Privacy-preserving edit distance computation using secret-sharing protocols

Universidad Nacional de Colombia
Faculty of Sciences
Medellín, Colombia
2023

Privacy-preserving edit distance computation using secret-sharing protocols

Hernán Darío Vanegas Madrigal

Thesis presented as a requirement to obtain the title of:
MSc. in Applied Mathematics

Supervisor:
PhD. Daniel Cabarcas Jaramillo

Line of research:
Cryptography

Universidad Nacional de Colombia
Faculty of Sciences
Medellín, Colombia
2023

Cómputo de la distancia de edición preservando la privacidad mediante el uso de protocolos basados en secreto compartido

Hernán Darío Vanegas Madrigal

Tesis presentada como requisito para obtener el título de:
Maestría en Ciencias - Matemática Aplicada

Director:
PhD. Daniel Cabarcas Jaramillo

Línea de Investigación:
Criptografía

Universidad Nacional de Colombia
Facultad de Ciencias
Medellín, Colombia
2023

We can only see a short distance ahead, but we can see plenty there that needs to be done.

– *Alan Turing.*

Acknowledgements

First of all, I want to say thank you to my supervisor Prof. Daniel Cabarcas Jaramillo. He is a truly good professor and supervisor. His continuous support was the key to finishing this work successfully. Due to his patience and his good attitude from the first ideas to the last moments, this experience has been amazing, very rewarding, and completely enjoyable. I owe a lot of things to Prof. Cabarcas: my interest in cryptography, the improvement in my writing style, my research skills, and the way to see problems. All the moments I needed his help, he was there, ready to answer my questions, emails, and messages. The most amazing thing is that he is both academically excellent and more important, an amazing person with great values and exceptional academic ethic. I only hope to give back all of the support, teachings, and friendship in the future. Also, I want to say thank you to Prof. Diego Aranha from Aarhus University, Denmark, for all of his support and comments on this work. Despite we did not have significant contact before, he accepted to help us with this work and give us comments in his free time. Also, I want to say thank you to Daniel Escudero, because I consider him a very valuable person in my academic development. He was the one that introduced me to the marvelous world of MPC since my undergraduate degree. Also, he supported me by giving me material, books, and papers to read. The knowledge I have at this moment, and maybe the path I will follow in the future is thanks to him. Another important person for this project was Marcel Keller, the maintainer of the MP-SPDZ framework. He is also a key part of this work because all the times I had a silly question about MP-SPDZ, I posted it in the issues of the repository, and he always gives a correct, clear, and useful response back. Also, if I required some non-existent functionality or report a bug, he was always willing to answer back, solve the problem and improve the framework. Without his efforts to keep improving the library, this work would be way more difficult.

I want to say thank you to Verónica Valencia Hernández. She is an important person to me and she was very important during the development of this work. She always is there to give me advice to improve my work, to give me strength when something went wrong, and for being my resting space in my life. Thank you to her for her love and her company. Thank you to my family for their continuous support during all my studies. They were always there to help me in all the moments. Also, I want to say thank you to my friends from the School of Mathematics: Andrés Felipe Uribe, Juan Pablo Cardona, Santiago Echavarría, Valentina Guarín, Estiven Carvajal, and Maria Antonia

Rincón. Their friendship was always very valuable and the time I spent with them was the best experience that a person can get in life. I will be there for them as they are for me. I want to say thank you to Liliana Parra from the administrative staff in the School of Mathematics. She always is there to solve all kind of problems to all the students with commitment and dedicated effort.

Also, I say thank you to David Basin and Martín Ocha Ronderos for giving me the opportunity to go to ETH Zürich, Switzerland, to do an internship and work with them. It was one of the most amazing experiences in my life. Thank you for trusting in me and giving me the opportunity to know a lot of new places, new academic environments, and new researchers. I also want to say thank you to José David Mosquera for his friendship and help during this trip. He helped me a lot with advice about Zürich and we enjoyed together good moments in this amazing country. Also, I want to say thanks to Saskia Wolf and Vivien Klomp from the administrative staff of the Information Security Group for all of the help with the formalities of this internship. They were always willing to help me to have a good experience during my stay at ETH Zürich and their good energy made this time very enjoyable. Finally, thank you to the folks of the Information Security Group, to my office mate and friend Andrin Bertschi, the Stäheli family, Andras Wagner, and Noemi Conod. They were always an amazing group of people that helped me and made me feel welcome in Zürich and at ETH Zürich. Thank you for giving me one of the most amazing times of my life, and thank you for always treating me well and making me happy.

Abstract

The edit distance between two strings in an alphabet is the minimum number of insertions, deletions, and replacements that need to be done to transform one of the strings into the other. This metric is widely used in genomic applications to determine the similarity of two DNA chains which has its uses in medical and biological studies. Despite the benefits of computing the edit distance between DNA chains, there are privacy risks like re-identification, where an adversary having a DNA chain can extract private information about its owner. To attend to such privacy concerns, we propose a two-party MPC protocol using mixed-circuit computations through secret-sharing schemes like Tinier and SPD$\mathbb{Z}_{2^k}$ to compute the edit distance while preserving the privacy of the DNA chains used as inputs. Also, we use daBits to perform domain conversion and edaBits to perform arithmetic comparisons. Our work focuses on protocols whose underlying computational domains are rings of the form $\mathbb{Z}_{2^k}$. We implement our proposal in the MP-SPDZ framework, and through experimental evaluation simulating a local area network, we show that our proposal reaches a reduction in the execution time of approximately a 64% for active security and 78% for passive security with respect to a traditional implementation of the Wagner-Fischer algorithm. In the experiments, we show that our protocol has a reduction in the data sent of approximately 57-99% compared to a garbled circuit implementation and a reduction of the execution time of approximately 40% with respect to approaches using homomorphic encryption found in previous works.

Keywords: secure multi-party computation, edit distance, secret-sharing.

Título en español

Cómputo de la distancia de edición preservando la privacidad mediante el uso de protocolos basados en secreto compartido.

Resumen

La distancia de edición entre dos cadenas en un alfabeto es el mínimo número de inserciones, borrados y reemplazamientos que se necesitan para transformar una de las cadenas en la otra. Esta métrica es ampliamente utilizada en aplicaciones de la genómica para determinar la similitud de dos cadenas de ADN, lo cual tiene sus usos en estudios médicos y biológicos. A pesar de los beneficios de computar la distancia de edición entre dos cadenas de ADN, existen riesgos a la privacidad como la reidentificación, donde un adversario que posee una cadena de ADN puede extraer información privada de su propietario. Para atender estos riesgos a la privacidad, hemos propuesto un protocolo para dos participantes usando circuitos mixtos mediante esquemas de secreto compartido como Tinier y SPD$\mathbb{Z}_{2^k}$ para computar la distancia de edición preservando la privacidad de las cadenas usadas en el cómputo. Además, usamos daBits para realizar conversiones entre dominios, y edaBits para computar comparaciones aritméticas. Nuestro trabajo se enfoca en protocolos cuyo dominio computacional subyacente son anillos de la forma $\mathbb{Z}_{2^k}$. En este trabajo implementamos nuestra propuesta en el framework MP-SDPZ, y mediante una evaluación experimental simulando una red de área local, hemos encontrado que nuestra propuesta alcanza una reducción en el tiempo de ejecución de aproximadamente un 64% en el caso de seguridad activa, y un 78% en el caso de seguridad pasiva con respecto a una implementación tradicional del algoritmo Wagner-Fischer. En los experimentos mostramos que nuestro protocolo tiene una reducción de datos enviados a la red entre un 57-99% aproximadamente en comparación a una implementación usando *garbled circuits*, y una reducción de 40% aproximadamente con respecto a implementaciones que usan encripción homomórfica encontradas en trabajos anteriores.

Palabras clave: computación segura de múltiples participantes, distancia de edición, secreto compartido.

List of Figures

List of Tables

Contents

1 Introduction

Given an alphabet of symbols Σ, the edit distance between two strings in Σ^* is the minimum cost of a sequence of editing operations (insertions, deletions, and changes) to transform one string into the other [Ukk85]. Intuitively, the smaller the edit distance between two strings, the more similar they are. Algorithms to compute the edit distance have been studied for many years and the most recognized solutions are based on dynamic programming, such as the Wagner-Fischer algorithm [WF74]. Such algorithms are useful tools in fields like genomics, where the similarity between two genomic sequences is used in activities like disease diagnosis and treatment [Zhe+19]. Nowadays, the scientific community uses more complex algorithms to measure string similarity such as the Smith-Waterman algorithm [SW81] used in BLAST[1], a software widely used for querying databases of proteins and DNA sequences [BWY21]. Although the current approaches to solving the problem of string similarity are more sophisticated, the original proposal of the edit distance problem and its solutions laid the foundation for such modern approaches.

Despite the benefits of computing similarities between genomic data, there are risks that come from revealing such kind of information. One of the main risks is called *re-identification*, where a subject can be identified from its genomic data. As an example, consider a person who shares his genomic data for some medical study, and such data is leaked to the public. From such data, someone can spot genomic information that can express the probability of suffering from a certain disease. This knowledge along with a process of re-identification may allow some organizations to know the identity of the owner of the genomic data. Then, organizations may make unfair decisions considering the spotted information. For example, they may reject the owner of the genomic information in an employment or health insurance process [Oes+21]. There are other concerns like ancestry identification where an individual can identify his ancestors having their genomic information. Also, there are the so-called *attribute disclosure attacks via DNA*, where an attacker can spot a sensitive attribute about some person having a DNA sample and access to a database of samples related to such sensitive attribute [EN14].

In the context of our work, the previous concerns motivate the use of computational techniques to compute algorithms on genomic data that preserves the privacy of the data

[1] https://blast.ncbi.nlm.nih.gov/Blast.cgi

owners. Specifically, we will deal with the problem of how to compute the edit distance of two chains using the Wagner-Fischer algorithm without revealing the chains used as inputs. The scenario can be described as follows: suppose that Alice and Bob are connected via a secure communication channel; each one has a DNA chain represented as a list of nucleotides and they want to use the Wagner-Fischer algorithm to compute the edit distance of both strings, but Alice does not want to reveal his string to Bob and vice versa. In this work, we will come up with a technique to accomplish this task.

The Wagner-Fischer algorithm is a dynamic programming solution to find the edit distance between two chains $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$. The core of the algorithm is to compute a matrix D in which the following recursive equation holds:

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1, \\ D(i,j-1) + 1, \\ D(i-1,j-1) + t(i,j) \end{cases},$$

where

$$t(i,j) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } a_i \neq b_i \\ 0 & \text{otherwise} \end{cases},$$

$1 \leq i \leq n$ and $1 \leq j \leq m$. In this algorithm, two operations have high relevance when considering a secure solution. First, the minimum computation requires the secure comparison between integer numbers which needs the extraction of the most significant bit of the integer representation. Second, the computation of t requires the secure equality test between a pair of symbols in the chains.

As we will see in Section 1.1, the problem mentioned above can be solved using a variety of cryptographic techniques. Particularly, we will focus on a technique called *secure multi-party computation* (MPC). In an MPC protocol, a set of parties, each one holding part of the input of a function, want to compute such function correctly while preserving the privacy of the inputs. In such a protocol, the parties exchange messages and perform local computations. In the end, the parties may obtain the correct result, and the messages exchanged between them do not reveal any information about the inputs they provide. In our case, the parties are Alice and Bob, and the inputs are the chains that they are holding; also, the function will be the edit-distance function which is computed using the Wagner-Fischer algorithm.

Several previous works use Yao's garbled circuits which is a particular MPC protocol that has good performance for computing bit-wise operations securely. In our work, we will focus on another type of MPC protocol called *secret-sharing schemes* which are efficient for computing arithmetic operations securely.

In this work, we will answer two research questions:

1. Can we design an efficient protocol to compute edit distance using secret-sharing schemes efficiently enough to compete with the current state-of-the-art solutions?

2. Are protocols in $\mathbb{Z}_{2^k}$ the right choice to compute the edit distance securely?

1.1 Related work

Nowadays, there is an active field of research dealing with the computation of edit distance securely. The most widely used techniques to compute the edit distance are homomorphic encryption (HE) and garbled circuits (GC). To our knowledge, the specific study of secret-sharing schemes is not commonly used. However, we found some works in that direction. We will revisit some results related to the secure computation of edit distance.

In the case of homomorphic encryption schemes (HE), the work of [Zhe+19] proposes an architecture where a data owner holding a database of DNA sequences and an authorized user want to compute the edit distance between two DNA sequences. The user owns a sequence Q and wants to compute the edit distance between Q and a sequence S (unknown to the user) from the database of the data owner. So, the user sends his sequence Q to the cloud server and the data owner sends an encryption of the particular sequence S which is requested by the user. The cloud is considered powerful in both computational capabilities and storage, so it will be in charge of computing the edit distance between Q and the encryption of S using a modified version of the Paillier cryptosystem [BCP03].

On the other hand, works like [DZ16; RS10; Oha20] show how to compute the edit distance using dynamic programming algorithms, as we do in the present work. In particular, the work of [RS10] addresses the problem of computing a minimum between three elements using homomorphic encryption and uses this minimum computation repeatedly to find the edit distance. Finally, the work of [CKL15] is of high relevance in the present work. We build our work upon the ideas of Cheon et al. and extend their work further. In such work, Cheon et al. propose a technique to reduce the depth of the circuits to compute them securely using HE by expressing the edit distance in terms of the computation of the minimum of a list of numbers. This way of expressing the edit distance solution is actively used in this work. However, the work of Cheon et al. does not prove the correctness and optimality of their method to generate such a list of numbers, and they focus on strings with the same length. In this work, we extend their strategy to solve both problems. We use their idea to express the edit distance in terms of the minimum of a list of numbers but focusing also on finding the edit distance between DNA chains that do not necessarily

have the same length. Also, we present theorems that prove that our generalized technique to generate the mentioned list of numbers is both correct and optimal.

Another technique that is actively used to compute the edit distance securely is garbled circuits. The survey of [DZ16] shows a wide application of Yao's GC to implement standard dynamic programming approaches to compute the edit distance. One of the earliest works in this field appears in [JKS08] which uses one circuit for each basic operation in the algorithm: add by one, computing the minimum, and equality test. In [DZ16; Zha+19], they show further improvements to the approach of Jha et al. in both theoretical and practical aspects like improving the amount of secured data, the memory usage, the communication complexity, and the use of specialized hardware to compute the garbling and de-garbling processes in parallel. An important work appears in [ZH22], which uses GC to compute the edit distance in both active and passive threat models. The work of Zhu & Huang claims to outperform the best existing GC-based protocols. As in our work, they consider the secure computation of the edit distance using the Wagner-Fischer algorithm and take advantage of the structure of the minimization problem to find bounds that allow them to improve the performance of the protocol. Unfortunately, they do not report experimental measures of performance in the actively secure setting. As an additional note, some works like [AAM17] consider not the precise computation of edit distance but its approximation; they take advantage of this fact to improve the performance of the computations.

Compared to HE and GC, the secret-sharing techniques have less work devoted to computing the edit distance securely. The work of Rane & Sun [RS10] uses additive secret-sharing alongside homomorphic encryption, but they do not rely on secret-sharing schemes to perform the operation but on homomorphic encryption schemes. The work that is closest to our techniques appears in [ST19]. They use the ideas from [Ash+18] to compute an approximation of the edit distance using the ABY framework [DSZ15]. Such framework allows designing protocols using mixed circuit computation against passive adversaries, which means that by using ABY they can compute some sections of their protocol in binary domains or arithmetic domains, and move the secrets from one domain to another. There are significant differences between the work of Schneider et al. and our work. First, they limited their work to be secure against passive adversaries, contrary to our work where we explore both passive and active adversaries. Second, they use the techniques from Asharov et al. to improve the efficiency of the computation by approximating the edit distance; instead, we focus on the computation of the exact edit distance. This approximation of the edit distance includes the computation of a lookup table that is calculated by aligning the sequences to a publicly known reference genome. Given the optimizations proposed by Asharov et al., Schneider et al. report in their experiments that for a database with 1,000 sequences with a length of 3,470, they can extract the 5 most similar sequences to a query DNA chain in 1.89 seconds using a LAN.

1.2 Contributions

We apply recently developed secret-sharing schemes such as SPD$\mathbb{Z}_{2^k}$ and Tinier, as well as protocols such as daBits and edaBits to compute the edit distance securely. To the best of our knowledge, we are the first to propose a solution to the secure edit distance computation using the techniques mentioned above. For the computation of the edit distance, we divide the Wagner-Fischer algorithm into two sections: the preamble which is in charge of computing t, and the arithmetic section where the matrix D is computed. Given that division, we optimize each section separately.

For the computation of the matrix t, we encode the nucleotides using a binary representation, and through bit-wise operations, we propose a protocol to compute the equality test between a pair of nucleotides using Tinier. Once we compute t, we obtain binary shares of each possible value of the function, and using daBits, we transform such binary shares into arithmetic shares to be used in the arithmetic section.

For the arithmetic part, we take the ideas presented in [CKL15] and generalize them in two directions. First, as in Cheon et al., we expand the recursion equations from the Wagner-Fischer algorithm to compute the matrix D not as the minimum of three numbers, but as the minimum of a longer list of numbers. This allows us to divide the matrix D into sub-boxes and computing them reduces the number of rounds to compute the matrix. However, this strategy also increases the number of multiplications and comparisons in the protocol raising a trade-off in the execution time and the data sent during the protocol execution. This trade-off is studied both theoretically and empirically. Also, Cheon et al. consider a sub-box that matches the size of the matrix D and focuses only on DNA chains that have the same length. Instead, we generalize their work considering a sub-box of arbitrary size which works for DNA chains with different lengths.

As a part of this generalization using sub-boxes of arbitrary length, we propose an algorithm to automatically generate the equations to compute each sub-box. This algorithm arises from representing the recursive equations of the Wagner-Fischer algorithm as a graph. Then, with such representation, we prove both the correctness and the optimality of the equation generation. We need to point out that Cheon et al. use a different graphical method to compute their own equations but they do not prove its correctness and optimality.

We also perform experimental evaluations of our method using the MP-SPDZ framework. Through the experiments, we show that our algorithm has a significant reduction in the execution time using a LAN with respect to a naive implementation of the Wagner-Fischer algorithm. Also, we successfully explain the trade-off in the performance of the protocol

as the size of the size of the sub-box increases. Additionally, we find that our protocol is competitive with the techniques currently used to solve the edit distance problem like Yao's garbled circuits, and outperforms HE and even techniques like BMR, which is an actively secure protocol based on garbled circuits. Moreover, we empirically prove that protocols in $\mathbb{Z}_{2^k}$ are the best suited for our implementation and we give arguments that support this statement. All the source code with the implementations for this work can be found on https://github.com/hdvanegasm/sec-edit-distance.

1.3 Organization of the document

This document is organized as follows. In Chapter 2, we present all the concepts needed to understand the proposed solution. First, we will present a formal definition of the edit distance problem and the Wagner-Fischer algorithm. Second, we show what is secure multiparty computations and the main terminology to classify the protocols according to their security and correctness of the output; also, we show some metrics that will allow us to compare protocols to select the one that is best suited to our requirements. Finally, we present the protocols that we use to compute the edit distance algorithm securely. In Chapter 3, we show a solution to the edit distance computation based on secret-sharing schemes with its respective complexity analysis. In Chapter 4, we show an algorithm based on graph theory techniques to obtain the minimal number of terms to be parameters of the minimum function to compute the edit distance algorithm correctly. Finally, in Chapter 5, we present some experiments to evaluate the performance of our solution and to compare it with the cryptographic techniques used in the current state-of-the-art.

