non-interactive property. Finally, its *output length* (i.e., the number of resulted ciphertexts from server computation) is exponential in the depth of the tree, while the output length of our binary instantiation is at most linear in the depth of the tree and the integer instantiation can use SIMD to considerably reduce it. A comparison of decision tree protocols is summarized in Tables 6.1 and 6.2. A more detailed complexity analysis is described in Section 6.6.

6.2 The Basic Construction

The security definition is similar to the previous chapter, so we refer to Section 5.2. In this section, we present a modular description of our basic protocol. We start by describing the data structure.

6.2.1 Data Structure

We follow the idea of some previous protocols [34, 55, 175] of marking edges of the tree with comparison results. So if the comparison at node v is the bit b then we mark the right edge outgoing from v with b and the left edge with $1-b$. For convenience, we will instead store this information at the child nodes of v and refer to it as cmp. We extend the data structure of Definition 5.3.1 by adding to each node a pointer to its parent node and a field for storing cmp.

Definition 6.2.1 (Data Structure)**.** *For a decision tree model $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att})$, we let* Node *be a data structure that for each node v defines the following fields:*

- $v.\mathsf{threshold}$ *stores the threshold* $\mathsf{thr}(v)$ *of the node v*
- $v.\mathsf{aIndex}$ *stores the associated index* $\mathsf{att}(v)$
- $v.\mathsf{parent}$ *stores the pointer to the parent node which is null for the root node*
- $v.\mathsf{left}$ *stores the pointer to the left child node which is null for each leaf node*
- $v.\mathsf{right}$ *stores the pointer to the right child node which is null for each leaf node*
- $v.\mathsf{cmp}$ *is computed during the tree evaluation and stores the comparison bit*

  $$b \leftarrow [x_{\mathsf{att}(v.\mathsf{parent})} \geq \mathsf{thr}(v.\mathsf{parent})]$$

  if v is a right node. Otherwise it stores $1-b$.
- $v.\mathsf{cLabel}$ *stores the classification label if v is a leaf node and the empty string otherwise.*

We use $\mathcal{D}$ to denote the set of all decision nodes and $\mathcal{L}$ the set of all leaf nodes of $\mathcal{M}$. As a result, we use the equivalent notation $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att}) = (\mathcal{D}, \mathcal{L})$.

With the data structure defined above, we now define the classification function as follows.

Definition 6.2.2 (Classification Function)**.** *Let $x = (x_0, \ldots, x_{n-1})$ be the attribute vector and $\mathcal{M} = (\mathcal{D}, \mathcal{L})$ be the decision tree model. We define the* classification function *to be*

$$\mathsf{f}_c(x, \mathcal{M}) = \mathsf{tr}(x, \mathsf{root}),$$

where root *is the root node and* tr *is the* traverse function *define as:*

$$\mathsf{tr}(x, v) = \begin{cases} \mathsf{tr}(x, v.\mathsf{left}) & \text{if } v \in \mathcal{D} \text{ and } x_{v.\mathsf{aIndex}} < v.\mathsf{threshold} \\ \mathsf{tr}(x, v.\mathsf{right}) & \text{if } v \in \mathcal{D} \text{ and } x_{v.\mathsf{aIndex}} \geq v.\mathsf{threshold} \\ v & \text{if } v \in \mathcal{L} \end{cases}$$

Lemma 6.2.3. *Let $x = (x_0, \ldots, x_{n-1})$ be an attribute vector and $\mathcal{M} = (\mathcal{T}, \mathsf{thr}, \mathsf{att}) = (\mathcal{D}, \mathcal{L})$ a decision model. We have*

$$\mathcal{T}(x) = b \cdot \mathsf{tr}(x, \mathsf{root.right}) + (1-b) \cdot \mathsf{tr}(x, \mathsf{root.left}),$$

where $b = [x_{\mathsf{att}(\mathsf{root})} \geq \mathsf{thr}(\mathsf{root})]$ is the comparison at the root node.

Proof. The proof follows by induction on the depth of the tree. In the base case, we have a tree of depth one (i.e., the root and two leaves). In the induction step, we have two trees of depth d and we join them by adding a new root. □

```
1: function EvalDnode(𝒟, ⟦x⟧)
2:     for each v ∈ 𝒟 do
3:         ⟦b⟧ ← ⟦[x_{v.aIndex} ≥ v.threshold]⟧
4:         ⟦v.right.cmp⟧ ← ⟦b⟧
5:         ⟦v.left.cmp⟧ ← ⟦1 − b⟧
```

Algorithm 6.1: Computing a Decision Bit

```
1: function EvalPaths(𝒟, ℒ)
2:     let Q be a queue
3:     Q.enqueue(root)
4:     while Q.empty() = false do
5:         v ← Q.dequeue()
6:         ⟦v.left.cmp⟧ ← ⟦v.left.cmp⟧ ⊡ ⟦v.cmp⟧,
7:         ⟦v.right.cmp⟧ ← ⟦v.right.cmp⟧ ⊡ ⟦v.cmp⟧
8:         if v.left ∈ 𝒟 then
9:             Q.enqueue(v.left)
10:        if v.right ∈ 𝒟 then
11:            Q.enqueue(v.right)
```

Algorithm 6.2: Aggregating Decision Bits

6.2.2 Algorithms

Initialization. The Initialization consists of a one-time key generation. The client generates appropriate triple $(\mathsf{pk}, \mathsf{sk}, \mathsf{ek})$ of public, private and evaluation keys for a homomorphic encryption scheme. Then the client sends $(\mathsf{pk}, \mathsf{ek})$ to the server. For each input classification, the client just encrypts its input and sends it to the server. To reduce the communication cost of sending client's input, one can use a trusted *randomizer* that does not take part in the real protocol and is not allowed to collaborate with the server. The trusted *randomizer* generates a list of random strings r and sends the encrypted strings $[\![r]\!]$ to the server and the list of r's to the client. For an input x, the client then sends $x+r$ to the server in the real protocol. This technique is similar to the commodity based cryptography [16] with the difference that the client can play the role of the *randomizer* itself and sends the list of $[\![r]\!]$'s (when the network is not too busy) before the protocol's start.

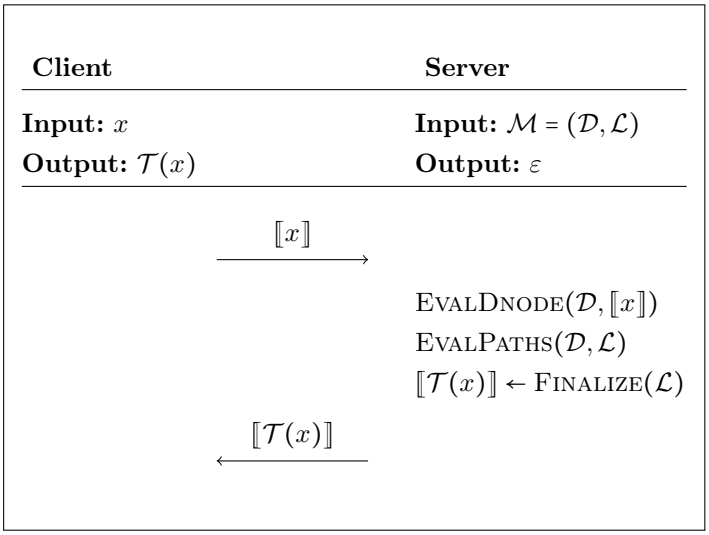
Computing Decision Bits. The server starts by computing for each node $v \in \mathcal{D}$ the comparison bit $b \leftarrow [x_{\mathsf{att}(v)} \geq \mathsf{thr}(v)]$ and stores b at the right child node ($v.\mathsf{right.cmp} = b$) and $1-b$ at the left child node ($v.\mathsf{left.cmp} = 1-b$). It is illustrated in Algorithm 6.1.

Aggregating Decision Bits. Then for each leaf node v, the server aggregates the comparison bits along the path from the root to v. We implement it using a queue and traversing the tree in BFS as illustrated in Algorithm 6.2.

Finalizing. After Aggregating the decision bits along the path to the leaf nodes, each leaf node v stores either $v.\mathsf{cmp} = 0$ or $v.\mathsf{cmp} = 1$. Then, the server aggregates the decision bits at the leaves by computing for each leaf v the value $[\![v.\mathsf{cmp}]\!] \odot [\![v.\mathsf{cLabel}]\!]$ and summing all the results. This is illustrated in Algorithm 6.3.

```
1: function FINALIZE(ℒ)
2:     [[result]] ← [[0]]
3:     for each v ∈ ℒ do
4:         [[result]] ← [[result]] ⊞ ([[v.cmp]] ⊡ [[v.cLabel]])
5:     return [[result]]
```

Algorithm 6.3: Finalizing



Protocol 6.4: The Basic Protocol

Putting It All Together. As illustrated in Protocol 6.4, the whole computation is performed by the server. It sequentially computes the algorithms described above and sends the resulting ciphertext to the client. The client decrypts and outputs the resulting classification label. The correctness is straightforward and follows from Lemma 6.2.3. The algorithms are straightforward and easy to understand. However, their naive application is inefficient.

6.3 Binary Implementation

In this section, we describe PDT-BIN, an instantiation of the basic scheme that requires encoding the plaintexts using their bit representation. Hence, ciphertexts encrypt bits and arithmetic operations are done mod 2.

6.3.1 Input Encoding

In this implementation, we encrypt plaintext bitwise. For each plaintext x_i with bit representation $x_i^{\mathsf{b}} = x_{i\mu} \dots x_{i1}$, we use $[\![x_i^{\mathsf{b}}]\!]$ to denote the vector $([\![x_{i\mu}]\!], \dots, [\![x_{i1}]\!])$, consisting of encryptions of the bits of x_i. As a result, the client needs to send $n\mu$ ciphertexts for the n attribute values. Unfortunately, homomorphic ciphertexts might be quite large. We can already use the trusted *randomizer* as explained before to send blinded inputs instead of ciphertexts in this phase. This, however, improves only the online communication. We additionally leverage the SVCP SIMD technique

that allows to pack many plaintexts into the same ciphertext and manipulate them together during homomorphic operations.

6.3.2 Ciphertext Packing

In the binary encoding, ciphertext packing means that each ciphertext encrypts s bits, where s is the number of slots in the ciphertext. Then we can use this property in three different ways. First, one could pack the bit representation of each classification label in a single ciphertext and allow the server to send back a single ciphertext to the client. Second, one could encrypt several attributes together and classify them with a single protocol evaluation. Finally, one could encrypt multiple decision node thresholds that must be compared to the same attribute in the decision tree model.

Packing Classification Label's Bits. Aggregating the decision bits using Algorithm 6.2 produces for each leaf $v \in \mathcal{L}$ a decision bit $[\![b_v]\!]$ which encrypts 1 for the classification leaf and 0 otherwise. Moreover, because of SVCP, the bit b_v is replicated to all slots. Now, let k be the number of classification labels (i.e., $|\mathcal{L}| = k$) and its bitlength be $|k|$. For each $v \in \mathcal{L}$, we let c_v denote the classification label $v.\mathsf{cLabel}$ which is $|k|$-bit long and has bit representation $c_v^{\mathsf{b}} = c_{v|k|} \dots c_{v1}$ with corresponding packed encryption $[\![\vec{c_v}]\!] = [\![c_{v|k|}| \dots |c_{v1}|0| \dots |0]\!]$. As a result, computing $[\![b_v]\!] \odot [\![\vec{c_v}]\!]$ for each leaf $v \in \mathcal{L}$ and summing over all leaves results in the correct classification label. Note that, this assumes that one is classifying only one vector and not many as in the next case.

Packing Attribute Values. Let $x^{(1)}, \dots, x^{(s)}$ be s possible attribute vectors with $x^{(l)} = [x_1^{(l)}, \dots, x_n^{(l)}]$, $1 \le l \le s$. For each $x_i^{(l)}$, let $x_i^{(l)^{\mathsf{b}}} = x_{i\mu}^{(l)}, \dots, x_{i1}^{(l)}$ be the bit representation. The client generates for each x_i the ciphertexts $[\![cx_{i\mu}]\!], \dots, [\![cx_{i2}]\!], [\![cx_{i1}]\!]$ as illustrated in Equation 6.1.

$$\begin{aligned} [\![cx_{i1}]\!] &= [\![x_{i1}^{(1)}|x_{i1}^{(2)}| \dots |x_{i1}^{(s)}]\!] \\ [\![cx_{i2}]\!] &= [\![x_{i2}^{(1)}|x_{i2}^{(2)}| \dots |x_{i2}^{(s)}]\!] \\ &\dots \\ [\![cx_{i\mu}]\!] &= [\![x_{i\mu}^{(1)}|x_{i\mu}^{(2)}| \dots |x_{i\mu}^{(s)}]\!] \end{aligned} \qquad \text{Packing many attribute values} \tag{6.1}$$

To shorten the notation, let y_j denote the threshold of the j-th decision node (i.e., $y_j = v_j.\mathsf{threshold}$) and assume $v_j.\mathsf{aIndex} = i$. The server just encrypts each threshold bitwise which automatically replicates the bit to all slots. This is illustrated in Equation 6.2.

$$\begin{aligned} [\![cy_{j1}]\!] &= [\![y_{j1}|y_{j1}| \dots |y_{j1}]\!] \\ [\![cy_{j2}]\!] &= [\![y_{j2}|y_{j2}| \dots |y_{j2}]\!] \\ &\dots \\ [\![cy_{j\mu}]\!] &= [\![y_{j\mu}|y_{j\mu}| \dots |y_{j\mu}]\!] \end{aligned} \qquad \text{Packing a single threshold value} \tag{6.2}$$

Note that $([\![cy_{j\mu}]\!], \dots, [\![cy_{j1}]\!]) = [\![y_j^{\mathsf{b}}]\!]$ holds because of SVCP. The above described encoding allows comparing s attribute values together with one threshold. This is possible because the routine SheCmp is compatible with SVCP such that we have:

$$\begin{aligned} \textsc{SheCmp}(([\![cx_{i\mu}]\!], \dots, [\![cx_{i1}]\!]), ([\![cy_{j\mu}]\!], \dots, [\![cy_{j1}]\!])) = \\ ([\![b_{ij}^{(1)}|b_{ij}^{(2)}| \dots |b_{ij}^{(s)}]\!], [\![b_{ji}^{(1)}|b_{ji}^{(2)}| \dots |b_{ji}^{(s)}]\!]), \end{aligned} \tag{6.3}$$

where $b_{ij}^{(l)} = [x_i^{(l)} > y_j]$ and $b_{ji}^{(l)} = [y_j > x_i^{(l)}]$. This results in a single ciphertext such that the l-th slot contains the comparison result between $x_i^{(l)}$ and y_j.

Aggregating decision bits remains unchanged as described in Algorithm 6.2. This results in a packed ciphertext $[\![b_v]\!] = [\![b_v^{(1)}|\dots|b_v^{(s)}]\!]$ for each leaf $v \in \mathcal{L}$, where $b_v^{(l)} = 1$ if $x^{(l)}$ classifies to leaf v and $b_u^{(l)} = 0$ for any other leaf $u \in \mathcal{L} - \{v\}$.

For the classification label c_v of a leaf $v \in \mathcal{L}$, let $[\![c_v^{\mathsf{b}}]\!] = ([\![c_{v|k|}]\!], \dots, [\![c_{v1}]\!])$ denote the encryption of the bit representation $c_v^{\mathsf{b}} = c_{v|k|} \dots c_{v1}$. To select the correct classification label Algorithm 6.3 is updated as follows. We compute $[\![c_{v|k|}]\!] \odot [\![b_v]\!], \dots, [\![c_{v1}]\!] \odot [\![b_v]\!]$ for each leaf $v \in \mathcal{L}$ and sum them component-wise over all leaves. This results in the encrypted bit representation of the correct classification labels.

Packing Threshold Values. In this case, the client encrypts a single attribute in one ciphertext, while the server encrypts multiple threshold values in a single ciphertext. Hence, for an attribute value x_i, the client generates the ciphertexts as in Equation 6.4. Let m_i be the number of decision nodes that compare to the attribute x_i (i.e., $m_i = |\{v_j \in \mathcal{D} : v_j.\mathsf{aIndex} = i\}|$). The server packs all corresponding threshold values in $\lceil \frac{m_i}{s} \rceil$ ciphertext(s) as illustrated in Equation 6.5.

$$\begin{aligned} [\![cx_{i1}]\!] &= [\![x_{i1}|x_{i1}|\dots|x_{i1}]\!] \\ [\![cx_{i2}]\!] &= [\![x_{i2}|x_{i2}|\dots|x_{i2}]\!] \\ &\dots \\ [\![cx_{i\mu}]\!] &= [\![x_{i\mu}|x_{i\mu}|\dots|x_{i\mu}]\!] \end{aligned} \qquad \text{Packing a single attribute value} \tag{6.4}$$

$$\begin{aligned} [\![cy_{j1}]\!] &= [\![y_{j_1 1}|\dots|y_{j_{m_i} 1}|\dots]\!] \\ [\![cy_{j2}]\!] &= [\![y_{j_1 2}|\dots|y_{j_{m_i} 2}|\dots]\!] \\ &\dots \\ [\![cy_{j\mu}]\!] &= [\![y_{j_1 \mu}|\dots|y_{j_{m_i} \mu}|\dots]\!] \end{aligned} \qquad \text{Packing many threshold values} \tag{6.5}$$

The packing of threshold values allows comparing one attribute value against multiple threshold values together. Unfortunately, we do not have access to the slots while performing homomorphic operations. Hence, to aggregate the decision bits, we make m_i copies of the resulting packed decision bits and shift left each decision bit to the first slot. Then the aggregation of the decision bits and the finalizing algorithm work as in the previous case with the only difference that only the result in the first slot matters and the remaining can be set to 0.

6.3.3 Efficient Path Evaluation

As explained above, the encryption algorithm Enc adds noise to the ciphertext which increases during homomorphic evaluation. While addition of ciphertexts increases the noise slightly, the multiplication increases it significantly [36]. The noise must be kept low enough to prevent incorrect decryption. To keep the noise low, one can either keep the circuit's depth low enough or use the refresh algorithm. In this section, we will focus on keeping the circuit depth low.

Definition 6.3.1 (Multiplicative Depth)**.** *Let f be a function and C_f be a Boolean circuit that computes f and consists of AND-gates or multiplication (modulo 2) gates and XOR-gates or addition (modulo 2) gates. The* circuit depth *of C_f is the maximal length of a path from an input gate to an output gate. The* multiplicative depth *of C_f is the path from an input gate to an output gate with the largest number of multiplication gates.*

For example, consider the function $f([a_1,\dots,a_n]) = \Pi_{i=1}^{n} a_i$. A circuit that successively multiplies the a_i has multiplicative depth n. However, a circuit that divides the array in two halves, multiplies the elements in each half and finally multiplies the result, has multiplicative depth $\lceil \frac{n}{2} \rceil + 1$. This gives the intuition for the following lemma.

Lemma 6.3.2 (Logarithmic Multiplicative Depth Circuit)**.** *Let $[a_1,\dots,a_n]$ be an array of n integers and f be the function defined as follows:*

$$f([a_1,\dots,a_n]) = [a'_1,\dots a'_{\lceil \frac{n}{2} \rceil}],$$

where

$$a'_i = \begin{cases} a_{2i-1} \cdot a_{2i} & \text{if } (n \bmod 2 = 0) \vee (i < \lceil \frac{n}{2} \rceil), \\ a_n & \text{if } (n \bmod 2 = 1) \wedge (i = \lceil \frac{n}{2} \rceil). \end{cases}$$

Moreover, let f be the iterated function where f^i is the i-th iterate of f defined as follows:

$$f^i([a_1,\dots,a_n]) = \begin{cases} [a_1,\dots,a_n] & \text{if } i = 0, \\ f(f^{i-1}([a_1,\dots,a_n])) & \text{if } i \geq 1. \end{cases}$$

The $|n|$-th iterate $f^{|n|}$ of f computes $\Pi_{i=1}^{n} a_i$ and has multiplicative depth $|n|-1$ if n is a power of two and $|n|$ otherwise, where $|n| = \log n$ is the bitlength of n:

$$f^{|n|}([a_1,\dots,a_n]) = [\Pi_{i=1}^{n} a_i]$$

Proof. For the proof we consider two cases: n is a power of two (i.e., $n = 2^l$ for some l), and n is not a power of two.

The Power of Two Case. The proof is inductive. Assume $n = 2^l$, we show by induction on l. The base case trivially holds. For the inductive step, we assume the statement holds for $n = 2^l$ and show it holds for $n' = 2^{l+1}$. By dividing the array $[a_1,\dots,a_{n'}]$ in exactly two halves, the inductive assumption holds for each half. Multiplying the results of both halves concludes the proof.

The Other Case. The proof is constructive. Assume n is not a power of two and let n'' be the largest power of two such that $n'' < n$, hence $|n''| = |n|$. We divide $[a_1,\dots,a_n]$ in two halves $A_1 = [a_1,\dots,a_{n''}]$ and $A' = [a_{n''+1},\dots,a_n]$. We do this recursively for A' and get a set of subsets of $[a_1,\dots,a_n]$ which all have a power of two number of elements. The claim then holds for each subset (from the power of two case above) and A_1 has the largest multiplicative depth which is $|n''|-1$. By joining the result from A_1 and A', we get the product $\Pi_{i=1}^{n} a_i$ with one more multiplication resulting in a multiplicative depth of $|n''| = |n|$. □

Now, we know that sequentially multiplying comparison results on the path to a leaf results in a multiplicative depth which is linear in the depth of the tree, and increases the noise significantly. Instead of doing the multiplication sequentially, we will therefore do it in such a way as to preserve a logarithmic multiplicative depth. This is described in Algorithm 6.5. Algorithm 6.5 consists of a main function and a sub-function. The main function EvalPathsE collects for each leaf v encrypted comparison results on the path from the root to v and passes it as an array to the sub-function EvalMul which is a *divide and conquer* type. The sub-function follows the construction described in the proof of Lemma 6.3.2. It divides the array in two

Require: leaves set $\mathcal{L}$, decision nodes set $\mathcal{D}$
Ensure: Updated $v.\mathsf{cmp}$ for each $v \in \mathcal{L}$
1: **function** EvalPathsE($\mathcal{L}$, $\mathcal{D}$)
2: **for each** $v \in \mathcal{L}$ **do**
3: **let** d = number of nodes on the path $(\mathsf{root} \to v)$
4: **let** path be an empty array of length d
5: $l \leftarrow d$
6: $w \leftarrow v$
7: **while** $w \neq \mathsf{root}$ **do** ▷ construct path to root
8: $\mathsf{path}[l] \leftarrow [\![w.\mathsf{cmp}]\!]$
9: $l \leftarrow l - 1$
10: $w \leftarrow w.\mathsf{parent}$
11: $[\![v.\mathsf{cmp}]\!] \leftarrow \text{EvalMul}(1, d, \mathsf{path})$

Require: integers from and to, array of nodes path
Ensure: Product of elements in path
1: **function** EvalMul(from, to, path)
2: **if** from ≥ to **then**
3: **return** path[from]
4: $n \leftarrow \mathsf{to} - \mathsf{from} + 1$
5: $\mathsf{mid} \leftarrow 2^{|n-1|-1} + \mathsf{from} - 1$ ▷ $|n|$ bitlength of n
6: $[\![\mathsf{left}]\!] \leftarrow \text{EvalMul}(\mathsf{from}, \mathsf{mid}, \mathsf{path})$
7: $[\![\mathsf{right}]\!] \leftarrow \text{EvalMul}(\mathsf{mid} + 1, \mathsf{to}, \mathsf{path})$
8: **return** $[\![\mathsf{left}]\!] \boxdot [\![\mathsf{right}]\!]$

Algorithm 6.5: Paths Evaluation with log Multiplicative Depth

parts (left and right) such that the left part has a power of two number of elements. Then it calls the recursion on the two parts and returns the product of their results.

The two functions in Algorithm 6.5 correctly compute the multiplication of decision bits for each path. While highly parallelizable, it is still not optimal, as each path is considered individually. Since multiple paths in a binary tree share a common prefix (from the root), one would ideally want to handle common prefixes one time and not many times for each leaf. This can be solved using a *memoization* technique which is an optimization that stores results of expensive function calls such that they can be used later if needed. Unfortunately, naive memoization would require a complex synchronization in a multi-threaded environment and linear multiplicative depth. In the next paragraph, we propose a pre-computation on the tree, that would allow us to have the best of both worlds – multiplication with logarithmic depth along the paths, while reusing the result of common prefixes, thus, avoiding unnecessary work.

6.3.4 Improving Path Evaluation with Pre-Computation

The idea behind this optimization is to use a directed acyclic graph which we want to define first.

Definition 6.3.3 (DAG)**.** *A* directed acyclic graph *(DAG) is a graph with directed edges in which there are no cycles. A vertex v of a DAG is said to be* reachable *from another vertex u if there exists a path that starts at u and ends at v. The reachability relationship is a partial order $\leq$ and we say that two vertices u and v are ordered as $u \leq v$ if there exists a directed path from u to v.*

We require our DAGs to have a unique maximum element. The edges in the DAG define dependency relation between vertices.

Definition 6.3.4 (Dependency Graph)**.** *Let h be the function that takes two DAGs G_1, G_2 and returns a new DAG G_3 that connects the maxima of G_1 and G_2. We define the function $g([a_1, \ldots, a_n])$ that takes an array of integers and returns:*

- *a graph with a single vertex labeled with a_1 if $n = 1$,*
- *$h(g([a_1, \ldots, a_{n'}]), g([a_{n'+1}, \ldots, a_n]))$ if $n > 1$ holds, where $n' = 2^{|n|-1}$ and $|n|$ denotes the bitlength of n.*

We call the DAG G generated by $G = g([a_1, \ldots, a_n])$ a dependency graph*. For each edge (a_i, a_j) in G such that $i < j$, we say that a_j* depends *on a_i and denote this by adding a_i in the* dependency list *of a_j. We require that if $L(j) = [a_{i_1}, \ldots, a_{i_{|L(j)|}}]$ is the dependency list of a_j then it holds $i_1 > i_2 > \ldots i_{|L(j)|}$.*

An example of dependency graph generated by the function $g([a_1, \ldots, a_n])$ is illustrated in Figure 6.6 for $n = 4$ and $n = 5$.

Lemma 6.3.5. *Let $[a_1, \ldots, a_n]$ be an array of n integers. Then $g([a_1, \ldots, a_n])$ as defined above generates a DAG whose maximum element is marked with a_n.*

Lemma 6.3.6. *Let $[a_1, \ldots, a_n]$ be an array of n integers, $G = g([a_1, \ldots, a_n])$ be a DAG as defined above and $L(j) = [a_{i_1}, \ldots, a_{i_{|L(j)|}}]$ be the dependency list of a_j. Then Algorithm 6.7 computes $\Pi_{i=1}^{n} a_i$ and has a multiplicative depth of $\log(n)$.*

The proofs of Lemmas 6.3.5 and 6.3.6 follow by induction similar to Lemma 6.3.2. Before describing the improved path evaluation algorithm, we first extend our Node data structure by adding to it a new field representing a stack denoted dag, that stores

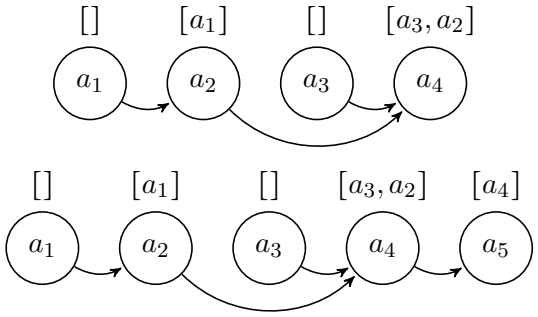


Figure 6.6: Dependency Graph for $n = 4$ and $n = 5$

```
1: for j = 1 to j = n do
2:     for l = 1 to l = |L(j)| do
3:         a_j ← a_j · a_{i_l}
```

Algorithm 6.7: Multiplication using Dependency Lists

the dependency list. Moreover, we group the nodes of the decision tree by level and use an array denoted $\mathsf{level}[]$, such that $\mathsf{level}[0]$ stores a pointer to the root and $\mathsf{level}[i]$ stores pointers to the child nodes of $\mathsf{level}[i-1]$ for $i \geq 1$. Now, we are ready to describe the improved path evaluation algorithm which consists of a pre-computation step and an online step.

The pre-computation is a one-time computation that depends only on the structure of the decision tree and requires no encryption. As described in Algorithm 6.8, its main function ComputeDag uses the leveled structure of the tree and the dependency graph defined above to compute the dependency list of each node in the tree (i.e., the DAG defined above). The sub-function AddEdge is used to actually add nodes to the dependency list of another node (i.e., by adding edges between these nodes in the DAG).

The online step is described in Algorithm 6.9. It follows the idea of Algorithm 6.7 by multiplying decision bit level-wise depending on the dependency lists. The correctness follows from Lemma 6.3.6.

6.4 Arithmetic Implementation

In this section, we describe Pdt-Int, an instantiation of the basic scheme that encodes the plaintexts such that the computation is done using an arithmetic circuit. This means that a ciphertext now encrypts an integer and that arithmetic operations are no longer mod 2, but mod 2^l for some $l > 2$.

6.4.1 Modified Lin-Tzeng Comparison Protocol

We first describe our modified version of the Lin-Tzeng comparison protocol [135]. The main idea of their construction is to reduce the greater-than comparison to the set intersection problem of prefixes. Let x_i and y_j be inputs of client and server, respectively, with the goal to compute $[x_i > y_j]$.

Require: integers up and low
Ensure: Computed $v.\mathsf{dag}$ for each $v \in \mathcal{D} \cup \mathcal{L}$
1: **function** ComputeDag($\mathsf{up}, \mathsf{low}$)
2: **if** $\mathsf{up} \geq \mathsf{low}$ **then**
3: **return** ▷ end the recursion
4: $\eta \leftarrow \mathsf{low} - \mathsf{up} + 1$
5: $\mathsf{mid} \leftarrow 2^{|\eta-1|-1} - 1 + \mathsf{up}$ ▷ $|\eta|$ bitlength of η
6: **for each** $v \in \mathsf{level}[\mathsf{low}]$ **do**
7: AddEdge($v, \mathsf{low}, \mathsf{mid}$)
8: **for** $i = \mathsf{mid} + 1$ **to** $\mathsf{low} - 1$ **do** ▷ non-deepest leaves
9: **for each** $v \in \mathsf{level}[i] \cap \mathcal{L}$ **do**
10: AddEdge(v, i, mid)
11: ComputeDag($\mathsf{up}, \mathsf{mid}$)
12: ComputeDag($\mathsf{mid} + 1, \mathsf{low}$)

Require: Node v, integers $\mathsf{currLvl}$ and $\mathsf{destLvl}$
Ensure: Updated $v.\mathsf{dag}$
1: **function** AddEdge(v, $\mathsf{currLvl}$, $\mathsf{destLvl}$)
2: $w \leftarrow v$
3: **while** $\mathsf{currLvl} > \mathsf{destLvl}$ **do**
4: $w \leftarrow w.\mathsf{parent}$
5: $\mathsf{currLvl} \leftarrow \mathsf{currLvl} - 1$
6: $v.\mathsf{dag.push}(w)$ ▷ dag is a stack

Algorithm 6.8: Pre-computation of Multiplication DAG

Require: set of nodes stored by level in array level
Ensure: Updated $v.\mathsf{cmp}$ for each $v \in \mathcal{L}$
1: **function** EvalPathsP
2: **for** $i = 1$ **to** d **do** ▷ from top to bottom level
3: **for each** $v \in \mathsf{level}[i]$ **do**
4: **while** $v.\mathsf{dag.empty}() = false$ **do** ▷ dag is a stack
5: $w \leftarrow v.\mathsf{dag.pop}()$
6: $[\![v.\mathsf{cmp}]\!] \leftarrow [\![v.\mathsf{cmp}]\!] \boxdot [\![w.\mathsf{cmp}]\!]$

Algorithm 6.9: Aggregate Decision Bits with precomputed DAG

Input Encoding. Let $\text{INT}(z_\eta \cdots z_1) = z$ be a function that takes a bit string of length η and parses it into the η–bit integer $z = \sum_{l=1}^{\eta} z_l \cdot 2^{l-1}$. The *0-encoding* $V_{x_i}^0$ and *1-encoding* $V_{x_i}^1$ of an integer input x_i are the following vectors: $V_{x_i}^0 = (v_{i\mu}, \cdots, v_{i1}), V_{x_i}^1 = (u_{i\mu}, \cdots, u_{i1})$, such that $\forall l \in \{1 \dots \mu\}$

$$v_{il} = \begin{cases} \text{INT}(x_{i\mu} x_{i\mu-1} \cdots x_{il'} 1) & \text{if } x_{il} = 0 \\ r_{il}^{(0)} & \text{if } x_{il} = 1 \end{cases}$$

$$u_{il} = \begin{cases} \text{INT}(x_{i\mu} x_{i\mu-1} \cdots x_{il}) & \text{if } x_{il} = 1 \\ r_{il}^{(1)} & \text{if } x_{il} = 0, \end{cases}$$

where $l' = l + 1$, and $r_{il}^{(0)}$, $r_{il}^{(1)}$ are random numbers of a fixed bitlength $\nu > \mu$ (e.g. $2^\mu \le r_{il}^{(0)}, r_{il}^{(1)} < 2^{\mu+1}$) with $LSB(r_{il}^{(0)}) = 0$ and $LSB(r_{il}^{(1)}) = 1$ (LSB is the least significant bit). If the INT function is used the compute the element at position l, then we call it a *proper encoded element* otherwise we call it a *random encoded element.* Note that a random encoded element $r_{il}^{(1)}$ at position l in the 1-encoding of x_i is chosen such that it is guaranteed to be different to a proper or random encoded element at position l in the 0-encoding of y_j, and vice versa. Hence, it enough if $r_{il}^{(1)}$ and $r_{il}^{(0)}$ are one or two bits longer than any possible proper encoding element at position l. Also note that the bit string $x_{i\mu} x_{i\mu-1} \cdots x_{il}$ is interpreted by the function INT as a bit string $z_{\mu-l+1} \cdots z_1$ with length $\mu - l + 1$ where $z_1 = x_{il}, z_2 = x_{i(l+1)}, \dots, z_{\mu-l+1} = x_{i\mu}$. If we see $V_{x_i}^0, V_{y_j}^1$ as sets, then $x_i > y_j$ if and only if they have exactly one common element.

For example, if $\mu = 3$, $x_i = 6 = 110_2$ and $y_j = 2 = 010_2$ then

$$V_{x_i}^1 = (\text{INT}(1), \text{INT}(11), r_{i1}^{(1)}) = (1, 3, r_{i1}^{(1)}),$$

$$V_{y_j}^0 = (\text{INT}(1), r_{j2}^{(0)}, \text{INT}(011)) = (1, r_{j2}^{(0)}, 3),$$

and $V_{x_i}^1 - V_{y_j}^0 = (0, r_2, r_1)$, where $r_2 = 3 - r_{j2}^{(0)}$ and $r_1 = r_{i1}^{(1)} - 3$ are random numbers. On the other hand, if $x_i = 2 = 010_2$ and $y_j = 6 = 110_2$ then $V_{x_i}^1 = (r_{i3}^{(1)}, 1, r_{i1}^{(1)})$, $V_{y_j}^0 = (r_{j3}^{(0)}, r_{j2}^{(0)}, 7)$ and $V_{x_i}^1 - V_{y_j}^0 = (r_3, r_2, r_1)$ where $r_3 = r_{i3}^{(1)} - r_{j3}^{(0)}$, $r_2 = 1 - r_{j2}^{(0)}$ and $r_1 = r_{i1}^{(1)} - 7$ are all random numbers.

Lemma 6.4.1. *Let x_i and y_j be two integers, then $x_i > y_j$ iff $V = V_{x_i}^1 - V_{y_j}^0$ has a unique position with 0.*

Proof. If $V = V_{x_i}^1 - V_{y_j}^0$ has a unique 0 at a position $l, (1 \le l \le \mu)$ then u_{il} and v_{il} have bit representation $z_{\mu-l+1} \cdots z_1$, where for each $h, \mu - l + 1 \ge h \ge 2$, $z_h = x_{ig} = x_{jg}$ with $g = l + h - 1$, and $z_1 = x_{il} = 1$ and $x_{jl} = 0$. It follows that $x_i > y_j$.
If $x_i > y_j$ then there exists a position l such that for each $h, \mu \ge h \ge l+1$, $x_{ih} = x_{jh}$ and $x_{il} = 1$ and $x_{jl} = 0$. This implies $u_{il} = v_{il}$.

For $h, \mu \ge h \ge l + 1$, either u_{ih} bit string is a prefix of x_i while v_{jh} is random, or u_{ih} is random while v_{jh} bit string is a prefix of y_j. From the choice of $r_{ih}^{(0)}$, $r_{ih}^{(1)}$, we have $u_{ih} \neq v_{ih}$.

For $h, l-1 \ge h \ge 1$ there are three cases: u_{ih} and v_{ih} (as bit string) are both prefixes of x_i and y_j, only one of them is a prefix, both are random. For the first case the difference of the bits at position l and for the other cases the choice of $r_{ih}^{(0)}$ imply that $u_{ih} \neq v_{ih}$. □

1: **function** LinCompare($[\![V^1_{x_i}]\!], [\![V^0_{y_j}]\!]$)
2: **parse** $[\![V^1_{x_i}]\!]$ **as** $[\![u_{i\mu}]\!], \ldots, [\![u_{i1}]\!]$
3: **parse** $[\![V^0_{y_j}]\!]$ **as** $[\![v_{i\mu}]\!], \ldots, [\![v_{i1}]\!]$
4: **for** $l := 1$ **to** μ **do**
5: **choose** a random r_l from the plaintext space
6: $c_l = [\![(u_{il} - v_{jl}) \cdot r_l]\!]$
7: **choose** a random permutation π
8: **return** $\pi(c_\mu, \cdots, c_1)$

Algorithm 6.10: Modified Lin-Tzeng Protocol Using AHE

The Protocol. Let $[\![V^0_{x_i}]\!] = [\![v_{i\mu}]\!], \ldots, [\![v_{i1}]\!]$ (respectively $[\![V^1_{x_i}]\!] = [\![u_{i\mu}]\!], \ldots, [\![u_{i1}]\!]$) denote the componentwise encryption of $V^0_{x_i}$ (resp. $V^1_{x_i}$). The client sends $[\![V^0_{x_i}]\!], [\![V^1_{x_i}]\!]$ to the server. To determine the comparison result for $x_i > y_j$, the server evaluates the function LinCompare($[\![V^1_{x_i}]\!], [\![V^0_{y_j}]\!]$) (Algorithm 6.10) which returns μ ciphertexts among which exactly one encrypts zero if an only if $x_i > y_j$.

Algorithm LinCompare (Algorithm 6.10) requires only AHE and would be interactive. For the decision tree evaluation, the server omits the randomization in Step 6 and the random permutation in Step 7, since this is not the final result. Moreover, the server collects the difference ciphertexts c_l in an array and uses the multiplication algorithm with logarithmic multiplicative depth. Hence, we will instead use LinCompareDT as described in Algorithm 6.11.

Difference to the original protocol. In contrast to the original protocol of Lin and Tzeng [135], we note the following differences:

- Additively HE instead of multiplicative: As explained above multiplication increases the noise significantly while addition increases it slightly.
- The Int function: Instead of relying on a collision-free hash function as Lin and Tzeng [135], we use the Int function which is simpler to implement and more efficient as it produces smaller values.
- The choice of random encoded elements $r^{(0)}_{il}$, $r^{(1)}_{il}$: We choose the random encoded elements as explained above and encrypt them, while the original protocol uses ciphertexts chosen randomly in the ciphertext space.
- Encrypting the encodings on both sides: In the original protocol, the evaluator has access to y_j in plaintext and does not need to choose randomly encoded elements. By encoding as explained in our modified version, we can encrypt both encodings and delegate the evaluation to a third party which is not allowed to have access to the inputs in plaintext.
- Aggregation: The multiplication of the ciphertexts returned by Algorithm 6.10 returns a ciphertext encrypting either 0 or a random number.

The modified comparison algorithm as used for PDTE is illustrated in Algorithm 6.11. Note that, this can be computed using binary gates as well, by encrypting the 0/1-encodings binary-wise resulting in μ blocks of ciphertexts, computing XOR-gates in parallel for each block, then computing OR-gates in parallel for each block and finally summarizing the results using AND-gates. The multiplicative depth will be 2μ.

1: **function** LinCompareDT($[\![V^1_{x_i}]\!], [\![V^0_{y_j}]\!]$)
2: **parse** $[\![V^1_{x_i}]\!]$ **as** $[\![u_{i\mu}]\!], \ldots, [\![u_{i1}]\!]$
3: **parse** $[\![V^0_{y_j}]\!]$ **as** $[\![v_{i\mu}]\!], \ldots, [\![v_{i1}]\!]$
4: **let** arr be an empty array of size μ
5: **for** $l \coloneqq 1$ **to** μ **do**
6: $arr[l] \leftarrow [\![u_{il} - v_{jl}]\!]$
7: **return** EvalMul$(1, \mu, arr)$

Algorithm 6.11: Modified Lin-Tzeng Protocol for PDTE Using FHE

6.4.2 Arithmetic Decision Tree Evaluation

In this section, we use the modified Lin-Tzeng comparison explained above for the decision tree evaluation. We follow the structure of the basic protocol as described in Protocol 6.4.

Encoding and Encrypting Input. The protocol starts with the client encoding, encrypting and sending its input to the server. For each attribute value x_i the client sends the encryptions $[\![V^0_{x_i}]\!] = ([\![v_{i\mu}]\!], \ldots, [\![v_{i1}]\!])$ and $[\![V^1_{x_i}]\!] = ([\![u_{i\mu}]\!], \ldots, [\![u_{i1}]\!])$ of the 0-encoding $V^0_{x_i}$ and 1-encoding $V^1_{x_i}$ of x_i. We let

$$[\![V_x]\!] = [([\![V^0_{x_1}]\!], [\![V^1_{x_1}]\!]), \ldots ([\![V^0_{x_n}]\!], [\![V^1_{x_n}]\!])]$$

denote the encoding and encryption of all attribute values in x.

The server does the same with the threshold values. For each threshold value y_j the server computes the encryptions $[\![V^0_{y_j}]\!] = [\![v_{j\mu}]\!], \ldots, [\![v_{j1}]\!]$ and $[\![V^1_{y_j}]\!] = [\![u_{j\mu}]\!], \ldots, [\![u_{j1}]\!])$ of the 0-encoding $V^0_{y_j}$ and 1-encoding $V^1_{y_j}$ of y_j. We let

$$[\![V_{\mathcal{D}}]\!] = \{([\![V^0_{y_j}]\!], [\![V^1_{y_j}]\!]) | v \in \mathcal{D} \wedge v.\mathsf{threshold} = y_j\}.$$

denote the encoding and encryption of all threshold values in $\mathcal{D}$. This computation is illustrated by Encode(x) and Encode($\mathcal{D}$) in Protocol 6.15. Note that, this is still compatible with the trusted *randomizer* technique, where we will use sequences of random integers.

Evaluating Decision Node. The server evaluates the comparison at each decision node. For each decision node $v \in \mathcal{D}$, let $v.\mathsf{threshold} = y_j$ and $i = v.\mathsf{aIndex}$. We assume that $x_i \neq y_j$ for all i, j. The parties can ensure this by having the client adding a bit 0 to the bit representation of each x_i, and the server adding a bit 1 to the bit representation of each y_j before encoding the values. Then from the definition of the tree evaluation, we move to the right if $[x_i \geq y_j]$ or the left otherwise. This is equivalent of testing $[x_i > y_j]$ or $[y_j > x_i]$, since we assume $x_i \neq y_j$. Therefore, for each decision node y_j with corresponding attribute x_i, the server uses LinCompareDT$([\![V^1_{x_i}]\!], [\![V^0_{y_j}]\!])$ to mark the edge right to y_j (i.e., store it at the right child node of v) and LinCompareDT$([\![V^1_{y_j}]\!], [\![V^0_{x_i}]\!])$ to mark the edge left to y_j (i.e., store it at the left child node of v). It is illustrated in Algorithm 6.12.

Aggregating Decision Results. Then for each leaf node $v \in \mathcal{L}$, the server aggregates the comparison results along the path from the root to v. As a result of

```
1: function EvalDnode(𝒟, [[V_𝒟]], [[V_x]])
2:     for each v ∈ 𝒟 do
3:         let y_j ← v.threshold and i ← v.aIndex
4:         [[v.right.cmp]] ← LinCompareDT([[V^1_{x_i}]], [[V^0_{y_j}]])
5:         [[v.left.cmp]] ← LinCompareDT([[V^1_{y_j}]], [[V^0_{x_i}]])
```

Algorithm 6.12: Computing Decision Bits

```
1: function EvalPaths(𝒟, ℒ)
2:     for each v ∈ ℒ do
3:         let P_v be the array of nodes on the path (root → v)
4:         [[cost_v]] ← [[0]]
5:         for each u ∈ P_v do
6:             [[cost_v]] ← [[cost_v]] ⊞ [[u.cmp]]
```

Algorithm 6.13: Aggregating Decision Bits

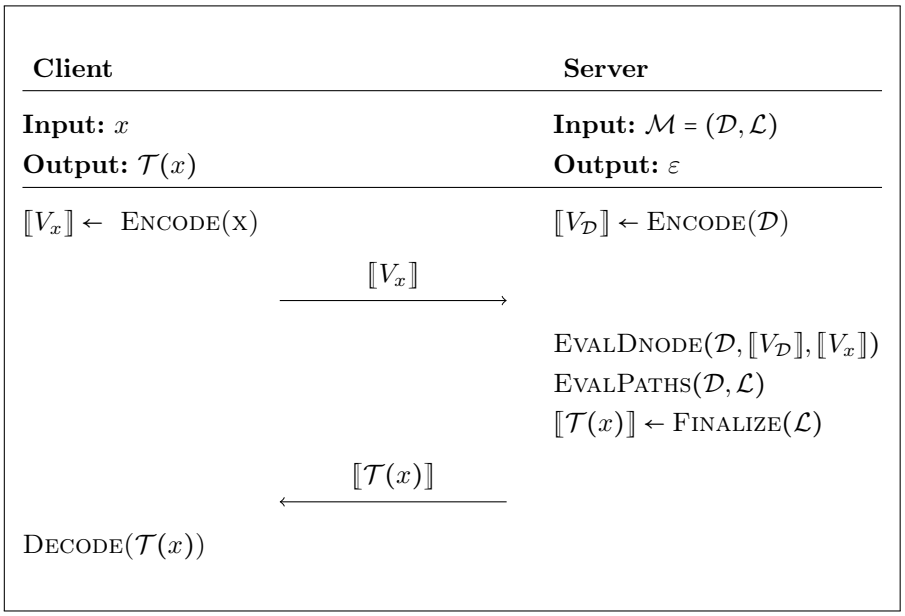
Algorithm 6.12, one edge of each decision node is marked with a ciphertext of 0, while the other will be marked with a ciphertext of a random plaintext. It follows that the sum of marks along each path of the tree, will result to an encryption of 0 for the classification path and an encryption of a random plaintext for other paths. This computation is illustrated in Algorithm 6.13.

Finalizing. To reveal the final result to the client, we do the following. For each ciphertext $[\![cost_v]\!]$ of Algorithm 6.13, the server chooses a random number r_v, computes $[\![result_v]\!] \leftarrow [\![cost_v \cdot r_v + v.\mathsf{cLabel}]\!]$ and sends the resulting ciphertexts to the client in a random order. This is illustrated in Algorithm 6.14. As a result, the server sends back $\mathcal{O}\left(2^d\right)$ ciphertexts (in the worse case $|\mathcal{L}| = 2^d$). Alternatively, the server can make a trade-off between communication and computation by using the shift operation to pack many $result_v$ in a single ciphertext. This would require additionally $\mathcal{O}\left(2^d\right)$ cryptographic operations, but reduce the communication cost to $\mathcal{O}\left(\left\lceil \frac{2^d}{s} \right\rceil\right)$, where s is the number of slots.

Putting It All Together. The whole protocol is illustrated in Protocol 6.15. The client encrypts 0-encodings and 1-encodings of its input and sends the ciphertexts to the Server. The whole computation is performed by the server. It sequentially computes the algorithms described above and sends the resulting ciphertexts to the

```
1: function Finalize(ℒ)
2:     for each v ∈ ℒ do
3:         choose a random number r_v
4:         [[result_v]] ← [[cost_v · r_v + v.cLabel]]
```

Algorithm 6.14: Finalizing



Protocol 6.15: Overview of Protocol Pdt-Int

client. The client then decrypts and decodes the outputs resulting in the classification label. The algorithms are straightforward and easy to understand. However, ciphertext packing and parallelization can improve the performance of the protocol.

6.4.3 Optimization

We describe in the following how ciphertext packing can be used to improve computation and communication. Ciphertext packing means that each ciphertext encrypts s bits, where s is the number of slots in the ciphertext. Then we can use this property in three different ways. First, one could pack each encoding (0 or 1) of each value (attribute or threshold) in a single ciphertext allowing the server to compute the difference in Step 6 of Algorithm 6.11 with one homomorphic operation (instead of μ). Second, one could encrypt several attributes together and classify them with a single protocol evaluation. Finally, one could encrypt multiple decision node thresholds that must be compared to the same attribute in the decision tree model.

Packing 0-Encodings and 1-Encodings. Recall that our modified Lin-Tzeng comparison requires only component-wise subtraction and a multiplication of all components. Therefore, the client can pack the 0-encoding of each x_i in one ciphertext and sends $[\![v_{i\mu}|\dots|v_{i1}|0|\dots|0]\!]$ instead of $[\![V^0_{x_i}]\!]$ (and similar for the 1-encoding). Then the server does the same for each threshold value and evaluates the decision node by computing the differences $[\![d_{ij}]\!] \leftarrow [\![u_{i\mu} - v_{j\mu}|\dots|u_{i1} - v_{j1}|0|\dots|0]\!]$ with one homomorphic subtraction. To multiply the μ relevant components in $[\![d_{ij}]\!]$, we use $|\mu|$ (bitlength of μ) left shifts and $|\mu|$ multiplications to shift $\Pi^{\mu}_{l=1}(u_{il} - v_{jl})$ to the first slot. The path evaluation and the computation of the result's ciphertext remain as explained above.

Packing Attribute Values. Let $x^{(1)}, \dots, x^{(s)}$ be s possible attribute vectors with $x^{(l)} = [x^{(l)}_1, \dots, x^{(l)}_n]$, $1 \le l \le s$. For each $x^{(l)}_i$, let $V^{0(l)}_{x_i} = v^{(l)}_{i\mu}, \dots, v^{(l)}_{i1}$ be the 0-encoding.