2 Preliminaries

In this chapter, we will introduce all the concepts needed to understand both the problem we are dealing with and the solution that we will propose in the later chapters. The preliminaries are divided into two main parts. In the first part, we present the edit distance problem. The presentation is done in a general way, although we will direct our attention to a particular case of the general version of the problem. The second part will cover all the cryptographic techniques that we will use to compute the edit distance securely, namely, the techniques based on secure multi-party computation. The presentation of the cryptography topics is divided into three sections. In the first section, we cover a technique called *garbled circuits* (GC). Although this technique is not the main focus of our study, in the bibliography review we found that GC is a widely used technique in the secure computation of edit distance. So, our purpose is to present the basic concepts here to later show an experimental comparison between our solution and the solution using GC. In the second section, we will present the techniques based on *secret-sharing* that we will use to solve our problem at hand. Finally, in the third section, we will show a couple of techniques called *daBits* and *edaBits*, which will help us to optimize the proposed solution.

In some specific points of this chapter, we fill some details or give a deeper intuition about some of the explained concepts that, in our opinion, are not very clear from the original papers or bibliographic sources. We hope that this will help the reader to easily understand the ideas behind such techniques. In other points, we prefer to refer the reader to the original source to avoid having an extensive presentation of the topics.

Notation. In what follows in the rest of this work, we will adopt the following conventions. We define the set $\mathbb{J}_n \subseteq \mathbb{N}$ to be the set $\{1, 2, \ldots, n\}$, for some $n \in \mathbb{N}$. Also, let $a, b \in \mathbb{Z}$; we denote by $c = a \mod b$ to say that c is the residue resulting from the division of a over b, and we denote $c \equiv a \mod b$ to say that c is congruent with a modulo b. We denote $\mathbb{Z}_b$ to be the ring of integers modulo b.

2.1 The edit distance problem

As we mentioned before, the edit distance problem is the main point of attention in this work. Let us formulate the problem precisely according to the definition presented in [Ukk85]. Let Σ be an alphabet of symbols and let $A \stackrel{\text{def}}{=} (a_1, a_2, \ldots, a_m)$ be a string over the alphabet Σ. We define the possible *editing operations* on A as follows:

1. We can *delete* any position of the string A, namely the i-th position, to obtain the string $(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_m)$.

2. We can *insert* a symbol $b \in \Sigma$ in between the positions i-th and $i+1$-th of the string A to obtain the string $(a_1, \ldots, a_i, b, a_{i+1}, \ldots, a_m)$.

3. We can *change* the symbol in the position i-th of the string A to a new symbol $b \in \Sigma$ to obtain the string $(a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_m)$.

We denote the empty string as $\oslash$. An edit operation can be represented as a pair $(a, b) \in (\Sigma^* \times \Sigma^*) \setminus \{(\oslash, \oslash)\}$ where each component of the tuple is a string of length at most 1, and is denoted as $a \to b$. We say that a string B is obtained from the application of the operation $a \to b$ over the string A, if $A = \sigma a \tau$ and $B = \sigma b \tau$, for some $\sigma, \tau \in \Sigma^*$. We can assign a non-negative real number as a cost for each one of the operations which is determined by a cost function $\gamma : (\Sigma^* \times \Sigma^*) \setminus \{(\oslash, \oslash)\} \to \mathbb{R}^+ \cup \{0\}$. If $S = (s_1, \ldots, s_r)$ is a sequence of edit operations over a string A, we denote by $\gamma(S) \stackrel{\text{def}}{=} \sum_{i=1}^{r} \gamma(s_i)$ the overall cost of applying such sequence of changes. We will assume two properties over γ:

1. $\gamma(a \to a) = 0$.

2. If $a \to b$ and $b \to c$ are editing operations, it holds that $\gamma(a \to c) \leq \gamma(a \to b) + \gamma(b \to c)$.

These two properties allow us to argue that the cost of one change has the minimum cost; for example, a change $a \to a$ could be done as $a \to b$, then $b \to c$, and finally $c \to a$. With these properties, we are saying that the cost of redundant steps in an editing operation are ignored. Although we will present a general treatment of the edit distance problem in this section, our work will be limited to the case where each editing operation has a cost of 1 if there is an effective change on the string; this means that, in our case, deleting or inserting a symbol will have a cost of 1, and the change of a symbol will account as 1 if the change is $a \to b$, where $a \neq b$, and both a and b are different from the empty string.

Notice that by applying these operations one by one, we can transform any string $A \in \Sigma^*$ into any other string $B \in \Sigma^*$. Naively, we can delete all the elements of the string A and

then, we can add the symbols corresponding to the string B. Formally, given $A, B \in \Sigma^*$, the *edit distance problem* consists in finding the sequence of editing operations to be applied on A to convert it into the string B so that the sum of the costs of each operation in the sequence is minimized. In this work, we are not interested in outputting the sequence of operations but only in the overall minimum cost. Such minimum cost is called the *edit distance* of A and B.

To solve this problem, Wagner and Fischer propose an algorithm based on dynamic programming [WF74], and that is the approach we will consider in this work. Let us establish some notation. For the rest of the discussion, let us fix an alphabet Σ and let $A \stackrel{\text{def}}{=} (a_1, a_2, \ldots, a_m)$ and $B \stackrel{\text{def}}{=} (b_1, b_2, \ldots, b_n)$ be two strings in Σ of lengths m and n respectively. For $i \in \mathbb{J}_m$, denote the sub-string $A^{(i)} \stackrel{\text{def}}{=} (a_1, a_2, \ldots, a_i)$. An analogous definition holds for the string B. We denote the edit distance between $A^{(i)}$ and $B^{(j)}$ as $D(i, j)$. With the notation established, the goal is to find $D(m, n)$. To accomplish this task, we will show the original approach proposed by Wagner and Fischer in [WF74], and then we make some adaptions to our specific case.

Although the goal is to obtain an edit sequence with minimum cost, Wagner and Fischer state the problem in a more graphical way using a tool called *traces*. Before the formal presentation of traces, we will give an intuitive idea of what a trace is. A trace is a graphical way to represent how an edit sequence S transforms a string A into B but without showing the order in which the modifications happen and avoiding the redundancies in S. Let us consider Figure 2-1 as an example of a trace taken from [WF74] directly. In this example, each line from the element a_i of A to the element b_j of B means that b_j was derived from a_i either directly if $a_i = b_j$, or indirectly if b_j was obtained from a_i by applying one or more edit operations on a_i. Also, some of the elements of the string A are not connected to any line; these positions are the deleted ones in the process of transforming A into B using the sequence S. Similarly, there are elements in B that are not connected to any line; these positions represent characters in B that are added to A in the conversion process following the sequence S.

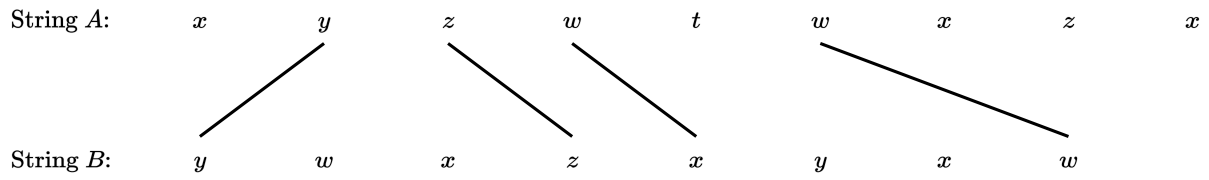


Figure 2-1: Example of a trace taken from [WF74].

Definition 2.1.1 (Trace). *A trace from A to B is triple (T, A, B), where T is a set of ordered pairs of integers (i, j) that satisfies the following properties:*

1. $1 \leq i \leq |A|$ *and* $1 \leq j \leq |B|$.

2. *For any two distinct pairs* (i_1, j_1) *and* (i_2, j_2)*, it holds that:*

   a) $i_1 \neq i_2$ *and* $j_1 \neq j_2$.

   b) $i_1 < i_2$ *if and only if* $j_1 < j_2$.

When there is no confusion about the strings of a trace (T, A, B)*, we denote it as* T*. If* $(a, b) \in T$*, we say that both* a *and* b *are touched by* T*.*

Following the intuitive idea presented before, a pair $(i, j) \in T$ corresponds to a line that goes from a_i into b_j as in Figure **2-1**. The properties above enforce some graphical properties on the traces. On the one hand, Condition 1 in Definition 2.1.1 ensures that the lines in a trace connect two elements inside the strings. On the other hand, Condition 2a ensures that a position of a string is connected to one line, and Condition 2b ensures that no two lines cross.

For a trace (T, A, B) from A to B, we can define the cost of the trace as follows: let I and J be the sets of positions that are not touched by T in A and B respectively; the *cost for the trace* T is

$$\text{Cost}(T) = \sum_{(i,j)\in T} \gamma\left(a_i \to b_j\right) + \sum_{i \in I} \gamma\left(a_i \to \oslash\right) + \sum_{j \in J} \gamma\left(\oslash \to b_j\right)$$

Looking at traces as relations, we can define a notion of the composition of traces as the composition of relations. If T_1 is a trace from A to B and T_2 is a trace from B to C, we can define the trace $T \stackrel{\text{def}}{=} T_1 \circ T_2$ to be the trace from A to C where

$$T \stackrel{\text{def}}{=} \{(a, c) \mid (a, b) \in T_1 \ \wedge \ (b, c) \in T_2\}.$$

With this notion of composition we have the following lemma taken from [WF74].

Lemma 2.1.2 (Composition of traces)**.** *If* (T_1, A, B) *and* (T_2, B, C) *are traces, then* $\mathit{Cost}(T_1 \circ T_2) \leq \mathit{Cost}(T_1) + \mathit{Cost}(T_2)$.

Considering Lemma 2.1.2, we can make a connection between the cost of a trace and the cost of a sequence of edit operations. Such relation derives from the following two properties which can be proven:

1. For every trace (T, A, B), there exists a sequence of editing operations S that transforms A into B such that $\text{Cost}(T) = \gamma(S)$.

2. For every sequence S of editing operations that converts the string A into the string B, there exists a trace (T, A, B) such that $\text{Cost}(T) \leq \gamma(S)$.

These two properties allow us to state the following property that helps us to solve the edit distance problem using traces.

Theorem 2.1.3. *Let A and B be strings in the alphabet Σ. Let S_{min} be a sequence of editing operations that convert A into B with minimum cost, and let (T_{min}, A, B) be a trace with minimum trace cost. Then $\text{Cost}(T_{min}) = \gamma(S_{min})$.*

Proof. Applying Property 1 for T_{min}, there exists a sequence S of editing operations that transform A into B, such that $\text{Cost}(T_{\text{min}}) = \gamma(S)$. Similarly, applying Property 2 for S_{min}, there exists a trace (T, A, B) such that $\text{Cost}(T) \leq \gamma(S_{\text{min}})$. Joining the results, it holds that

$$\text{Cost}(T_{\text{min}}) = \gamma(S) \geq \gamma(S_{\text{min}}) \geq \text{Cost}(T) \geq \text{Cost}(T_{\text{min}}),$$

which shows that $\text{Cost}(T_{\text{min}}) = \gamma(S_{\text{min}})$. □

Before proposing a method to compute the edit distance, we need to observe how the cost of the traces interacts with the concatenation of strings. Let $A \stackrel{\text{def}}{=} A_1 A_2$ be a string defined as the concatenation of string A_1 and A_2, and let $B \stackrel{\text{def}}{=} B_1 B_2$ be the concatenation of the strings B_1 and B_2. Let (T, A, B) a trace that does not connect elements of A_i and B_j for $i \neq j$ and $i, j \in \{1, 2\}$. Then, we can decompose the trace (T, A, B) into two traces (T_1, A_1, B_1) and (T_2, A_2, B_2) such that

$$\text{Cost}(T) = \text{Cost}(T_1) + \text{Cost}(T_2).$$

Moreover, if T is a trace with minimum cost, then (T_1, A_1, B_1) and (T_2, A_2, B_2) will also be traces with minimal cost in their respective strings. This will be useful for developing a strategy to solve the edit distance problem using dynamic programming techniques.

Now, we are ready to present the main theorem to solve the edit distance problem using dynamic programming.

Theorem 2.1.4. *Let $A \stackrel{def}{=} (a_1, a_2, \ldots, a_m)$ and $B \stackrel{def}{=} (b_1, b_2, \ldots, b_n)$ be strings in the alphabet Σ. Then*

$$D(i,j) = \min \begin{cases} D(i-1, j-1) + \gamma(a_i \to b_j) \\ D(i, j-1) + \gamma(\varnothing \to b_j) \\ D(i-1, j) + \gamma(a_i \to \varnothing) \end{cases},$$

for all $1 \leq i \leq |A|$ and $1 \leq j \leq |B|$.

Proof. Let T be a trace from $A^{(i)}$ to $B^{(j)}$ with minimum cost. If a_i and b_j are touched by lines in T simultaneously, then they are touched by the same line, otherwise, the lines will cross and because of Condition 2b in the definition of trace, this is not possible. Therefore, there are three cases:

1. Suppose that there is a line in T that connects a_i and b_j. So, because of our previous discussion, we can split the strings $A = A^{(i-1)}a_i$ and $B = B^{(j-1)}b_j$. Then, we can decompose T into two minimal traces T_1 from $A^{(i-1)}$ to $B^{(j-1)}$, and T_2 from the string a_i to b_j. Also, it holds that $\text{Cost}(T) = \text{Cost}(T_1) + \text{Cost}(T_2)$. Knowing that T_1 and T_2 are traces with minimum cost, we have that $\text{Cost}(T_1) = D(i-1, j-1)$, and $\text{Cost}(T_2) = \gamma(a_i \to b_j)$. Therefore, we conclude that $\text{Cost}(T) = D(i-1, j-1) + \gamma(a_i \to b_j)$.

2. Suppose that a_i is untouched by any line in T. Therefore we can split the string A so that $A = A^{(i-1)}a_i$. This split allows us to decompose T into two minimal traces T_1 from $A^{(i-1)}$ to B, and T_2 from the string a_i to the empty string $\varnothing$. Again, $\text{Cost}(T) = \text{Cost}(T_1) + \text{Cost}(T_2)$, where $\text{Cost}(T_1) = D(i-1, j)$ and $\text{Cost}(T_2) = \gamma(a_i \to \varnothing)$. Then, $\text{Cost}(T) = D(i-1, j) + \gamma(a_i \to \varnothing)$.

3. Suppose that b_j is untouched by any line in T. We can proceed as in the previous case making the split $B = B^{(j-1)}b_j$. Following similar steps we conclude that $\text{Cost}(T) = D(i, j-1) + \gamma(\varnothing \to b_j)$.

Due to Theorem 2.1.3, we have that $\text{Cost}(T) = D(i, j)$. Also, one of the three cases must hold, and $D(i, j)$ is the minimum cost, so $D(i, j)$ must be equal to the minimum of the three possible situations. In conclusion,

$$D(i,j) = \min \begin{cases} D(i-1, j-1) + \gamma(a_i \to b_j), \\ D(i, j-1) + \gamma(\varnothing \to b_j), \\ D(i-1, j) + \gamma(a_i \to \varnothing) \end{cases} .$$

□

We have left a remaining piece to complete a dynamic programming solution. In the strategy of dynamic programming, it is common to have base cases that are the starting point of the computation. The following theorem establishes the base cases for the edit distance problem.

Theorem 2.1.5. *Let* $A \stackrel{def}{=} (a_1, a_2, \ldots, a_m)$ *and* $B \stackrel{def}{=} (b_1, b_2, \ldots, b_n)$ *be strings in the alphabet* Σ. *Then,* $D(0,0) = 0$. *Also,*

$$D(i,0) = \sum_{k=0}^{i} \gamma\left(a_k \to \varnothing\right), \quad \textit{and} \quad D(0,j) = \sum_{k=0}^{j} \gamma\left(\varnothing \to b_k\right).$$

Proof. Notice that when $i = 0$ or $j = 0$, there is only one possible trace which is $T = \emptyset$ which trivially has the least cost. So, the equations derive immediately from the definition of the cost of a trace presented in Equation 2.1. □

As we mentioned before, we are interested in a particular case where the cost of an operation is 1 if there is an effective change in the string. For this reason, we restate Theorems 2.1.4 and 2.1.5 for our particular case in the following two theorems.

Theorem 2.1.6. *Let* $A \stackrel{def}{=} (a_1, a_2, \ldots, a_m)$ *and* $B \stackrel{def}{=} (b_1, b_2, \ldots, b_n)$ *be strings in the alphabet* Σ*. Then*

$$D(i,j) = \min \begin{cases} D(i-1, j-1) + t(i,j), \\ D(i, j-1) + 1, \\ D(i-i, j) + 1 \end{cases},$$

where

$$t(i,j) = \begin{cases} 1, & \text{if } a_i \neq b_j \\ 0, & \text{otherwise} \end{cases}.$$

Theorem 2.1.7. *Let* $A \stackrel{def}{=} (a_1, a_2, \ldots, a_m)$ *and* $B \stackrel{def}{=} (b_1, b_2, \ldots, b_n)$ *be strings in the alphabet* Σ*. Then,* $D(0,0) = 0$*. Also,* $D(i,0) = i$*, and* $D(0,j) = j$*.*

With both theorems, we can design an algorithm to find the edit distance between two strings by applying the results from the theorems in a direct way. The corresponding algorithm is presented in Algorithm 1.

In the presentation of the algorithm, we separate the computation of the matrix t from the main iteration. This allows us to parallelize the comparison of the nucleotides easier than computing them into the main loop. Such parallelization reduces the number of rounds which improves the performance of the overall algorithm. Moreover, these comparisons can not be done in parallel inside the main loop due to practical limitations in the MP-SPDZ framework. More specifically, the dependencies between the data of the matrix D do not allow the optimization algorithms of MP-SPDZ to parallelize the main loops in a consistent way, which produces errors in the final result. In the rest of this work, we will call the *preamble* to the section of the algorithm between Lines 1 and 10. Also, we will call the *arithmetic section* to the section of the algorithm between Lines 11 and 22.

Algorithm 1 Edit distance algorithm

Input: two chains $P = [p_1, \cdots, p_n]$ and $Q = [q_1, \cdots, q_m]$.
Output: an integer value with the edit distance between the chains P and Q.

Let t be a matrix with dimensions $n \times m$.
for $i = 1$ to n **do**
    for $j = 1$ to m **do**
        if $p_i \neq q_j$ **then**
            $t(i, j) = 1$
        else
            $t(i, j) = 0$
        end if
    end for
end for
Let D be a matrix with dimensions $(n + 1) \times (m + 1)$ initialized with zeros.
for $i = 0$ to n **do**
    $D(i, 0) = i$
end for
for $j = 0$ to m **do**
    $D(0, j) = j$
end for
for $i = 1$ to n **do**
    for $j = 1$ to m **do**

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 \\ D(i, j-1) + 1 \\ D(i-1, j-1) + t(i, j) \end{cases}$$

    end for
end for
return $D(n, m)$

2.2 Secure multi-party computation

In the framework of secure multi-party computation (MPC), there is a set of *parties* or *players* $P_1, P_2, \ldots, P_n$, such that the party P_j holds a value x_j and they agree to compute a function f that takes n inputs. The goal is to compute the value $y = f(x_1, \ldots, x_n)$ satisfying the following two conditions:

- Correctness: the correct value y is obtained by all the parties.
- Privacy: the only information learned by the parties is y.

This means that we want the party P_j to learn only his input x_j and the correct output of the function y. The computation of f under such conditions is referred to as computing f *securely* [CDN15, Section 1.3].