The client generates for each attribute x_i the ciphertexts $[\![cx^0_{i\mu}]\!], \ldots, [\![cx^0_{i2}]\!], [\![cx^0_{i1}]\!]$ as illustrated in Equation 6.6. The client does similarly for the 1-encodings.

$$\begin{aligned} [\![cx^0_{i1}]\!] &= [\![v^{(1)}_{i1} | v^{(2)}_{i1} | \ldots | v^{(s)}_{i1}]\!] \\ [\![cx^0_{i2}]\!] &= [\![v^{(1)}_{i2} | v^{(2)}_{i2} | \ldots | v^{(s)}_{i2}]\!] \\ &\ldots \\ [\![cx^0_{i\mu}]\!] &= [\![v^{(1)}_{i\mu} | v^{(2)}_{i\mu} | \ldots | v^{(s)}_{i\mu}]\!] \end{aligned} \qquad \text{Packing of multiple } V^0_{x_i} \tag{6.6}$$

To shorten the notation, let y_j denote the threshold of j-th decision node (i.e., $y_j = v_j.\mathsf{threshold}$) and let $V^0_{y_j} = v_{j\mu}, \ldots, v_{j1}$ be the corresponding 0-encoding. The server encrypts $V^0_{y_j}$ as illustrated in Equation 6.7. The 1-encoding is encrypted similarly.

$$\begin{aligned} [\![cy^0_{j1}]\!] &= [\![v_{j1} | v_{j1} | \ldots | v_{j1}]\!] \\ [\![cy^0_{j2}]\!] &= [\![v_{j2} | v_{j2} | \ldots | v_{j2}]\!] \\ &\ldots \\ [\![cy^0_{j\mu}]\!] &= [\![v_{j\mu} | v_{j\mu} | \ldots | v_{j\mu}]\!] \end{aligned} \qquad \text{Packing of a single } V^0_{y_j} \tag{6.7}$$

The above described encoding allows comparing s attribute values together with one threshold. This is possible because the routine LinCompareDT() is compatible with SVCP such that we have:

$$\begin{aligned} \textsc{LinCompareDT}(([\![cx^1_{i\mu}]\!], \ldots, [\![cx^1_{i1}]\!]), ([\![cy^0_{j\mu}]\!], \ldots, [\![cy^0_{j1}]\!])) = \\ [\![b^{(1)}_{ij} | b^{(2)}_{ij} | \ldots | b^{(s)}_{ij}]\!], \end{aligned} \tag{6.8}$$

where $b^{(l)}_{ij} = 0$ if $x^{(l)}_i > y_j$ and $b^{(l)}_{ij}$ is a random number otherwise. This results in a single ciphertext such that the l-th slot contains the comparison result between $x^{(l)}_i$ and y_j.

Aggregating decision bits remains unchanged as described in Algorithm 6.13. This results in a packed ciphertext $[\![b_v]\!] = [\![b^{(1)}_v | \ldots | b^{(s)}_v]\!]$ for each leaf $v \in \mathcal{L}$, where $b^{(l)}_v = 0$ if $x^{(l)}$ classifies to leaf v and $b^{(l)}_u$ is a random number for any other leaf $u \in \mathcal{L} - \{v\}$. Algorithm 6.14 remains unchanged as well.

Packing Threshold Values. In this case, the client encrypts a single attribute in one ciphertext, while the server encrypts multiple threshold values in a single ciphertext. Hence, for an attribute value x_i, the client generates the ciphertexts as in Equation 6.9 for the 0-encoding and handles the 1-encoding similarly.

$$\begin{aligned} [\![cx^0_{i1}]\!] &= [\![v_{i1} | v_{i1} | \ldots | v_{i1}]\!] \\ [\![cx^0_{i2}]\!] &= [\![v_{i2} | v_{i2} | \ldots | v_{i2}]\!] \\ &\ldots \\ [\![cx^0_{i\mu}]\!] &= [\![v_{i\mu} | v_{i\mu} | \ldots | v_{i\mu}]\!] \end{aligned} \qquad \text{Packing of a single } V^0_{x_i} \tag{6.9}$$

Let m_i be the number of decision nodes that compare to the attribute x_i (i.e., $m_i = |\{v_j \in \mathcal{D} : v_j.\mathsf{aIndex} = i\}|$). The server packs all corresponding threshold values in $\left\lceil \frac{m_i}{s} \right\rceil$ ciphertext(s) as illustrated in Equation 6.10 for the 0-encoding and handles the 1-encoding similarly.

$$\begin{aligned}
[\![cy^0_{j1}]\!] &= [\![v_{j_1 1}|\ldots|v_{j_{m_i} 1}|\ldots]\!] \\
[\![cy^0_{j2}]\!] &= [\![v_{j_1 2}|\ldots|v_{j_{m_i} 2}|\ldots]\!] \\
&\ldots \\
[\![cy^0_{j\mu}]\!] &= [\![v_{j_1 \mu}|\ldots|v_{j_{m_i} \mu}|\ldots]\!]
\end{aligned} \qquad \text{Packing of multiple } V^0_{y_j} \tag{6.10}$$

The packing of threshold values allows comparing one attribute value against multiple threshold values together. Unfortunately, we do not have access to the slot while performing homomorphic operations. Hence, to aggregate the decision bits, we make m_i copies of the resulting packed decision results and shift left each decision result to the first slot. Then the aggregation of the decision results and the finalizing algorithm work as in the previous case with the only difference that only the result in the first slot matters and the remaining slots can be set to 0.

6.5 Security Analysis

We briefly discuss the correctness and the security of our construction.

Correctness. The correctness for the basic scheme follows directly from Lemma 6.2.3. For the binary implementation, we proved with Lemmas 6.3.2, 6.3.5, 6.3.6 that aggregating the paths using Algorithms 6.5 and 6.9 is correct. For the integer implementation, Lemma 6.4.1 ensures the correctness of the comparison. The classification path is marked with 0 on all edges while the other paths are marked with at least one random number. As a result, summing up the marks along the paths returns 0 for the classification path and a random number for all other paths.

Security. It is straightforward to see that our protocols are secure. There is no interaction with the client during the computation and a semi-honest server sees only IND-CPA ciphertexts. A semi-honest client only learns the encryption of the result (and additional encryptions of random elements for Pdt-Int). A malicious server can only return a false classification result. This is inherent to private function evaluation where the function (the decision tree in our case) is an input to the computation. A malicious client can send a too "noisy" ciphertext, such that after the computation at the server a correct decryption is not possible, leaking some information. This attack works only with level FHE and is easy to deal with, namely the computation of a ciphertext capacity is a public function which the server can use to check the ciphertexts before starting the computation. Therefore, we state the following:

Theorem 6.5.1. *Our protocols correctly and securely implement the PDTE functionality* $\mathcal{F}_{\mathsf{PDTE}}$.

As Pdt-Bin returns the bit representation of the resulted classification label whose bitlength is public (i.e., the set of possible classification labels is known to the client), there is no leakage beyond the final output. Pdt-Int returns as many ciphertexts as there are leaves and, therefore, leaks the number of decision nodes.

6.6 Complexity Analysis

We now analyze the complexity of our scheme, distinguishing between the binary and the integer implementations. In the following, we assume that the decision tree

is a complete tree with depth d. The case for sparse trees is similar by using the corresponding number of decision nodes in the tree.

6.6.1 Complexity of the Binary Implementation

The SHE comparison circuit has multiplicative depth $|\mu-1|+1$ and requires $\mathcal{O}(\mu\cdot|\mu|)$ multiplications [45, 46, 47]. That is, the evaluation of all decision nodes requires $\mathcal{O}\big(2^d\mu\cdot|\mu|\big)$ multiplications. The path evaluation has a multiplicative depth of $|d-1|$ and requires for all 2^d paths $\mathcal{O}\big(d2^d\big)$ multiplications. The evaluation of the leaves has a multiplicative depth of 1 and requires in total 2^d multiplications. The total multiplicative depth for Pdt-Bin is, therefore, $|\mu-1|+|d-1|+2\approx|\mu|+|d|+2$ while the total number of multiplications is $\mathcal{O}\big(2^d\mu\cdot|\mu|+d2^d+2^d\big)\approx\mathcal{O}\big(d2^d\big)$.

For the label packing, the bit representation of each classification label is packed in one ciphertext. This holds for the final result as well. As a result, if the tree is complete and all classification labels are distinct, then the server sends $\lceil\frac{d}{s}\rceil$ ciphertext(s) to the client. In practice, however, $\lceil\frac{d}{s}\rceil=1$ holds as d is smaller than the number s of slots.

For threshold packing, the decision bit at node v will be encrypted as $[\![b_v|0|...|0]\!]$. Then if we encrypt the classification label $c_i=c_{i|k|}...c_{i1}$ as $[\![c_{i|k|}|0|...|0]\!],...,[\![c_{i1}|0|...|0]\!]$, the final result c_l will be encrypted similarly such that with extra shifts, we can build the ciphertext $[\![c_{l|k|}|...|c_{l1}|0|...|0]\!]$. As a result, the server sends only 1 ciphertext back to the client.

For other cases (e.g., attribute packing, or no packing at all as in the current implementation of TFHE), the bits of a classification label are encrypted separately which holds for the final result as well. As a result, the server sends back d ciphertexts to the client.

6.6.2 Complexity of the Integer Implementation

The modified Lin-Tzeng comparison circuit has multiplicative depth $|\mu-1|$ and requires $\mathcal{O}(\mu-1)$ multiplications. As a result, the evaluation of all decision nodes requires $\mathcal{O}\big((\mu-1)2^d\big)$ multiplications. In Pdt-Int, the path evaluation does not requires any multiplication. However, the leave evaluation has a multiplicative depth of 1 and requires in total 2^d multiplications. The total multiplicative depth for Pdt-Int is, therefore, $|\mu-1|+1\approx|\mu|+1$ while the total number of multiplications is $\mathcal{O}\big((\mu-1)2^d+2^d\big)\approx\mathcal{O}\big(2^d\big)$.

For Pdt-Int, it is not possible to aggregate the leaves as in Pdt-Bin. If the client is classifying many inputs, the server must send 2^d ciphertexts back. If the client is classifying only one input, then the server can use shifts to pack the result in $\lceil\frac{2^d}{s}\rceil$ ciphertext(s).

6.7 Evaluation

In this section, we discuss some implementation details and evaluate our schemes.

6.7.1 Implementation Details

We implemented our algorithms using HElib [100] and TFHE [51, 52]. HElib is a C++ library that implements FHE. The current version includes an implementation of the leveled FHE BGV scheme [36]. HElib also includes various optimizations that make FHE runs faster, including the Smart-Vercauteren ciphertext packing (SVCP) techniques [173].

TFHE is a C/C++ library that implements FHE proposed by Chillotti et al. [48, 49]. It allows evaluating any Boolean circuit on encrypted data. The current version implements a very fast gate-by-gate bootstrapping, i.e., bootstrapping is performed after each gate evaluation. Future versions will include leveled FHE and ciphertext packing as described by Chillotti et al. [50]. Dai and Sunar [60, 61] propose an implementation of TFHE on CUDA-enabled GPUs that is 26 times faster.

We evaluated our implementation on an AWS instance with Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz running Ubuntu 18.04.2 LTS. The Instance has 36 CPUs, 144 GB Memory, and 8 GB SSD. As the bottleneck of our scheme is the overhead of the homomorphic computation, we focus on the computation done by the server. We start by generating appropriate encryption parameters and evaluating the performance of basic operations.

6.7.2 Basic Operations

Recall that FHE schemes – as considered in this thesis – are usually defined over a ring $\mathbb{Z}[X]/(X^N + 1)$ and that the encryption scheme might be a leveled FHE with parameter L. For HElib, the parameters N and L determines how to generate encryption keys for a security level λ which is at least 128 in all our experiments. The degree of the ring polynomial in HElib is not necessarily a power of 2. The ring polynomial is chosen among the cyclotomic polynomials. For HElib, we abuse the notation and use N to denote the N-th cyclotomic polynomial. Given the value of L and other parameters, the HElib function $\mathsf{FindM}(\cdot)$ computes the N-th cyclotomic polynomial, that guarantees a security level at least equal to a given security parameter λ. Table 6.3 summarizes the parameters we used for key generation and the resulting sizes for encryption keys and ciphertexts. We will refer to it as *homomorphic context* or just *context*. For HElib, one needs to choose L large enough than the depth of the circuit to be evaluated and then computes an appropriate value for N that ensures a security level at least 128. We experimented with three different contexts $(\text{HElib}_{\text{small}}, \text{HElib}_{\text{med}}, \text{HElib}_{\text{big}})$ for the binary representation used in Pdt-Bin and the context $\text{HElib}_{\text{int}}$ for the integer representation used in Pdt-Int. For TFHE, the default value of N is 1024 and the security level can be chosen up to 128 while L is infinite because of the gate-by-gate bootstrapping. We used the context TFHE_{128} to evaluate Pdt-Bin with TFHE. Table 6.4 reports the average runtime for encryption and decryption over 100 runs. The columns "Enc Vector" and "Dec Vector" stand for encryption and decryption using SIMD encoding and decoding, which is not supported by TFHE yet.

6.7.3 Homomorphic Operations in HElib

The opposite of the notion of *ciphertext noise* is the notion of *ciphertext capacity* or just *capacity* which is also determined by L and estimates the *capacity* of a ciphertext to be used in homomorphic operations. In Figure 6.16 and 6.17, we reported the remaining capacity of a ciphertext after a number of consecutive additions or multiplications starting from the values of L in Table 6.3. They show that the capacity is reduced only slightly after addition, but significantly after multiplication. Note that the encryption operation already has an impact on L. Figure 6.18 shows that doing the multiplication with logarithmic depth (Lemma 6.3.2) reduced the capacity sublinearly instead of linearly as in Figure 6.17. The sublinear complexity is also illustrated in Figure 6.19 for the comparison circuit which also has a logarithmic multiplicative depth [45, 46, 47]. Figure 6.20 illustrates runtimes (best and average) for addition and multiplication

Name	L	N	λ (bits)	**Slots**	sk (MB)	pk (MB)	**Ctxt** (MB)
$\text{HElib}_{\text{small}}$	200	13981	151	600	52.2	51.6	1.7
$\text{HElib}_{\text{med}}$	300	18631	153	720	135.4	134.1	3.7
$\text{HElib}_{\text{big}}$	500	32109	132	1800	370.1	367.1	8.8
$\text{HElib}_{\text{int}}$	450	24793	138.161	6198	370.1	367.1	8.8
TFHE_{128}	∞	1024	128	1	82.1	82.1	0.002

Table 6.3: Key Generation's Parameters and Results: For HElib, the value in the column N is not the degree of the ring polynomial, but the N-th cyclotomic polynomial. It is computed in HElib using a function called $\mathsf{FindM}(\cdot)$.

HElib Context	Enc Single (ms)	Enc Vector (ms)	Dec Single (ms)	Dec Vector (ms)
$\text{HElib}_{\text{small}}$	59.21	59.41	26.08	26.38
$\text{HElib}_{\text{med}}$	124.39	124.93	54.31	54.92
$\text{HElib}_{\text{big}}$	283.49	284.31	127.11	128.32
$\text{HElib}_{\text{int}}$	323.41	488.77	88.63	93.50
TFHE_{128}	0.04842	n/a	0.00129	n/a

Table 6.4: Encryption/Decryption Runtime

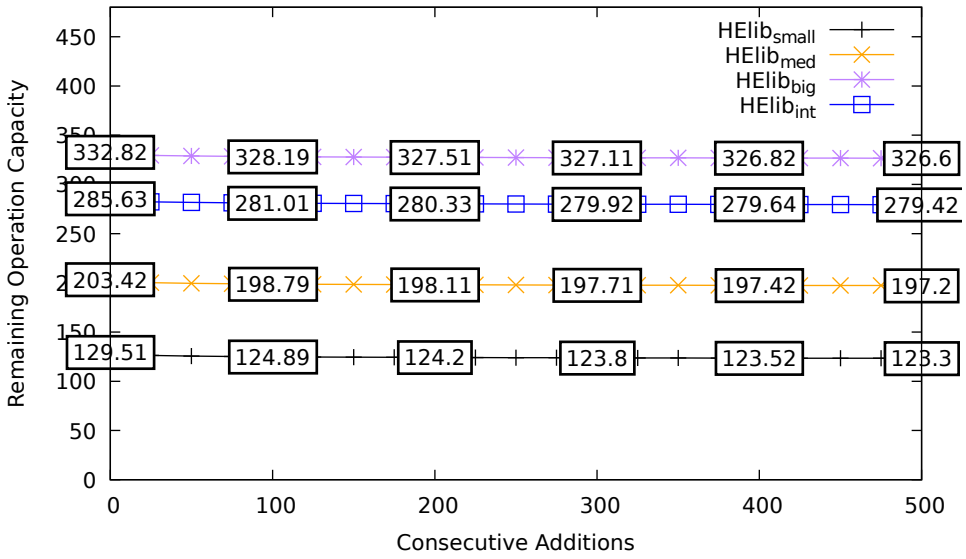


Figure 6.16: Capacity after consecutive additions

in HElib over 100 runs showing that homomorphic addition is really fast compared to multiplication. We report the runtime for the comparison circuit in Figure 6.21 comparing runtime for HElib and TFHE. While both are linear in the bitlength, the runtime for HElib increases very quickly.

6.7.4 Homomorphic Operations in TFHE

As already mentioned earlier, the current version of TFHE only supports binary gates. According to Chillotti et al. [49, 51], gate bootstrapping and gate evaluation cost about 13 ms for all binary gates except for the MUX gate, which costs 26 ms on a modern processor. For a full list of available gates, we refer to Chillotti et al. [52]. In Table 6.5, we illustrate the runtime of TFHE's gate evaluation with our testbed. The figures are given as average over 1000 runs.

6.7.5 Performance of the Binary Implementation

In this section, we report on our experiment with PDT-BIN on complete trees. Recall that for FHE supporting SIMD, we can use attribute values packing that allows evaluating many attribute vectors together. We, therefore, focus on attribute packing to show the advantage of SIMD. Figure 6.22 illustrates the amortized runtime of PDT-BIN with HElib. That is, the time of one PDTE evaluation divided by the number of slots provided by the used homomorphic context. As one can expect, the runtime clearly depends on the bitlength of the attribute values and the depth of the tree. The results show a clear advantage of HElib when classifying large data sets. For paths aggregation, we proposed EVALPATHSE (Algorithm 6.5) and EVALPATHSP (Algorithm 6.9). Figure 6.23 illustrates PDT-BIN runtime using these algorithms in a multi-threaded environment and shows a clear advantage of EVALPATHSP which will be used in the remaining experiments with PDT-BIN. Figure 6.24 illustrates the runtime of PDT-BIN with HElib$_{med}$ showing that the computation cost is dominated by the computation of decision bits which involves homomorphic evaluation of comparison circuits. In Figure 6.25, we report the evaluation of PDT-BIN using TFHE,

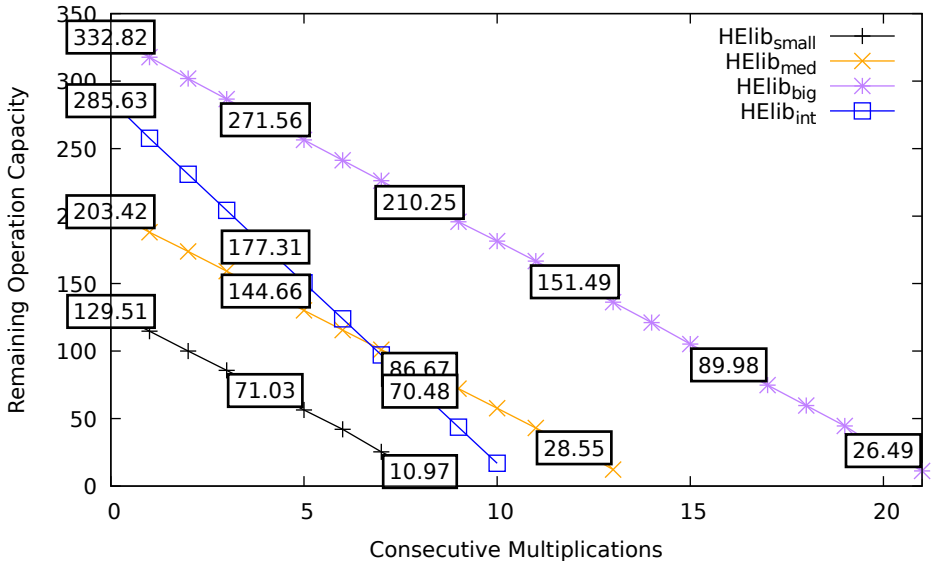


Figure 6.17: Capacity after consecutive multiplications

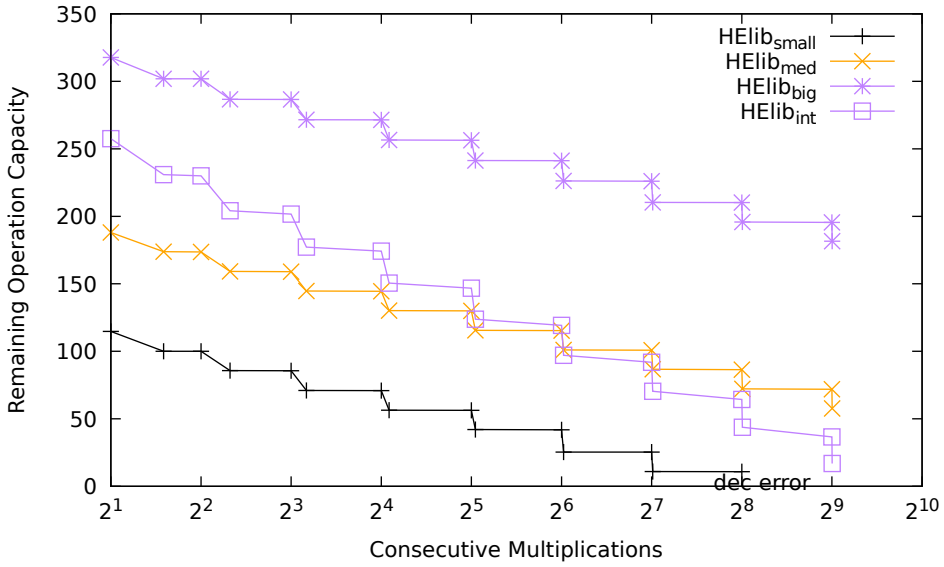


Figure 6.18: Capacity after multiplication with log Depth

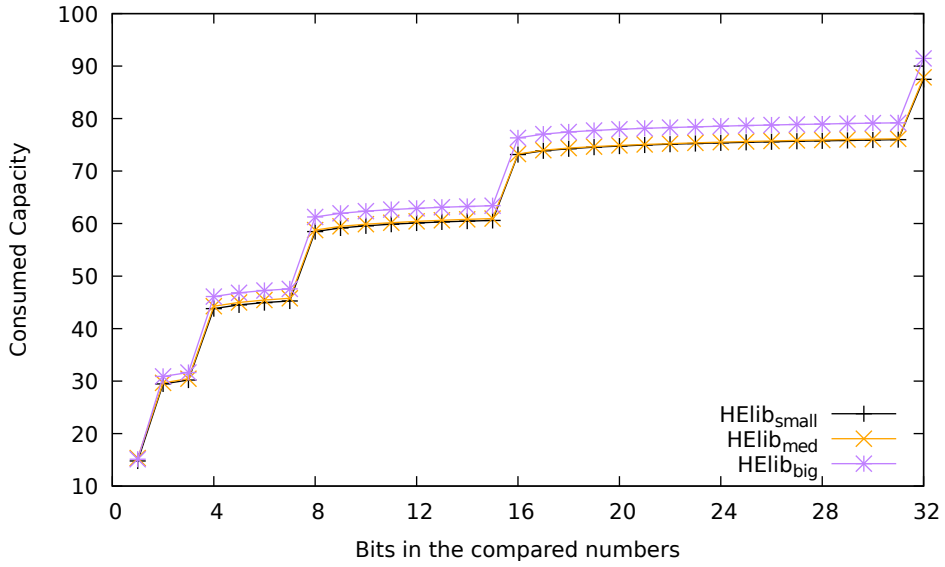


Figure 6.19: Comparison Capacity Consumption in HElib

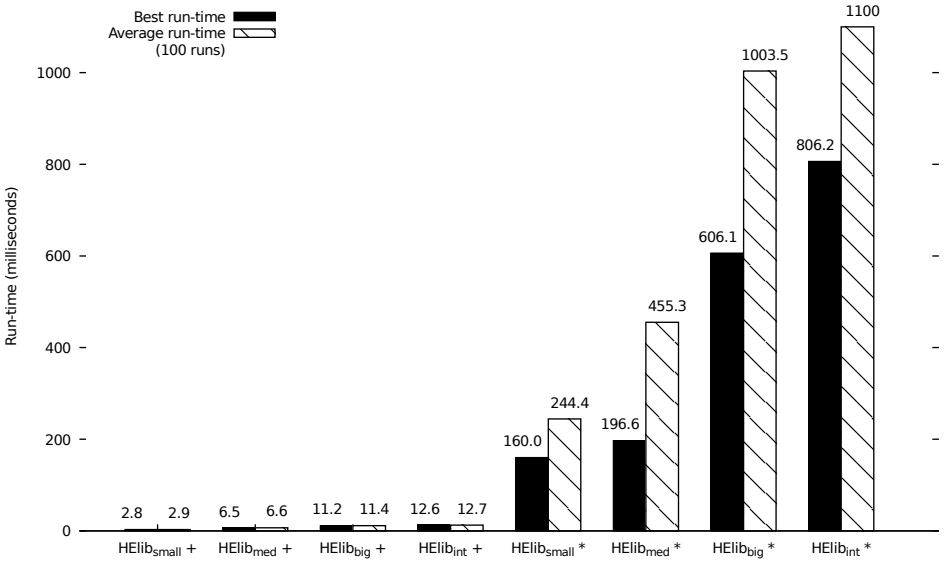


Figure 6.20: Runtime for Addition and Multiplication in HElib

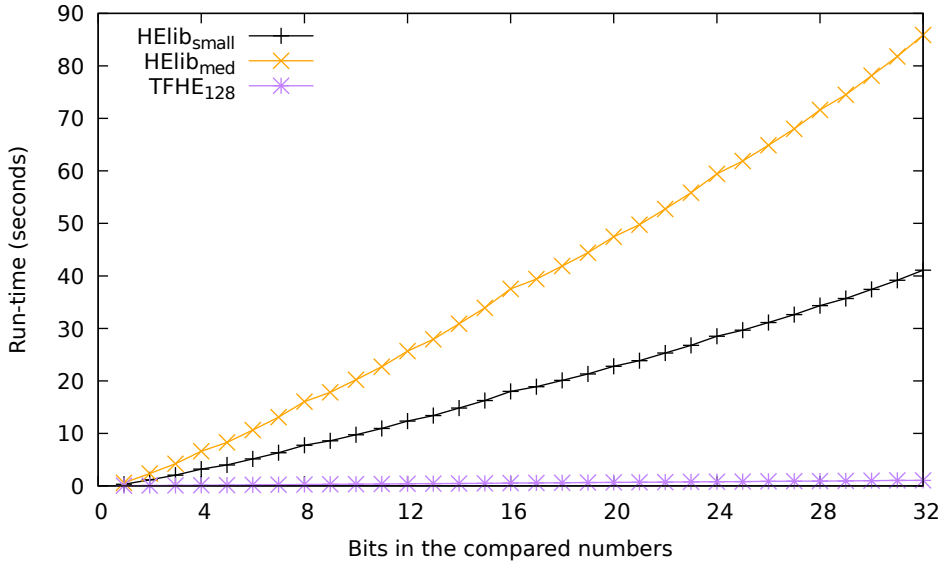


Figure 6.21: Comparison Run-time Cost

Gate Name	Gate Functionality	Run-time (ms) 128-bit security
CONSTANT	$result = encode(int)$	0.00052
NOT	$result = \neg a$	0.00051
COPY	$result = a$	0.00035
NAND	$result = \neg(a \wedge b)$	11.32751
OR	$result = a \vee b$	11.40669
AND	$result = a \wedge b$	11.38739
XOR	$result = a + b \bmod 2$	11.39326
XNOR	$result = (a = b)$	11.39418
NOR	$result = \neg(a \vee b)$	11.39813
ANDNY	$result = \neg a \wedge b$	11.39255
ANDYN	$result = a \wedge \neg b$	11.39737
ORNY	$result = \neg a \vee b$	11.40777
ORYN	$result = a \vee \neg b$	11.39940
MUX	$result = a?b : c$	21.29517

Table 6.5: TFHE Binary Bootstrapping Gates

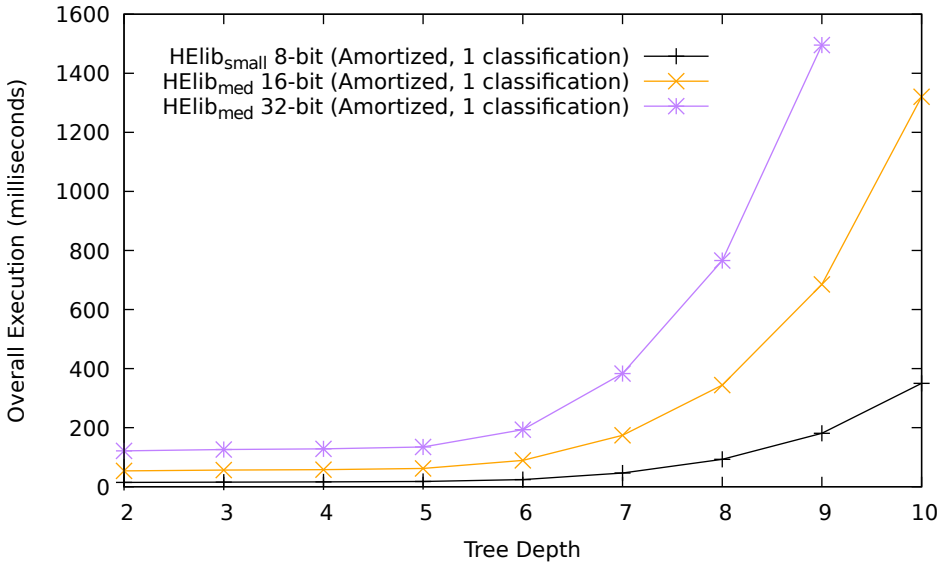


Figure 6.22: Amortized Pdt-Bin Runtime with HElib

which shows a clear advantage compare to HElib. For the same experiment with 72 threads, TFHE evaluates a complete tree of depth 10 and 64-bit input in less than 80 seconds, while HElib takes about 400 seconds for 16-bit input. Recall that, a CUDA implementation [60, 61] of TFHE can further improve the time of Pdt-Bin using TFHE.

6.7.6 Performance of our Schemes on Real Datasets

We also performed experiments on real datasets from the UCI repository [185]. We performed experiments for both Pdt-Bin and Pdt-Int for the datasets illustrated in Table 6.6 (parameters n, d, m are defined in Table 2.2). For Pdt-Bin, we reported the costs for HElib (single and amortized) and the costs for TFHE. Since TFHE evaluates only Boolean circuits, we only have implementation and evaluation of Pdt-Int with HElib. We also illustrate in Table 6.7 the costs of two best previous works that rely only on homomorphic encryption, whereby the figures are taken from the respective papers [145, 175]. For one protocol run, Pdt-Bin with TFHE is much faster than Pdt-Bin with HElib which is also faster than Pdt-Int with HElib. However, because of the large number of slots, the amortized cost of Pdt-Bin with HElib is better. For 16-bit inputs, our amortized time with HElib and our time with TFHE outperform XCMP [145] which used 12-bit inputs. For the same input bitlength, XCMP is still much better than our one run using HElib, since the multiplicative depth is just 3. However, our schemes still have a better communication and Pdt-Bin has no leakage. While the scheme of Tai et al. [175] in the semi-honest model has a better time for 64-bit inputs than our schemes for 16-bit inputs, it requires a fast network communication and at least double cost in the malicious model. The efficiency of Tai et al. is in part due to their ECC implementation of the lifted ElGamal [76], which allows a fast runtime and smaller ciphertexts, but is not secure against a quantum attacker, unlike lattice-based FHE as used in our schemes.

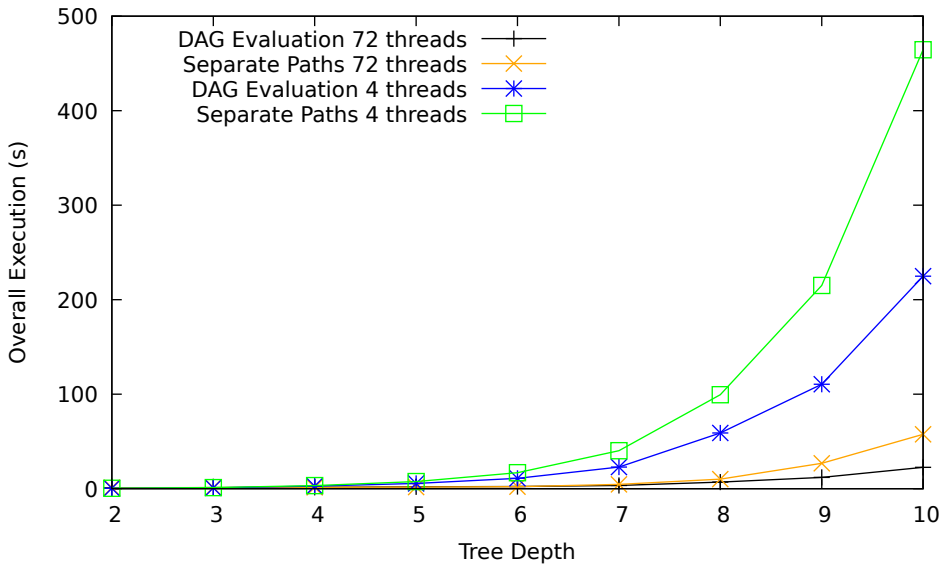


Figure 6.23: PDT-BIN Runtime with $\text{HElib}_{\text{small}}$ Comparing EVALPATHSP (DAG) vs. EVALPATHSE (Separate Paths)

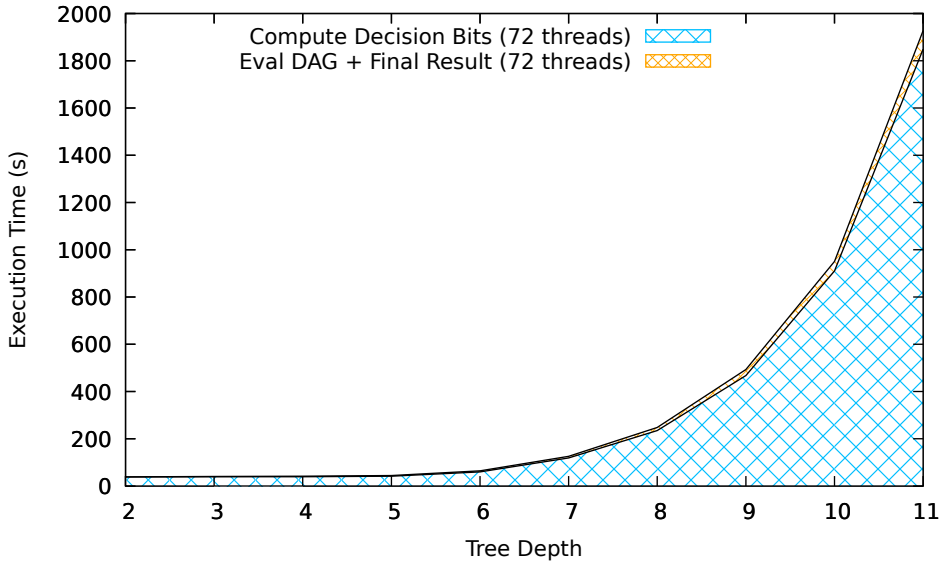


Figure 6.24: PDT-BIN Runtime with $\text{HElib}_{\text{med}}$ for 16-bit inputs

	Heart-disease (HDI)	Housing (HOU)	Spambase (SPA)	Artificial (ART)
n	13	13	57	16
d	3	13	17	10
m	5	92	58	500

Table 6.6: Real Datasets and Model Parameters

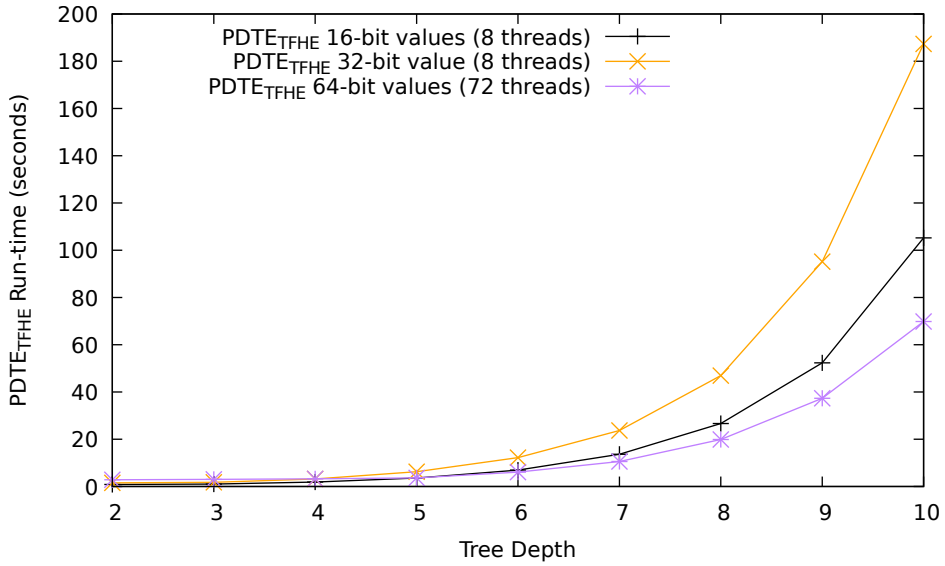


Figure 6.25: PDT-BIN Runtime with TFHE

	PDT-BIN (TFHE)	PDT-BIN (HElib)		PDT-INT (HElib)		[145] (HElib)	[175] (mcl)
λ	128	150		135		128	128
μ	16	16		16		12	64
#thd	16	16		16		16	-
	one	am.	one	am.	one	one	one
HDI	0.94	0.05	40.61	0.0073	45.59	0.59	0.25
HOU	6.30	0.35	252.38	0.90	428.23	10.27	1.98
SPA	3.66	0.24	174.46	0.72	339.60	6.88	1.80
ART	22.39	1.81	1303.55	0.75	2207.13	56.37	10.42

Table 6.7: Runtime (in seconds) of PDTE on Real Datasets: λ is the security level. μ is the input bit length. #thd is the number of threads. mcl[149] is a pairing-based cryptography library. Column "one" reports the time for one protocol run while "am." reports the amortized time (e.g., the time for one run divided by s).

1: **for** $j = 1$ **to** N **do**
2: $\quad [\![R_j^{\mathsf{b}}]\!] \leftarrow \text{PDT-BIN}_S(\mathcal{T}_j, x)$
3: $\quad$ **for** $i = 1$ **to** k **do**
4: $\quad\quad [\![b_{ij}]\!] \leftarrow \text{SHEEQUAL}([\![R_j^{\mathsf{b}}]\!], [\![c_i^{\mathsf{b}}]\!])$
5: $[\![\mathsf{result}]\!] \leftarrow [\![0]\!]$
6: **for** $i = 1$ **to** k **do**
7: $\quad [\![f_i^{\mathsf{b}}]\!] \leftarrow \text{SHEFADDER}([\![b_{i1}]\!], \ldots, [\![b_{iN}]\!])$
8: $\quad [\![e_i]\!] \leftarrow \text{SHECMP}([\![f_i^{\mathsf{b}}]\!], [\![t^{\mathsf{b}}]\!])$
9: $\quad [\![\mathsf{result}]\!] \leftarrow [\![\mathsf{result}]\!] \boxplus ([\![e_i]\!] \boxdot [\![\vec{c}_i]\!])$
10: **return** $[\![\mathsf{result}]\!]$

Algorithm 6.26: Private Random Forest With Majority Voting

1: Compute $[\![f_i^{\mathsf{b}}]\!]$ as in Algorithm 6.26 Lines 1 to 7
2: **for** $i := 1$ **to** k **do**
3: $\quad [\![\beta_{ii}]\!] \leftarrow [\![1]\!]$
4: $\quad$ **for** $j := i + 1$ **to** k **do**
5: $\quad\quad ([\![\beta_{ij}]\!], [\![\beta_{ji}]\!]) \leftarrow \text{SHECMP}([\![f_i^{\mathsf{b}}]\!], [\![f_j^{\mathsf{b}}]\!])$
6: **for** $i := 1$ **to** k **do**
7: $\quad [\![r_i^{\mathsf{b}}]\!] \leftarrow \text{SHEFADDER}([\![\beta_{i1}]\!], \ldots, [\![\beta_{ik}]\!])$
8: **for** $i := 1$ **to** k **do**
9: $\quad [\![e_i]\!] \leftarrow \text{SHEEQUAL}([\![r_i^{\mathsf{b}}]\!], [\![k^{\mathsf{b}}]\!])$
10: **for** $i := 1$ **to** k **do**
11: $\quad [\![\mathsf{result}]\!] \leftarrow [\![\mathsf{result}]\!] \boxplus ([\![e_i]\!] \boxdot [\![\vec{c}_i]\!])$
12: **return** $[\![\mathsf{result}]\!]$

Algorithm 6.27: Private Random Forest with Maximum Voting

6.8 Extension To Random Forests

In this section, we briefly describe how the binary implementation PDT-BIN can be extended to evaluate a random forest non-interactively. A random forest is a generalization of decision trees which consists of many trees. A classification with a random forest then evaluates each tree in the forest and outputs the classification label which occurs most often. Hence, the classification labels are ranked by their number of occurrences and the final result is the best ranked one.

Let the random forest consists of trees $\mathcal{T}_1, \ldots, \mathcal{T}_N$ and let $\text{PDT-BIN}_S(\mathcal{T}_j, x)$ denote the evaluation of the decision tree $\mathcal{T}_j$ on input vector x resulting in $\mathcal{T}_j(x) = R_j$, which is encrypted as $[\![R_j^{\mathsf{b}}]\!] = ([\![R_{j|k|}]\!], \ldots, [\![R_{j1}]\!])$, where $R_j^{\mathsf{b}} = R_{j|k|} \ldots R_{j1}$. Let's assume, there are k classification labels $c_1, \ldots, c_k$ with $c_i^{\mathsf{b}} = c_{i|k|} \ldots c_{i1}$ and each c_i has encryptions $[\![c_i^{\mathsf{b}}]\!] = ([\![c_{i|k|}]\!], \ldots, [\![c_{i1}]\!])$ and $[\![\vec{c}_i]\!] = [\![c_{i|k|}| \ldots |c_{i1}]\!]$. Let f_i denote the number of occurrences of c_i after evaluating the N trees, with encryption $[\![f_i^{\mathsf{b}}]\!] = ([\![f_{i|N|}]\!], \ldots, [\![f_{i1}]\!])$, where $f_i^{\mathsf{b}} = f_{i|N|} \ldots f_{i1}$.

The computation requires the routines SHECMP, SHEFADDER, and SHEEQUAL for greater-than comparison, full adder, and equality testing. To select the best label, the random forest algorithm either uses majority voting or argmax. For majority voting, c_i is the final result if an only if $f_i \geq t$, where $t = \left\lceil \frac{N}{2} \right\rceil$ with bit representation

$t^{\mathsf{b}} = t_{|N|} \dots t_1$ and encryption $[\![t^{\mathsf{b}}]\!] = ([\![t_{|N|}]\!], \dots, [\![t_1]\!])$. The computation is described in Algorithm 6.26. For argmax, c_i is the final result if an only if f_i is larger than all other $f_j, j \neq i$. The computation is described in Algorithm 6.27.

6.9 Summary

While almost all existing PDTE protocols require many interactions between the client and the server, we designed and implemented novel client-server protocols that delegate the complete evaluation to the server while preserving privacy and keeping the overhead low. Our solutions rely on SHE/FHE and evaluate the tree on ciphertexts encrypted under the client's public key. Since current SHE/FHE schemes have high overhead, we combine efficient data representations with different algorithmic optimizations to keep the computational overhead and the communication cost low. As a result, we provided the first non-interactive protocol, that allows the client to delegate the evaluation to the server by sending an encrypted input and receiving only the encryption of the result.

Chapter 7

Secure k^{th}-Ranked Element

In previous chapters, we looked at private classification using decision trees. A generalization of a decision tree is a random forest that consists of many decision trees. A classification with a random forest then evaluates each decision tree in the forest and outputs the classification label which occurs most often. Hence, the classification labels are ranked by their number of occurrences and the final result is the best ranked one. This is related to the problem of computing the k^{th}-ranked element which is the topic of this chapter. However, in this chapter, we consider the computation of the k^{th}-ranked element in a distributed and privacy-preserving setting. That is, given n parties each holding a private integer, the problem is to securely compute the element ranked k (for a given k such that $1 \leq k \leq n$) among these n integers. The goal is to reveal to the parties only the k^{th}-ranked element (or the party holding it) and nothing else. The chapter is structured as follows. We start by describing the problem, the applications, and the solution approaches in Section 7.1 and then we introduce correctness and security definitions in Section 7.2. We describe different approaches to compute the k^{th}-ranked element in Sections 7.3, 7.4, 7.5, and 7.6. We analyze correctness and security in Section 7.7 and present a complexity analysis in Section 7.8. We discuss the evaluation results in Section 7.9 before summarizing the chapter in Section 7.10.

7.1 Problem Definition

We start with the description of the problem and present the applications. Then we discuss solutions in the so-called standard model, our own approach and conclude the section with our design goals.

7.1.1 Description

Assume there are n clients $C_1, \ldots, C_n$ holding each a private integer $x_1, \ldots, x_n$ and that the integers are all pairwise distinct. We consider the problem of securely computing the k^{th}-ranked element (KRE) of these n integers. That is, the goal is to reveal to the clients the integer x_i with rank k and nothing else. Specific examples are computation of the minimum (smallest ranked), the maximum (largest ranked), or the median (middle ranked) element among the integers. Another ranked-based statistic is the best-in-class which is the mean of the top 25% best ranked elements. Hence, best-in-class computation has elements from the mean and median computation.

7.1.2 Application

The computation of the k^{th}-ranked element is of particular interest in settings such as collaborative benchmarking and auctions, where the individual inputs are sensitive

data, yet the k^{th}-ranked element (or the party holding it) is of mutual interest to all parties [2, 27, 63, 119, 155]. Benchmarking is a management process where a company compares its key performance indicator (KPI) to the statistics of the same KPIs of a group of competitors from a peer group. In secure auctions, bids are concealed and only the winner is revealed to the bidders. A big challenge for these applications is that KPIs and bids are sensitive and confidential, even within a single company [119, 155]. Confidentiality is of the utmost importance in benchmarking, since KPIs allow the inference of sensitive information. Companies are, therefore, hesitant to share their business performance data due to the risk of losing a competitive advantage [119]. The confidentiality issue can be addressed using SMC [23, 87, 195], which guarantees that no party will learn more than the output of the protocol, i.e., the other parties' inputs remain confidential.

7.1.3 Standard Model Solutions

There exist several secure protocols that can be used for keeping KPIs or bids confidential while comparing them [2, 23, 87, 195]. They require a communication channel between each pair of input parties. We will refer to this approach as the *standard model*. Protocols for securely computing the k^{th}-ranked element in the standard model do not scale easily to a large number of parties as they require a communication channel between any pair of parties and are highly interactive, resulting in high latency. Moreover, they are difficult to deploy as special arrangements are required between each pair of parties to establish a secure connection [42].

7.1.4 Our Solution Approach

A promising approach for overcoming the limitations of the standard model is to use the *server model* described in Section 3.1.2. In this model, the servers make their computational resources available for the computation, but have no input to the computation and receive no output [114, 119]. For example, Jakobsen et al. [109] propose a framework in which the input parties (the clients) delegate the computation to a set of untrusted workers. Relying on multiple non-colluding servers requires a different business model for the service provider of a privacy-preserving service. The service provider has to share benefits with an almost equal peer offering its computational power [120]. We, therefore, use a communication model consisting of clients (with private inputs) and a server. In this model, the server provides no input to the computation and does not learn the output, but makes its computational resources available to the clients [114, 119]. Moreover, there are communication channels only between each client and the server. Hence, it is a centralized communication pattern, i.e., a star network. As a result, the clients will only communicate with the server, but never directly amongst each other. This model naturally fits with the client-server architecture of Internet applications and allows a service provider to play the server's role. It can simplify the secure protocol, and improve its performance and scalability [42, 113, 114].

7.1.5 Design Goals

We propose different approaches for securely computing the k^{th}-ranked element (KRE) in a star network using either garbled circuits (GC) or additive homomorphic encryption (AHE) or somewhat homomorphic encryption (SHE):

- Our first scheme KRE-YGC uses Yao's GC [20, 139] to compare clients' inputs at the server without revealing to any party (including the server) the actual comparison result.
- Our second scheme KRE-AHE1 is based on threshold AHE. We first propose a modified variant of the Lin-Tzeng comparison protocol [135]. The server then uses it to homomorphically compare inputs encrypted with AHE. Using the threshold decryption property the parties jointly decrypt the comparison results.
- In our third scheme KRE-AHE2, we continue with threshold AHE, however, we perform the comparison using the DGK protocol [63]. In contrast to the previous scheme, the inputs are compared interactively.
- The fourth scheme KRE-SHE is based on threshold SHE and allows the server to non-interactively compute the KRE such that the clients only interact to jointly decrypt the result.

We design the above schemes with the following goals:

- Number of rounds: In contrast to [1], all our protocols have a constant number of rounds.
- Collusion-resistance: This is a protocol property that is measured by the number of parties that can collude without violating the privacy of the non-colluding ones. In KRE-YGC, a collusion with the server completely breaks the security, while KRE-AHE1 and KRE-AHE2 can tolerate the collusion of several clients with the server as long as the number of colluding clients is smaller than a threshold t. If the server does not collude, then KRE-YGC can tolerate up to $n-1$ colluding clients. Aggarwal et al.'s scheme [1] is collusion-resistant if implemented with a threshold scheme.
- Fault-tolerance: It is a protocol property that is measured by the number of parties that can fail without preventing the protocol to properly compute the intended functionality. Our server model can only tolerate clients' failure. KRE-YGC is not fault-tolerant while KRE-AHE1 and KRE-AHE2 can tolerate the failure of up to $n-t$ clients. Aggarwal et al.'s scheme [1] is fault-tolerant if implemented with a threshold scheme.
- Complexity: This refers to the asymptotic computation complexity as well as the communication complexity. Our goal is to keep the complexity as low as possible. A summary is illustrated in Table 7.2. We provide a detailed analysis in Section 7.8.

How our schemes achieve these properties is summarized in Tables 7.1 and 7.2. We also illustrate in these tables the scheme of Aggarwal et al. [1] which has been described with other related work in Section 3.6.

7.2 Security Model and Techniques

This section provides definitions related to our model and security requirements. We start by defining the k^{th}-ranked element of a sequence of integers.

	Kre-Ygc	Kre-Ahe1 Kre-Ahe2	Kre-She	[1]
# Rounds	4	4	2	$O(\mu)$
Collusion-resis.	$n-1 \mid 0$	$t-1 \mid t$	$t-1 \mid t$	$t-1 \mid$ n/a
Fault-tolerance	0	$n-t$	$n-t$	$n-t$

Table 7.1: Schemes' properties: The collusion row refers to the number of parties that can collude - server excluded | server included - without breaking the privacy of non-colluding clients. The fault-tolerance row refers to the number of parties that can fail without preventing the protocol to properly compute the intended functionality.

	Kre-Ygc		Kre-Ahe1 Kre-Ahe2	Kre-She	[1]
	sym.	asym.			
CC-C	$O(n\mu)$	$O(n)$	$O(n\mu)$	$O(\mu)$	$O(n\mu^2)$
CC-S	$O(n^2\mu)$	$O(n \log n)$	$O(n^2\mu)$	$O(n^2\mu \log \mu)$	n/a
BC-C	$O(n\mu\lambda)$	$O(n\kappa)$	$O(n\mu\kappa)$	$O((\mu+n)\kappa)$	$O(n\mu^2\lambda)$
BC-S	0	$O(n^2\kappa)$	$O(n^2\mu\kappa)$	$O(n\kappa)$	n/a

Table 7.2: Schemes' Complexity: Rows CC-C/S and BC-C/S denote the computation and communication (bit) complexity for each client and the server, respectively. The columns "sym." and "asym." denote symmetric and asymmetric operations in Kre-Ygc.

7.2.1 The Model

Definition 7.2.1. *Let $\mathbb{X} = \{x_1, ..., x_n\}$ be a set of n distinct integers and let $\tilde{x}_1, \ldots, \tilde{x}_n$ be the corresponding sorted set, i.e., $\tilde{x}_1 \leq \ldots \leq \tilde{x}_n$, and $\mathbb{X} = \{\tilde{x}_1, \ldots, \tilde{x}_n\}$. The* rank *of an element $x_i \in \mathbb{X}$ is j, such that $x_i = \tilde{x}_j$. The* k^{th}-ranked element (KRE) *is the element $\tilde{x}_k$ with rank k.*

If the rank is $k = \left\lceil \frac{n}{2} \right\rceil$ then the element is called *median*. If $k = 1$ (resp. $k = n$) then the element is called *minimum* (resp. *maximum*).