The goal is reached by means of an *MPC protocol*, which is a set of rules that the parties follow. Such rules involve both local computation and the exchange of information. This set of instructions depends on the inputs from the parties and they may involve some random sampling. This means that an MPC protocol is defined as a set of probabilistic instructions [Esc21, Section 1.1].

In cryptography, it is very common to express the desired properties of a cryptographic construction by using an ideal scenario in which all of the properties hold almost in a trivial and direct way. In the case of an MPC protocol, our *ideal world* consists of all the parties sending and receiving messages only from a trusted-third party (TTP). The parties provide the input values to the TTP and the TTP returns to each party the correct value of $y = f(x_1, x_2, \ldots, x_n)$. It is important to make clear that the TTP does not have any other behavior but just receives inputs and returns the correct value of the computation. Given that the parties only have communication with the TTP and the TTP does not reveal any information but the output, this ideal world fulfills all the requirements of privacy and correctness mentioned above. While in the ideal world everything works just fine, in the *real world*, the scenario is precisely the MPC protocol where all the parties have communication between them and there is no TTP involved at all. In Figure 2-2, we show an example of the ideal and real-world considering four parties. To prove that an MPC protocol in the real world is secure, we use the ideal world at our disposal. Intuitively, we need to show that the real and ideal world are "indistinguishable".

Given that the definition of security given above is rather intuitive, we need to formalize these concepts using mathematical means. This will allow us to prove formally when an MPC protocol is secure or not. To formalize the notion of security, it is often used the concept of *adversary*. An adversary is an algorithm that controls a set of parties

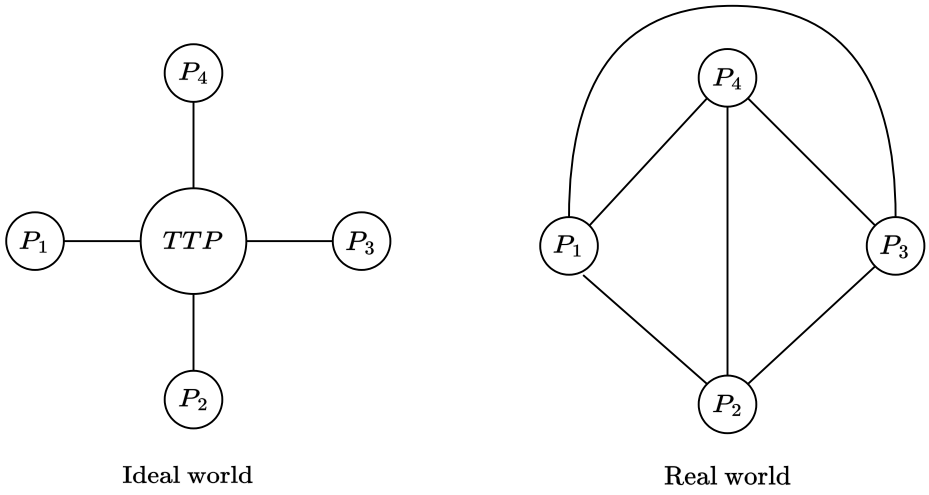


Figure 2-2: Example of an ideal and real world for 4 parties.

$\mathcal{C} \subseteq \{P_1, P_2, \ldots, P_n\}$, such that $|\mathcal{C}| \leq t$, for some $t < n$. The parties in the set $\mathcal{C}$ are called the *corrupted parties*. The adversary can access the messages that the corrupted parties send and receive, and in some cases, it can control the behavior of the corrupted parties during the protocol execution. The goal of the adversary is to gather additional information beyond the function output or affect the correctness of the function output. Therefore, a protocol will be secure if for every adversary corrupting a set of parties $\mathcal{C}$, it can not learn anything but the output of the function from the protocol execution and it can not deprive the honest parties of learning the correct output of the function. Taking into account the ideal and real world, the protocol is secure if the protocol execution is indistinguishable from the ideal world even in presence of an arbitrary adversary controlling a set of corrupted parties. The capabilities of the adversary to gather additional information or to prevent the honest parties to learn the correct output depend on the computational capabilities of the adversary, the subset of corrupted parties it can control, and the possibility of the adversary behaving outside of the instructions of the protocol.

2.2.1 Types of adversaries

There are two main types of adversaries. On the one hand, we have adversaries that want to learn more information beyond the output of the function but they do not deviate from the protocol instructions. These kinds of adversaries are called *passive* or *semi-honest* adversaries. On the other hand, some adversaries want to learn more than the output of the function but they can deviate from the protocol instructions. For example, they can send messages to other parties they want at any moment without any restriction. These kinds of adversaries are called *malicious* or *active* adversaries. An MPC protocol

that is private and correct in presence of active adversaries is called a *robust* protocol [CDN15, Section 4.1]. The protocols that are secure against passive adversaries are more efficient than the protocols that are secure against active adversaries, because the latter need additional mechanisms to ensure privacy and correctness when a corrupted party performs an unexpected action.

Another classification of the adversaries depends on the subset of corrupted parties it can control. It is commonly used to classify adversaries according to the number of corrupted parties t under its control. The main classifications are $t < n$ which is called *dishonest majority*, $t < n/2$ which is called *honest majority*, and there is a third type that is very useful which is $t < n/3$. The number of honest and corrupt parties is important because the protocol designers can devise strategies to reach privacy and correctness efficiently. For example, there are protocols that have a mechanism to identify honest parties, and once such identification is made, the protocol can delegate to such parties some critical computations that need honest and not curious behavior [DEK21].

2.2.2 Privacy guarantees

We mentioned before that a protocol is secure if its execution looks "close" to the ideal world where the TTP takes place in the interaction. Let us make more precise what "close" means. There are three main types of security that reflect how close is a protocol to the ideal world. It is important to remember that an MPC protocol is a set of probabilistic instructions, so the messages and local computations are in fact random variables that have a statistical distribution. The following classification is shown as it is explained in [Esc21, Section 1.1.2].

The first type is *perfect security*. In this case, the ideal world and the MPC protocol execution have the exact same distribution. And from a point of view of the adversary, it can not learn anything from the protocol execution beyond the output of the function regardless of its computational power. The drawback of perfect security is that it is not always achievable. For that reason, we need other definitions of security that are not so strong but they are still useful in practice.

The second type is *statistical security*. This type of security means that the similarity between the real and ideal world is controlled by a certain parameter in the MPC protocol that makes both worlds as close as desired. Both worlds do not have the same distribution but both distributions can be as close as we want by increasing or diminishing the security parameter. From an adversarial point of view, the protocol is secure, but there is a small probability that the adversary can break the security completely regardless of his

computational capabilities. This probability of leakage is controlled by the parameter and it can be as small as we want.

Finally, the third type of security is *computational security*. Here, the security is restricted only to efficient adversaries, which means that the adversary is an algorithm that runs in polynomial time. Both the ideal and the real world will be indistinguishable as long as the adversary has a computational power bounded by some polynomial.

The first two types of security are called *information-theoretic security*. And it commonly happens that the information-theoretic secure protocols are more efficient than the protocols which are computationally secure. This happens because for the latter it is common to use cryptographic primitives that rely on computationally hard problems as subroutines for these protocols. However, information-theoretic protocols are impossible to reach in some situations. For those cases, we are obliged to use computationally secure protocols. But in the practice, such protocols are enough for real-world applications.

2.2.3 Output guarantees

Remember that an important part of the security of a protocol is to be correct, that is, at the end of the execution of the protocol the parties should obtain the correct evaluation of the function f. Related to this, there is a classification of the protocols according to their capabilities to ensure the correctness property.

First, there is the *guaranteed output delivery* (GOD) which means that the protocol always returns the correct value of the function evaluation to the honest parties, no matter what actions are performed by the adversary.

The second type is the *fairness*. Here, the corrupt parties can cause the honest parties not to obtain the output of the computation, which is referred to as causing the parties to *abort*. However, if this abort occurs, the protocol guarantees that the corrupt parties also do not learn the output of the functionality.

The third and weaker guarantee is called *security with abort*. In this output guarantee, the adversaries can cause the honest parties not to obtain the output of the computation, but the corrupt parties may still learn the output.

It is important to highlight, as it is mentioned by [Esc21, Section 1.1.3], that if a protocol is secure against passive adversaries, it will have guaranteed output delivery, because all the parties do not deviate from the protocol instructions. Also, it can happen that in some situations, the guaranteed output delivery can not be reached, but there are some cases

where fairness is still useful in practice.

2.2.4 Complexity measures

In this work, as in the theory and practice of MPC, it is common to compare protocols to determine which one is more suitable than the other for a given task. To make this comparison, three metrics can be used to compare protocols. It is important to mention that all the metrics are orthogonal, and determining which protocol is better than the other is not always an easy task. This choice will occasionally depend on the context where the protocol itself is executed. Aspects like the computational power of the parties and the aspects of the network used to execute the protocol will help to determine what protocol is the best choice.

On the one hand, we have a metric that corresponds to the amount of local computation that the parties need to use to execute the protocol. This metric can be measured similarly to the running time measured in the stand-alone algorithms commonly used in computer science areas.

It is important to remember that the MPC protocols are distributed tasks. In practice, these protocols are executed using multiple machines that are connected to a network running a computer process that receives, sends, and locally processes information. For this reason, the amount of bits sent through the network is an important metric, and it is related to the bandwidth. The fewer bits a protocol sends to the network, the better.

Finally, another metric that needs to be taken into account is the number of rounds used in a protocol. Let us define properly what we mean by a round. Assume that during the protocol execution, the communication between the parties is done by one-to-one channels. This means that each party is connected to every other party by a secure channel to send and receive messages. An *invocation* is a secure computation that is achieved by a single interaction between parties; during such interaction, each party sends a message to all of the other parties [IUS09, Section 2.3.1]. Suppose that the protocol execution is organized in an optimized way where all the invocations that can be done in parallel are computed in one chunk. This organization divides the protocol execution into blocks of computation that start when a group of invocations is computed in parallel and finish when some information is needed and the parallel computation can not continue. Such a number of blocks is what we call the number of rounds. It is important to notice that the number of rounds does not account for the amount of data transmitted. Instead, the number of rounds is related to the amount of computation that can be parallelized which is closely related to the amount of time of the protocol execution. This means that the higher the

number of rounds, the more time is needed for the protocol to finish. Also notice that if we can parallelize a group of invocations, the information coming from such a group can be sent through the network in one package instead of multiple packages, one for each invocation. This means that a protocol with a high number of rounds will have a higher execution time because the latency will take effect once each round finishes and the information needed for the next round is sent through the network.

Given these three metrics, it is important to say that none of them is a definitive metric to choose a protocol. For example, if we have a network comprised of multiple powerful computers, a high bandwidth, and a high latency, we can prefer a protocol with a low number of rounds but with a possibly high number of bits sent through the network. On the contrary, if we have an infrastructure comprised of computers with low processing power but with good network characteristics, we may select a protocol that has a low computational complexity but a high number of rounds and that sends a high number of bits. Everything will depend on the tools and infrastructure we are using to execute the protocol.

2.3 Garbled circuits

One of the goals of this work is to compare our solution with the current solutions for the secure computation of edit distance. As we saw in Section 1.1, the garbled circuits (GC) are one of the most widely used techniques to compute the edit distance securely. In this section, we will cover some basic theory of garbled circuits and their more significant constructions.

2.3.1 Yao's garbled circuits

The *Yao's GC* is a cryptographic technique originally proposed by Andrew Yao [Yao82; Yao86] and then formalized by Goldreich, Micali, and Wigderson [GMW87]. In fact, this cryptographic technique was one of the starting points in the area of MPC and led to the development of more protocols for such end. In this section, we will expose the basics of Yao's GC at a high-level. The ideas revisited in this section are taken from [EKR18, Section 3.1]. However, a more formal treatment of this technique can be found in resources like [BHR12].

The original proposal of Yao's GC was designed for two parties. In such case, two parties P_1 and P_2 want to compute a function f that receives two arguments. In this scenario, the

party P_1 holds x and the party P_2 holds the value y, and they want to compute $f(x, y)$ securely.

The strategy to solve the problem is to consider the function f as a lookup table. Suppose that X is the domain of the inputs coming from party P_1. Similarly, let Y be the domain of the inputs coming from party P_2. Assuming that X and Y are finite sets, we can see the function f as a lookup table T with $|X| \cdot |Y|$ rows using a proper enumeration of the input pairs $(x, y) \in X \times Y$. The row enumerated by (x, y) of the table, which will be written as $T_{x,y}$, will contain the value of $f(x, y)$.

This representation of the function f allows us to privately evaluate f as follows. Let κ be a security parameter and let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric-key cryptographic scheme. P_1 will encrypt the table T by assigning to each input pair $(x, y) \in X \times Y$ a pair of strong keys $k_x = \mathsf{Gen}(1^\kappa)$ and $k_y = \mathsf{Gen}(1^\kappa)$. Then, P_1 encrypts each position of the table T by computing $\mathsf{Enc}_{k_x,k_y}(T_{x,y})$ and obtaining the table $[\mathsf{Enc}_{k_x,k_y}(T_{x,y})]_{(x,y)\in X\times Y}$. Finally, P_1 makes a random permutation of the rows of the encrypted table and sends it to P_2.

When P_2 receives the encrypted table, it needs to decrypt the position that corresponds to the input to the secure computation (x, y). This means that P_2 needs to know somehow the values of k_x and k_y to be able to decrypt the output $T_{x,y}$. Given that P_1 knows x and he computed k_x, then P_1 can send k_x to P_2. Notice that if k_x is generated in a random secure way, it will not reveal anything about x. From a point of view of a passive adversary, k_x is a random string. It remains for P_2 to learn k_y. This can be done using a 1-out-of-$|Y|$ oblivious transfer (OT) protocol as a subroutine. In this oblivious transfer protocol, the party P_2 gives y as input to the protocol, and P_1 gives the set $\{k_r\}_{r\in Y}$ as input. At the end of the execution of the OT protocol, P_2 will learn only the value of k_y and P_1 will learn nothing. There are multiple ways to realize this OT protocol, but we will not include them here given that it is not the main subject of this section. In the MPC terminology, P_1 is also called the *garbler*, and P_2 is called the *evaluator*.

Point-and-permute. In the previous description, we mentioned that P_1 returns the encrypted table with its rows randomly permuted. But, how will P_2 know which position he needs to decrypt? This problem is solved using a technique called *point-and-permute* proposed in [BMR90]. In this technique, P_1 will encode a pointer to the position to decrypt in the last bits of the keys k_x and k_y. The encoding is such that it avoids collisions between positions and it needs to maintain the level of security of the keys. So, P_1 will encode the pointer in $\lceil \log_2 |X| \rceil$ bits and append them to k_x and also using $\lceil \log_2 |Y| \rceil$ bits and append them to k_y. Appending the bits at the end of the keys will ensure that the length of the keys (and so, the security level of the encryption) is not affected.

Notice that the previous approach depends heavily on the sizes of X and Y. So, when the

domains of the inputs are very large, the construction of the table T is computationally impractical. To address this problem, we can represent f as a Boolean circuit C and evaluate each gate as a lookup table of 4 entries. The strategy will be to encrypt each position of the tables as mentioned before, but we need to keep the output value of each gate secure to avoid revealing the intermediate steps in the evaluation of the circuit. To keep the intermediate values hidden, we also make the output of each gate also a key.

Let C be the Boolean circuit associated with the function f. For each wire w_i of the circuit, we will have two keys k_i^0 and k_i^1. These keys represent the two possible values of the wire but notice that they do not reveal the plaintext value associated with the wire. These keys are called the *wire labels* and the plaintext wire values are called the *wire values*. When we are in the middle of the evaluation of the circuit C, each wire will have a specific wire value that depends on the inputs to the circuit and also a specific wire label. These specific values and labels are called *active values* and *active labels* respectively. The goal is to transmit just the active label, and neither the active value nor the inactive label to keep the computation private.

Let G be a gate in C with input wires w_i and w_j, and an output wire w_t. For P_1 to construct the garbled table for G, he computes

$$T_G \stackrel{\text{def}}{=} \begin{pmatrix} \mathsf{Enc}_{k_i^0,k_j^0}\left(k_t^{G(0,0)}\right) \\ \mathsf{Enc}_{k_i^0,k_j^1}\left(k_t^{G(0,1)}\right) \\ \mathsf{Enc}_{k_i^1,k_j^0}\left(k_t^{G(1,0)}\right) \\ \mathsf{Enc}_{k_i^1,k_j^1}\left(k_t^{G(1,1)}\right) \end{pmatrix}.$$

This means that, for each gate, P_1 encrypts the labels corresponding to the output computed by the gate using the respective labels of the input wires as keys. Once P_1 computes the tables for each gate, he permutes the entries of the tables and sends them to P_2. These tables are commonly known as *garbled tables* or *garbled gates*. Also, similarly to the discussion at the beginning of the section, P_1 sends to P_2 only the active labels corresponding to the input wires. Once again, P_1 knows the labels associated with his own input wires. The labels corresponding to the input wires whose values are in possession of P_2 are sent using a 1-out-of-2 OT protocol.

When P_2 receives the garbled tables and the active input labels, he can proceed to decrypt sequentially the positions according to the active labels. In the middle of the process, P_2 will decrypt the intermediate keys which are the active labels and he will use these intermediate labels as inputs of the subsequent garbled tables until he reaches the end. In the end, P_2 will end up with the active label of the output wire. This final output label can be sent to P_1 again to obtain the active value because P_1 was the party that computed the keys, so it knows the value associated with the active label of the output. This final

step can be avoided just by letting P_1 send a decoding table that maps only the output wires to their corresponding plaintext values.

It is important to mention that this construction of Yao's GC is secure only against semi-honest adversaries. This technique can be extended to an arbitrary number of parties, and also to be secure against active adversaries as we will see in the next section.

2.3.2 BMR

Although Yao's GC solves the problem of secure function evaluation in the context of semi-honest adversaries for two parties, the technique alone is not enough if we want to reach security against malicious adversaries and an arbitrary number of corrupted parties. To solve these limitations, Beaver, Micali, and Rogaway proposed a technique known as BMR [BMR90]. The original proposal of BMR solves the problem of extending Yao's GC protocol to more than two parties. However, the BMR protocol is secure against malicious adversaries only when a minority of the parties are corrupted. To solve this problem, Lindell et al. propose a modification of the original BMR protocol to allow a dishonest majority setting and a malicious adversary in a constant number of rounds [Lin+19]. In this chapter, we will revisit both the original proposal of BMR and the improvement of Lindell et al. for the malicious adversaries in the dishonest majority setting.

The idea behind the BMR protocol is to use another auxiliary protocol to compute the garbling gates. The BMR protocol is divided into two sections: an *offline* phase where the garbling gates are constructed, and an *online* phase where the garbled gates are exchanged and the parties evaluate the garbled circuit locally. For the rest of this section, let κ be a security parameter, and let n be the number of parties. Also, suppose that the parties agree to compute a function f.

The original BMR protocol [BMR90]

The original version of BMR considers two types of *seeds* for each wire in the circuit of f. On the one hand, each party P_j assigns to each wire w two seeds: $s^j_{w,0}$, $s^j_{w,1}$, one for each possible value in the wire. The former is often called 0-seed and the latter is called 1-seed. On the other hand, in the garbling process, the parties produce two *superseeds* defined as the concatenation of the 0-seeds and 1-seeds as follows:

$$S_{w,0} \stackrel{\text{def}}{=} s^1_{w,0} \| s^2_{w,0} \| \cdots \| s^n_{w,0} \quad \text{and} \quad S_{w,1} \stackrel{\text{def}}{=} s^1_{w,1} \| s^2_{w,1} \| \cdots \| s^n_{w,1}. \tag{2-1}$$

Also, let us define $L \stackrel{\text{def}}{=} n \cdot \kappa$.