Definition 7.2.2. *Let $C_1, \ldots, C_n$ be n clients each holding a private μ-bit integer $x_1, \ldots, x_n$ and S be a server which has no input. Our* ideal functionality $\mathcal{F}_{\text{KRE}}$ *receives $x_1, \ldots, x_n$ from the clients, computes the* KRE $\tilde{x}_k$ *and outputs $\tilde{x}_k$ to each client C_i. Moreover, $\mathcal{F}_{\text{KRE}}$ outputs a* leakage $\mathcal{L}_i$ *to each C_i and $\mathcal{L}_S$ to S.*

The leakage is specific to each protocol and contains information such as n, t, λ, κ, μ (see Table 2.2). It can be inferred from the party's view which is all that the party is allowed to learn from the protocol execution. In case of limited collusion (i.e., the number of colluding parties is smaller than a given threshold as given in Table 7.1) additional leakage might include comparison results between some pairs of inputs or the rank of some inputs. Recall that the *view* of the i-th party during an execution of the protocol on input $\vec{x} = (x_1, \ldots, x_n)$ is denoted by: $\mathsf{View}_i^{\Pi_{\text{KRE}}}(\vec{x}) = \{x_i, r_i, m_{i1}, m_{i2}, \ldots\}$, where r_i represents the outcome of the i-th party's internal coin tosses, and m_{ij} represents the j-th message it has received. Since the server is a party without input, x_i in its view will be replaced by the empty string.

We assume that parties follow the protocol specification, but the adversary keeps a record of all messages received by corrupted parties and tries to infer as much information as possible. Our schemes are, therefore, secure in the semi-honest model.

Definition 7.2.3. *Let $\mathcal{F}_{\text{KRE}} : (\{0,1\}^{\mu})^n \mapsto \{0,1\}^{\mu}$ be the functionality that takes n μ-bit inputs $x_1, \ldots, x_n$ and returns their KRE. Let $I = \{i_1, \ldots, i_t\} \subset \{1, \ldots, n+1\}$ be a subset of indexes of corrupted parties (Server's input x_{n+1} is empty), $\vec{x} = (x_1, \ldots, x_n)$ and*

$$\mathsf{View}_{\mathsf{I}}^{\Pi_{\text{KRE}}}(\vec{x}) = (I, \mathsf{View}_{\mathsf{i}_1}^{\Pi_{\text{KRE}}}(\vec{x}), \ldots, \mathsf{View}_{\mathsf{i}_t}^{\Pi_{\text{KRE}}}(\vec{x})).$$

*A protocol t-*privately computes $\mathcal{F}_{\text{KRE}}$ in the semi-honest model *if there exists a polynomial-time simulator $\mathsf{Sim}_{\mathsf{I}}^{\text{kre}}$ such that: $\forall I, |I| = t$ and $\mathcal{L}_I = \bigcup_{i \in I} \mathcal{L}_i$, it holds:*

$$\mathsf{Sim}_{\mathsf{I}}^{\text{kre}}(I, (x_{i_1}, \ldots, x_{i_t}), \mathcal{F}_{\text{KRE}}(x_1, \ldots, x_n), \mathcal{L}_I) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{I}}^{\Pi_{\text{KRE}}}(x_1, \ldots, x_n).$$

7.2.2 Technical Overview

Each scheme consists of two phases. In an initialization phase, clients generate and exchange necessary cryptographic keys through the server. We stress that the initialization phase is run once and its complexity does not depend on the functionality that we want to compute. In the following, we, therefore, focus on the actual computations.

We determine the KRE in the main protocol by computing the rank of each x_i and selecting the right one. To achieve that, we compare pairs of inputs $(x_i, x_j), 1 \le i, j, \le n$ and denote the result by a comparison bit b_{ij}.

Definition 7.2.4. *Let $x_i, x_j, 1 \le i, j, \le n$, be integer inputs of C_i, C_j. Then the* comparison bit *b_{ij} of the pair (x_i, x_j) is defined as 1 if $x_i \ge x_j$ and 0 otherwise. The computation of $x_i \ge x_j$ is distributed and involves C_i, C_j, where they play different roles, e.g., generator and evaluator. Similar to the functional programming notation of an ordered pair, we use* head *and* tail *to denote C_i and C_j.*

For each input x_i, we then add all bits $b_{ij}, 1 \le j \le n$ to get its rank r_i.

Lemma 7.2.5. *Let $x_1, \ldots, x_n$ be n distinct integers, and let $r_1, \ldots, r_n \in \{1, \ldots, n\}$ be their corresponding ranks and b_{ij} the comparison bit for (x_i, x_j). It holds $r_i = \sum_{j=1}^{n} b_{ij}$.*

Proof. Since r_i is the rank of x_i, x_i is by definition larger or equal to r_i elements in $\{x_1, \ldots, x_n\}$. This means that r_i values among $b_{i1}, \ldots, b_{in}$ are 1 and the remaining $n - r_i$ values are 0. It follows that $\sum_{j=1}^{n} b_{ij} = r_i$. □

The above lemma requires distinct inputs. To make sure that clients' inputs are indeed distinct before the protocol execution, we borrow the idea of [2] and use the index of parties as differentiator. Each party C_i represents its index i as a $\log n$-bit string and appends it at the end (i.e., in the least significant positions) of the binary string of x_i, resulting in a new input of length $\mu + \log n$. For simplicity, we assume in the remainder of the chapter, that the x_i's are all distinct μ-bit integers. Therefore, it is not necessary to compare all pairs $(x_i, x_j), 1 \le i, j \le n$, since we can deduce b_{ji} from b_{ij}.

As explained in Definition 7.2.4, C_i, C_j play different roles in the comparison for (x_i, x_j). Therefore, we would like to equally distribute the roles among the clients. As example for $n = 3$, we need to compute only three (instead of nine) comparisons resulting in three *head* roles and three *tail* roles. Then we would like each of the three clients to play the role *head* as well as *tail* exactly one time. We will use Definition 7.2.6 and Lemma 7.2.7 to equally distribute the roles *head* and *tail* between clients.

Definition 7.2.6. *Let $\mathbb{X} = \{x_1, \ldots, x_n\}$ be a set of n integers. We define the predicate* PAIRED *as follows:*

$$\begin{aligned}
\text{PAIRED}(i,j) := {} & (i \equiv 1 \pmod 2 \wedge i > j \wedge j \equiv 1 \pmod 2) \vee && (7.1a)\\
& (i \equiv 1 \pmod 2 \wedge i < j \wedge j \equiv 0 \pmod 2) \vee && (7.1b)\\
& (i \equiv 0 \pmod 2 \wedge i > j \wedge j \equiv 0 \pmod 2) \vee && (7.1c)\\
& (i \equiv 0 \pmod 2 \wedge i < j \wedge j \equiv 1 \pmod 2). && (7.1d)
\end{aligned}$$

Lemma 7.2.7. *Let $\mathbb{X} = \{x_1, \ldots, x_n\}$ be a set of n integers and the predicate* PAIRED *be as above. Then comparing only pairs (x_i, x_j) such that* PAIRED$(i,j) = true$ *is enough to compute the rank of all elements in $\mathbb{X}$.*

Proof. Let $\mathbb{P} = \{(x_i, x_j) : x_i, x_j \in \mathbb{X} \wedge i \neq j\}$, $\mathbb{P}_1 = \{(x_i, x_j) : x_i, x_j \in \mathbb{X} \wedge \text{PAIRED}(i,j) = true\}$, $\mathbb{P}_2 = \{(x_i, x_j) : x_i, x_j \in \mathbb{X} \wedge Q(i,j) = true\}$, where $Q(i,j)$ is defined as follows:

$$\begin{aligned}
Q(i,j) := {} & (i \equiv 1 \pmod 2 \wedge i < j \wedge j \equiv 1 \pmod 2) \vee && (7.2a)\\
& (i \equiv 0 \pmod 2 \wedge i > j \wedge j \equiv 1 \pmod 2) \vee && (7.2b)\\
& (i \equiv 0 \pmod 2 \wedge i < j \wedge j \equiv 0 \pmod 2) \vee && (7.2c)\\
& (i \equiv 1 \pmod 2 \wedge i > j \wedge j \equiv 0 \pmod 2). && (7.2d)
\end{aligned}$$

Clearly, $\mathbb{P}$ contains the maximum number of comparisons required to compute the rank of every $x_i \in \mathbb{X}$. Now it suffices to show that:

1. $\mathbb{P}_1$ and $\mathbb{P}_2$ form a partition of $\mathbb{P}$
2. $\forall\ (x_i, x_j) \in \mathbb{P} : (x_i, x_j) \in \mathbb{P}_1 \Leftrightarrow (x_j, x_i) \in \mathbb{P}_2$

$\mathbb{P}_1$ and $\mathbb{P}_2$ are clearly subsets of $\mathbb{P}$. For each $(x_i, x_j) \in \mathbb{P}$, (i,j) satisfies exactly one of the conditions (7.1a), ..., (7.1d), (7.2a), ..., (7.2d), hence $\mathbb{P} \subseteq \mathbb{P}_1 \cup \mathbb{P}_2$. Moreover, for each $(x_i, x_j) \in \mathbb{P}$, either PAIRED$(i,j) = true$ or $Q(i,j) = true$. It follows that $\mathbb{P}_1 \cap \mathbb{P}_2 = \varnothing$ which concludes the proof of claim 1. To prove claim 2, it suffices to see that, (i,j) satisfies condition (7.1a) if and only if (j,i) satisfies condition (7.2a). The same holds for (7.1b) and (7.2b), (7.1c) and (7.2c), (7.1d) and (7.2d). □

For example, if $n = 3$, we compute comparison bits only for (x_1, x_2), (x_2, x_3), (x_3, x_1) and deduce the remaining comparison bits from the computed ones. If $n = 4$, we compare only (x_1, x_2), (x_1, x_4), (x_2, x_3), (x_3, x_1), (x_3, x_4), (x_4, x_2).

The predicate PAIRED (Equation 7.1) is used in our schemes to reduce the number of comparisons and to equally distribute the computation task of the comparisons among the clients. Let $\#head_i$ (resp. $\#tail_i$) denote the number of times PAIRED$(i,j) = true$ (resp. PAIRED$(j,i) = true$) holds. For example, if $n = 3$, we have $\#head_i = \#tail_i = 1$ for all clients. However, for $n = 4$, we have $\#head_1 = \#head_3 = 2$, $\#tail_1 = \#tail_3 = 1$, $\#head_2 = \#head_4 = 1$ and $\#tail_2 = \#tail_4 = 2$.

Lemma 7.2.8. *Let $\mathbb{X} = \{x_1, \ldots, x_n\}$ be a set of integers and assume the predicate* PAIRED *is used to sort $\mathbb{X}$. If n is odd then: $\#head_i = \#tail_i = \frac{n-1}{2}$. If n is even then:*

$$\#head_i = \begin{cases} \frac{n}{2} & \text{if } i \text{ odd} \\ \frac{n}{2} - 1 & \text{if } i \text{ even} \end{cases} \quad \#tail_i = \begin{cases} \frac{n}{2} - 1 & \text{if } i \text{ odd} \\ \frac{n}{2} & \text{if } i \text{ even.} \end{cases}$$

Proof. This is actually a corollary of the proof of Lemma 7.2.7. It follows from the fact that $(x_i, x_j) \in \mathbb{P}_1 \Leftrightarrow (x_j, x_i) \in \mathbb{P}_2$ and any x_i is involved in $n-1$ comparisons (since we need $b_{i1}, \ldots, b_{in}$ to compute $r_i = \sum_{j=1}^{n} b_{ij}$, where we trivially have $b_{ii} = 1$ without comparison). This proves the case when n is odd. If n is even then the odd case applies for $n' = n-1$. Then for each $i \in \{1, \ldots, n'\}$, we have $\textsc{Paired}(i, n) = \mathit{true}$ if i is odd (condition 7.1b) and $\textsc{Paired}(n, i) = \mathit{true}$ if i is even (condition 7.1c). □

7.3 A GC-Based Construction

This section describes Kre-Ygc (Protocol 7.1) based on GC which consists of an initialization and a main protocol. During initialization, parties generate and distribute cryptographic keys. The online protocol uses GC to compare the inputs and AHE to compute the rank of each x_i from the comparison bits. We denote an AHE ciphertext with $[\![\cdot]\!]$ (see Table 2.2).

7.3.1 Initialization

The initialization consists of public key distribution and Diffie-Hellman (DH) key agreement. Each client C_i sends its public key pk_i (e.g., using a pseudonym certificate) of an AHE to the server. The server then distributes the public keys to the clients. In our implementation, we use the Paillier [159] scheme, but any AHE scheme such as [127] will work as well. Then each pair (C_i, C_j) of clients runs DH key exchange through the server to generate a common secret key $ck_{ij} = ck_{ji}$. The common key ck_{ij} is used by C_i and C_j to seed the pseudorandom number generator (PRNG) of the garbling scheme that is used to generate a comparison GC for x_i and x_j, i.e. $Gb(1^\lambda, ck_{ij}, f_>)$, where $f_>$ is a Boolean comparison circuit.

7.3.2 GC-Based Main Protocol

Protocol 7.1 is a four-round protocol in which we use GC to compare pairs of inputs and to reveal a blinded comparison bit to the server. Then we use AHE to unblind the comparison bits, compute the ranks and the KRE without revealing anything to the parties. Let $f_>$ be defined as: $f_>((a_i, x_i), (a_j, x_j)) = a_i \oplus a_j \oplus b_{ij}$, where $a_i, a_j \in \{0,1\}$, i.e., $f_>$ compares x_i, x_j and blinds the comparison bits b_{ij} with a_i, a_j.

Comparing Inputs. For each pair (x_i, x_j), if $\textsc{Paired}(i, j) = \mathit{true}$ the parties do the following:

- Client C_i chooses a masking bit $a_i^{ij} \xleftarrow{\$} \{0,1\}$ and extends its input x_i to (a_i^{ij}, x_i). Then using the common key ck_{ij}, it computes $(F_>^{ij}, e) \leftarrow Gb(1^\lambda, ck_{ij}, f_>)$ and $(\bar{a}_i^{ij}, \bar{x}_i^{ij}) \leftarrow En(e, (a_i^{ij}, x_i))$, and sends $F_>^{ij}$, $(\bar{a}_i^{ij}, \bar{x}_i^{ij})$ to the server S.

- Client C_j chooses a masking bit $a_j^{ij} \xleftarrow{\$} \{0,1\}$ and extends its input x_j to (a_j^{ij}, x_j). Then using the common key $ck_{ji} = ck_{ij}$, it computes $(F_>^{ij}, e) \leftarrow Gb(1^\lambda, ck_{ji}, f_>)$ and $(\bar{a}_j^{ij}, \bar{x}_j^{ij}) \leftarrow En(e, (a_j^{ij}, x_j))$, and sends only $(\bar{a}_j^{ij}, \bar{x}_j^{ij})$ to the server S.

- We have $b'_{ij} \leftarrow F_>^{ij}((\bar{a}_i^{ij}, \bar{x}_i^{ij}), (\bar{a}_j^{ij}, \bar{x}_j^{ij})) = a_i^{ij} \oplus a_j^{ij} \oplus b_{ij}$ (i.e. b_{ij} is hidden to S). The server then evaluates all GCs (Steps 1 to 5).

Unblinding Comparison Bits. Using AHE, the parties unblind each GC result $b'_{ij} = a_i^{ij} \oplus a_j^{ij} \oplus b_{ij}$, where a_i^{ij} is known to C_i and a_j^{ij} is known to C_j, without learning anything. As a result, $[\![b_{ij}]\!]_i$ and $[\![b_{ij}]\!]_j$ are revealed to S encrypted under pk_i and pk_j. This is illustrated in Steps 6 to 16 and works as follows:

- S sends b'_{ij} to C_i and C_j. They reply with $[\![a_j^{ij} \oplus b_{ij}]\!]_i$ and $[\![a_i^{ij} \oplus b_{ij}]\!]_j$.
- S forwards $[\![a_i^{ij} \oplus b_{ij}]\!]_j$, $[\![a_j^{ij} \oplus b_{ij}]\!]_i$ to C_i, C_j. They reply with $[\![b_{ij}]\!]_j$, $[\![b_{ij}]\!]_i$.
- S sets $[\![b_{ji}]\!]_j = [\![1 - b_{ij}]\!]_j$.

Computing the Rank. The computation of the rank is done at the server by homomorphically adding comparison bits. Hence for each i, the server computes the ciphertext $[\![r_i]\!]_i = [\![\sum_{j=1}^{n} b_{ij}]\!]_i$. Then, it chooses a random number α_i and computes $[\![\beta_i]\!]_i = [\![(r_i - k) \cdot \alpha_i]\!]_i$ (Steps 17 to 19). The ciphertext $[\![\beta_i]\!]_i$ encrypts 0 if $r_i = k$ (i.e., x_i is the k^{th}-ranked element) otherwise it encrypts a random plaintext.

Computing the KRE's Ciphertext. Each client C_i receives $[\![\beta_i]\!]_i$ encrypted under its public key pk_i and decrypts it. Then if $\beta_i = 0$, C_i sets $m_i = x_i$ otherwise $m_i = 0$. Finally, C_i encrypts m_i under each client's public key and sends $[\![m_i]\!]_1, \ldots, [\![m_i]\!]_n$ to the server (Steps 20 to 22).

Revealing the KRE's Ciphertext. In the final steps (Steps 23 to 24), the server adds all $[\![m_j]\!]_i$ encrypted under pk_i and reveals $[\![\sum_{j=1}^{n} m_j]\!]_i$ to C_i.

Brief Summary. KRE-YGC protocol correctly computes the KRE. The proof trivially follows from the correctness of the GC protocol, Lemmas 7.2.5 and 7.2.7, and the correctness of the AHE scheme. KRE-YGC is not fault-tolerant and a collusion with the server reveals all inputs to the adversary.

7.4 An AHE-Based Construction

This section describes KRE-AHE1 (Protocol 7.3) based on threshold AHE. KRE-AHE1 compares all inputs (using our modified variant of the Lin-Tzeng protocol as described in Algorithm 6.10) at the server which then randomly distributes encrypted comparison bits to the clients for threshold decryption.

7.4.1 Threshold Initialization

We assume threshold key generation. Hence, there is a public/private key pair $(\mathsf{pk}, \mathsf{sk})$ for an AHE, where the private key sk is split in n shares $\langle \mathsf{sk} \rangle_1, \ldots, \langle \mathsf{sk} \rangle_n$ such that client C_i gets share $\langle \mathsf{sk} \rangle_i$ and at least t shares are required to reconstruct sk. Additionally, each client C_i has its own AHE key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ and publishes pk_i to all clients. We denote by $[\![x_i]\!], [\![x_i]\!]_j$ encryptions of x_i under $\mathsf{pk}, \mathsf{pk}_j$ respectively (Table 2.2).

7.4.2 Main Protocol

Protocol 7.3 is a four-round protocol in which the clients send their inputs encrypted using AHE under the common public key pk to the server. The server homomorphically evaluates comparison circuits on the encrypted inputs using our modified variant of the Lin-Tzeng protocol [135]. Then the clients jointly decrypt the comparison results and compute the rank of each x_i.

1: **for** $i := 1, j := i+1$ **to** n **do**
2: **if** PAIRED(i,j) **then**
3: $C_i \to S$: $F_>^{ij}, (\bar{a}_i^{ij}, \bar{x}_i^{ij})$
4: $C_j \to S$: $(\bar{a}_j^{ij}, \bar{x}_j^{ij})$
5: S: **let** $b'_{ij} \leftarrow F_>^{ij}(\bar{x}_i^{ij}, \bar{x}_j^{ij})$
6: **for** $i := 1, j := i+1$ **to** n **do**
7: **if** PAIRED(i,j) **then**
8: $S \to C_i$: $b'_{ij} = a_i^{ij} \oplus a_j^{ij} \oplus b_{ij}$
9: $S \to C_j$: $b'_{ij} = a_i^{ij} \oplus a_j^{ij} \oplus b_{ij}$
10: $C_i \to S$: $[\![a_j^{ij} \oplus b_{ij}]\!]_i$
11: $C_j \to S$: $[\![a_i^{ij} \oplus b_{ij}]\!]_j$
12: $S \to C_i$: $[\![a_i^{ij} \oplus b_{ij}]\!]_j$
13: $S \to C_j$: $[\![a_j^{ij} \oplus b_{ij}]\!]_i$
14: $C_i \to S$: $[\![b_{ij}]\!]_j$
15: $C_j \to S$: $[\![b_{ij}]\!]_i$
16: S: **let** $[\![b_{ji}]\!]_j \leftarrow [\![1 - b_{ij}]\!]_j$
17: **for** $i := 1$ **to** n **do**
18: $S : [\![r_i]\!]_i \leftarrow [\![\sum_{j=1}^n b_{ij}]\!]_i$ ▷ $b_{ii} = 1$
19: $S \to C_i$: $[\![\beta_i]\!]_i \leftarrow [\![(r_i - k) \cdot \alpha_i]\!]_i$, for a random α_i
20: **for** $i := 1$ **to** n **do**
21: C_i: $m_i := \begin{cases} x_i & \text{if } \beta_i = 0 \\ 0 & \text{if } \beta_i \neq 0 \end{cases}$
22: $C_i \to S$: $[\![m_i]\!]_1, \ldots, [\![m_i]\!]_n$
23: **for** $i := 1$ **to** n **do**
24: $S \to C_i$: $[\![\sum_{j=1}^n m_j]\!]_i$

Protocol 7.1: KRE-YGC Protocol

Uploading Ciphertexts. Using the common public key pk, each client C_i sends $[\![x_i]\!], [\![V_{x_i}^0]\!], [\![V_{x_i}^1]\!]$ to the server as illustrated in Step 2 of Protocol 7.3.

Comparing Inputs. The server compares the inputs by computing

$$g_{ij} \leftarrow \text{LINCOMPARE}([\![V_{x_i}^1]\!], [\![V_{x_j}^0]\!])$$

for each $1 \leq i, j, \leq n$. Let G be the $n \times n$ matrix $[g_{11}, \ldots, g_{1n}, \ldots, g_{n1}, \ldots, g_{nn}]$. The server chooses $n+1$ permutations $\pi, \pi_1, \ldots, \pi_n \xleftarrow{\$} \mathfrak{S}_n$ that hide the indexes of g_{ij} to the clients during threshold decryption: π permutes the rows of G and each $\pi_i, 1 \leq i \leq n$ permutes the columns of row i. Let $G'_1, \ldots, G'_n$ be the rows of the resulting matrix G' (after application of the permutations to G). Using Algorithm 7.2, the server computes for each C_i a $t \times n$ matrix $G^{(i)}$ consisting of the rows:

$$G'_{i-t+1 \bmod n}, \ldots, G'_{i-1 \bmod n}, G'_i.$$

The matrix $G^{(i)}$ and the list of combiners for rows in $G^{(i)}$ are sent to C_i in Step 11. An example is illustrated in Table 7.3.

G	G'	Dec_p	Dec_f
$g_{11},\ g_{12},\ g_{13}$	$g_{32},\ g_{33},\ g_{31}$	$C_1,\ C_2$	C_1
$g_{21},\ g_{22},\ g_{23}$	$g_{11},\ g_{13},\ g_{12}$	$C_2,\ C_3$	C_2
$g_{31},\ g_{32},\ g_{33}$	$g_{23},\ g_{21},\ g_{22}$	$C_1,\ C_3$	C_3

Table 7.3: Threshold Decryption Example ($n = 3, t = 2$): The elements of G are permuted resulting in G'. Clients in columns "Dec_p" run $\mathsf{Dec}_p()$ on the corresponding row and send the result to the client in column "Dec_f" for the final decryption.

Lemma 7.4.1 shows that the ciphertexts generated from Algorithm 7.2 allow to correctly decrypt the matrix $G = [g_{11}, \ldots, g_{nn}]$, i.e., each g_{ij} is distributed to exactly t different clients. By applying the lemma to the set of rows of G, the first part shows that each client receives exactly a subset of t different rows of G. The second part shows that each row of G is distributed to exactly t different clients, which allows a correct threshold decryption of each row.

Lemma 7.4.1. *Let $\mathbb{X} = \{x_1, \ldots, x_n\}$ be a set of n elements, $\mathbb{X}_i = \{x_{i-t+1}, \ldots, x_i\}$, $1 \le i \le n$, where the indexes in $\mathbb{X}_i$ are computed modulo n, and $t \le n$. Then:*

- *Each subset $\mathbb{X}_i$ contains exactly t elements of $\mathbb{X}$ and*
- *each $x \in \mathbb{X}$ is in exactly t subsets $\mathbb{X}_i$.*

Proof. It is clear from the definition that $\mathbb{X}_i \subseteq \mathbb{X}$ for all i and since $i - (i - t + 1) + 1 = t$, $\mathbb{X}_i$ has exactly t elements. Let x_i be in $\mathbb{X}$, then from the definition, x_i is an element of only the subsets $\mathbb{X}_i, \mathbb{X}_{i+1}, \ldots, \mathbb{X}_{i+t-1}$, where indexes of the $\mathbb{X}_i$ are computed mod n. Again, it holds $(i + t - 1) - i + 1 = t$. □

After receiving $G^{(i)}$, each client C_i performs its partial decryption for each ciphertext, re-encrypts each line l ($l \in I^{(i)}$) with the public key pk_l of client C_l. This prevents the server to learn comparison bits. Then C_i sends the result $h_{l,j}^{(i)} = [\![\mathsf{Dec}_p(\langle \mathsf{sk} \rangle_i, g_{lj})]\!]_l$, $(1 \le j \le n)$ to the server (Step 17). Client C_l will be the combiner of the ciphertexts in line l. In Step 19, the server forwards the encrypted partial decryption results $h_{lj}^{(i_1)}, \ldots, h_{lj}^{(i_t)}$ of line l and the corresponding $c_l = [\![x_{\pi^{-1}(l)}]\!]$ to C_l. Client C_l decrypts and reconstructs each comparison result resulting in the comparison bits $b_{l1}, \ldots, b_{ln}$ as illustrated in Steps 24 and 25.

Computing the KRE's Ciphertext. Each combiner C_l computes the rank $r_l = \sum_{j=1}^{n} b_{lj}$ (Step 27) and ciphertext $\tilde{c}_l$ that is either a re-encryption of c_l if $r_l = k$ or an encryption of 0 otherwise (Step 28). The ciphertext $\tilde{c}_l$ is sent back to the server. The server multiplies all $\tilde{c}_l$ (Step 29) resulting in a ciphertext $\tilde{c}$ of the KRE which is sent to a subset $\mathbb{I}_t$ of t clients for threshold decryption (Step 32). Each client in $\mathbb{I}_t$ performs a partial decryption (Step 34), encrypts the result for all clients, and sends the ciphertexts to the server (Step 35). Finally, the server forwards the encrypted partial decryption to the clients (Step 37) that they use to learn the KRE (Step 38).

Brief Summary. KRE-AHE1 protocol correctly computes the KRE. The proof trivially follows from the correctness of the Lin-Tzeng comparison protocol [135], Lemmas 7.2.5 and 7.4.1 and the correctness of AHE. KRE-AHE1 executes all $O(n^2)$ comparisons non-interactively at the server, but requires threshold decryption for $O(n^2)$

```
1: function DecReq(G, i, t, π)
2:     parse G as [g_11, ..., g_1n, ..., g_n1, ..., g_nn]
3:     let G^(i) = [q_11^(i), ..., q_1n^(i), ..., q_t1^(i), ..., q_tn^(i)]
4:     for u := 1 to t do
5:         j ← i − t + u mod n
6:         if j ≤ 0 then
7:             j ← j + n                                   ▷ 1 ≤ j ≤ n
8:         I^(i) ← I^(i) ∪ {j}
9:         w ← π(j)
10:        for v := 1 to n do
11:            q_uv^(i) ← g'_wv
12:    return (G^(i), I^(i))
```

Algorithm 7.2: Decryption Request in Kre-Ahe1 and Kre-Ahe2

elements. The next protocol runs the $O(n^2)$ comparisons in parallel with the help of the clients while requiring threshold decryption of only $O(n)$ elements.

7.5 Improving the AHE-Based Construction

In this section, we describe Kre-Ahe2 (Protocol 7.6) which instantiates the comparison with the DGK protocol [63]. The initialization is similar to the previous case. We start by briefly describing the DGK protocol [63] using the server.

7.5.1 DGK Comparison Protocol With the Server

The comparison with the server is illustrated in Protocol 7.4. For each pair C_i, C_j such that $\textsc{Paired}(i,j)$ holds, the clients C_i and C_j run the DGK protocol with the server. The server forwards $[\![x_i^{\mathsf{b}}]\!]_i = [\![x_{i\mu}]\!]_i, \ldots, [\![x_{i1}]\!]_i$ (encrypted under pk_i) to C_j. Client C_j runs $\textsc{DgkEval}([\![x_i^{\mathsf{b}}]\!]_i, x_j^{\mathsf{b}})$ and obtains $\delta_{ji}, ([\![z_{i\mu}]\!]_i, \ldots, [\![z_{i1}]\!]_i)$ as result. It then encrypts δ_{ji} under the common public key and sends back $[\![z_{i\mu}]\!]_i, \ldots, [\![z_{i1}]\!]_i, [\![\delta_{ji}]\!]$ to client C_i via the server. Client C_i runs $\textsc{DgkDecrypt}([\![z_{i\mu}]\!]_i, \ldots, [\![z_{i1}]\!]_i)$, obtains a shared bit δ_{ij} and sends back $[\![\delta_{ij} \oplus \delta_{ji}]\!]$ to the server. After the computation, the clients C_i and C_j hold random shared bits δ_{ij} and δ_{ji} such that $b_{ij} = [x_i \leq x_j] = \delta_{ij} \oplus \delta_{ji}$ holds. The server learns the encryption $[\![b_{ij}]\!]$ of the comparison bit b_{ij}.

7.5.2 Main Protocol

Kre-Ahe2 is a 4-round protocol in which inputs are compared interactively using the DGK protocol. The resulting comparison bit is encrypted under pk and revealed to the server which then computes the ranks of the x_i's and triggers a threshold decryption.

Uploading Ciphertext. Each party C_i sends $[\![x_i]\!]$ (encrypted under the common public key pk) and $[\![x_i^{\mathsf{b}}]\!]_i = ([\![x_{i\mu}]\!]_i, \ldots, [\![x_{i1}]\!]_i)$ (encrypted under its own public key pk_i) to the server. This is illustrated in Step 2 of Protocol 7.6. The server then initializes a matrix $G = [g_{11}, \ldots, g_{nn}]$, where $g_{ii} = [\![1]\!]$ and $g_{ij} (i \neq j)$ will be computed in the DGK protocol as $g_{ij} = [\![b_{ij}]\!]$ if $\textsc{Paired}(i,j)$ is true, and an array $X = [[\![x_1]\!], \ldots, [\![x_n]\!]]$ (Step 3).

for $i \coloneqq 1$ **to** n **do**
  $C_i \to S$: $[\![x_i]\!], [\![V_{x_i}^0]\!], [\![V_{x_i}^1]\!]$
for $i, j \coloneqq 1$ **to** n $(i \neq j)$ **do**
  S: $g_{ij} \leftarrow \textsc{LinCompare}([\![V_{x_i}^1]\!], [\![V_{x_j}^0]\!])$
S : **choose** permutations $\pi_0, \ldots, \pi_t \xleftarrow{\$} \mathfrak{S}_n$
S : **let** $G = [g_{11}, \ldots, g_{nn}]$
S : **let** $(c_1 \ldots, c_n) \leftarrow \pi_0([\![x_1]\!], \ldots, [\![x_n]\!])$
for $u, v \coloneqq 1$ **to** n **do**
  S : $g_{uv} \leftarrow g_{\pi_0(u)\pi_u(v)}$
for $i \coloneqq 1$ **to** n **do**
  $S \to C_i$: $(G^{(i)}, I^{(i)}) \leftarrow \textsc{DecReq}(G, i, t, \pi_0)$
  C_i: **parse** $G^{(i)}$ **as** $[q_{u_1 1}^{(i)}, \ldots, q_{u_1 n}^{(i)}, \ldots, q_{u_t 1}^{(i)}, \ldots, q_{u_t n}^{(i)}]$
    parse $I^{(i)}$ **as** $[I_1^{(i)}, \ldots, I_t^{(i)}]$
for $i \coloneqq 1$ **to** n **do**
  for $u \coloneqq 1$ **to** t **do**
    for $v \coloneqq 1$ **to** n **do**
      for each $l \in I_u^{(i)}$ **do**
        $C_i \to S$: $h_{uv}^{(i)} \leftarrow [\![\mathsf{Dec}_p(\langle \mathsf{sk} \rangle_i, q_{uv}^{(i)})]\!]_l$
let $\{l_1, \ldots, l_t\}$ be the indexes of partial decryptors of the l-th row of G
for $l \coloneqq 1$ **to** n **do**
  $S \to C_l$: c_l
  for $j \coloneqq 1$ **to** n **do**
    $S \to C_l$: $h_{lj}^{(l_1)}, \ldots, h_{lj}^{(l_t)}$
for $l, j \coloneqq 1$ **to** n **do**
  C_l: $d_1 = Dec(\mathsf{sk}_l, h_{lj}^{(l_1)}), \ldots, d_t = Dec(\mathsf{sk}_l, h_{lj}^{(l_t)})$
  C_l: $b_{lj} \leftarrow \mathsf{Dec}_f(d_1, \ldots, d_t)$
for $l \coloneqq 1$ **to** n **do**
  C_l: $r_l \leftarrow \sum_{j=1}^{n} b_{lj}$
  $C_l \to S$: $\tilde{c}_l \coloneqq \begin{cases} c_l \cdot [\![0]\!] & \text{if } r_l = k \\ [\![0]\!] & \text{if } r_l \neq k \end{cases}$
S: **let** $\tilde{c} \leftarrow \prod_{l=1}^{n} \tilde{c}_l$
S: **let** $\mathbb{I}_t = \{i_1, \ldots, i_t\} \xleftarrow{\$} \{1, \ldots, n\}$
for all $i \in \mathbb{I}_t$ **do**
  $S \to C_i$: $\tilde{c}$
for all $i \in \mathbb{I}_t$ **do**
  C_i: $m^{(i)} = \mathsf{Dec}_p(\langle \mathsf{sk} \rangle_i, \tilde{c})$
  $C_i \to S$: $[\![m^{(i)}]\!]_1, \ldots, [\![m^{(i)}]\!]_n$
for $i \coloneqq 1$ **to** n **do**
  $S \to C_i$: $[\![m^{(i_1)}]\!]_i, \ldots, [\![m^{(i_t)}]\!]_i$
  C_i: $\mathsf{Dec}_f(m^{(i_1)}, \ldots, m^{(i_t)})$

Protocol 7.3: KRE-AHE1 Protocol

Comparing Inputs. In this step, pairs of clients run DGKCOMPARE with the server as illustrated in Algorithm 7.4. If (i, j) satisfies the predicate PAIRED, then C_i runs

1: **function** DGKCOMPARE(i, j)
2: **if** PAIRED(i, j) **then**
3: $S \to C_j$: $[\![x_i^{\mathsf{b}}]\!]_i = [\![x_{i\mu}]\!]_i, \ldots, [\![x_{i1}]\!]_i$
4: C_j: $(\delta_{ji}, Z) \leftarrow$ DGKEVAL($[\![x_i^{\mathsf{b}}]\!]_i, x_j^{\mathsf{b}}$)
5: C_j: **parse** Z **as** $([\![z_{i\mu}]\!]_i, \ldots, [\![z_{i1}]\!]_i)$
6: $C_j \to S$: $[\![z_{i\mu}]\!]_i, \ldots, [\![z_{i1}]\!]_i, [\![\delta_{ji}]\!]$
7: $S \to C_i$: $[\![z_{i\mu}]\!]_i, \ldots, [\![z_{i1}]\!]_i, [\![\delta_{ji}]\!]$
8: C_i: $\delta_{ij} \leftarrow$ DGKDECRYPT($[\![z_{i\mu}]\!]_i, \ldots, [\![z_{i1}]\!]_i$)
9: $C_i \to S$: $[\![\delta_{ij} \oplus \delta_{ji}]\!]$
10: S : **return** $[\![b_{ij}]\!] = [\![\delta_{ij} \oplus \delta_{ji}]\!]$

Protocol 7.4: DGK Comparison Protocol With the Server

1: **function** COMPUTEKREAHE(G, X, k)
2: **parse** G **as** $[g_{11}, \ldots, g_{nn}]$ and X **as** $[[\![x_1]\!], \ldots, [\![x_n]\!]]$
3: **for** $i \coloneqq 1$ **to** n **do**
4: $[\![r_i]\!] \leftarrow g_{ii}$
5: **for** $j \coloneqq 1$ **to** n $(j \neq i)$ **do**
6: **if** PAIRED(i, j) **then**
7: $[\![r_i]\!] \leftarrow [\![r_i]\!] \cdot g_{ij}$
8: **else**
9: $[\![r_i]\!] \leftarrow [\![r_i]\!] \cdot [\![1]\!] \cdot g_{ji}^{-1}$
10: **for** $i \coloneqq 1$ **to** n **do**
11: $y_i \leftarrow ([\![r_i]\!] \cdot [\![k]\!]^{-1})^{\alpha_i} \cdot [\![x_i]\!]$
12: **return** $[y_1, \ldots, y_n]$

Algorithm 7.5: Computing the KRE's ciphertext in KRE-AHE2

the DGK protocol as generator, and C_j is the evaluator. After the computation, C_i and C_j get shares δ_{ij} and δ_{ji} of the comparison bit which is encrypted under pk as $[\![b_{ij}]\!] = [\![\delta_{ij} \oplus \delta_{ji}]\!]$ and revealed to the server.

Computing the KRE's Ciphertext. After all admissible comparisons have been computed (and the result stored in the matrix G), the server uses Algorithm 7.5 to compute the rank of each input x_i by homomorphically adding the comparison bits involving x_i. Let $[\![r_i]\!]$ be a ciphertext initially encrypting 0 and let $b_{ij} = \delta_{ij} \oplus \delta_{ji}$. For each j, if PAIRED(i, j) is true (i.e., $[\![b_{ij}]\!]$ has been computed) then we compute $[\![r_i]\!] \leftarrow [\![r_i + b_{ij}]\!]$. Otherwise (i.e., $[\![b_{ij}]\!]$ has not been computed but we can deduce it from $[\![b_{ji}]\!]$) we compute $[\![r_i]\!] \leftarrow [\![r_i + 1 - b_{ij}]\!]$. Now, the server has the encrypted rank $[\![r_1]\!], \ldots, [\![r_n]\!]$, where exactly one $[\![r_i]\!]$ encrypts k. Since we are looking for the element whose rank is k, the server then computes $y_i = ([\![r_i]\!] \cdot [\![k]\!]^{-1})^{\alpha_i} \cdot [\![x_i]\!] = [\![(r_i - k)\alpha_i + x_i]\!]$ for all i, where α_i is a number chosen randomly in the plaintext space.

Therefore, for the ciphertext $[\![r_i]\!]$ encrypting k, y_i is equal to $[\![x_i]\!]$. Otherwise y_i encrypts a random plaintext.

Decrypting the KRE's Ciphertext. In Step 12, the server distributes the result $Y = [y_1, \ldots, y_n]$ of Algorithm 7.5 to the clients for threshold decryption. For that, the array Y is passed as $n \times 1$ matrix to Algorithm 7.2. In Step 16, the server

receives partial decryption results from the clients, forwards them to the corresponding combiner (Step 18). Each combiner C_j performs a final decryption (Step 21) resulting in a message $\tilde{x}_j$ whose bitlength is less or equal to μ if it is the KRE. Combiner C_j then sets $m^{(j)} = \tilde{x}_j$ if $|\tilde{x}_j| \leq \mu$, otherwise $m^{(j)} = 0$ (Step 22). Then $m^{(j)}$ is encrypted with the public key of all clients and send to the server (Step 23). Finally, the server reveals the KRE to all clients (Step 25).

1: **for** $i \coloneqq 1$ **to** n **do**
2: $\quad C_i \to S$: $[\![x_i]\!], [\![x_i^{\mathsf{b}}]\!]_i$
3: S : **let** $G = [g_{11}, \ldots, g_{nn}]$ and $X = [[\![x_1]\!], \ldots, [\![x_n]\!]]$
4: **for** $i \coloneqq 1, j \coloneqq i+1$ **to** n **do**
5: $\quad C_i, C_j, S$: $g_{ij} \leftarrow$ DGKCOMPARE(i, j)
6: S: $Y \leftarrow$ COMPUTEKREAHE(G, X, k)
7: S: **let** $\pi \xleftarrow{\$} \mathfrak{S}_n$ be a permutation
8: S: **parse** Y **as** $[y_1, \ldots, y_n]$
9: **for** $i \coloneqq 1$ **to** n **do**
10: $\quad S$: $y_i \leftarrow y_{\pi(i)}$
11: **for** $i \coloneqq 1$ **to** n **do**
12: $\quad S \to C_i$: $(Z^{(i)}, I^{(i)}) \leftarrow$ DECREQ(Y, i, t, π)
13: $\quad C_i$: **parse** $Z^{(i)}$ as $[z_{j_1}^{(i)}, \ldots, z_{j_t}^{(i)}]$
14: **for** $i \coloneqq 1$ **to** n **do**
15: $\quad$ **for each** j **in** $I^{(i)}$ **do**
16: $\quad\quad C_i \to S$: $h_j^{(i)} \leftarrow [\![\mathsf{Dec}_p(\langle \mathsf{sk} \rangle_i, z_j^{(i)})]\!]_j$
17: **for** $j \coloneqq 1$ **to** n **do**
18: $\quad S \to C_j$: $(h_j^{(i_1)}, \ldots, h_j^{(i_t)})$
19: **for** $j \coloneqq 1$ **to** n **do**
20: $\quad C_j$: $d_1 = Dec(\mathsf{sk}_j, h_j^{(i_1)}), \ldots, d_t = Dec(\mathsf{sk}_j, h_j^{(i_t)})$
21: $\quad C_j$: $\tilde{x}_j \leftarrow \mathsf{Dec}_f(d_1, \ldots, d_t)$
22: $\quad C_j$: $m^{(j)} \coloneqq \begin{cases} \tilde{x}_j & \text{if } |\tilde{x}_j| \leq \mu \\ 0 & \text{if } |\tilde{x}_j| > \mu \end{cases}$
23: $\quad C_j \to S$: $[\![m^{(j)}]\!]_1, \ldots, [\![m^{(j)}]\!]_n$
24: **for** $i \coloneqq 1$ **to** n **do**
25: $\quad S \to C_i$: $[\![\sum_{j=1}^{n} m^{(j)}]\!]_i$
26: $\quad C_i$: $Dec(\mathsf{sk}_i, [\![\sum_{j=1}^{n} m^{(j)}]\!]_i)$

Protocol 7.6: KRE-AHE2 Protocol

Brief Summary. KRE-AHE2 protocol correctly computes the KRE. This trivially follows from the correctness of DGK protocol [63], Lemmas 7.2.5 and 7.4.1, and the correctness of AHE. KRE-AHE2 evaluates comparisons interactively but requires threshold decryption for $O(n)$ elements. Notice that KRE-AHE2 can be instantiated with the Lin-Tzeng protocol [135] as well. To compare x_i, x_j, C_j will receive both $[\![V_{x_i}^0]\!], [\![V_{x_i}^1]\!]$ and randomly choose between evaluating LINCOMPARE either with $[\![V_{x_i}^1]\!], [\![V_{x_j}^0]\!]$ or with $[\![V_{x_j}^1]\!], [\![V_{x_i}^0]\!]$. It will then set $\delta_{ji} \leftarrow 0$ or $\delta_{ji} \leftarrow 1$ accordingly. This improves the running time (LINCOMPARE is more efficient than DGKEVAL) while increasing the communication (μ more ciphertexts are sent to C_j).

```
1: function ComputeKreShe(X, Z, [[k^b]])
2:     parse X as [[x_1^b]], ..., [[x_n^b]]] and Z as [[[x_1]], ..., [[x_n]]]
3:     for i := 1 to n do
4:         [[b_ii]] ← [[1]]
5:         for j := i + 1 to n do
6:             ([[b_ij]], [[b_ji]]) ← SheCmp([[x_i^b]], [[x_j^b]])
7:     for i := 1 to n do
8:         [[r_i^b]] ← SheFadder([[b_i1]], ..., [[b_in]])
9:     for i := 1 to n do
10:        [[β_i]] ← SheEqual([[r_i^b]], [[k^b]])
11:    for i := 1 to n do
12:        [[y_i]] ← SheMult([[x_i]], [[β_i]])
13:    return SheAdd([[y_1]], ..., [[y_n]])
```

Algorithm 7.7: Computing the KRE's Ciphertext in Kre-She

In Kre-Ahe1 and Kre-Ahe2, we evaluated either the comparison (Kre-Ahe1) or the rank (Kre-Ahe2) completely at the server. In the next scheme, we compute the KRE's ciphertext non-interactively at the server such that clients are only required for the threshold decryption of one ciphertext.

7.6 A SHE-Based Construction

This section describes Kre-She based on SHE. Hence, $[\![x]\!]$ now represents a SHE ciphertext of the plaintext x. The initialization and threshold decryption are similar to Kre-Ahe1. We will use homomorphic routines for addition, multiplication, greater-than comparison, full adder, equality test, and shift as described in Section 2.1.4.

7.6.1 Main Protocol

In Protocol 7.8 the server S receives encrypted inputs from clients. For each client's integer x_i, the encrypted input consists of:

- an encryption of the bit representation: $[\![x_i^{\mathsf{b}}]\!] = ([\![x_{i\mu}]\!], \ldots, [\![x_{i1}]\!])$ and
- an encryption of the packed bit representation: $[\![\vec{x}_i]\!] = [\![x_{i\mu}|\ldots|x_{i1}|0|\ldots|0]\!]$.

Then the server runs Algorithm 7.7 which uses SheCmp to pairwise compare the inputs resulting in encrypted comparison bits $[\![b_{ij}]\!]$. Then SheFadder is used to compute the rank of each input by adding comparison bits. The result is an encrypted bit representation $[\![r_i^{\mathsf{b}}]\!]$ of the ranks. Using the encrypted bit representations $[\![k^{\mathsf{b}}]\!]$, $[\![r_i^{\mathsf{b}}]\!]$ of k and each rank, SheEqual checks the equality and returns an encrypted bit $[\![\beta_i]\!]$. Recall that because of SVCP the encryption of a bit β_i is automatically replicated in all slots, i.e., $[\![\beta_i]\!] = [\![\beta_i|\beta_i|\ldots|\beta_i]\!]$, such that evaluating $[\![\vec{y}_i]\!] \leftarrow \textsc{SheMult}([\![\vec{x}_i]\!], [\![\beta_i]\!])$, $1 \le i \le n$, and $\textsc{SheAdd}([\![\vec{y}_1]\!], \ldots, [\![\vec{y}_n]\!])$ returns the KRE's ciphertext.

Correctness and security follow trivially from Lemma 7.2.5, correctness and security of SHE. The leakage is $\mathcal{L}_S = \mathcal{L}_i = \{n, t, \kappa, \lambda, \mu\}$.

for $i \coloneqq 1$ **to** n **do**
  $C_i \to S$: $[\![x_i^{\mathsf{b}}]\!], [\![\vec{x}_i]\!]$
S : **let** $X = [[\![x_1^{\mathsf{b}}]\!], \ldots, [\![x_n^{\mathsf{b}}]\!]]$ and $Z = [[\![\vec{x}_1]\!], \ldots, [\![\vec{x}_n]\!]]$
$S : [\![x_{i^*}]\!] \leftarrow \textsc{ComputeKreShe}(X, Z, [\![k^{\mathsf{b}}]\!])$
S: **let** $\mathbb{I}_t = \{i_1, \ldots, i_t\} \xleftarrow{\$} \{1, \ldots, n\}$
for all $i \in \mathbb{I}_t$ **do**
  $S \to C_i$: $[\![x_{i^*}]\!]$
for all $i \in \mathbb{I}_t$ **do**
  C_i: $m^{(i)} \leftarrow \mathsf{Dec}_p(\langle \mathsf{sk} \rangle_i, [\![x_{i^*}]\!])$
  $C_i \to S$: $[\![m^{(i)}]\!]_1, \ldots, [\![m^{(i)}]\!]_n$
for $i \coloneqq 1$ **to** n **do**
  $S \to C_i$: $[\![m^{(i_1)}]\!]_i, \ldots, [\![m^{(i_t)}]\!]_i$
  C_i: $\mathsf{Dec}_f(m^{(i_1)}, \ldots, m^{(i_t)})$

Protocol 7.8: Kre-She Protocol

7.7 Security Analysis

Let the inherent leakage be $\mathcal{L} = \{k, n, t, \kappa, \lambda, \mu\}$, i.e., the parameters of the protocol.

Theorem 7.7.1. *If the server S is non-colluding and the AHE scheme is IND-CPA secure, then* Kre-Ygc *1-privately computes $\mathcal{F}_{\text{KRE}}$ in the semi-honest model with leakage $\mathcal{L}_S = \mathcal{L}_i = \mathcal{L}$. Hence, let the view of a party P in the protocol* Kre-Ygc *be denoted by $\mathsf{View}_{\mathsf{P}}^{\Pi_{\text{KRE}}}(x_1, \ldots, x_n)$. Then there are simulators $\mathsf{Sim}_{\mathsf{C_i}}^{\text{kre}}$ for each C_i and $\mathsf{Sim}_{\mathsf{S}}^{\text{kre}}$ for S such that:*

$$\mathsf{Sim}_{\mathsf{S}}^{\text{kre}}(\varnothing, \mathcal{L}_S) \overset{c}{\equiv} \mathsf{View}_{\mathsf{S}}^{\Pi_{\text{KRE}}}(x_1, \ldots, x_n) \text{ and}$$

$$\mathsf{Sim}_{\mathsf{C_i}}^{\text{kre}}(x_i, \mathcal{F}_{\text{KRE}}(x_1, \ldots, x_n), \mathcal{L}_i) \overset{c}{\equiv} \mathsf{View}_{\mathsf{C_i}}^{\text{Kre-Ygc}}(x_1, \ldots, x_n).$$

Proof. The view $\mathsf{View}_{\mathsf{C_i}}^{\Pi_{\text{KRE}}}$ consists of:

$$(F_{>}^{ij}, (\bar{a}_i^{ij}, \bar{x}_i^{ij}), b'_{ij}, [\![a_j^{ij} \oplus b_{ij}]\!]_i, [\![a_i^{ij} \oplus b_{ij}]\!]_j, [\![b_{ij}]\!]_j)_{1 \le j \le n (i \ne j)}, \\ [\![\beta_i]\!]_i, \beta_i.$$

The garbled circuit $F_{>}^{ij}$ and garbled input $(\bar{a}_i^{ij}, \bar{x}_i^{ij})$ are generated by client C_i itself. The remaining elements of the view are either an IND-CPA ciphertext $[\![\cdot]\!]_j$ or a random bit b'_{ij}, $a_j^{ij} \oplus b_{ij}$ (a_j^{ij} is a random bit chosen by C_j) or random integer β_i (β_i is a random integer or 0. It is 0 if and only if x_i is the k^{th}-ranked element, which is revealed anyway at the end of the protocol). For each $m \in \mathsf{View}_{\mathsf{C_i}}^{\Pi_{\text{KRE}}}$, $\mathsf{Sim}_{\mathsf{C_i}}^{\text{kre}}$ only needs the bit length $|m|$ of m and simulates it by choosing a random bit strings of length $|m|$. As a result, the leakage to the adversary is $\mathcal{L}$.