Remember that f can be written as a circuit C_f composed of wires and gates. Let g be a gate in C_f computing a function $f_g : \{0,1\} \times \{0,1\} \to \{0,1\}$. Following the same idea of Yao's GC, the garbled gate of g is computed such that the superseed associated with the output of g is encrypted using the superseeds associated with the inputs of g according to the truth table of g. As an example, if we want to use the superseed $S_{w,0}$ to encrypt a value M of length L, we compute

$$M \oplus \left(\bigoplus_{i=1}^{n} G\left(s_{w,0}^{i}\right) \right),$$

where, G is a pseudo-random generator that takes a seed of length κ and outputs a string of length L. Notice that to decrypt a value using a superseed, all the associated seeds need to be known to compute the decryption of such value.

Notice that using just seeds to hide the plain value of a wire is not enough. For example, an adversary may know that the first key is associated with the 0 value, and the second one is associated with the 1 value. To overcome this problem, the protocol uses a random independent masking bit λ_w associated with each wire w. Let ρ_w be the plain value going through the wire w. To ensure that all the real values are kept hidden, the masking bits λ_w are unknown to all the parties during the execution of the protocol. The masked values that are known to the parties, denoted by Λ_w, are called *external values* and they are defined to be

$$\Lambda_w \stackrel{\text{def}}{=} \lambda_w \oplus \rho_w.$$

When the parties are evaluating the circuits, the only values they see are the external values, which do not reveal any information about the real values ρ_w, unless they know the masking bit λ_w. The case of the input wires is a special one because the parties know the real value of their own input. So, the masking values λ_w are given to the party that owns the input of w, so that the party can compute the external value locally and send it to the other parties.

Now, let us define what a garbled circuit is in this case. A BMR garbled circuit consists of a set of garbled gates. Each garbled gate is constructed using an ideal functionality using some inputs that we will define shortly. Suppose that g is a gate with inputs wires a and b, and an output wire c. Each party P_i inputs to the functionality the seeds $s_{a,0}^{i}$, $s_{a,1}^{i}$, $s_{b,0}^{i}$, $s_{b,1}^{i}$, $s_{c,0}^{i}$ and $s_{c,1}^{i}$. With these values, the functionality assembles the superseeds $S_{a,0}$, $S_{a,1}$, $S_{b,0}$, $S_{b,1}$, $S_{c,0}$, and $S_{c,1}$ as defined in Equation (2-1). Additionally, each party inputs the output of a pseudo-random generator G applied to each of his seeds. Finally, the construction of the gates assumes that the masking bits λ_w are shared among all the parties, for $w \in \{a, b\}$. This means that, for each $w \in \{a, b\}$, the party P_i holds a value λ_w^i such that $\lambda_w = \bigoplus_{i=1}^{n} \lambda_w^i$. So, to construct the garbled gate, the functionality also receives the masking bits λ_a^i, λ_a^i and λ_a^i from each party P_i.

The output of the functionality is a garbled gate which is a table of four ciphertexts. Each ciphertext is the encryption of either $S_{c,0}$ or $S_{c,1}$. The output of the functionality has the property that given superseeds for wires a and b, it is possible to decrypt one cyphertext and reveal the superseed of c based on the values of the input wires and the evaluation of such values in the gate g.

Functionality 2.1 is in charge of garbling a gate. This functionality was taken from [Lin+19] and is part of the offline phase of the BMR protocol. To realize this functionality, we need the help of another MPC protocol as we will see later for the case of the work of Lindell et al. in Section 2.3.2. In the specification of Functionality 2.1, we denote the negation of a bit by $\overline{b} \stackrel{\text{def}}{=} b \oplus 1$, for $b \in \{0,1\}$.

To complete the specification of all the parts of the BMR protocol, we explain how to perform the online phase. The online phase consists in taking the garbled gates and evaluating the garbled circuit to obtain the output. In this phase, the parties need to obtain a superseed for the input gates, and then they can evaluate the circuit locally. In the end, the parties obtain the superseed associated with the external value of the output wire of the entire circuit. It is important to remark that once the parties obtain the superseeds for the input, the evaluation of the garbled circuit can be done with no interaction between the parties at all. In Protocol 2.1, there is a specification of the online-phase-BMR protocol. It is important to explain that, in Step 2 of the protocol, the parties have all the information needed to compute the mask required to decrypt the value of S_{c,Λ_c} from the garbled gate. Intuitively, recall that under honest behavior, the mask used to obtain the ciphertext of the garbled gate is computed as the XOR of strings of the form $G(s^i_{w,0})$ or $G(s^i_{w,1})$, for $w \in \{a,b\}$. But, all the parties know the values of s^i_{a,Λ_a} and s^i_{b,Λ_b}, because they have the superseeds S_{a,Λ_a} and S_{b,Λ_b} at their disposal. Therefore, they can compute the mask to decrypt S_{c,Λ_c} as required.

Notice that the BMR protocol is correct. Let us explain this fact using one of the four combinations of values for Λ_a and Λ_b. To prove the correctness, we need to verify that, for all possible values of Λ_a and Λ_b, the resulting value from Step 2 in Protocol 2.1 is S_{c,Λ_c}. Suppose that $(\Lambda_a, \Lambda_b) = (0,1)$. According to Protocol 2.1, the parties need to use the cyphertext B_g for the decryption. Once they decrypt such value, they obtain $S_{c,B}$, as described in Functionality 2.1. Notice that $\Lambda_a = \rho_a \oplus \lambda_a = 0$, then $\rho_a = \lambda_a$. Similarly, $\Lambda_a = \rho_b \oplus \lambda_b = 1$, which means that $\rho_b = \overline{\lambda_b}$. By the definition of f_g, we have that $f_g(\rho_a, \rho_b) = \rho_c$, then it holds that $f_g(\rho_a, \rho_b) = f_g(\lambda_a, \overline{\lambda_b})$. There are two cases:

- If $f_g(\lambda_a, \overline{\lambda_b}) = \lambda_c$, this means that $S_{c,B} = S_{c,0}$. Given that $\rho_c \stackrel{\text{def}}{=} f_g(\rho_a, \rho_b) = f_g(\lambda_a, \overline{\lambda_b}) = \lambda_c$, then $\Lambda_c \stackrel{\text{def}}{=} \rho_c \oplus \lambda_c = 0$. Therefore, the decrypted value $S_{c,B}$ is equal to S_{c,Λ_c}.

Functionality 2.1: garble-gate-BMR

Let κ be a security parameter and $G : \{0,1\}^\kappa \to \{0,1\}^{2\kappa n}$ a pseudo-random generator. Denote G_1 the first $L \stackrel{\text{def}}{=} n \cdot \kappa$ bits of the output of the function G. Similarly, denote G_2 the last L bits of the output of G. Let g be a gate computing the function $f_g : \{0,1\} \times \{0,1\} \to \{0,1\}$ with input wires a and b, and output wire c.

Inputs: for each input gate, the input of the functionality are the following:

- Seeds: each party P_i inputs uniform seeds $s^i_{w,b} \in \{0,1\}^\kappa$, for $b \in \{0,1\}$ and $w \in \{a,b,c\}$.
- Stretched seed: each party P_i inputs L-bit strings $\tilde{\gamma}^i_{w,b}$ and $\gamma^i_{w,b}$, for $b \in \{0,1\}$ and $w \in \{a,b,c\}$. Notice that if P_i is honest, $\tilde{\gamma}^i_{w,b} = G_1\left(s^i_{w,b}\right)$ and $\gamma^i_{w,b} = G_2\left(s^i_{w,b}\right)$.
- Masking bit: each party P_i inputs a uniform $\lambda^i_w \in \{0,1\}$, for $w \in \{a,b,c\}$.

Output: The functionality computes $\lambda_w = \bigoplus_{i=1}^n \lambda^i_w$. Then, it sets the following values:

$$
\begin{aligned}
S_{c,A} &= \left(f_g\left(\lambda_a, \lambda_b\right) = \lambda_c \;?\; S_{c,0} \;:\; S_{c,1}\right), \\
S_{c,B} &= \left(f_g\left(\lambda_a, \overline{\lambda_b}\right) = \lambda_c \;?\; S_{c,0} \;:\; S_{c,1}\right), \\
S_{c,C} &= \left(f_g\left(\overline{\lambda_a}, \lambda_b\right) = \lambda_c \;?\; S_{c,0} \;:\; S_{c,1}\right), \\
S_{c,D} &= \left(f_g\left(\overline{\lambda_a}, \overline{\lambda_b}\right) = \lambda_c \;?\; S_{c,0} \;:\; S_{c,1}\right).
\end{aligned}
$$

The functionality outputs the following four ciphertexts that correspond to the garbled gate:

$$
\begin{aligned}
A_g &= \left(\bigoplus_{i=1}^n \tilde{\gamma}^i_{a,0}\right) \oplus \left(\bigoplus_{i=1}^n \tilde{\gamma}^i_{b,0}\right) \oplus S_{c,A}, \\
B_g &= \left(\bigoplus_{i=1}^n \gamma^i_{a,0}\right) \oplus \left(\bigoplus_{i=1}^n \tilde{\gamma}^i_{b,1}\right) \oplus S_{c,B}, \\
C_g &= \left(\bigoplus_{i=1}^n \tilde{\gamma}^i_{a,1}\right) \oplus \left(\bigoplus_{i=1}^n \gamma^i_{b,0}\right) \oplus S_{c,C}, \\
D_g &= \left(\bigoplus_{i=1}^n \gamma^i_{a,1}\right) \oplus \left(\bigoplus_{i=1}^n \gamma^i_{b,1}\right) \oplus S_{c,D}.
\end{aligned}
$$

Protocol 2.1: online-phase-BMR

Step 1. Send external values:

1. Every party P_i broadcasts the external values related to his own input wires. Recall that each party know its own real input value ρ_w and also λ_w. So, each party can compute the external value Λ_w related to its own input wires. At the end of the step, each party knows all the external values Λ_w, where w is an input wire of the circuit.

2. Every party P_i broadcasts the Λ_w-seed for each circuit-input wire. At the end of the step, each party will know the Λ_w-superseeds for each input wire of the circuit.

Step 2. Evaluate the circuit: the parties evaluate the circuit in a topological order. Let g be a gate with input wires a and b, and with output wire c. Given the values of S_{a,Λ_a} and S_{b,Λ_b} the parties can obtain Λ_a and Λ_b. Each party P_i can do that by looking at its fragment seed s^i_{w,Λ_w} inside S_{w,Λ_w}, for $w \in \{a, b\}$. Once Λ_a and Λ_b are known, the parties use the following value to decrypt the value of S_{c,Λ_c}:

- If $(\Lambda_a, \Lambda_b) = (0, 0)$, the parties use the ciphertext A_g.
- If $(\Lambda_a, \Lambda_b) = (0, 1)$, the parties use the ciphertext B_g.
- If $(\Lambda_a, \Lambda_b) = (1, 0)$, the parties use the ciphertext C_g.
- If $(\Lambda_a, \Lambda_b) = (1, 1)$, the parties use the ciphertext D_g.

- If $f_g(\lambda_a, \overline{\lambda_b}) \neq \lambda_c$, it holds that $S_{c,B} = S_{c,1}$. Notice that $\rho_c \stackrel{\text{def}}{=} f_g(\rho_a, \rho_b) = f_g(\lambda_a, \overline{\lambda_b}) \neq \lambda_c$, which means that $\rho_c = \overline{\lambda_c}$. Computing Λ_c, it holds that $\Lambda_c \stackrel{\text{def}}{=} \rho_c \oplus \lambda_c = 1$. Therefore, the value of $S_{c,B}$ is equals to S_{c,Λ_c} again.

The remaining three cases for the other possible values for (Λ_a, Λ_b) have a very similar proof.

In this thesis, we decided not to show the original protocol that realizes the ideal functionality to construct the garbled gates. Remember that the original proposal of BMR is secure against active adversaries in the honest majority setting. Instead, in the next section, we will show how to accomplish this task by using the approach of Lindell et al. that brings security against an active adversary in the dishonest majority setting.

The general BMR protocol [Lin+19]

As we mentioned before, the final job to do is to construct a protocol that realizes Functionality 2.1. In this thesis, we will show a general construction proposed in [Lin+19]. Before Lindell's work, the proposals to realize the entire BMR protocol were either secure against semi-honest adversaries or secure against malicious adversaries but in the honest majority setting. No proposal cover both the security against malicious adversaries in the dishonest majority case. However, Lindel et al. propose a protocol that is secure in such a desirable threat model. Moreover, such a proposal is sufficiently efficient to be implemented in practice (or what they call *concretely efficient*).

The baseline of the protocol relies on the original version of BMR presented in the previous section along with some changes. First of all, instead of using the XOR of bit strings, the protocol uses arithmetic additions in a finite field. Then, all the computations are not over binary circuits but arithmetic circuits. The second main modification is the replacement of the seeds $s^i_{w,b}$ for randomly generated keys $k^i_{w,b}$. Remember that the seeds were bit strings that serve as input of a pseudo-random generator. Instead, the keys are finite field elements that will be inputs of a pseudo-random function. The other modification is that, compared to the original version, this proposal does not need to encode the external values of the wires using superseeds. This is because the parties can spot the external value encoded in the superseed just by looking at their own fragment seed inside. The fourth modification is that the protocol does not check that the random input to the garbling functionality is the output of a pseudo-random function. Remember that at this moment we are dealing with malicious adversaries, so in the inputs of the Functionality 2.1, a corrupted party can provide a string that is not an output of a pseudo-random generator. In other references, this is solved by executing expensive zero-knowledge proofs that make

the protocol impractical. Instead, Lindell et al. prove that this kind of adversarial behavior just makes the protocol abort, but does not reveal sensitive information. This situation is fine in this context because the best that can be done in the dishonest majority case with an active adversary is to have security with abort [Esc21, Section 1.3.4]. Moreover, this new approach allows the protocol to treat the pseudo-random function as a black box instead of considering the circuit of such a function to perform a zero-knowledge proof. This reduction allows a simplification of the protocol and an improvement in its performance. The last change is that the garbled values are encrypted as vectors, namely, vectors of length n. Having vectors allows the protocol to encode n values in $\{0,1\}^\kappa$ in a field instead of encoding values in $\{0,1\}^L$ in a larger field.

For the protocol, each party P_i chooses a pair of keys $k^i_{w,0}$ and $k^i_{w,0}$, for each wire w. Such keys are elements of a finite field $\mathbb{F}_p$, with p prime such that $2^\kappa < p < 2^{\kappa+1}$. In particular, p will be the smallest prime greater than 2^κ. The pseudo-random function is denoted as F, and the evaluation of an element x in the function using the key k is denoted as $F_k(x)$. The output of the pseudo-random function, as well as the keys, are interpreted as elements of $\mathbb{F}_p$. In practical implementations, we can use a block cipher in the CBC-MAC mode to instantiate F [Lin+19].

The big picture of the improved BMR protocol is shown in Figure **2-3**. The main goal is to design a protocol Π_{SFE}, where SFE stands for "Secure Function Evaluation". This protocol will realize a functionality $\mathcal{F}_{\mathsf{SFE}}$ that allows to securely evaluate a function f represented as a circuit $\mathcal{C}_f$ under a malicious adversary in the dishonest majority case. The protocol Π_{SFE} will be constructed in the $\mathcal{F}_{\mathsf{offline}}$-hybrid model, where $\mathcal{F}_{\mathsf{offline}}$ is the functionality in charge of garbling the gates of the circuit. This means that the protocol Π_{SFE} will use a sub-protocol Π_{offline} that realizes the functionality $\mathcal{F}_{\mathsf{offline}}$. But the protocol Π_{offline} will be constructed also in the $\mathcal{F}_{\mathsf{MPC}}$-hybrid model. The $\mathcal{F}_{\mathsf{MPC}}$ will provide the commands Add and Multiply to perform additions and multiplications securely. These two commands are used in Π_{offline} between elements of $\mathbb{F}_p$ to construct the garbled gates. Specifically, Lindell et al. show both a general method to construct Π_{offline} using the functionality $\mathcal{F}_{\mathsf{MPC}}$, and also, they show how to realize the offline phase using the SPDZ protocol [Dam+12].

As well as in the original version of BMR, Π_{SFE} is divided into two phases: the online phase and the offline phase. The offline phase is divided into two sub-phases: preprocessing-I and preprocessing-II. Here we present a brief summary of both sub-phases. For more details about preprocessing-I and preprocessing-II, we refer to the reader to [Lin+19, Functionality 4].

The command preprocessing-I is in charge of generating and sending the random values needed to generate the garbled gates. Such values include the keys and the masking values. Specifically, the masking values $\lambda_w \in \{0,1\}$ are chosen at random and stored for

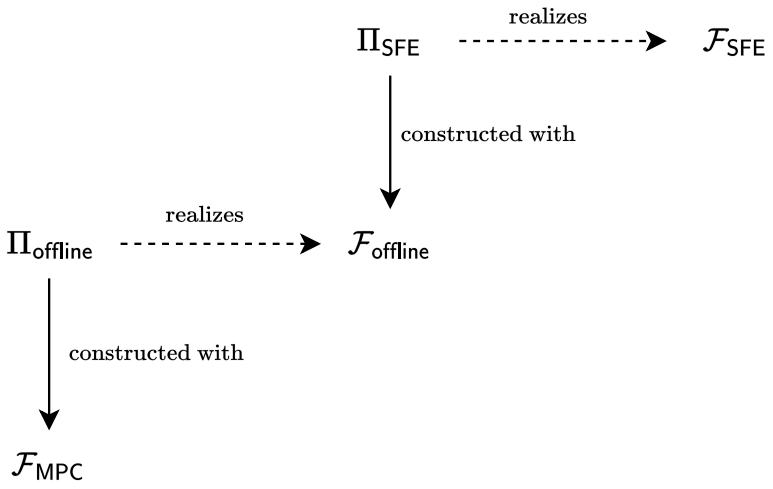


Figure 2-3: Scheme of protocol and functionality dependency for the $\mathcal{F}_{\mathsf{offline}}$ functionality. This scheme is adapted from [Lin+19].

each wire w. In particular, the command Multiply provided by $\mathcal{F}_{\mathsf{MPC}}$ is used to generate the masking bits (see [Lin+19, Apendix A]). Also, for each party P_i, the functionality chooses the keys $k^i_{w,0}$ and $k^i_{w,1}$, and sends them to the corresponding party.

The command preprocessing-II is in charge of constructing the garbled gates. First of all, preprocessing-II opens the values of λ_w, for each input wire w, to the corresponding party owning the input wire. Also, the command opens λ_w for the output wires to all the parties. These two operations do not harm the privacy because the parties know their own inputs and they are intended to learn the output of the function f. After that, each party uses the keys and the pseudo-random function to provide random values to the command preprocessing-II. For the party P_i, these random values are computed as $F_{k^i_{w,b}}(0\|j\|g)$, for $b \in \{0,1\}$ and $i,j \in \mathbb{J}_n$. Finally, the garbled gate are four vectors $\mathbf{A}_g = (A^1_g, \ldots, A^n_g)$, $\mathbf{B}_g = (B^1_g, \ldots, B^n_g)$, $\mathbf{C}_g = (C^1_g, \ldots, C^n_g)$ and $\mathbf{D}_g = (D^1_g, \ldots, D^n_g)$, where each position of each vector is the sum of values of the form $F_{k^i_{w,b}}(0\|j\|g)$ along with a key $k^j_{c,Q}$, where $Q \in \{0,1\}$ is a bit that depends on the function f_g that computes the gate g. To compute such additions, it is needed the command Add provided by the functionality $\mathcal{F}_{\mathsf{MPC}}$.

The protocol Π_{SFE} executes the offline phase by executing the commands in $\mathcal{F}_{\mathsf{offline}}$. These commands generate all the random values needed to construct the garbled gates and evaluate them in the online phase. In the online phase, the parties compute the external values of the input wires as in the original protocol. In this improved version of the protocol, we do not have any superseed to encode the external values, instead, we have keys. These keys with the form k^i_{w,Λ_w} encode the external values Λ_w. In this phase, the parties receive the garbled gates with the form $\mathbf{A}_g$, $\mathbf{B}_g$, $\mathbf{C}_g$, $\mathbf{D}_g$, for every gate g. Again,

similarly to the original protocol, the party P_i passes through each gate g of the circuit in topological order to obtain the keys k^i_{c,Λ_c} where c is the output wire of g. The party will choose to use one of the values from $\mathbf{A}_g$, $\mathbf{B}_g$, $\mathbf{C}_g$ and $\mathbf{D}_g$, according to the values of Λ_a and Λ_b, where a and b are the input wires of g. To obtain the values of k^i_{c,Λ_c}, the party undoes the operations performed in the command preprocessing-II used to mask the value of k^i_{c,Λ_c}. Remember that for undoing the operations, the protocol does not perform XOR as in the original version, but additions and subtractions. Different from the original protocol, the parties need to check if the keys that they are obtaining from the wire c match with the set of keys $\{k^i_{c,0}, k^i_{c,1}\}$ they have. If not, the parties abort. In the end, all the parties will obtain Λ_w, where w is an output wire of the entire circuit, and they can unmask $\rho_w = \lambda_w \oplus \Lambda_w$ given that λ_w is obtained in the pre-processing step.

This improved BMR protocol has a constant number of rows. Precisely, the protocol has 12 communication rounds, where 3 of them are in the online phase. This means that this protocol is suitable for network architectures of high latency. However, even when this protocol has a low number of rounds, the data sent in this type of protocol tends to be large and does not scale very well with the size of the circuit because one party constructs a garbled table for each gate in the circuit and sends it to the second party. This lack of scalability with respect to the data sent harms the performance when using a network architecture with low bandwidth. It is important to mention that the work of Lindell et al. is the baseline in later works like the one of Keller and Yanai [KY18]. Moreover, such later work is the one that is implemented in the framework MP-SPDZ [Kel20], which is extensively used in the experiment section of this work.

2.4 Secret-sharing schemes

In the previous section, we revisited some techniques based on garbled circuits. However, in the area of MPC, there is a wide variety of techniques that use different tools to compute a function securely, and each of them has its pros and cons. In this section, we will explain another technique called *secret-sharing*. This type of technique is of high relevance in our work because we intend to compare the performance of both the secret-sharing schemes and garbled circuits to compute the edit distance.

Throughout this introductory part, we fix a finite field $\mathbb{F}$, which will be the domain of computation of the secret-sharing schemes. If another algebraic structure is intended to be used, we will state that explicitly. Also suppose that n is the number of parties participating in the protocol, such that they agree to compute a function $f : \mathbb{F}^n \to \mathbb{F}$.

The main idea behind the secret-sharing schemes is that an element $s \in \mathbb{F}$ will be frag-

mented into n random parts called *shares*, such that each part will be held by each party. These parts are computed in such a way that some subsets of parties are not allowed to learn anything about s, while other subsets are able to learn s completely. In this work, we will deal with *threshold* secret-sharing schemes, which are schemes that do not leak information about s as long as less or equal to t shares are known. At the moment that $t+1$ shares are available, they allow revealing s completely. More specifically, a secret-sharing scheme provides tools to compute a tuple $(s_1, s_2, \ldots, s_n) \in \mathbb{F}^n$, such that:

- For any set $A \subseteq \mathbb{J}_n$, such that $|A| \leq t$, the set $\{s_i\}_{i \in A}$ does not leak any information about s.
- For any set $B \subseteq \mathbb{J}_n$, such that $|B| \geq t + 1$, the set $\{s_i\}_{i \in B}$ can be used to fully reconstruct the value s.

When an element s is divided into parts as specified above, we say that s is *secret-shared*. If s is secret-shared using shares $(s_1, s_2, \ldots, s_n)$, we denote this as

$$[\![s]\!] \stackrel{\text{def}}{=} (s_1, s_2, \ldots, s_n).$$

The common strategy to compute the output of an agreed function f using secret-sharing schemes works as follows:

1. The function f is represented as an arithmetic circuit C_f.
2. Each party P_i takes its own input x and compute the shares $[\![x]\!] \stackrel{\text{def}}{=} (x_1, \ldots, x_n)$. For each party P_j, P_i sends x_j to P_j.
3. The parties execute a protocol to evaluate the arithmetic gates in such a way that the parties obtain the output of each gate also in a secret-shared format. This will prevent the parties to learn something about the intermediate steps of the computation.
4. At the end, each party obtains a share of the output of the circuit C_f. Therefore, they can join their shares to reconstruct the output of the function.

A particular case of this type of scheme is the *linear* secret-sharing schemes. Formally, let $x, y \in \mathbb{F}$. Suppose that $[\![x]\!] = (x_1, \ldots, x_n)$ and $[\![y]\!] = (y_1, \ldots, y_n)$. In a linear secret-sharing scheme, it holds that $[\![x \pm y]\!] = (x_1 \pm y_1, x_2 \pm y_2, \ldots, x_n \pm y_n)$ [Esc21]. Considering the common strategy for secret-sharing protocols presented above, this fact means that in a linear secret-sharing scheme there is no need for communication for computing additive gates because each party can compute it locally. Specifically, the party P_i holds x_i and y_i, so he can compute $x_i \pm y_i$, which is a valid share for $x \pm y$. An immediate consequence of this fact is that, for a publicly known value $c \in \mathbb{F}$, it holds that $[\![c \cdot x]\!] = (c \cdot x_1, \ldots, c \cdot x_n)$,

which means that this operation does not need communication too.

Notation. Let $x \in \mathbb{F}$, and suppose that $[\![x]\!] = (x_1, \ldots, x_n)$. When a party P_i has a share of x, we abuse the notation saying that the party holds $[\![x]\!]$ to state that P_i has x_i. It needs to be clear that the party does not hold the complete array of shares, because it will break privacy. This is especially useful when we require to say that if the party has shares of x and y, he can obtain a valid share of $x + y$ by saying that the party can obtain a share $[\![x + y]\!]$ by doing $[\![x]\!] + [\![y]\!]$. Formally, this means that the party will have a valid share of $x + y$ by adding $x_i + y_i$. The ambiguity with the notation should be clear from the context, otherwise, we will state its meaning explicitly.

The linear secret-shared schemes allow to cheaply compute addition gates, however, to compute a multiplication gate, it is needed more resources. That is, if we have shares $[\![x]\!]$ and $[\![y]\!]$, it requires some work to obtain a share $[\![x \cdot y]\!]$. In fact, it can be proven that to compute the share of a product, the use of some sort of communication between the parties is unavoidable. So, to design a linear secret sharing scheme, it is needed to specify how the shares are constructed and shared between parties, how to reconstruct values from the shares, and more importantly, how to compute the share of a multiplication gate.

2.4.1 SPD$\mathbb{Z}_{2^k}$

In this section, we will revisit a technique based on secret-sharing called SPD$\mathbb{Z}_{2^k}$. This technique was proposed in [Cra+18] being the first scheme whose domain of computation are rings of the form $\mathbb{Z}_M$ for an integer M. In particular, they put special attention to the case $M = 2^k$, and they claim that the generalization to an arbitrary M is done easily from the particular case. This work has relevance because it is the first proposal that achieves security against active adversaries in the dishonest majority case using rings instead of finite fields. This change in the algebraic structure comes with some challenges because the security of schemes based on finite fields like SPDZ [Dam+12] relies on the fact that every non-zero element has an inverse. On the contrary, this property does not hold for rings of the form $\mathbb{Z}_{2^k}$. So, the work of Cramer et al. proposes new techniques to achieve security despite this theoretical loss.

The main motivation behind the use of rings for MPC protocols is that the arithmetic modulo 2^k is closer to what happens in a standard CPU. This allows the protocol designers to take advantage of the CPU architectures to improve the performance of procedures like comparisons and bit-wise operations. In our particular case, in the arithmetic section of the edit distance algorithm, we require to perform a high number of comparisons to compute the minimum of a list of numbers. To compute comparisons, we need to compute

the most significant bit and to perform truncations of secret shared values. For those tasks, it is necessary to compute basic operations that can be performed easier in $\mathbb{Z}_{2^k}$ than in $\mathbb{Z}_p$ with p prime. For example, the reduction of a number modulo 2^k is easier in $\mathbb{Z}_{2^k}$ because we can take the least $k-1$ least significant bits. The reduction modulo p, for p prime, requires a bit more effort. Therefore, the presence of a high number of comparisons of arithmetic shares is the main motivation behind our choice of SPD$\mathbb{Z}_{2^k}$ as the main point of study in this thesis to compute the edit distance securely.

Notation. In this section, we will use the same notation as in [Cra+18]. We denote by $\mathbb{Z}_M$ the integers modulo M, where we take the representatives in the set $\{0, 1, \ldots, M-1\}$ as usual. Also, the congruence $x \equiv y \mod 2^k$ will be denoted as $x \equiv_k y$.

The baseline of the SPD$\mathbb{Z}_{2^k}$ scheme is the *unconditional secure message authentication codes* (MACs). In summary, these are secure message authentication codes that prevent a message to be forged even in presence of computationally powerful adversaries. To get a better idea about information-theoretic MACs, we refer to the reader to [KL14, Section 4.6], where Katz and Lindell bring a formal definition of this type of security and some basic constructions. In particular, the SPD$\mathbb{Z}_{2^k}$ protocol takes the idea of unconditional MACs from the SPDZ protocol [Dam+12], where the security is defined according to the following game: let x be a value, α be a random MAC key, and $m \stackrel{\text{def}}{=} \alpha \cdot x$ the MAC tag in some finite field $\mathbb{F}$. The adversary receives just the value x and it is asked to find errors e_m, e_x, e_α to compute $m' \stackrel{\text{def}}{=} m + e_m$, $\alpha' \stackrel{\text{def}}{=} \alpha + e_\alpha$ and $x' \stackrel{\text{def}}{=} x + e_x$. The adversary wins if $x \neq x'$ and $m' = \alpha' \cdot x'$. In the case of finite fields, the probability that the adversary wins is $1/|\mathbb{F}|$, and the security of SPDZ relies on this fact. However, in the SPD$\mathbb{Z}_{2^k}$, where the underlying algebraic structure is $\mathbb{Z}_{2^k}$, the adversary can find an attack to win with probability $1/2$. To show this fact with details, suppose that the adversary chooses $x' \stackrel{\text{def}}{=} x + 2^{k-1}$. Given that the adversary wants $m' = \alpha' \cdot x'$, we have that

$$\begin{aligned} m' &= \alpha' \cdot x' \\ &= (\alpha + e_\alpha)(x + e_x) \\ &= \alpha x + \alpha 2^{k-1} + e_\alpha x + e_\alpha 2^{k-1} \\ &= m + \alpha 2^{k-1} + e_\alpha x + e_\alpha 2^{k-1}. \end{aligned}$$

So, if the adversary chooses $e_m \stackrel{\text{def}}{=} e_\alpha x + e_\alpha 2^{k-1}$ (which are known values), then it holds that its probability of winning

$$\Pr[m' = \alpha' \cdot x'] = \frac{1}{2},$$

which can be obtained by considering cases for α odd and α even. This attack comes from the fact that in a ring of the form $\mathbb{Z}_{2^k}$, the even elements do not have an inverse.

In SPD$\mathbb{Z}_{2^k}$, for an element $x \in \mathbb{Z}_{2^k}$, the problem is solved by considering a MAC key α at random from $\mathbb{Z}_{2^s}$, where s is a security parameter, and the MAC tag is computed as $\alpha \cdot x$ in the ring $\mathbb{Z}_{2^{k+s}}$. The work of Cramer et al. consists in defining a secret-shared MAC scheme properly and using it to build an authenticated secret-sharing scheme over $\mathbb{Z}_{2^k}$.

The information-theoretic MAC scheme proposed by Cramer et.al. has two parameters. The parameter k is the one that determines the size of the ring where the computation occurs. This means that each party will provide elements in $\mathbb{Z}_{2^k}$ as secret inputs to the MPC protocol to compute certain function f. The second parameter is s, which is a security parameter.

The MAC scheme has a single global key α which is divided into n shares $(\alpha^i)_{i=1}^n$ such that

$$\alpha = \sum_{i=1}^{n} \alpha^i \mod 2^{k+s}.$$

Each share α^i belongs to $\mathbb{Z}_{2^s}$, and the party P_i holds the share α^i. On the other hand, for an authenticated value $x \in \mathbb{Z}_{2^k}$, each party holds a share $x^i \in \mathbb{Z}_{2^{k+s}}$, such that

$$x' \stackrel{\text{def}}{=} \sum_{i=1}^{n} x^i \mod 2^{k+s},$$

and $x \equiv_k x'$. Also, each party has an additive share $m^i \in \mathbb{Z}_{2^{k+s}}$ of the MAC tag $m \stackrel{\text{def}}{=} \alpha \cdot x' \mod 2^{k+s}$. Joining all the elements, the share of x can be defined as

$$[\![x]\!] \stackrel{\text{def}}{=} \left(x^i, m^i, \alpha^i\right)_{i=1}^n \in \left(\mathbb{Z}_{2^{k+s}} \times \mathbb{Z}_{2^{k+s}} \times \mathbb{Z}_{2^s}\right)^n,$$

such that

$$\sum_{i=1}^{n} m^i \equiv_{k+s} \left(\sum_{i=1}^{n} x^i\right) \cdot \left(\sum_{i=1}^{n} \alpha^i\right).$$

An important property is that this form of secret-shared MAC is linear. This means that if the parties have shares $[\![x]\!]$ and $[\![y]\!]$, they can compute $[\![x + y]\!]$ without interaction. The same holds for $[\![c \cdot x]\!]$ and $[\![x + c]\!]$, for any public value $c \in \mathbb{Z}_{2^k}$.

The first step in the construction of the MAC scheme is to define an ideal functionality that generates and distributes shares of a global MAC key α, and also authenticates a set of secret-shared values $[\![x_1]\!], \ldots, [\![x_t]\!]$. These tasks are defined in the functionality $\mathcal{F}_{\mathsf{MAC}}$ presented in [Cra+18, Figure 5]. Once this functionality is defined, they propose a procedure to open and check a secret-shared value $[\![x]\!]$ by doing the following steps (taken from [Cra+18, Procedure SingleCheck]):

1. Generate a random shared and authenticated value $[\![r]\!]$ using the functionality $\mathcal{F}_{\mathsf{MAC}}$, where $r \in \mathbb{Z}_{2^s}$.
2. Each party computes $[\![y]\!] \stackrel{\text{def}}{=} [\![x + 2^k r]\!]$ locally.
3. Each party broadcasts its own share y^i and reconstructs $y = \sum_{i=1}^{n} y^i \mod 2^{k+s}$.
4. Each party commits to the value $z^i = m^i - y \cdot \alpha^i \mod 2^{k+s}$, where m^i is the MAC share on y.
5. All parties open their commitments and check that $\sum_{i=1}^{n} z^i \equiv_{k+s} 0$.
6. If the check passes, then the procedure outputs $y \mod 2^k$.

The key step in this procedure is Step 2. Specifically, this step aims to keep the privacy in the case of x being obtained as a linear combination of other private inputs. For example, suppose that the value x is obtained as $x = w + z$, where w and z are private values. Then, opening $x' = \sum_{i=1}^{n} x^i$ and then checking the MAC tag on x' is not secure because opening x' can leak if there was an overflow performing the sum $w + z$ by checking its s most significant bits. So Step 2 is designed to hide the s most significant bits by generating a random s-bit value r. If we observe the value of y carefully, its s most significant bits will be random, and the k least significant bits will hold the value of x. This last fact can be confirmed in Step 6, because doing $y \mod 2^k$ will extract the k least significant bits of y, namely, the value of x.

One of the most significant results in the work of Cramer et al. is the proposal of a procedure for opening a batch of secret-shared values and checking their MACs at once. This procedure is known as *batch MAC checking*. The core of the solution to this problem is to check a single random linear combination of the MACs of the values. This approach has a lower communication complexity compared to opening and checking each element of the batch one by one. The details of this approach can be found in [Cra+18, Section 3.2], specifically, in the procedure BatchCheck.

Once the authentication mechanism is established, we can use it to define an MPC protocol to compute functions securely. The SPD$\mathbb{Z}_{2^k}$ protocol belongs to a family of protocols that has a structure called the *preprocessing model*. These types of protocols are divided into two phases: a *preprocessing phase* (also called *offline phase*) and an *online phase*. In the preprocessing phase, the protocol generates all of the material needed to perform the computation. This material is completely independent of the user inputs, therefore, this phase can be executed even before each party provides its own input to the computation of the function. The material obtained from this phase corresponds commonly to multiplication triples needed to evaluate the communication gates, among other randomness used in

the online phase. The online phase corresponds to the evaluation of the arithmetic circuit itself. There, the parties bring their inputs to the protocol, and the circuit is evaluated using the randomness generated in the preprocessing phase. In this separation, it commonly happens that the offline phase is more computationally expensive than the online phase. So, this separation allows us to execute the protocol with lower latency compared to making both phases together [Fre+15].

To compute the preprocessing phase, Cramer et al. propose a functionality $\mathcal{F}_{\mathsf{Prep}}$ that is in charge of generating the raw material for the online phase. It has two main commands. The Input command generates the randomness for the party P_i to provide its input. The command distributes an authenticated share $[\![r]\!]$, where $r \in \mathbb{Z}_{2^{k+s}}$ is a random element, and gives r to P_i. The Triple command is in charge of generating a *multiplication triple* to evaluate the multiplication gates in the arithmetic circuit. A multiplication triple is a triple of secret-shared numbers $([\![a]\!], [\![b]\!], [\![c]\!])$, such that a and b are random numbers in $\mathbb{Z}_{2^k}$, and $c = a \cdot b \mod 2^k$.

On the other hand, the online phase is in charge of distributing shares of the inputs from the parties, evaluating the gates of the arithmetic circuit, and outputting values that are secret-shared. To accomplish these tasks, Cramer et al. propose the functionality $\mathcal{F}_{\mathsf{Online}}$. This functionality is realized by the protocol Π_{Online} in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model. In the Protocol 2.2, we present the details of Π_{Online} as they are presented in [Cra+18]. In this protocol, the most obscure step is the Multiplication command which we will explain next intuitively. If we consider the values not secret-shared but in clear text, we have that $\epsilon \stackrel{\text{def}}{=} x - a$ and $\delta \stackrel{\text{def}}{=} x - b$, therefore, $x = \epsilon + a$ and $y = \delta + b$. With these values, we can compute

$$\begin{aligned} x \cdot y &= (\epsilon + a) \cdot (\delta + b) \\ &= \epsilon \cdot \delta + \epsilon \cdot b + \delta \cdot a + a \cdot b \\ &= c + \epsilon \cdot b + \delta \cdot a + \epsilon \cdot \delta. \end{aligned}$$

So, if we take this value for $x \cdot y$ and include the secret-share notation, we have that $[\![x \cdot y]\!] = [\![c + \epsilon \cdot b + \delta \cdot a + \epsilon \cdot \delta]\!]$. But given that the secret-sharing scheme is linear, we have that $[\![x \cdot y]\!] = [\![c]\!] + \epsilon \cdot [\![b]\!] + \delta \cdot [\![a]\!] + \epsilon \cdot \delta$. Notice that the values of ϵ and δ were opened, so they are public and can be treated as constants.