The view of the server consists of:

$$\langle F_{>}^{ij}, (\bar{a}_i^{ij}, \bar{x}_i^{ij}), (\bar{a}_j^{ij}, \bar{x}_j^{ij}), b'_{ij}, \\ [\![a_j^{ij} \oplus b_{ij}]\!]_i, [\![a_i^{ij} \oplus b_{ij}]\!]_j, [\![a_i^{ij} \oplus b_{ij}]\!]_j, [\![a_j^{ij} \oplus b_{ij}]\!]_i, \\ [\![a_{ij}]\!]_j, [\![b_{ij}]\!]_i\rangle_{\text{Paired}(i,j)=true}, \langle [\![r_i]\!]_i, [\![\beta_i]\!]_i\rangle_{1 \le i \le n}, \langle [\![m_i]\!]_j\rangle_{1 \le i,j \le n}.$$

The simulation is similar to the case of the client. For each $m \in \mathsf{View}_{\mathsf{S}}^{\Pi_{\mathrm{KRE}}}$, $\mathsf{Sim}_{\mathsf{S}}^{\mathrm{kre}}$ chooses random bit strings of length $|m|$. □

Theorem 7.7.2. *Let $t \in \mathbb{N}$ and $\tau < t$. If the server S is non-colluding and the AHE scheme is IND-CPA secure, then* Kre-Ahe1 *and* Kre-Ahe2 *τ-privately compute $\mathcal{F}_{\mathrm{KRE}}$ in the semi-honest model with leakage $\mathcal{L}_S = \mathcal{L}_i = \mathcal{L}$. Hence, let $I = \{i_1, \ldots, i_\tau\}$ denote the indexes of corrupt clients, $\mathcal{L}_I = \bigcup_{i \in I} \mathcal{L}_i$ denote their joint leakages and $\mathsf{View}_{\mathsf{I}}^{\Pi_{\mathrm{KRE}}}(x_1, \ldots, x_n)$ denote their joint views, there exists a simulator $\mathsf{Sim}_{\mathsf{I}}^{\mathrm{kre}}$ such that:*

$$\mathsf{Sim}_{\mathsf{I}}^{\mathrm{kre}}((x_{i_1}, \ldots, x_{i_\tau}), \mathcal{F}_{\mathrm{KRE}}(x_1, \ldots, x_n), \mathcal{L}_I) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{I}}^{\Pi_{\mathrm{KRE}}}(x_1, \ldots, x_n).$$

Proof. If the server S is non-colluding and the number of colluding clients is smaller than t, then parties see in Kre-Ahe1 only random strings (IND-CPA ciphertexts, random shares, or random bits). Hence, all messages can be simulated by choosing random bit strings of the corresponding length. However, the simulation of Step 19 must be coherent with Step 25. Each client receives random shares in Step 19, runs the final decryption $\mathsf{Dec}_f(.)$ in Step 25 and learns a random bit. Let C_l be a client with $l \in I$. To simulate Steps 19 and 25, the simulator chooses t random values for Step 19 such that running $\mathsf{Dec}_f(.)$ returns the random bit simulated in Step 25.

For example, if the underlying AHE is ECC ElGamal (ECE), then a ciphertext has the form $c = (\alpha_1, \alpha_2) = (r \cdot P, m \cdot P + r \cdot \mathsf{pk})$. For each ECE ciphertext $c = (\alpha_1, \alpha_2) = (r \cdot P, m \cdot P + r \cdot \mathsf{pk})$ that must be decrypted in Step 25, C_l gets α_2 and t partial decryption results $\alpha_{11}, \ldots, \alpha_{1t}$ of α_1 in Step 19. To simulate this, the simulator chooses a random bit b and a random $\tilde{\alpha}_2$. Then it computes $\tilde{\alpha}_1 = \tilde{\alpha}_2 - b \cdot P$ and generates random $\tilde{\alpha}_{11}, \ldots, \tilde{\alpha}_{1t}$ such that $\sum_{i=1}^{t} \tilde{\alpha}_{1i} = \tilde{\alpha}_1$ in $\mathbb{G}$. An adversary controlling a number of clients smaller than t can only deduce the length of random strings, the number of ciphertexts, the number of iterations. As a result, the leakage to that adversary is $\mathcal{L}$.

The proof for Kre-Ahe2 is similar. Ciphertexts and random shares are simulated with equally long random strings and Steps 18 and 22 in Kre-Ahe2 are simulated as above for Steps 19 and 25 in Kre-Ahe1. □

Theorem 7.7.3. *Let $t \in \mathbb{N}$ and $\tau < t$. If the server S is non-colluding and the SHE scheme is IND-CPA secure, then* Kre-She *τ-privately computes $\mathcal{F}_{\mathrm{KRE}}$ in the semi-honest model with leakage $\mathcal{L}_S = \mathcal{L}_i = \mathcal{L}$. Hence, let $I = \{i_1, \ldots, i_\tau\}$ denote the indexes of corrupt clients, $\mathcal{L}_I = \bigcup_{i \in I} \mathcal{L}_i$ denote their joint leakages and $\mathsf{View}_{\mathsf{I}}^{\Pi_{\mathrm{KRE}}}(x_1, \ldots, x_n)$ denote their joint views, there exists a simulator $\mathsf{Sim}_{\mathsf{I}}^{\mathrm{kre}}$ such that:*

$$\mathsf{Sim}_{\mathsf{I}}^{\mathrm{kre}}((x_{i_1}, \ldots, x_{i_\tau}), \mathcal{F}_{\mathrm{KRE}}(x_1, \ldots, x_n), \mathcal{L}_I) \stackrel{c}{\equiv} \mathsf{View}_{\mathsf{I}}^{\Pi_{\mathrm{KRE}}}(x_1, \ldots, x_n).$$

Proof. If the server S is non-colluding and the number of colluding clients is smaller than t, then parties see in Kre-She only random strings (IND-CPA ciphertexts or partial decryption results). The security is also straightforward as the computation is almost completely done by the server alone and encrypted under an IND-CPA encryption. Moreover, the partial decryption reveals only a partial result to each decryptor. □

Recall that our adversary is semi-honest. In Kre-Ygc, a server collusion reveals all inputs to the adversary. In Kre-Ahe1 and Kre-Ahe2, a server collusion only increases the leakage as long as the number of corrupted clients is smaller than t. For example in Kre-Ahe1, the adversary can learn the order of the inputs whose comparison bits are final-decrypted by a corrupted client in Step 25. In Kre-She,

the KRE is homomorphically computed by the server such that the clients are only required for the decryption of one ciphertext encrypting the KRE. Moreover, the ciphertexts are encrypted using the threshold public key. As a result, assuming a semi-honest adversary and a collusion set containing less than t clients, a server collusion leaks no more information than $k, n, t, \kappa, \lambda, \mu$.

7.8 Complexity Analysis

In this section, we discuss the complexity of our schemes. We will use κ and λ as length of asymmetric ciphertext and symmetric security parameter.

7.8.1 Complexity of the GC-Based Protocol

A GC for the comparison of two μ-bit integers consists of μ AND-gates resulting in 4μ symmetric ciphertexts [130, 131]. It can be reduced by a factor of 2 using the *halfGate* optimization [197] at the cost of performing two cheap symmetric operations (instead of one) during GC evaluation.

We do the analysis for the case where n is odd (the even case is similar). From Lemma 7.2.8, each client generates $(n-1)/2$ GCs resulting in $(n-1)\mu$ symmetric operations. The computation of encrypted comparison bits (Steps 6 to 16) and the computation of the KRE's ciphertext require $O(n)$ asymmetric operations to each client. Finally, each client has to decrypt one ciphertext in Step 23. As a result, the computation complexity of each client is, therefore, $O((n-1)\mu)$ symmetric and $O(2n+1)$ asymmetric operations. In communication, this results in $n\kappa$ bits for the asymmetric ciphertexts, $2\mu\lambda(n-1)/2$ bits for the GCs and $\mu\lambda(n-1)/2$ for the garbled inputs and $n\kappa$ bits for handling the server's leakage. In total each client sends $2n\kappa + \frac{3\mu\lambda(n-1)}{2}$.

The server evaluates $n(n-1)/2$ GCs each consisting of 2μ symmetric ciphertexts. Computing the rank (Steps 17 to 19) requires $O(n \log n + n)$ operations to the server. Finally, the server evaluates $\log n + n$ asymmetric operations to compute the KRE ciphertext for each client (Steps 23 to 24). The total computation complexity of the server is $O(n(n-1)\mu)$ symmetric and $O((n+1)\log n + 2n)$. In communication, the server sends $n(n-1)$ asymmetric ciphertexts in Steps 6 to 16, n asymmetric ciphertexts in Steps 17 to 19 and n asymmetric ciphertexts in Steps 23 to 24. This results in a total of $(n^2+n)\kappa$ bits.

7.8.2 Complexity of the AHE-Based Protocol

Each client performs $O(\mu)$ operations in Step 2, $O(n\mu t)$ operations in Step 17, $O(n\mu t)$ operations in Step 24, $O(\log t)$ operations in Step 25, $O(1)$ operations in Step 28, eventually $O(1)$ and $O(n)$ operations in Steps 34 and 35, and $O(\log t)$ operations in Step 38. This results in a total of $O(\mu + 2n\mu t + 2\log t + n + 1)$ asymmetric operations.

Each client sends $(2\mu+1)\kappa$ bits in Step 2, $n\mu t\kappa$ bits in Step 17, κ bits in Step 28, eventually $n\kappa$ bits in Step 35. This results in a total of $(2\mu + n\mu t + n + 2)\kappa$ bits for each client.

The main cryptographic operations of the server happen in the evaluation of the Lin-Tzeng protocol in Step 4. The comparison of two values takes 2μ asymmetric operations. As a result, the server performs $O(2\mu n^2)$ asymmetric operations for all comparisons.

The server sends $n^2\mu t\kappa$ bits in Step 11 and $(n^2t+1)\kappa$ bits in Step 19, $t\kappa$ bits in Step 32 and $nt\kappa$ bits in Step 37. This results in a total of $(n^2\mu t + n^2t + nt + t + 1)\kappa$ bits for the server.

7.8.3 Complexity of the Improved AHE-Based Protocol

Since Kre-Ahe2 also requires the predicate Paired as Kre-Ygc, we do the analysis for the case where n is odd (the even case is similar).

Each client performs $O(\mu+1)$ operations in Step 2, $O(\frac{7\mu(n-1)}{2})$ operations in Step 5, $O(t)$ operations in Step 16 and $O(\log t)$ in Step 21, $O(n)$ operations in Step 23 and $O(1)$ operations in Step 26. This results in a total of $O(\mu + \frac{7\mu(n-1)}{2} + t + \log t + n + 1)$ asymmetric operations.

Each client sends $(\mu+1)\kappa$ bits in Step 2, $\frac{\kappa(n-1)}{2}$ bits (when the client is head) and $\frac{(\mu+1)\kappa(n-1)}{2}$ (when the client is tail) in Step 5, $t\kappa$ bits in Step 16 and $n\kappa$ bits in Step 23. This results in a total of $(\mu\frac{(n+1)}{2} + 2n + t)\kappa$ bits for each client.

The cryptographic operations of the server happen in ComputeKreAhe (Algorithm 7.5) that is called in Step 6 of Protocol 7.6. The server performs $O(n^2+n)$ asymmetric operations.

The server sends $\frac{(\mu\kappa+(\mu+1)\kappa)n(n-1)}{2}$ bits in Step 5, $nt\kappa$ bits in Steps 12 and 18, $n\kappa$ bits in Step 25. This results in a total of $(\frac{(2\mu+1)n(n-1)}{2} + 2nt + n)\kappa$ bits for the server.

7.8.4 Complexity of the SHE-Based Protocol

Each client has $O(\mu)$ computation cost ($\mu+1$ encryptions in Step 2 and eventually one partial decryption in Step 10) and a communication cost of $(\mu+n+1)\kappa$ bits.

The cryptographic operations of the server happen in ComputeKreShe (Algorithm 7.7) that is called in Step 4 of Protocol 7.8. The SHE comparison circuit has depth $\log(\mu-1)+1$ and requires $O(\mu\log\mu)$ homomorphic multiplications [46, 47]. For all comparisons, the server, therefore, performs $O(n^2\mu\log\mu)$ multiplications. In Step 10 of Algorithm 7.7, the computation of $[\![\prod_{j=1,j\neq k}^{n}(r_i-j)]\!]$ has depth $\log n$ and requires $O(n\log n)$ homomorphic multiplications. Step 12 of Algorithm 7.7 adds an additional circuit depth and requires $O(n)$ homomorphic multiplications. As a result, Algorithm 7.7 has a total depth of $\log(\mu-1)+\log n+2$ and requires $O(n^2\mu\log\mu + n\log n + n)$ homomorphic multiplications.

The server sends $t\kappa$ bits in Step 7 and $nt\kappa$ bits in Step 12 resulting in a total of $(t+nt)\kappa$ bits.

7.9 Evaluation

In this section, we describe implementation details and report on the experimental results of our implementations.

7.9.1 Implementation Details

We implemented Kre-Ygc, Kre-Ahe1, Kre-Ahe2 as client-server Java applications while using SCAPI [75]. As Kre-She mostly consists of the homomorphic evaluation by the server, we implemented Algorithm 7.7 using HElib [100].

The threshold decryption in Kre-Ahe1, Kre-Ahe2 has been implemented using elliptic curve ElGamal (ECE) [127]. We briefly present ECE and its threshold decryption [33]. Let $\mathbb{G}$ be an elliptic curve group generated by a point P of prime

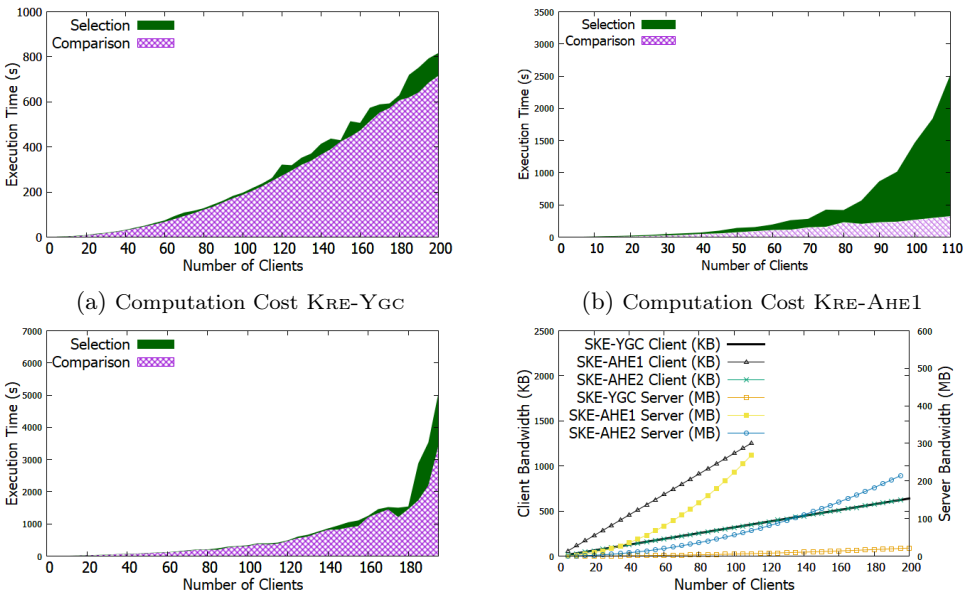
order p. The key generation chooses $s \xleftarrow{\$} \mathbb{Z}_p$ and outputs $\mathsf{sk} = s$ and $\mathsf{pk} = s \cdot P$ as private and public key. To encrypt an integer m, one chooses $r \xleftarrow{\$} \mathbb{Z}_p$ and outputs the ciphertext $c = (r \cdot P, m \cdot P + r \cdot \mathsf{pk})$. To decrypt a ciphertext $c = (\alpha_1, \alpha_2)$, one computes $Q = \alpha_2 - \alpha_1 \cdot \mathsf{sk}$ and solves the discrete logarithm on $\mathbb{G}$.
Let n, t be integers such that $t \leq n$. To support t-out-of-n threshold decryption the secret key $\mathsf{sk} = s$ is secret-shared using Shamir secret sharing scheme [170] as described in Section 2.1.3. The threshold key generation outputs secret key shares $\langle \mathsf{sk} \rangle_i = (\langle s \rangle_i, l_i), 1 \leq i \leq n$, where $\langle s \rangle_i = f(i)$, and f and l_i are defined in Equations 2.1 and 2.2, respectively. Let $\mathbb{I}_t \subseteq \{1, \ldots, n\}$ be a subset of t clients and assume for simplicity $\mathbb{I}_t = \{1, \ldots, t\}$. To decrypt a ciphertext $c = (\alpha_1, \alpha_2)$ each client $C_i, i \in \mathbb{I}_t$ computes $m_i = \alpha_1 \cdot s_i \cdot l_i$. Then the combiner receives all m_i, computes $\alpha_2 - \sum_{i=1}^{t} m_i = \alpha_2 - \alpha_1 \cdot \sum_{i=1}^{t} s_i \cdot l_i = \alpha_2 - \alpha_1 \cdot s = Q$ and solve the discrete logarithm on $\mathbb{G}$. This requires $O(1)$ to each client $C_i, i \in \mathbb{I}_t$ and $O(\log t)$ asymmetric operations to the combiner.

7.9.2 Experimental Setup

For Kre-Ygc, Kre-Ahe1, Kre-Ahe2, we conducted experiments using for the server a machine with a 6-core Intel(R) Xeon(R) E-2176M CPU @ 2.70GHz and 32GB of RAM, and for the clients two machines with each two Intel(R) Xeon(R) CPU E7-4880 v2 @ 2.50GHz. The client machines were equipped with 8GB and 4 GB of RAM, and were connected to the server via WAN. Windows 10 Enterprise was installed on all three machines. For each experiment, about 3/5 of the clients were run on the machine with 8 GB RAM while about 2/5 were run on the machine with 4 GB RAM. We ran all experiments using JRE version 8.

Since the main computation of Kre-She is done on the server, we focus on the evaluation Algorithm 7.7 on a Laptop with Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz running 16.04.1-Ubuntu with 4.10.0-14-lowlatency Kernel version.

7.9.3 Results



(a) Computation Cost Kre-Ygc

(b) Computation Cost Kre-Ahe1

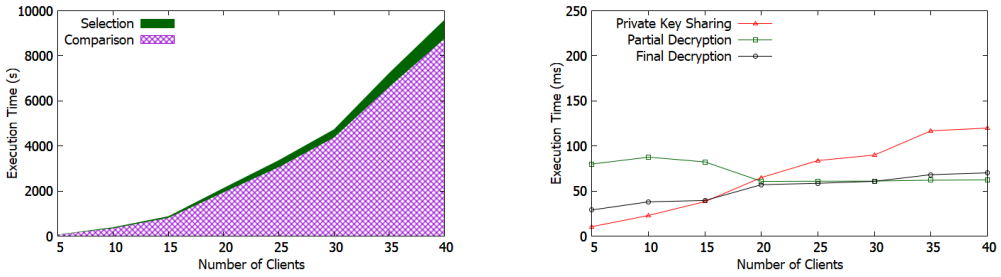
(c) Computation Cost Kre-Ahe2

(d) Communication Cost in MB

Figure 7.9: Performance Results for Kre-Ygc, Kre-Ahe1, Kre-Ahe2

	KRE-YGC	KRE-AHE2			KRE-AHE1	
t	n/a	1	2	100	1	2
Time (s)	197.00	353.00	336.00	441.00	1024.00	1749.00
C-Bits (MB)	0.31	0.30	0.30	0.32	0.56	1.11
S-Bits (MB)	5.42	56.07	56.12	60.56	111.37	222.67

Table 7.4: Performance Comparison for 100 clients: C-Bits (resp. S-Bits) denotes the number of bits sent by each client (resp. the server). t is the secret sharing threshold, i.e., the number of clients that must contribute to the threshold decryption.



(a) Server Computation Cost (b) Threshold Decryption Cost

Figure 7.10: Performance Results KRE-SHE

We evaluated the schemes KRE-YGC, KRE-AHE1, KRE-AHE2 at security level $\lambda = 128$, bitlength $\mu = 32$, and (minimal) threshold $t = 2$ for threshold decryption. We instantiated KRE-AHE1 and KRE-AHE2 with Elliptic Curve ElGamal using elliptic curve `secp256r1`. Figure 7.9 shows our performance results which are summarized in Table 7.4 for $n = 100$ clients. In Table 7.4, we also illustrate the costs when $t = 1$ (i.e., each C_i knows `sk`) for both KRE-AHE2 and KRE-AHE1 and when $t = n$ (i.e. all C_i must participate in the threshold decryption) for KRE-AHE2.

KRE-YGC is the most efficient in both computation and communication and takes 203 seconds to each client to compute the KRE of 100 clients in a WAN setting. The communication is 0.31 MB for each client and 5.42 MB for the server. However, KRE-YGC is neither collusion-resistant nor fault-tolerant.

KRE-AHE2 is the second most efficient and is collusion-resistant and fault-tolerant. Although it requires more interactions to compute comparisons, we batched many comparisons together and were able to run threshold decryption for $O(n)$ elements, instead of $O(n^2)$ as in KRE-AHE1. The computation of the KRE of 100 values takes to each client 353 seconds (for $t = 1$), 336 seconds (for $t = 2$) and 441 seconds (for $t = 100$). The communication is 0.3 MB, 0.3 MB, 0.32 MB for each client and 56.07 MB, 56.12 MB, 60.56 MB for the server when $t = 1, 2, 100$, respectively.

While being collusion-resistant and fault-tolerant as well, KRE-AHE1 is less efficient than KRE-YGC and KRE-AHE2. The computation of the KRE of 100 values takes to each client 1024 seconds (for $t = 1$), 1749 seconds (for $t = 2$). The communication is 0.56 MB, 1.11 MB for each client, and 111.37 MB, 222.67 MB for the server when $t = 1, 2$, respectively. For $t = 100$, our testbed ran out of memory.

We evaluated Algorithm 7.7 of KRE-SHE at security level at least 110. The result is illustrated in Figure 7.10a for inputs with bitlength $\mu = 16$. The computation is dominated by the inputs' comparison and takes less than one hour for 25 clients. We

also evaluated in Figure 7.10b the performance of the threshold decryption with an n-out-of-n secret sharing. For up to 40 clients threshold decryption costs less than 0.15 second. KRE-SHE is practically less efficient than all other schemes, but has the best asymptotic complexity.

As a result, KRE-YGC is suitable for a setting where the server is non-colluding and clients cannot fail. If collusion and failure are an issue, then either KRE-AHE1 or KRE-AHE2 or even KRE-SHE is suitable. KRE-AHE1 can be more time efficient than KRE-AHE2 for up to 30 clients and a highly parallelizable server. KRE-SHE has the best asymptotic complexity, however, it requires more efficient somewhat homomorphic encryption schemes.

7.10 Summary

In this chapter, we considered the problem of computing the KRE (with applications to benchmarking) of n clients' private inputs using a server. The general idea of our solution is to sort the inputs, compute the rank of each input, use it to compute the KRE. The computation is supported by the server which coordinates the protocol and undertakes as much computations as possible. We proposed and compared different approaches based on garbled circuits or threshold HE. The server is oblivious, and does not learn the input of the clients. We also implemented and evaluated our schemes. As a result, KRE-YGC is suitable for a setting where the server is non-colluding and clients cannot fail. If collusion and failure are an issue, then either KRE-AHE2 or KRE-SHE is suitable. KRE-SHE has the best asymptotic complexity, however, it requires more efficient somewhat homomorphic encryption schemes.

Chapter 8

Conclusion

This chapter concludes the thesis. In Section 8.1, we summarize the results of our work on order-preserving encryption, decision tree classification, and the k^{th}-ranked element. Section 8.2 describes an outlook of this thesis with respect to further research directions.

8.1 Summary

Order-preserving encryption (OPE) schemes are efficient and easy to deploy encryption schemes that allow encrypting data, while still enabling efficient range queries on the encrypted data. OPE is, therefore, suitable for encrypted cloud databases. However, OPE is symmetric, limiting the use case to one client and a server. In Chapter 4, we overcame this limitation by introducing the concept of oblivious OPE which is basically the equivalent of a public-key OPE. OOPE is based on stateful OPE where the order of plaintexts is encoded on a remote server using a search tree with semantically secure encrypted nodes. In OOPE, we replaced the encoding protocol by a mix-technique secure multiparty computation combining garbled circuit, homomorphic encryption, and secret sharing. Our evaluation showed acceptable performance which however depends on the network performance between the parties.

As a concrete application of OOPE, we mentioned a supply chain scenario, where a Data Analyst is a supplier (manufacturer) owning a private decision tree model and is interested in optimizing its manufacturing process using private data owned by its buyer (another supplier or distributor). Each path in the decision tree is basically a conjunction of range queries, which OOPE can handle while preserving the privacy of both Data Owner and Data Analyst.

We also addressed the scenario of evaluating private decision trees on private data which consists of a server holding a private decision tree and a client holding a private attribute vector. The goal is to classify the client's attribute vector using the server's decision tree model such that the result of the classification is revealed only to the client and nothing else is revealed neither to the client nor the server. We proposed two solutions to the problem.

In our first solution presented in Chapter 5, we represented the tree as an array and executed only d interactive comparisons (instead of 2^d as in existing solutions), where d denotes the depth of the tree. We used a small garbled circuit to perform each comparison and compute the index of the next node which we secret-shared between the parties. To actually select the next node in the tree, we used a primitive called oblivious array indexing (OAI) which, given an array and secret shares on an index, returns secret shares of the indexed element to the parties. We showed how to efficiently implement OAI using a garbled circuit, oblivious transfer, ORAM, and PIR.

In Chapter 6, we delegated the complete tree evaluation to the server using fully or somewhat homomorphic encryption where the ciphertexts are encrypted under the client's public key. As current homomorphic encryption schemes have high overhead, we combined efficient data representations with different algorithmic optimizations to keep the computational overhead and the communication cost low. While the first approach is suitable for settings with fast network communication, the second one is suitable for settings with a weak client and a powerful server.

Our schemes for privately evaluating decision trees extend to random forests which are a generalization of decision trees. A random forest consists of many decision trees such that the classification outputs the classification label which occurs most often among the trees in the forest. This is related to the problem of computing the k^{th}-ranked element which, we considered in Chapter 7 for a distributed and privacy-preserving setting with application in benchmarking and auction. We proposed different approaches for privately computing the k^{th}-ranked element in the server model, using either garbled circuits or threshold homomorphic encryption. Using garbled circuits is very efficient, but suitable for a setting where the server is non-colluding and clients cannot fail. Threshold additively homomorphic encryption allows dealing with collusion and failure, but is less efficient. Threshold fully homomorphic encryption additionally has the best asymptotic complexity, as it allows delegating almost all the computation to the server such that the clients are only required for decryption.