There are left to present two main building blocks of the protocol: how to authenticate an additively secret-shared value using the MAC scheme presented at the beginning of this section, and how to generate multiplication triples.

For the authentication protocol, Cramer et al. propose a protocol Π_{Auth} that realizes the MAC functionality in the $\mathcal{F}_{\mathsf{vOLE}}$-hybrid model. The functionality $\mathcal{F}_{\mathsf{vOLE}}$ is in charge of performing a *vector oblivious linear evaluation* between two parties. In summary, This

Protocol 2.2: Π_{Online}

The protocol has two parameters: k which is the word size in wich the operations are performed, and s which is the security parameter.

Initialize. The parties call the functionality $\mathcal{F}_{\mathsf{Prep}}$ as follows:

1. On input (Init) to get the MAC key shares $\alpha^j \in \mathbb{Z}_{2^s}$.
2. For each party P_i that will provide inputs to the computation, and for each input provided by P_i, it is called the functionality $\mathcal{F}_{\mathsf{Prep}}$ on input (Input, P_i) to obtain random sharings $[\![r]\!]$, where $r \in \mathbb{Z}_{2^{k+s}}$ and P_i learns r.
3. For each multiplication gate in the arithmetic circuit, the command (Triple) is called to get a multiplication triple $([\![a]\!], [\![b]\!], [\![c]\!])$.

Input. Suppose that P_i holds an input value to the functionality $x \in \mathbb{Z}_{2^k}$. To distribute additive shares x^i of x to each party, the parties execute the following instructions:

1. P_i broadcasts $\epsilon \stackrel{\text{def}}{=} x - r \mod 2^{k+s}$, where $[\![r]\!]$ is the next unused input mask generated in the Initialize step.
2. The parties compute $[\![x]\!] \stackrel{\text{def}}{=} [\![r]\!] + \epsilon$.

Add. To compute an addition gate whose inputs are two secret-shared values $[\![x]\!]$ and $[\![y]\!]$, the parties compute locally $[\![z]\!] \stackrel{\text{def}}{=} [\![x]\!] + [\![y]\!]$.

Multiply. To compute a multiplication gate with input values $[\![x]\!]$ and $[\![y]\!]$, the parties execute the following instructions:

1. The parties open $[\![x]\!] - [\![a]\!]$ as ϵ and $[\![y]\!] - [\![y]\!]$ as δ, using the Command Open in the BatchCheck procedure, where $([\![a]\!], [\![b]\!], [\![c]\!])$ is the next unused multiplication triple generated in the Initialize step.
2. The parties locally compute $[\![x \cdot y]\!] = [\![c]\!] + \epsilon \cdot [\![b]\!] + \delta \cdot [\![a]\!] + \epsilon \cdot \delta$.

Output. To output a secret-shared value $[\![y]\!]$ the parties execute the following steps:

1. Call the procedure BatchCheck to check the MACs that have been opened so far to compute the multiplication gates.
2. If the previous step does not abort, the parties open and check the MAC of $[\![y]\!]$ using the procedure SingleCheck.

functionality has three parameters: t, l, and r. In this functionality, two parties P_A and P_B are involved. The party P_A will input a value $\alpha \in \mathbb{Z}_{2^s}$. On the other hand, the party P_B will input a vector $\mathbf{x}$, such that the first t entries are in $\mathbb{Z}_{2^r}$ and the $(t+1)$-th entry is in $\mathbb{Z}_{2^l}$. The functionality will generate a random vector $\mathbf{b} \in \mathbb{Z}_{2^l}^{t+1}$, compute $\mathbf{a} \stackrel{\text{def}}{=} \mathbf{b} + \alpha \cdot \mathbf{x} \mod 2^l$, returns $\mathbf{a}$ to P_A, and return the random vector $\mathbf{b}$ to P_B. To perform the authentication, the protocol Π_{Auth} is used to authenticate a vector of values $\mathbf{x} \stackrel{\text{def}}{=} (x_1, \ldots, x_t)$ over $\mathbb{Z}_{2^l}$ where each party P_i has a share of each component denoted by $\mathbf{x}^i$. An instance $\mathcal{F}_{\mathsf{vOLE}}$ is used between each pair of parties (P_i, P_j), with $i \neq j$, to obtain the linear evaluation $\mathbf{a}^{i,j} \stackrel{\text{def}}{=} \mathbf{b}^{j,i} + \alpha^i \cdot \tilde{\mathbf{x}}^j$, such that:

- $\mathbf{a}^{i,j}$ and $\mathbf{b}^{j,i}$ are the outputs of the functionality $\mathcal{F}_{\mathsf{vOLE}}$ executed between parties P_i and P_j.
- α^i is the share of the MAC key α. This share is sampled by P_i at random from $\mathbb{Z}_{2^s}$.
- $\tilde{\mathbf{x}}^j \stackrel{\text{def}}{=} \left[\mathbf{x}^j \mid x^j_{t+1}\right] \in \mathbb{Z}_{2^r}^t \times \mathbb{Z}_{2^{l'}}$, where $l' \stackrel{\text{def}}{=} \max\{l, r+s, 2s\}$, and x^j_{t+1} is sampled at random from $\mathbb{Z}_{2^{l'}}$.

The outputs $\mathbf{a}^{i,j}$ and $\mathbf{b}^{j,i}$ are used to compute the share of the MAC tag, and the random value x^j_{t+1} is used to perform a consistency check of the generated MACs. For more details about the protocol Π_{Auth}, we refer to the reader to [Cra+18, Section 5].

For the multiplication triples, Crammer, et al. propose the protocol Π_{Triple} that realizes the functionality for multiplication triple generation in the $(\mathcal{F}_{\mathsf{ROT}}, \mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{MAC}})$-hybrid model. The functionality $\mathcal{F}_{\mathsf{ROT}}$ is in charge of computing a *random oblivious transfer* between two parties. On the other hand, the functionality $\mathcal{F}_{\mathsf{Rand}}$ generates and distributes random numbers among the parties. Also, the functionality $\mathcal{F}_{\mathsf{MAC}}$ is in charge of generating shares of the global MAC key and distributing the shares of the MAC tag. The protocol for the generation of triples has four main stages. The first stage is called Multiply. Here the protocol generates two secret shared-vectors $[\![\mathbf{a}]\!]$ and $[\![\mathbf{c}]\!]$, and a share $[\![b]\!]$, such that $\mathbf{c} = b \cdot \mathbf{a}$, where the product is done component-wise. In the second stage, they construct shares $[\![a]\!]$, $[\![b]\!]$, $[\![c]\!]$, $[\![\hat{a}]\!]$ and $[\![\hat{c}]\!]$ such that

$$c \equiv_k a \cdot b \quad \text{and} \quad \hat{c} \equiv_k \hat{a} \cdot b$$

In particular, the shares $[\![a]\!]$, $[\![c]\!]$, $[\![\hat{a}]\!]$, and $[\![\hat{c}]\!]$ are constructed as random linear combinations of the components of the vectors $\mathbf{a}$ and $\mathbf{c}$ respectively. The third stage is called Sacrifice, which is a technique that is commonly used to create multiplication triples under active adversaries. This technique uses (and sacrifices) the shares $[\![\hat{a}]\!]$ and $[\![\hat{c}]\!]$ to check the correctness of the triple $([\![a]\!], [\![b]\!], [\![c]\!])$. The fourth stage is the Output stage, in which it is generated a share $[\![r]\!]$, where $r \in \mathbb{Z}_{2^s}$ is a random element, and finally it returns the triple $([\![a]\!], [\![b]\!], [\![c + 2^k r]\!])$. This last stage masks the s most significant bits of c to avoid

leakage of information. For a more detailed exposition of the protocol Π_{Triples}, we refer to the reader to [Cra+18, Section 6].

2.4.2 Tinier

In our proposal for the edit distance protocol, we divide the algorithm into two parts: one that is done in the binary domain, i.e. $\mathbb{Z}_2$, and another one that is done in an arithmetic domain. In this section, we will expose the protocol Tinier, which is used to compute the binary computations of the first part of our proposal. This protocol was proposed in [Fre+15] as an improvement of the generation of multiplication triples in fields of characteristic two. In particular, Frederiksen et al. focus their attention on the preprocessing stage of the protocols TinyOT [Nie+12], MiniMAC [DZ13] and SPDZ [Dam+12], showing a way to generate multiplication triples using oblivious transfer extensions. In this section, we will show how Tinier generates the multiplication triples for the particular case of $\mathbb{Z}_2$, which are needed to evaluate the multiplication gates in the binary circuit of our proposal.

In this section, we suppose that the set of parties involved in the protocol is $\{P_1, \ldots, P_n\}$. Also, we set κ to be a security parameter and $\mathbb{F}$ to be a finite field. Similar to the case of SPD$\mathbb{Z}_{2^k}$, Tinier considers the additive secret-sharing approach. This means that, if we want to secret-share a value $x \in \mathbb{F}$, we randomly generate shares $x^{(1)}, \ldots, x^{(n)} \in \mathbb{F}$ such that $x = x^{(1)} + \cdots + x^{(n)}$, where the party P_i holds the value $x^{(i)}$. Also, they consider the authentication of secret-shared elements using information-theoretic MACs. In the particular case of Tinier, they use a fixed global key $\Delta \in \mathbb{F}_{2^M}$, for $M \geq \kappa$, which is additively secret-shared among the parties. In the general case, the authenticated secret-share of a value $x \in \mathbb{F}$, with $\mathbb{F} = \mathbb{F}_{2^u}$ and $u | M$, is represented as follows:

$$[\![x]\!] \stackrel{\text{def}}{=} \left(x^{(i)}, \mathbf{m}_x^{(i)}, \Delta^{(i)}\right)_{i=1}^{n}$$

where, $\mathbf{m}_x^{(i)}$ is the additive share of the MAC tag $\mathbf{m}_x \stackrel{\text{def}}{=} x \cdot \Delta$. Each share of the MAC tag $\mathbf{m}_{\mathbf{x}}^{(\mathbf{i})}$ is represented as an element in $\mathbb{F}_2^M$. Although the work of Frederiksen et al. considers a general case of elements in the field $\mathbb{F}_{2^u}$, for simplicity, we will focus on the case of $u = 1$, which is the binary case.

The techniques presented in the Tinier work rely heavily on *oblivious transfer* (OT) techniques. In an OT functionality, two parties P_S and P_R are involved. P_S inputs two messages $\mathbf{v}_0$ and $\mathbf{v}_1$ belonging to $\mathbb{F}_2^\kappa$ and P_R input a bit b. At the end of the execution, P_R only learns the value $\mathbf{v}_b$.

A variant of the OT functionality is where the messages sent by party P_S are correlated. That means that P_S sends messages $\mathbf{v}_0$ and $\mathbf{v}_1$ such that $\mathbf{v}_0 + \mathbf{v}_1 = \Delta$ for some constant Δ.

This functionality is called *correlated* OT. In the specific case of Tinier, it does not use an actively secure protocol to compute correlated OT, but a passive protocol that allows the adversary to input errors in the middle of the process which will introduce errors in the output of the functionality. This version is called *correlated OT with errors*. This functionality denoted $\mathcal{F}_{\mathsf{COTe}}^{\kappa,l}$ is of high importance for the exposition of the next protocols, so we show all the details in Functionality 2.2 taken from [Fre+15, Figure 2]. The security of this functionality is proven in [Nie07].

Functionality 2.2: $\mathcal{F}_{\mathsf{COTe}}^{\kappa,l}$

The Initialize step is independent from its inputs and it is called once. After the initialization, Extend can be called multiple times. The functionality has two parameters: l which is the number of correlated OTs to be performed and the bit length κ. In this functionality there are two parties involved: P_S and P_R, and there is an ideal adversary $\mathcal{A}$.

Initialize. Upon receiving $\Delta \in \mathbb{F}_2^\kappa$ from P_S, the functionality stores Δ.

Extend. On input $(P_R, (\mathbf{x}_1, \ldots, \mathbf{x}_l))$ from P_R, where $\mathbf{x}_i \in \mathbb{F}_2^\kappa$, the functionality performs the following steps:

1. It samples $\mathbf{t}_h \in \mathbb{F}_2^\kappa$, for $h = 1, 2, \ldots, l$, which will be sent to P_R. If P_R is corrupted, the functionality waits for $\mathcal{A}$ to input $\mathbf{t}_h$.
2. It computes the values $\mathbf{q}_h \stackrel{\text{def}}{=} \mathbf{t}_h + \mathbf{x}_h * \Delta$, for $h = 1, 2, \ldots, l$, and sends them to P_S. If P_S is corrupted, the functionality waits for $\mathcal{A}$ to input $\mathbf{q}_h$, and then it outputs to P_R the values of $\mathbf{t}_h$ consistent with the adversary inputs.

Before going into the triple generation, Frederiksen et al. propose a protocol to authenticate an additively shared value $x \in \mathbb{F}_2$. They first propose an ideal functionality $\mathcal{F}_{[\![\cdot]\!]}$ which specifies how the authentication works. In summary, the functionality has an initialization phase, where it samples and distributes shares for the MAC key Δ. Then, in the authentication process, the functionality receives the additive shares as inputs, reconstructs the secret values based on such shares, computes the MAC tags for each reconstructed value using the key Δ, and finally distributes shares of the MAC tag to each party. It is important to mention that the functionality allows the adversary to provide errors to the MAC tag to represent the adversary's capabilities. The details of the functionality $\mathcal{F}_{[\![\cdot]\!]}$ are presented in [Fre+15, Figure 5].

In [Fre+15, Figure 6], Frederiksen et al. propose a protocol $\Pi_{[\![\cdot]\!]}$ that realizes the functionality $\mathcal{F}_{[\![\cdot]\!]}$ in the $\mathcal{F}_{\mathsf{COTe}}^{\kappa,l}$-hybrid model. In this protocol, each pair of parties (P_i, P_j) with

$i \neq j$ initialize an instance of $\mathcal{F}_{\mathsf{COTe}}^{M,l}$ where P_j inputs a random share of the MAC key $\Delta^{(j)} \in \mathbb{F}_{2^M}$. In this initialization process, the consistency of the shares of the MAC key is checked. This consistency check relies on a sub-protocol called Π_{MACCheck} presented in [Fre+15, Figure 16]. The sub-protocol Π_{MACCheck} takes as inputs a list of public values a_h and a list of authenticated shares $[\![b_h]\!]$, for $h = 1, \ldots, l$, and it checks if $a_h = b_h$. In other words, Π_{MACCheck} checks if a set of public values are consistent with a set of secret-shared values when using a secret-shared MAC key Δ. To check if the shares of the MAC key are consistent, the parties generate and authenticate a set of dummy values using the MAC key Δ. The dummy values are opened, so the parties run Π_{MACCheck} giving as inputs the dummy values in both versions public and secret-shared (the latter ones are authenticated using the MAC key Δ). The Π_{MACCheck} will determine if the public dummy values match the authenticated secret-shared dummy values. If both lists match, then we can conclude that the MAC key Δ was initialized correctly except with probability $2^{-\kappa}$. It is important to mention that Π_{MACCheck} is also used in the multiplication triples protocol to check if the MAC generated to authenticate the triple is consistent. Now, to authenticate a set of values $x_1, \ldots, x_l \in \mathbb{F}_2$ that are already secret-shared among the parties, the party P_i inputs its shares $x_1^{(i)}, \ldots, x_l^{(i)}$ to the instance $\mathcal{F}_{\mathsf{COTe}}^{M,l}$ that was initialized with the party P_j using the command Extend of the corresponding instance. The output provided by each instance of the functionality $\mathcal{F}_{\mathsf{COTe}}^{M,l}$ is then combined to construct a share of the MAC tag for each value.

For the triple generation, the Tinier work first proposes a functionality $\mathcal{F}_{\mathsf{Triples}}$ that is in charge of generating and distributing shares of a multiplication triple. Instead of specifying the details of the functionality, we will jump directly to the protocol $\Pi_{\mathsf{BitTriples}}$ that generates and distributes multiplication triples in $\mathbb{F}_2$. In [Fre+15, Theorem 2], they state that the protocol $\Pi_{\mathsf{BitTriples}}$ realizes the functionality $\mathcal{F}_{\mathsf{Triples}}$ in the $\left(\mathcal{F}_{\mathsf{COTe}}^{\kappa,l}, \mathcal{F}_{[\![\cdot]\!]}\right)$-hybrid model in the presence of a static adversary corrupting up to $n-1$ parties.

It is important to mention that the protocol $\Pi_{\mathsf{BitTriples}}$ is secure under the random *oracle model*. In some situations, the assumptions made on the security of a hash function may be not enough to prove the security of a cryptographic construction that uses such function. Instead of giving up on the proof of security of the construction, it is common to use the random oracle model, which is an idealization of a hash function. In the random oracle model, we assume the existence of a public function $H : \{0,1\}^* \to \{0,1\}^l$ treated as a black box. This means that the parties make queries with input x to the black box (which we refer to as query the oracle) and it answers with a value $H(x)$. The parties do not know how the function H works internally. Also, the parties do not know which values are queried or when the oracle has been queried by other parties, which matches with a real-world situation where the parties locally evaluate a hash function. There are two properties of importance for a random oracle:

1. If a string x has not been queried to H, the value of $H(x)$ is uniform.

2. If x is queried to H, the same value is returned in the next queries of x to H.

Although the random oracle allows proving the security of certain constructions, it is not a silver bullet to prove the real security of the construction because, in the real world, we are instantiating the random oracle with a concrete hash function which is not a truly random function. For more details on proofs using random oracles, and the security effects of using random oracles in constructions when instantiated using a concrete real-world hash function, we refer to the reader to [KL14, Section 5.5]. In [BR93], the reader will find the original proposal of the random oracle model.

That said, we proceed to the detailed specification of the protocol $\Pi_{\mathsf{BitTriples}}$, as we shown in Protocol 2.3 which is taken from [Fre+15, Figure 15].

There are some details we want to fill in about Protocol 2.3 to make a clearer exposition. First of all, let us show why this process generates a random triple. Intuitively, this can be explained by summing up the values of $z_h^{(i)}$ generated in Step 3 of the Triple generation. If we do this, we obtain

$$\begin{aligned}
\sum_{i=1}^{n} z_h^{(i)} &\stackrel{\text{def}}{=} \sum_{i=1}^{n} \left(\sum_{j \neq i} n_h^{(i,j)} + x_h^{(i)} \cdot y_h^{(i)} + \sum_{j \neq i} v_{0,h}^{(i,j)} \right) \\
&= \sum_{i=1}^{n} \sum_{j \neq i} v_{0,h}^{(j,i)} + \sum_{i=1}^{n} \sum_{j \neq i} x_h^{(i)} \cdot y_h^{(j)} + \sum_{i=1}^{n} x_h^{(i)} \cdot y_h^{(i)} + \sum_{i=1}^{n} \sum_{j \neq i} v_{0,h}^{(i,j)} \\
&= \sum_{i=1}^{n} \sum_{j \neq i} x_h^{(i)} \cdot y_h^{(j)} + \sum_{i=1}^{n} x_h^{(i)} \cdot y_h^{(i)} \\
&= \left(\sum_{i=1}^{n} x_h^{(i)} \right) \cdot \left(\sum_{i=1}^{n} y_h^{(i)} \right).
\end{aligned}$$

So, it holds that $z_h = \sum_{i=1}^{n} z_h^{(i)} = \left(\sum_{i=1}^{n} x_h^{(i)} \right) \cdot \left(\sum_{i=1}^{n} y_h^{(i)} \right) = x_h \cdot y_h$, which explains better why the protocol outputs a multiplication triple. The second identity we want to expand is the second equality in Step 2(b)ii of Triple generation because, in our experience, this equality is not immediate to come up with. We want to show why $n_h^{(i,j)} = v_{0,h}^{(j,i)} + x_h^{(i)} \cdot y_h^{(j)}$. Notice that

$$\begin{aligned}
n_h^{(i,j)} &= w_h^{(i,j)} + x_h \cdot s_h^{(j,i)} \\
&= w_h^{(i,j)} + x_h^{(i)} \left(v_{0,h}^{(j,i)} + v_{1,h}^{(j,i)} + y_h^{(j)} \right) \\
&= w_h^{(i,j)} + x_h^{(i)} \cdot v_{0,h}^{(j,i)} + x_h^{(i)} \cdot v_{1,h}^{(j,i)} + x_h^{(i)} \cdot y_h^{(j)}.
\end{aligned}$$

So, it remains to show that $w_h^{(i,j)} + x_h^{(i)} \cdot v_{0,h}^{(j,i)} + x_h^{(i)} \cdot v_{1,h}^{(j,i)} = v_{0,h}^{(j,i)}$. To show that, remember

Protocol 2.3: $\Pi_{\mathsf{BitTriples}}$

The protcol will generate a set of random triples $(\llbracket x_h \rrbracket, \llbracket y_h \rrbracket, \llbracket z_h \rrbracket)$, for $h = 1, \ldots, l$, such that $z_h = x_h \cdot y_h$, and both x_h and y_h are generated at random. The protocol has a parameter l which is the number of triples that will be generated. Also, the protocol assumes the acess to a random oracle $H : \{0,1\}^* \to \{0,1\}$.