8.2 Outlook

This work proposed an efficient protocol for OPE and showed how to extend it to ESEDS-OPE [125] which addresses recent plaintext guessing attacks on OPE, while not ruling out more sophisticated longitudinal attacks. As a result, the security of OPE is still not fully understood and required further research effort. It might also be interesting to build real-word applications based on ESEDS-OPE and OOPE.

The private decision protocol can be extended in several directions. If the attributes are categorical, then the tree is no longer binary and an equality test decides which branch should be followed.

A linear branching program (LBP) [14] is a decision tree in which each decision node contains, besides the threshold value, a vector that will be linearly combined with the attribute vector of the client before comparison. While a decision tree divides the space into axis-parallel regions, a linear branching program produces oblique regions resulting in smaller and more accurate trees [102, 152]. However, the evaluation of each decision node in an LBP computes an inner product followed by a greater-than comparison. This requires combining both an integer representation (for the inner product) and a binary representation (for the comparison). An idea toward the solution could be to generalize oblivious transfer by a new primitive that takes a pair of vectors (u_0, u_1) from the sender P_0, a bit b and a vector v from the receiver P_1, and outputs the inner product $\langle u_b, v\rangle$ to P_1. This primitive can then be generalized again to 1-out-of-n OT.

While a single classification reveals nothing to the client (leakage of tree parameters can be handled by adding dummy nodes to the tree), many classifications might allow the client to learn more about the model. It might be interesting to investigate differential privacy to deal with this problem.

In our scheme, both the model owner and the client must be online during the classification. In future work, one might consider outsourcing the computation to an untrusted third party while preserving the privacy of both input parties. The model

owner would outsource an encrypted version of the model to the third party which then runs a secure computation with the client. In contrast to the previous scenario, where the adversary can control either the model owner or the client (but not both), the adversary, in this case, can control either the server or a collection of clients or a collection of clients and the server. Zheng et al. [199] consider an outsourcing scenario, however, relying on two servers and evaluating every decision nodes using a comparison based on secret sharing. Another interesting direction could be to put both the model and the client input on a blockchain.

The computation of k^{th}-ranked element can be improved in different ways. Firstly, one can combine AHE with SHE by evaluating the comparison interactively using DGK or Lin-Tzeng (as in Kre-Ahe2), then encrypt the comparison bits using SHE and evaluate the selection of the k^{th}-ranked element as in Kre-She. Secondly, SCIB [28] leaks comparison bits to the parties but is secure in the malicious model. Our schemes do not leak comparison bits but are secure in the semi-honest model. Extending our schemes to support malicious adversaries can be investigated. Finally, our schemes support only one input per client and can be extended to several inputs per client. With one input per client and n clients, we have an $n \times n$ matrix of comparison bits. With m inputs per client and n clients, we will have to deal with an $n \times n$ block matrix where each block is an $m \times m$ matrix.

Bibliography

[1] Gagan Aggarwal, Nina Mishra, and Benny Pinkas. Secure computation of the k th-ranked element. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 40–55, 2004.

[2] Gagan Aggarwal, Nina Mishra, and Benny Pinkas. Secure computation of the median (and other elements of specified ranks). *J. Cryptology*, 23(3):373–401, 2010.

[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 563–574, New York, NY, USA, 2004. ACM.

[4] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Cambridge MA, March 2018.

[5] Abdelrahaman Aly, Marcel Keller, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. Scale–mamba v1.4 : Documentation. `https://homes.esat.kuleuven.be/~nsmart/SCALE/`, May 2019.

[6] Abdelrahaman Aly and Mathieu Van Vyve. Securely solving classical network flow problems. In *ICISC*, pages 205–221, 2014.

[7] Amazon web services. `https://aws.amazon.com/machine-learning`, 2019.

[8] Ghous Amjad, Seny Kamara, and Tarik Moataz. Breach-resistant structured encryption. *PoPETs*, 2019(1):245–265, 2019.

[9] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 535–548, New York, NY, USA, 2013. ACM.

[10] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 673–701. Springer, 2015.

[11] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions. *J. Cryptology*, 30(3):805–858, 2017.

[12] Mikhail J. Atallah, Marina Bykova, Jiangtao Li, Keith B. Frikken, and Mercan Topkara. Private collaborative forecasting and benchmarking. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*, pages 103–114, 2004.

[13] Mikhail J. Atallah, Hicham G. Elmongui, Vinayak Deshpande, and Leroy B. Schwarz. Secure supply-chain protocols. In *2003 IEEE International Conference on Electronic Commerce (CEC 2003), 24-27 June 2003, Newport Beach, CA, USA*, pages 293–302, 2003.

[14] Mauro Barni, Pierluigi Failla, Vladimir Kolesnikov, Riccardo Lazzeretti, Ahmad-Reza Sadeghi, and Thomas Schneider. Secure evaluation of private linear branching programs with medical applications. ESORICS'09, pages 424–439, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 479–488, New York, NY, USA, 1996. ACM.

[16] Donald Beaver. Commodity-based cryptography (extended abstract). STOC '97, pages 446–455, New York, NY, USA, 1997. ACM.

[17] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513. ACM, 1990.

[18] Amos Beimel. Secret-sharing schemes: A survey. In *Proceedings of the Third International Conference on Coding and Cryptology*, IWCC'11, pages 11–46, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. SP '13, pages 478–492, Washington, DC, USA, 2013. IEEE Computer Society.

[20] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 784–796, New York, NY, USA, 2012. ACM.

[21] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and applications. In *Proceedings on Advances in Cryptology*, CRYPTO '89, pages 547–557, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[22] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: A system for secure multi-party computation. In *CCS*, pages 257–266, New York, NY, USA, 2008. ACM.

[23] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10, New York, NY, USA, 1988. ACM.

[24] Bigml. `https://bigml.com/`, 2019.

[25] Ian F. Blake and Vladimir Kolesnikov. Strong conditional oblivious transfer and computing on intervals. In *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 2004.

[26] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIACCS*, pages 207–218, 2013.

[27] Erik-Oliver Blass and Florian Kerschbaum. Strain: A secure auction for blockchains. In *ESORICS (1)*, volume 11098 of *Lecture Notes in Computer Science*, pages 87–110. Springer, 2018.

[28] Erik-Oliver Blass and Florian Kerschbaum. Secure computation of the k^{th}-ranked integer on blockchains. *IACR Cryptology ePrint Archive*, 2019:276, 2019.

[29] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 192–206, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*, EUROCRYPT '09, pages 224–241, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO'11, pages 578–595, Berlin, Heidelberg, 2011. Springer-Verlag.

[32] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, pages 565–596, 2018.

[33] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. `https://crypto.stanford.edu/~dabo/cryptobook/`, September 2017.

[34] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

[35] Fabrice Boudot, Berry Schoenmakers, and Jacques Traoré. A fair and efficient solution to the socialist millionaires' problem. *Discrete Applied Mathematics*, 111(1-2):23–36, 2001.

[36] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *ECCC*, 18:111, 2011.

[37] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[38] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. CCS '07, pages 498–507, New York, NY, USA, 2007. ACM.

[39] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. *IACR Cryptology ePrint Archive*, 2015:472, 2015.

[40] Jason Catlett. Overpruning large decision trees. In *IJCAI*, pages 764–769, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[41] Octavian Catrina and Sebastiaan De Hoogh. Improved primitives for secure multiparty integer computation. In *Proceedings of the 7th International Conference on Security and Cryptography for Networks*, SCN'10, pages 182–199, Berlin, Heidelberg, 2010. Springer-Verlag.

[42] Octavian Catrina and Florian Kerschbaum. Fostering the uptake of secure multiparty computation in e-commerce. In *Proceedings of the The Third International Conference on Availability, Reliability and Security, ARES 2008, March 4-7, 2008, Technical University of Catalonia, Barcelona , Spain*, pages 693–700, 2008.

[43] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *STOC*, pages 11–19, New York, NY, USA, 1988. ACM.

[44] Nathan Chenette, Kevin Lewi, Stephen A. Weis, and David J. Wu. Practical order-revealing encryption with limited leakage. In *FSE '16*, pages 474–493, 2016.

[45] Jung Hee Cheon, Miran Kim, and Myungsun Kim. Search-and-compute on encrypted data. In *Financial Cryptography and Data Security - FC 2015 International Workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, pages 142–159, 2015.

[46] Jung Hee Cheon, Miran Kim, and Myungsun Kim. Optimized search-and-compute circuits and their application to query evaluation on encrypted data. *IEEE Trans. Information Forensics and Security*, 11(1):188–199, 2016.

[47] Jung Hee Cheon, Miran Kim, and Kristin E. Lauter. Homomorphic computation of edit distance. In *Financial Cryptography and Data Security - FC 2015 International Workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, pages 194–212, 2015.

[48] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, pages 3–33, 2016.

[49] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, pages 377–408, 2017.

[50] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Improving TFHE: faster packed homomorphic operations and efficient circuit bootstrapping. *IACR Cryptology ePrint Archive*, 2017:430, 2017.

[51] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *IACR Cryptology ePrint Archive*, 2018:421, 2018.

[52] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

[53] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 41–50, 1995.

[54] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In *Proceedings of the 4th International Conference on Progress in Cryptology – LATINCRYPT 2015 - Volume 9230*, pages 40–58, Berlin, Heidelberg, 2015. Springer-Verlag.

[55] Martine De Cock, Rafael Dowsley, Caleb Horst, Raj S. Katti, Anderson C. A. Nascimento, Wing-Sea Poon, and Stacey Truex. Efficient and private scoring of decision trees, support vector machines and logistic regression models based on pre-computation. *IEEE Trans. Dependable Sec. Comput.*, 16(2):217–230, 2019.

[56] Geoffroy Couteau. New protocols for secure equality test and comparison. In *ACNS*, volume 10892 of *Lecture Notes in Computer Science*, pages 303–320. Springer, 2018.

[57] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, pages 280–299, 2001.

[58] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, New York, NY, USA, 2015.

[59] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *CCS '06*, pages 79–88, 2006.

[60] Wei Dai and Berk Sunar. cuhe: A homomorphic encryption accelerator library. In *BalkanCryptSec 2015*, September 2015.

[61] Wei Dai and Berk Sunar. Cuda-accelerated fully homomorphic encryption library, August 2019. https://github.com/vernamlab/cuFHE.

[62] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 285–304, 2006.

[63] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Efficient and secure comparison for on-line auctions. In *ACISP*, pages 416–430, 2007.

[64] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In *PKC*, pages 160–179, 2009.

[65] Ivan Damgård, Martin Geisler, and Mikkel Kroigard. A correction to "efficient and secure comparison for on-line auctions". *Int. J. Appl. Cryptol.*, 1(4):323–324, August 2009.

[66] Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, PKC '01, pages 119–136, London, UK, UK, 2001. Springer-Verlag.

[67] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. ESORICS '13, pages 1–18, 2013.

[68] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[69] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. CRYPTO '12, pages 643–662, 2012.

[70] Ivan Damgård and Rune Thorbek. Efficient conversion of secret-shared values between different fields. *IACR Cryptology ePrint Archive*, 2008:221, 2008.

[71] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[72] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. CCS '17, pages 523–535, 2017.

[73] Wenliang Du and Mikhail J. Atallah. Privacy-preserving cooperative scientific computations. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, CSFW '01, pages 273–, Washington, DC, USA, 2001. IEEE Computer Society.

[74] Betül Durak, Thomas DuBuisson, and David Cash. What else is revealed by order-revealing encryption? Technical Report 786, IACR Cryptology ePrint Archive, 2016.

[75] Yael Ejgenberg, Moriya Farbstein, Meital Levy, and Yehuda Lindell. SCAPI: the secure computation application programming interface. *IACR Cryptology ePrint Archive*, 2012:629, 2012.

[76] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in Cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[77] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends in Privacy and Security*, 2(2-3):70–246, 2018.

[78] Uri Feige, Joe Killian, and Moni Naor. A minimal model for secure computation (extended abstract). In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 554–563, 1994.

[79] Marc Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, pages 457–472, 2001.

[80] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: an ANSI C compiler for secure two-party computations. In *CC '14*, pages 244–249, 2014.

[81] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1322–1333, 2015.

[82] Keith B. Frikken. Secure multiparty computation. In *Algorithms and Theory of Computation Handbook*, pages 14–14. Chapman & Hall/CRC, 2010.

[83] Juan Garay, Berry Schoenmakers, and José Villegas. Practical and secure solutions for integer comparison. In *Proceedings of the 10th International Conference on Practice and Theory in Public-key Cryptography*, PKC'07, pages 330–342, Berlin, Heidelberg, 2007. Springer-Verlag.

[84] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[85] Craig Gentry, Shai Halevi, Charanjit S. Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. In *ACNS*, volume 9092 of *Lecture Notes in Computer Science*, pages 172–191. Springer, 2015.

[86] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 182–194, New York, NY, USA, 1987. ACM.

[87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, pages 218–229, New York, NY, USA, 1987. ACM.

[88] Oded Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, pages 86–97, 1998.

[89] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, New York, NY, USA, 2001.

[90] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[91] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[92] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[93] Google cloud ml engine. `https://cloud.google.com/ml-engine/`, 2019.

[94] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 513–524, 2012.

[95] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure computation with sublinear amortized work. *IACR Cryptology ePrint Archive*, 2011:482, 2011.

[96] Thore Graepel, Kristin Lauter, and Michael Naehrig. Ml confidential: Machine learning on encrypted data. In *Proceedings of the 15th International Conference on Information Security and Cryptology*, ICISC'12, pages 1–21, Berlin, Heidelberg, 2013. Springer-Verlag.

[97] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[98] P. Grubbs, K. Sekniqi, V. Bindschaedler, M. Naveed, and T. Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 655–672, May 2017.

[99] Florian Hahn and Florian Kerschbaum. Searchable encryption with secure and efficient updates. In *CCS '14*, pages 310–320, 2014.

[100] Shai Halevi and Victor Shoup. Algorithms in helib. In *CRYPTO (1)*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014.

[101] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[102] David Heath, Simon Kasif, and Steven Salzberg. Induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2(2):1–32, 1993.

[103] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS*, pages 451–462, 2010.

[104] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Catherine Jones. Privacy-preserving machine learning in cloud. In *Proceedings of the 2017 on Cloud Computing Security Workshop*, CCSW '17, pages 39–43, New York, NY, USA, 2017. ACM.

[105] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[106] R. Impagliazzo and S. Rudich. Limits on the provable consequences of one-way permutations. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 44–61, New York, NY, USA, 1989. ACM.

[107] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 145–161, 2003.

[108] Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *CT-RSA*, volume 9610 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 2016.

[109] Thomas P. Jakobsen, Jesper Buus Nielsen, and Claudio Orlandi. A framework for outsourcing of secure computation. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, CCSW '14, pages 81–92, New York, NY, USA, 2014. ACM.

[110] Markus Jakobsson and Moti Yung. Proving without knowing: On oblivious, agnostic and blindolded provers. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, pages 186–200, 1996.

[111] Ayman Jarrous and Benny Pinkas. Secure hamming distance based computation and its applications. In *ACNS*, pages 107–124, 2009.

[112] Marc Joye and Fariborz Salehi. Private yet efficient decision tree evaluation. In *DBSec*, volume 10980 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2018.

[113] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multi-party computation. *IACR Cryptology ePrint Archive*, 2011:272, 2011.

[114] Seny Kamara, Payman Mohassel, and Ben Riva. Salus: A system for server-aided secure function evaluation. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 797–808, 2012.

[115] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014.

[116] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, pages 724–741, 2015.

[117] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. CCS '16, pages 830–842, 2016.

[118] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. ASIACRYPT '14, pages 506–525, 2014.

[119] Florian Kerschbaum. Building a privacy-preserving benchmarking enterprise system. *Enterprise IS*, 2(4):421–441, 2008.

[120] Florian Kerschbaum. Adapting privacy-preserving computation to the service provider model. In *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering, CSE 2009, Vancouver, BC, Canada, August 29-31, 2009*, pages 34–41, 2009.

[121] Florian Kerschbaum. *A privacy-preserving benchmarking platform.* PhD thesis, Karlsruhe Institute of Technology, 2010.

[122] Florian Kerschbaum. Privacy-preserving computation - (position paper). In *Privacy Technologies and Policy - First Annual Privacy Forum, APF 2012, Limassol, Cyprus, October 10-11, 2012, Revised Selected Papers*, pages 41–54, 2012.

[123] Florian Kerschbaum. Frequency-hiding order-preserving encryption. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 656–667, New York, NY, USA, 2015. ACM.

[124] Florian Kerschbaum and Axel Schröpfer. Optimal average-complexity ideal-security order-preserving encryption. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 275–286, 2014.

[125] Florian Kerschbaum and Anselme Tueno. An efficiently searchable encrypted data structure for range queries. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, pages 344–364, 2019.

[126] Ágnes Kiss, Masoud Naderpour, Jian Liu, N. Asokan, and Thomas Schneider. Sok: Modular and efficient private decision tree evaluation. *PoPETs*, 2019(2):187–208, 2019.

[127] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[128] Neal Koblitz, Alfred Menezes, and Scott A. Vanstone. The state of elliptic curve cryptography. *Des. Codes Cryptography*, 19(2/3):173–193, 2000.

[129] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In *CRYPTO (2)*, volume 8043 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2013.

[130] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology and Network Security, 8th International Conference, CANS 2009, Kanazawa, Japan, December 12-14, 2009. Proceedings*, pages 1–20, 2009.

[131] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, pages 486–498, 2008.

[132] Ben Kreuter, Benjamin Mood, Abhi Shelat, and Kevin Butler. Pcf: A portable circuit format for scalable two-party secure computation. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 321–336, Berkeley, CA, USA, 2013. USENIX Association.

[133] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 364–373, 1997.

[134] Kevin Lewi and David J. Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *CCS '16*, pages 1167–1178, 2016.

[135] Hsiao-Ying Lin and Wen-Guey Tzeng. An efficient solution to the millionaires' problem based on homomorphic encryption. In *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings*, pages 456–466, 2005.

[136] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In *Advances in cryptology-CRYPTO 2000*, volume 1880 of *Lecture notes in computer science*, pages 36–54, Berlin and New York, 2000. Springer.

[137] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. *Journal of Cryptology*, 15(3):177–206, 2002.

[138] Yehuda Lindell and Benny Pinkas. Secure multiparty computation for privacy-preserving data mining. *IACR Cryptology ePrint Archive*, 2008:197, 2008.

[139] Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptol.*, 22(2):161–188, April 2009.

[140] Yehuda Lindell and Benny Pinkas. Secure multiparty computation for privacy-preserving data mining. *The Journal of Privacy and Confidentiality*, 2009(1):59–98, 2009.

[141] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *Information Security, 8th International Conference, ISC 2005, Singapore, September 20-23, 2005, Proceedings*, pages 314–328. Springer, 2005.

[142] Helger Lipmaa and Tomas Toft. Secure equality and greater-than tests with sublinear online complexity. In *Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part II*, ICALP'13, pages 645–656, Berlin, Heidelberg, 2013. Springer-Verlag.

[143] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating efficient ram-model secure computation. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 623–638, Washington, DC, USA, 2014. IEEE Computer Society.

[144] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 359–376, Washington, DC, USA, 2015. IEEE Computer Society.

[145] Wen-jie Lu, Jun-jie Zhou, and Jun Sakuma. Non-interactive and output expressive private comparison from homomorphic encryption. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 67–74, New York, NY, USA, 2018. ACM.

[146] Lior Malka. Vmcrypt: Modular software architecture for scalable secure computation. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 715–724, New York, NY, USA, 2011. ACM.

[147] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay—a secure two-party computation system. In *SSYM*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[148] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. Modular order-preserving encryption, revisited. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 763–777, 2015.

[149] mcl library, September 2019. https://github.com/herumi/mcl/.

[150] Microsoft azure. `https://azure.microsoft.com/de-de/services/machine-learning/`, 2019.

[151] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38, 2017.

[152] Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A system for induction of oblique decision trees. *J. Artif. Int. Res.*, 2(1):1–32, August 1994.

[153] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 448–457, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[154] Moni Naor and Benny Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 18:1–35, Jan 2005.

[155] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, pages 129–139, New York, NY, USA, 1999. ACM.

[156] Muhammad Naveed, Seny Kamara, and Charles V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 644–655, New York, NY, USA, 2015. ACM.

[157] Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In *TCC*, pages 368–386, 2009.

[158] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Proceedings of the 10th International Conference on Practice and Theory in Public-key Cryptography*, PKC'07, pages 343–360, Berlin, Heidelberg, 2007. Springer-Verlag.

[159] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, pages 223–238, Berlin, Heidelberg, 1999. Springer-Verlag.

[160] Arpita Patra, Pratik Sarkar, and Ajith Suresh. Fast actively secure OT extension for short secrets. In *NDSS*. The Internet Society, 2017.

[161] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. *IACR Cryptology ePrint Archive*, 2009:314, 2009.

[162] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A strongly encrypted database system. *IACR Cryptology ePrint Archive*, 2016, 2016.

[163] Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 463–477, Washington, DC, USA, 2013. IEEE Computer Society.

[164] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[165] Predictionio. `http://predictionio.incubator.apache.org/index.html`, 2019.

[166] Pille Pullonen, Dan Bogdanov, and Thomas Schneider. The design and implementation of a two-party protocol suite for sharemind 3. Technical report, CYBERNETICA Institute of Information Security, 2012.

[167] Michael O. Rabin. How to exchange secrets with oblivious transfer. Technical report, Technical Report TR-81, Aiken Computation Lab, Harvard University, 1981.

[168] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 655–670, Washington, DC, USA, 2014. IEEE Computer Society.

[169] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 707–721, New York, NY, USA, 2018. ACM.

[170] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[171] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In *Proceedings of the 17th International Conference on The Theory and Application of Cryptology and Information Security*, ASIACRYPT'11, pages 197–214, Berlin, Heidelberg, 2011. Springer-Verlag.

[172] Radu Sion and Bogdan Carbunar. On the practicality of private information retrieval. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'07. The Internet Society, 2007.

[173] N. P. Smart and F. Vercauteren. Fully homomorphic simd operations. *Des. Codes Cryptography*, 71(1):57–81, April 2014.

[174] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 411–428, Los Alamitos, CA, USA, may 2015. IEEE Computer Society.

[175] Raymond K. H. Tai, Jack P. K. Ma, Yongjun Zhao, and Sherman S. M. Chow. Privacy-preserving decision trees evaluation via linear functions. ESORICS '17, pages 494–512, 2017.

[176] Fabian Taigel, Anselme K. Tueno, and Richard Pibernik. Privacy-preserving condition-based forecasting using machine learning. *Journal of Business Economics*, Jan 2018.

[177] Isamu Teranishi, Moti Yung, and Tal Malkin. Order-preserving encryption secure beyond one-wayness. In *Proceedings of the 20th International Conference on Advances in Cryptology*, ASIACRYPT, 2014.

[178] Tomas Toft. Sub-linear, secure comparison with two non-colluding parties. In *Public Key Cryptography*, volume 6571 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2011.

[179] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 601–618, 2016.

[180] Anselme Tueno, Yordan Boev, and Florian Kerschbaum. Non-interactive private decision tree evaluation. In *Data and Applications Security and Privacy XXXIV - 34th Annual IFIP WG 11.3 Conference, DBSec 2020, Regensburg, Germany, June 25-26, 2020, Proceedings*, pages 174–194, 2020.

[181] Anselme Tueno and Florian Kerschbaum. Efficient secure computation of order-preserving encryption. In *Proceedings of the 2020 on Asia Conference on Computer and Communications Security*, ASIACCS '20, 2020.

[182] Anselme Tueno, Florian Kerschbaum, Daniel Bernau, and Sara Foresti. Selective access for supply chain management in the cloud. In *2017 IEEE Conference on Communications and Network Security, CNS 2017, Las Vegas, NV, USA, October 9-11, 2017*, pages 476–482, 2017.

[183] Anselme Tueno, Florian Kerschbaum, and Stefan Katzenbeisser. Private evaluation of decision trees using sublinear cost. *PoPETs*, 2019(1):266–286, 2019.

[184] Anselme Tueno, Florian Kerschbaum, Stefan Katzenbeisser, Yordan Boev, and Mubashir Qureshi. Secure computation of the kth-ranked element in a star network. In *Financial Cryptography and Data Security (FC)*, 2020.

[185] Uci repository. `http://archive.ics.uci.edu/ml/index.php`, 2019.

[186] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proceedings of the 29th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'10, pages 24–43, Berlin, Heidelberg, 2010. Springer-Verlag.

[187] Thijs Veugen. Improving the DGK comparison protocol. In *2012 IEEE International Workshop on Information Forensics and Security, WIFS 2012, Costa Adeje, Tenerife, Spain, December 2-5, 2012*, pages 49–54, 2012.

[188] Thijs Veugen. Correction to "improving the DGK comparison protocol". *IACR Cryptology ePrint Archive*, 2018:1100, 2018.

[189] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 850–861, 2015.

[190] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, Abhi Shelat, and Elaine Shi. Scoram: Oblivious ram for secure computation. CCS '14, pages 191–202, 2014.

[191] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. CCS '14, pages 215–226, New York, NY, USA, 2014. ACM.

[192] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[193] David J. Wu, Tony Feng, Michael Naehrig, and Kristin Lauter. Privately evaluating decision trees and random forests. *PoPETs*, 2016(4):335–355, 2016.

[194] Xi Wu, Matthew Fredrikson, Somesh Jha, and Jeffrey F. Naughton. A methodology for formalizing model-inversion attacks. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 355–370, 2016.

[195] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[196] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.

[197] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. EUROCRYPT'15, pages 220–250, 2015.

[198] Yihua Zhang, Aaron Steele, and Marina Blanton. Picco: A general-purpose compiler for private distributed computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 813–826, New York, NY, USA, 2013. ACM.

[199] Yifeng Zheng, Huayi Duan, and Cong Wang. Towards secure and efficient outsourcing of machine learning classification. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, pages 22–40, 2019.