Initialize.

1. Each party P_i samples a random MAC key $\Delta^{(i)} \in \mathbb{F}_{2^\kappa}$, a second value $\tilde{\Delta}^{(i)} \in \mathbb{F}_{2^\kappa}$ and sets $\hat{\Delta}^{(i)} \stackrel{\text{def}}{=} \left(\tilde{\Delta}^{(i)} \,\|\, \Delta^{(i)}\right) \in \mathbb{F}_{2^{2\kappa}}$.

2. Each pair of parties (P_i, P_j), for $i \neq j$, initializes a new instance of $\mathcal{F}_{\mathsf{COTe}}^{2\kappa,l}$ and executes the command $\mathcal{F}_{\mathsf{COTe}}^{2\kappa,l}$.Initialize, where P_j inputs $\hat{\Delta}^{(j)}$. Also, they initialize an instance of $\mathcal{F}_{\llbracket\cdot\rrbracket}$, and execute the command $\mathcal{F}_{\llbracket\cdot\rrbracket}$.Init, where P_j inputs $\Delta^{(j)}$.

3. The parties check the consistency of the generated MAC key $\hat{\Delta} \stackrel{\text{def}}{=} \hat{\Delta}^{(1)} + \cdots + \hat{\Delta}^{(n)}$ by executing the following steps: each party P_i generates dummy random values $\hat{x}_1^{(i)}, \ldots, \hat{x}_\kappa^{(i)} \in \mathbb{F}_2$, and authenticate them using $\Pi_{\llbracket\cdot\rrbracket}$ to obtain $\llbracket \hat{x}_1 \rrbracket, \ldots, \llbracket \hat{x}_n \rrbracket$; then, each party P_i broadcasts only the additive shares $\hat{x}_1^{(i)}, \ldots, \hat{x}_\kappa^{(i)}$, and they compute $\overline{x}_h \stackrel{\text{def}}{=} \sum_{i=1}^n \hat{x}_h^{(i)}$; finally the parties call Π_{MACCheck} with inputs $\overline{x}_1, \ldots, \overline{x}_\kappa$, and the shares $\llbracket \hat{x}_1 \rrbracket, \ldots, \llbracket \hat{x}_n \rrbracket$; if Π_{MACCheck} fails, the protocol aborts.

COTe Extend. Each party P_i executes $\mathcal{F}_{\mathsf{COTe}}^{2\kappa,l}$.Extend with P_j, for all $j \neq i$, in the following way: P_i inputs $\mathbf{x}^{(i)} = \left(x_i^{(i)}, \ldots, x_i^{(i)}\right) \in \mathbb{F}_2^l$. From this execution, P_i receives $\left\{\hat{\mathbf{t}}_h^{(i,j)}\right\}_{h=1}^l$, and P_j receives $\hat{\mathbf{q}}_h^{(j,i)} = \hat{\mathbf{t}}_h^{(i,j)} + x_h^{(i)} * \hat{\Delta}^{(j)}$, for $h = 1, \ldots, l$.

Triple generation. Each party P_i divides the output of $\mathcal{F}_{\mathsf{COTe}}^{2\kappa,l}$ in two halves as follows: $\hat{\mathbf{t}}_h^{(i,j)} = \left(\tilde{\mathbf{t}}_h^{(i,j)} \,\|\, \mathbf{t}_h^{(i,j)}\right)$ and $\hat{\mathbf{q}}_h^{(i,j)} = \left(\tilde{\mathbf{q}}_h^{(i,j)} \,\|\, \mathbf{q}_h^{(i,j)}\right)$. Each party P_i uses only the first κ components of its shares, namely $\tilde{\mathbf{q}}_h^{(i,j)}$, $\tilde{\Delta}^{(i)}$ and $\tilde{\mathbf{t}}_h^{(i,j)}$ to do the following:

1. Each party P_i generates l random values $y_h^{(i)} \in \mathbb{F}_2$.

2. For each $i = 1, \ldots, n$ do:

   a) Using a random oracle $H : \{0,1\}^* \to \{0,1\}$, the party P_i breaks the correlation generated in the $\mathcal{F}_{\mathsf{COTe}}^{2\kappa,l}$ output to generate independent multiplication triples. On the one hand, for all $h = 1, \ldots, l$, P_i locally computes

$w_h^{(i,j)} \stackrel{\text{def}}{=} H\left(\tilde{\mathbf{t}}_h^{(i,j)}\right)$. On the other hand, for all $j \neq i$, and for all $h = 1, \ldots, l$, P_j computes $v_{0,h}^{(j,i)} \stackrel{\text{def}}{=} H\left(\tilde{\mathbf{q}}_h^{(j,i)}\right)$ and $v_{1,h}^{(j,i)} \stackrel{\text{def}}{=} H\left(\tilde{\mathbf{q}}_h^{(j,i)} + \tilde{\Delta}^{(j)}\right)$.

b) The parties generates the correlations related to y_h:

i. For all $j \neq i$, the party P_j construct a vector $\mathbf{s}^{(j,i)} \stackrel{\text{def}}{=} \left(\mathbf{s}_1^{(j,i)}, \ldots, \mathbf{s}_l^{(j,i)}\right)$ and send it to P_i, such that, for each $h = 1, \ldots, l$, $\mathbf{s}_h^{(j,i)} \stackrel{\text{def}}{=} v_{0,h}^{(j,i)} + v_{1,h}^{(j,i)} + y_h^{(j)}$.

ii. For all $j \neq i$, the party P_i computes $n_h^{(i,j)} \stackrel{\text{def}}{=} w_h^{(i,j)} + x_h^{(i)} \cdot s_h^{(j,i)} = v_{0,h}^{(j,i)} + x_h^{(i)} \cdot y_h^{(j)}$.

3. Each party P_i computes

$$z_h^{(i)} \stackrel{\text{def}}{=} \sum_{j \neq i} n_h^{(i,j)} + x_h^{(i)} \cdot y_h^{(i)} + \sum_{j \neq i} v_{0,h}^{(i,j)}.$$

Authentication.

1. For all $h = 1, \ldots, l$, the party P_i authenticate x_h by obtaining a share of the MAC tag. This is done by summing up the last κ components of the output of $\mathcal{F}_{\mathsf{COTe}}^{2\kappa,l}$. More specifically, the share of the MAC tag is defined by

$$m_h^{(i)} \stackrel{\text{def}}{=} x_h^{(i)} * \Delta^{(i)} + \sum_{j \neq i} \left(\mathbf{q}_h^{(i,j)} - \mathbf{t}_h^{(i,j)}\right).$$

This allows to the parties to obtain a authenticated share $[\![x_h]\!]$.

2. The parties use an instance of $\mathcal{F}_{[\![\cdot]\!]}$ to authenticate the shares of y_h and z_h to obtain authenticated shares $[\![y_h]\!]$ and $[\![z_h]\!]$, for $h = 1, \ldots, l$.

Check triples. In this step, the parties perform a process of sacrifice and combine called $\Pi_{\mathsf{CheckTriples}}$ [Fre+15, Figure 23] to check the correctness and privacy of the generated triples. This approach is a generalization of the bucket-based cut-and-choose technique presented by Larria et al. in [LOS14].

the following three equations:

$w_h^{(i,j)} \stackrel{\text{def}}{=} H\left(\tilde{\mathbf{t}}_h^{(i,j)}\right)$
$v_{0,h}^{(j,i)} \stackrel{\text{def}}{=} H\left(\tilde{\mathbf{q}}_h^{(j,i)}\right),$
$v_{1,h}^{(j,i)} \stackrel{\text{def}}{=} H\left(\tilde{\mathbf{q}}_h^{(j,i)} + \tilde{\Delta}^{(j)}\right).$

Also, from the protocol, we have that $\tilde{\mathbf{q}}_h^{(j,i)} \stackrel{\text{def}}{=} \tilde{\mathbf{t}}_h^{(i,j)} + x_h^{(i)} * \tilde{\Delta}^{(j)}$. So, let us take cases over $x_h^{(i)} \in \mathbb{F}_2$:

- If $x_h^{(i)} = 0$, then $\tilde{\mathbf{q}}_h^{(j,i)} = \tilde{\mathbf{t}}_h^{(i,j)}$. Therefore, $w_h^{(i,j)} \stackrel{\text{def}}{=} H\left(\tilde{\mathbf{t}}_h^{(i,j)}\right) = H\left(\tilde{\mathbf{q}}_h^{(j,i)}\right) = v_{0,h}^{(j,i)}$. And finally, it holds that $w_h^{(i,j)} + x_h^{(i)} \cdot v_{0,h}^{(j,i)} + x_h^{(i)} \cdot v_{1,h}^{(j,i)} = v_{0,h}^{(j,i)}$.

- If $x_h^{(i)} = 1$, then $\tilde{\mathbf{q}}_h^{(j,i)} = \tilde{\mathbf{t}}_h^{(i,j)} + \tilde{\Delta}^{(j)}$. Given the operation are in $\mathbb{F}_2$, adding up $\tilde{\Delta}^{(j)}$ in both sides, we obtain that $\tilde{\mathbf{q}}_h^{(j,i)} + \tilde{\Delta}^{(j)} = \tilde{\mathbf{t}}_h^{(i,j)}$. This means that $w_h^{(i,j)} \stackrel{\text{def}}{=} H\left(\tilde{\mathbf{t}}_h^{(i,j)}\right) = H\left(\tilde{\mathbf{q}}_h^{(j,i)} + \tilde{\Delta}^{(j)}\right) = v_{1,h}^{(j,i)}$. Finally, $w_h^{(i,j)} + x_h^{(i)} \cdot v_{0,h}^{(j,i)} + x_h^{(i)} \cdot v_{1,h}^{(j,i)} = v_{1,h}^{(j,i)} + v_{0,h}^{(j,i)} + v_{1,h}^{(j,i)} = v_{0,h}^{(j,i)}$.

Finally, we can see that the way Tinier defines the shares of a value is very similar to the SPD$\mathbb{Z}_{2^k}$ protocol. Therefore, we omit the description of the online phase for Tinier given that the authentication has the same spirit, both protocols rely on multiplication triples and, in both cases, the secret-sharing scheme is linear.

2.5 daBits and edaBits

In the previous sections, we discussed the main MPC protocols to compute both parts of the edit distance algorithm. On the one hand, we will use Tinier to compute functions in the binary domain, and on the other hand we will use SPD$\mathbb{Z}_{2^k}$ to compute functions in the arithmetic domain (specifically, in the ring $\mathbb{Z}_{2^k}$). However, we have not discussed two problems to fully compute the edit distance. First, we perform some computations in the binary domain using shares in $\mathbb{Z}_2$, and then, somehow, we need the result of this binary computation to perform an arithmetic computation in $\mathbb{Z}_{2^k}$. To do this we can not simply take the binary shares and use them in the SPD$\mathbb{Z}_{2^k}$ protocol because the algebraic structure is not the same. Instead, we need to find an efficient and secure mechanism that transforms these binary shares into arithmetic shares in such a way that the underlying value related to both secret-sharing methods is the same bit. The second problem is that the arithmetic section of the edit distance protocol relies heavily on arithmetic comparisons to compute the minimum of a list of numbers. This in particular is not an easy task in MPC protocols because it has a high communication complexity. Moreover, when we

use arithmetic domains, this task is even more difficult because it relies on the binary decomposition of the underlying secret value, which is expensive to obtain in such an arithmetic context. The problem at hand is to figure out what is the best way to perform these comparisons efficiently.

In this section, we will explain how to solve both problems stated above. In particular, we will show how we can use *daBits* to solve the first problem known as *domain conversion*, and *edaBits* to solve the second problem related to arithmetic comparisons.

Notation. In this section, we need to make a difference between values that are secret-shared in $\mathbb{Z}_2$ from values secret-shared in $\mathbb{Z}_M$, for some $M \in \mathbb{Z}$. For the first one, we will use the notation $[\![x]\!]_2$, and for the later, we will use the notation $[\![x]\!]_M$.

2.5.1 daBits

According to the treatment presented in [Esc+20], a double-shared authenticated bit, also known as *daBit*, is a random bit $b \in \{0, 1\}$ that is authenticated and secret-shared in both $\mathbb{Z}_2$ and $\mathbb{Z}_M$, for some fixed $M \in \mathbb{Z}$. Using the notation established at the beginning of the section, a daBit is a tuple $([\![b]\!]_2, [\![b]\!]_M)$, where b is a randomly chosen bit. Originally, the idea of daBits was formulated in [RW19] to make conversions between $\mathbb{Z}_p$ and $\mathbb{F}_{2^k}$. Then, the Zaphod protocol [Aly+19] shows an improvement over the daBit protocol of Rotaru and Wood, and they show how to use this approach to obtain shares of an element in $\mathbb{Z}_p$ whose bit decomposition is in $\mathbb{Z}_2$. Also, in [Esc+20], Escudero et al. propose a new approach to the daBit protocol for the particular case of conversions between $\mathbb{Z}_M$ and $\mathbb{Z}_2$ following the ideas of Zaphod, which will be the approach that we will present in this section. As we saw before, the change from the field of the form $\mathbb{Z}_p$ to rings of the form $\mathbb{Z}_M$ requires modifications to the protocols to meet the security requirements.

As we mentioned before, the protocol of daBits proposed in [Esc+20] is based on the protocol presented in Zaphod. Escudero et al. present two versions for the protocol. Both versions rely on an ideal functionality called the *arithmetic black box*, which is denoted by $\mathcal{F}_{\mathsf{ABB}}$. This functionality gives us the mechanisms to perform arithmetic computations between secret-shared values. These operations are the distribution of shares both in binary and arithmetic domains, linear combinations of shares, multiplications, random number generation, and the opening of secret-shared values.

In the first version of the daBits protocol, assuming that t is the number of corrupted parties, a group of $t + 1$ parties (without loss of generality, let us say that they are the first $t + 1$ of them) distributes a secret sharing of a random bit in both domains, namely $[\![b_i]\!]_2$ and $[\![b_i]\!]_M$, for $i = 1, \ldots, t + 1$. Then the parties perform an XOR of all the shares.

That is,

$$[\![b]\!]_M = \bigoplus_{i=1}^{t+1} [\![b_i]\!]_M,$$

and

$$[\![b]\!]_2 = \bigoplus_{i=1}^{t+1} [\![b_i]\!]_2.$$

The XOR in $\mathbb{Z}_2$ can be done locally because $a \oplus b = a + b \mod 2$. On the other hand, the arithmetic share can be computed knowing that $a \oplus b = a + b - 2ab$ when $a, b \in \{0, 1\}$. The later identity shows that one daBit is generated using t multiplications.

The previous version is for the general case where the arithmetic is done in $\mathbb{Z}_M$. However, the second version that we will discuss next is for the particular case where the arithmetic domain is $\mathbb{Z}_{2^k}$. In this case, if the parties have a bit that is secret-shared in the arithmetic domain, the parties can take the least significant bit of the corresponding shares, and then distribute a binary secret-share of this least significant bit. With such binary shares, they can XOR them locally to obtain a share of the same bit that was secret-shared in the arithmetic domain. The advantage here is that this process does not need multiplications, which reduces the communication complexity given that we do not need to generate multiplication triples. To express this property more formally, let b a secret-shared value between the parties and suppose that $\left\{b^{(i)}\right\}_{i=1}^{n}$ are the shares of b modulo 2^k, which means that $b = \sum_{i=1}^{n} b^{(i)} \mod 2^k$. Following the notation of the SPD$\mathbb{Z}_{2^k}$ protocol, we assume that each share is in $\mathbb{Z}_{2^{k+s}}$ for some security parameter s. The key to the improvement presented in the second version is the following equation:

$$\begin{aligned}
\bigoplus_{i=1}^{n} \left(b^{(i)} \mod 2\right) &= \sum_{i=1}^{n} \left(b^{(i)} \mod 2\right) \mod 2 \\
&= \left(\sum_{i=1}^{n} b^{(i)} \mod 2^k\right) \mod 2 \\
&= b \mod 2.
\end{aligned}$$

Let us expand on the details behind the second equality which is the less evident one. Specifically, we will show that

$$\sum_{i=1}^{n} \left(b^{(i)} \mod 2\right) \mod 2 = \left(\sum_{i=1}^{n} b^{(i)} \mod 2^k\right) \mod 2.$$

First, from a well-known result of congruence theory, we have that

$$\sum_{i=1}^{n} \left(b^{(i)} \mod 2\right) \mod 2 = \sum_{i=1}^{n} b^{(i)} \mod 2.$$

So, we need to show that

$$\sum_{i=1}^{n} b^{(i)} \mod 2 = \left(\sum_{i=1}^{n} b^{(i)} \mod 2^k\right) \mod 2.$$

Let us define

$$c \stackrel{\text{def}}{=} \left(\sum_{i=1}^{n} b^{(i)}\right) \mod 2^k.$$

This means that

$$2^k \left| \left(c - \sum_{i=1}^{n} b^{(i)}\right) \right.,$$

then

$$2 \left| \left(c - \sum_{i=1}^{n} b^{(i)}\right) \right..$$

This allows us to conclude that

$$c \equiv \sum_{i=1}^{n} b^{(i)} \mod 2.$$

This means that both c and $\sum_{i=1}^{n} b^{(i)}$ have the same parity. So,

$$\begin{aligned} c \mod 2 &= \sum_{i=1}^{n} b^{(i)} \mod 2 \\ &= \left[\sum_{i=1}^{n} b^{(i)} \mod 2\right] \mod 2. \end{aligned}$$

Finally, replacing the value of c, we obtain the desired equality:

$$\left[\sum_{i=1}^{n} b^{(i)} \mod 2^k\right] \mod 2 = \left[\sum_{i=1}^{n} b^{(i)} \mod 2\right] \mod 2.$$

Now that the key equation was presented, we show the protocol for the daBit generation in Protocol 2.4. The protocol generates a list of m daBits whose arithmetic domain is in $\mathbb{Z}_{2^k}$. Notice that the protocol relies on the $\mathcal{F}_{\mathsf{ABB}}$ functionality that represents the underlying protocols to manage both the arithmetic and binary computations. So, the protocol is general enough to work with Tinier and SPD$\mathbb{Z}_{2^k}$ as the ones in charge of the arithmetic and binary computations. There are two important things to point out here. First, in the Faulty daBit generation, the protocol creates $m + s$ supposed daBits. The first m ones will be the output of the protocol, and the last s ones are used to check the correctness of the generated daBits. The construction of these daBits may be faulty because, in Step 2,

the adversary may input values that are not consistent. For this reason, there is another section called Check, where the consistency of the bit pairs is checked. The check is similar to the Zaphod approach in the sense that we construct a random linear combination of the bits both in arithmetic and binary domains. Then, we open both linear combinations and verify if both values are the same, otherwise, the parties abort. It is important to see that due to the use of the protocol SPD$\mathbb{Z}_{2^k}$, it is guaranteed that the bits that are secret-shared in the arithmetic domain are the shares of bit values, so there is no need to verify this fact. As an additional note, this avoided verification is needed when the arithmetic computations are performed in $\mathbb{Z}_M$ and M is not a power of 2. Notice that Equation 2.5.1 ensures the correctness in both the Faulty daBit generation and the Check steps.

Domain conversion

As discussed at the beginning of this section, we need to transform binary secret-shared values in $\mathbb{Z}_2$ into shares in $\mathbb{Z}_{2^k}$, and to do this, we will use daBits. Protocol 2.5 shows how to use a daBit generated in a preprocessing phase to perform a domain conversion. This protocol is based on the protocol presented in [Dam+19, Figure 3] and adapted for the use of daBits.

2.5.2 edaBits

An *extended daBit*, also called *edaBit*, is an MPC primitive proposed in [Esc+20], which is a set of random bits $(r_{m-1}, \ldots, r_0)$ shared in a binary domain, along with the value $r = \sum_{i=0}^{m-1} r_i \cdot 2^i$ shared in an arithmetic domain. This primitive is considered as an extension of the daBits proposed in [RW19] in the sense that an edaBit can be constructed by generating m daBits and computing a linear combination of them. However, Escudero et al. state that using their approach, an edaBit can be generated much more efficiently than generating m daBits. As we will see later, this primitive allows us to compute comparisons in the arithmetic domain more efficiently given that the comparisons have a high communication complexity in such domains alone. In our case, the edit distance protocol relies heavily on these comparisons in the arithmetic section as a part of the computation of the minimum of a list of numbers, so we think that the use of edaBits will improve the performance of such computations. In this work, we will show a simplification of the protocol for edaBit generation to the particular case of $\mathbb{Z}_{2^k}$ which is the main focus of this work.

Let us explain the case of passive security first. To generate edaBits, each party P_i gen-

Protocol 2.4: Π_{daBit}

The protocol generates a set of m daBits $(\llbracket b_i \rrbracket_2, \llbracket b_i \rrbracket_{2^k})_{i=1}^m$, using a security parameter s. The protocol has access to the functionality $\mathcal{F}_{\mathsf{ABB}}$.

Faulty daBit generation. For $i = 1, \cdots, m+s$, do:

1. The parties generate a random bit $\llbracket b_i \rrbracket_{2^k}$ as in the SPD$\mathbb{Z}_{2^k}$ protocol [Cra+18].
2. Let $b_i^{(j)}$ the share of b_i belonging to the party P_j. Then each party P_j inputs $b_i^{(j)} \mod 2$ to the functionality $\mathcal{F}_{\mathsf{ABB}}$ as a binary share, in such a way that every party has access to the share $\left\llbracket b_i^{(j)} \mod 2 \right\rrbracket_2$.
3. The parties locally compute $\llbracket b_i \rrbracket_2 = \bigoplus_{j=1}^n \left\llbracket b_i^{(j)} \mod 2 \right\rrbracket_2$.

Check.

1. The parties execute the following steps s times:
   a) Generate m fresh public random bits r_i.
   b) Compute the linear combination $\llbracket \hat{r} \rrbracket_2 \stackrel{\text{def}}{=} \bigoplus_{k=1}^{m+s} r_i \cdot \llbracket b_i \rrbracket_2$ and open it.
   c) Compute $\llbracket r \rrbracket_{2^k} \stackrel{\text{def}}{=} \sum_{i=1}^{m+s} r_i \cdot \llbracket b_i \rrbracket_{2^k}$
   d) Call $r' \stackrel{\text{def}}{=} \text{Open}\left(\llbracket r \cdot 2^{k-1} \rrbracket_{2^k}\right)$ and compute
   $$\frac{r'}{2^{k-1}} = \frac{r \cdot 2^{k-1} \mod 2^k}{2^{k-1}} = r \mod 2$$
   e) If $\hat{r}$ and $(r \mod 2)$ do not match, the parties abort.
2. Discard $(\llbracket b_i \rrbracket_{2^k}, \llbracket b_i \rrbracket_2)_{i=m+1}^{m+s}$ and return $(\llbracket b_i \rrbracket_{2^k}, \llbracket b_i \rrbracket_2)_{i=1}^{m}$ as checked daBits.

Protocol 2.5: Π_{B2A}

The protocol takes as input a binary share $\llbracket x \rrbracket_2$, where $x \in \{0, 1\}$, and outputs an arithmetic share $\llbracket x \rrbracket_{2^k}$.

1. The parties take a fresh daBit $(\llbracket b \rrbracket_2, \llbracket b \rrbracket_{2^k})$.
2. The parties call $c \stackrel{\text{def}}{=} \text{Open}\left(\llbracket x \rrbracket_2 + \llbracket b \rrbracket_2\right)$
3. Output $\llbracket x \rrbracket_{2^k} \stackrel{\text{def}}{=} c + \llbracket b \rrbracket_{2^k} - 2 \cdot c \cdot \llbracket r \rrbracket_{2^k}$

erates bits $r_{i,0}, \ldots, r_{i,m-1}$ and computes $r_i = \sum_{j=0}^{m-1} r_{i,j} \cdot 2^j$. Then, the party P_i distributes binary shares of the bits so that each party obtains $[\![r_{i,0}]\!]_2, \ldots, [\![r_{i,m-1}]\!]_2$. Also, the party P_i distributes arithmetic shares of the value r_i, so that each party obtains $[\![r_i]\!]_{2^k}$. Given that the party P_i knows the edaBit generated by itself, we need to combine all the edaBits generated by all the parties in such a way that none of the parties know the underlying secret bits. The first kind of edaBits generated by each party are called *private* edaBits. When the private edaBits are combined, the resulting edaBit is called *global* edaBit. The combination of the private edaBits to obtain the global edaBit is performed as follows:

1. Each party compute

$$[\![r']\!]_{2^k} \stackrel{\text{def}}{=} \sum_{i=1}^{n} [\![r_i]\!]_{2^k}.$$

2. The binary shares can be computed using a binary adder with n inputs, which means that there is a procedure that takes $\left([\![r_{i,0}]\!]_2, \ldots, [\![r_{i,m-1}]\!]_2\right)_{i=1}^{n}$, and computes the binary addition to obtain the result in its bit representation $\left([\![r_0]\!]_2, \ldots, [\![r_{m+\log_2(n)-1}]\!]_2\right)$.

However, there are some subtle details to take into account in this process. Notice that it is possible that the sum of the r_i may overflow mod 2^k, but the binary adder will compute the binary sum without performing the reduction mod 2^k, so we need to make a correction to meet both values. Another complication that may arise is that the obtained edaBit should be an m-bit integer and the sum of the n private edaBits may have more than m bits, so again, we need to correct the computation to obtain an edaBit that meets the conditions. Fortunately, when we are dealing with arithmetic in $\mathbb{Z}_{2^k}$ (as in our case of application) both concerns can be solved easily. For such a case, there is no need to make a correction to the reduction modulo 2^k, because if we take the bits $r_{k-1}, \ldots, r_0$, they are already the bit representation of $r' \mod 2^k$. However, it could happen that after cropping the bits beyond the k-th position, there are more than m bits. The solution here is to crop the excess bits again and then correct the arithmetic share of r' by subtracting the value induced by the excess bits to match with the value induced by the remaining m bits. To make such subtraction, we need to use daBits and the protocol Π_{B2A} to transform the excess bits to the binary domain to be able to compute the subtraction with r'. As an additional note, for the case where the arithmetic is done in Z_M and M is not a power of two, it is needed to convert all the bits beyond the m-th one using daBits and then perform the subtraction to accomplish that both the length of the edaBit is m and the correction of the overflow modulo M. This suggests that the generation of edaBits for the case of $\mathbb{Z}_{2^k}$ is more efficient because the bits beyond the k-th one are discarded, so we do not need to generate daBits to convert them.

In Protocol 2.6, we show the protocol Π_{edaBits} to generate global edaBits with the simplifi-

cations for the case of arithmetic modulo 2^k. The protocol is taken from [Esc+20, Figure 3]. This protocol has access to an ideal functionality $\mathcal{F}_{\text{edaBitsPriv}}$ to compute the private edaBits for each party. Later, we will deal with a protocol to realize $\mathcal{F}_{\text{edaBitsPriv}}$. Also, the protocol has access to the $\mathcal{F}_{\text{ABB}}$ functionality to perform arithmetic computation in both $\mathbb{Z}_2$ and $\mathbb{Z}_{2^k}$ as well as domain conversion. Additionally, the protocol has a subroutine called nBitADD which is a function on n inputs that executes the binary adder protocol.

Protocol 2.6: Π_{edaBits}

Let $m \leq k$ the number of bits in the edaBits. The protocol has access to the ideal functionality $\mathcal{F}_{\text{edaBitsPriv}}$ to compute and distribute private edaBits, that is, edaBits whose underlying secret values are known to one party. Also, the protocol has access to the functionality $\mathcal{F}_{\text{ABB}}$. At the end of the protocol, the parties obtain $[\![r]\!]_{2^k}$ and $([\![r_0]\!]_2, [\![r_1]\!]_2, \ldots, [\![r_{m-1}]\!]_2)$, such that, $r_i \in \mathbb{Z}_2$ for all $i \in \mathbb{J}_{m-1}$, $r = \sum_{i=0}^{m-1} r_i \cdot 2^i$, and the bits are uniform to the adversary.

1. The parties call the functionality $\mathcal{F}_{\text{edaBitsPriv}}$ to get random shares $[\![r_i]\!]_{2^k}$ and $\left([\![r_{i,0}]\!]_2, [\![r_{i,1}]\!]_2, \ldots [\![r_{i,m-1}]\!]_2\right)$, for $i \in \mathbb{J}_n$. The party P_i additionally learns the values of $r_{i,0}, \ldots, r_{i,m-1}$, as well as $r_i = \sum_{j=0}^{m-1} r_{i,j} \cdot 2^j$.
2. The parties uses the functionality $\mathcal{F}_{\text{ABB}}$ to compute $[\![r']\!]_{2^k} \stackrel{\text{def}}{=} \sum_{i=1}^{n} [\![r_i]\!]_{2^k}$.
3. The parties call nBitADD $\left(\left([\![r_{i,0}]\!]_2, \ldots, [\![r_{i,m-1}]\!]_2\right)_{i=1}^{n}\right)$ to obtain $m + \log_2(n)$ bits $\left([\![b_0]\!]_2, \ldots, [\![b_{m+\log_2(n)-1}]\!]_2\right)$.
4. The parties use $\mathcal{F}_{\text{ABB}}$ to convert $[\![b_j]\!]_2$ into $[\![b_j]\!]_{2^k}$, for $m \leq j < k$.
5. The parties use $\mathcal{F}_{\text{ABB}}$ to compute $[\![r]\!]_{2^k} \stackrel{\text{def}}{=} [\![r']\!]_{2^k} - \sum_{j=m}^{k-1} [\![b_j]\!]_{2^k} \cdot 2^j$.
6. Output $[\![r]\!]_{2^k}$ and $([\![r_0]\!]_2, \ldots, [\![r_m]\!]_2)$.

Now, the task we have left is to realize the functionality $\mathcal{F}_{\text{edaBitPriv}}$. In the presence of a passive adversary, the functionality is straightforward to realize: the party P_i generates m random bits, computes the arithmetic share induced by those bits, and inputs them into the $\mathcal{F}_{\text{ABB}}$ functionality. However, in the case of an active adversary, the party creating the private edaBits could be corrupted and it may input inconsistent values to the $\mathcal{F}_{\text{ABB}}$ functionality. Namely, the party may generate random bits, but it may input an arithmetic share whose bit representation does not match the generated bits. To prevent this inconsistency, the parties engage in a *cut-and-chosse* procedure to check if the generated private edaBit is correct. In the cut-and-choose procedure, a subset of edaBits is opened and checked for consistency. The remaining edaBits are placed into buckets at random.

Then, for each bucket, the first edaBit of the bucket is checked against the rest of the edaBits in the same bucket by adding both edaBits in both the arithmetic and binary domain, opening the sum in both worlds and checking if both the binary and the arithmetic values coincide. If the checks pass, then the procedure returns the first edaBit of each bucket as a checked edaBit. Escudero et al. show that if all the checks pass, the returned edaBits will be correct with a high probability. For a formal proof of security of the cut-and-choose procedure, we refer to the reader to [Esc+20, Section 4].

There are some considerations to revisit here. First, the cut-and-choose technique is widely used as a checking method in the generation of multiplication triples. However, the use of this technique for the edaBit generation is expensive given that we need to add the edaBits in both worlds, but for the binary world, we need binary multiplication triples to evaluate a circuit for binary addition. Unfortunately, the process of generating such multiplication triples needs again to execute a cut-and-chose procedure to check the consistency of the triples. To solve this issue, Escudero et al. relax the consistency in the binary triples in two ways: (1) given that the multiplication triples are used in the addition process of edaBits whose underlying secret values are known by P_i, they allow P_i to input the triples itself into the functionality so that P_i also knows the underlying values of the multiplication triples; and (2), even if P_i is corrupt, they allow P_i to input incorrect multiplication triples, so the verification step can be omitted; however, the triples generated by P_i need to be authenticated (for example, using information theoretic MACs as we have seen in previous sections) to prevent that the introduced errors may be changed afterward. These two relaxations allow a faster triple generation compared to a traditional approach.

In Protocol 2.7, we show the specification of $\Pi_{\mathsf{edaBitPriv}}$ that realizes the functionality $\mathcal{F}_{\mathsf{edaBitsPriv}}$ for the specific case of $\mathbb{Z}_{2^k}$. This specification was taken from [Esc+20, Figure 4]. It is important to mention that the protocol uses the ideal functionality $\mathcal{F}_{\mathsf{ABB}}$ with two additional commands. The first command is Input triples where a party P_i can input a triple of bits $(a, b, c) \in \{0, 1\}^3$ into the functionality. In the real world, this means that the party P_i distributes binary shares of $[\![a]\!]_2$, $[\![b]\!]_2$, and $[\![c]\!]_2$ among the other parties. The second command is Faulty Multiplication, which takes a binary triple (a, b, c) previously inputted to the functionality and two bits x and y inputted to the functionality and outputs $z = x \cdot y \oplus (c \oplus x \cdot y)$. In the real world, a protocol that realizes this command will take a triple $([\![a]\!]_2, [\![b]\!]_2, [\![c]\!]_2)$ and two binary shares $[\![x]\!]_2$ and $[\![y]\!]_2$, and will output $[\![x \cdot y \oplus (c \oplus x \cdot y)]\!]_2$. These commands can be realized using share distribution protocols and Beaver's multiplication techniques which are very well studied.

Protocol 2.7: $\Pi_{\mathsf{edaBitsPriv}}$

The protocol generates N shared edaBits $\left\{\left(\llbracket r_j \rrbracket_{2^k}, \left(\llbracket r_{j,0} \rrbracket_2, \llbracket r_{j,1} \rrbracket_2, \ldots, \llbracket r_{j,m-1} \rrbracket_2\right)\right)\right\}_{j=1}^{N}$ of length m, such that the underlying secret values are known to the party P_i. The protocol has access to the $\mathcal{F}_{\mathsf{ABB}}$ functionality. The protocol takes inputs C and C' that are the number of edaBits and binary triples respectively that will be opened for checking during the cut-and-choose procedure. Also, the protocol takes input B which is the size of each bucket.

1. P_i samples $r_{j,0}, r_{j,1}, \ldots, r_{j,m-1} \in \mathbb{Z}_2$, for $j = 1, \ldots, NB + C$, and inputs them into the $\mathcal{F}_{\mathsf{ABB}}$ functionality in the binary domain.
2. P_i computes $r_j \overset{\text{def}}{=} \sum_{i=0}^{m-1} r_{j,i} \cdot 2^i \in \mathbb{Z}_{2^k}$ and input it into the functionality $\mathcal{F}_{\mathsf{ABB}}$ in the arithmetic domain.
3. P_i samples $(N(B-1) + C')\,m$ random bit triples and input these to the $\mathcal{F}_{\mathsf{ABB}}$ functionality using the command Input triples.
4. The parties run the CutNChoose procedure to check the consistency of the edaBits. If the check passes, the parties obtain N edaBits, otherwise, the parties abort.

Arithmetic comparison

We have seen that the algorithm to compute the edit distance requires the computation of the minimum of a list of integers. As in the basic implementation of the edit distance algorithm, the proposed protocol that we will show in the next chapters relies on the computation of such a function. But going even deeper, the core of the computation of this minimum function is the comparison between two integers. In the case of MPC, we need a protocol to compute if an integer is less than another one. However, we need to take care of some subtle considerations. In our problem, as in a wide variety of real-world problems, we need to compare two integers, but as we have seen, the arithmetic computations in MPC are performed over rings of the form $\mathbb{Z}_{2^k}$. Therefore, we need to represent the datatypes of the problem into $\mathbb{Z}_{2^k}$. Specifically, consider $\alpha \in \left[-2^{l-1}, 2^{l-1}\right) \subseteq \mathbb{Z}$, where $l \leq k$. We can represent α in $\mathbb{Z}_{2^k}$ by doing $a \stackrel{\text{def}}{=} \alpha \mod 2^k$. We will write that $a \stackrel{\text{def}}{=} \text{Rep}(\alpha)$ to say that $a \in \mathbb{Z}_{2^k}$ is the representation of the signed integer α.

With the signed integer representation, let us state the comparison problem precisely: let $\alpha, \beta \in \left[-2^{k-2}, 2^{k-2}\right)$, $a \stackrel{\text{def}}{=} \text{Rep}(\alpha)$ and $b \stackrel{\text{def}}{=} \text{Rep}(\beta)$; given $[\![a]\!]_{2^k}$ and $[\![b]\!]_{2^k}$, we need to compute $\left[\!\left[\alpha \stackrel{?}{<} \beta\right]\!\right]_{2^k}$, where $\left(\alpha \stackrel{?}{<} \beta\right) = 1$ if $\alpha < \beta$, and $\left(\alpha \stackrel{?}{<} \beta\right) = 0$ otherwise. In this case, $\alpha - \beta \in \left[-2^{k-1}, 2^{k-1}\right)$, so $a - b = \text{Rep}(\alpha - \beta)$. So, to check if $\alpha < \beta$, we can check if $\alpha - \beta < 0$, which reduces to calculate the most significant bit (MSB) of $\alpha - \beta$. If the MSB is 1, this means that $\alpha < \beta$, otherwise, it means that $\alpha \geq \beta$. Some approaches to extract the most significant bit can be found in [Dam+19]. However, Escudero et al. take another path to reach this goal using edaBits. Following [Esc+20], to extract the most significant bit, we use the fact that

$$\text{MSB}(\alpha) = -\left\lfloor \frac{\alpha}{2^{k-1}} \right\rfloor \mod 2^k.$$

So, to solve this problem, we need a method for *truncation* where the underlying MPC computation domain is $\mathbb{Z}_{2^k}$. Specifically, let us state the problem of truncation in a more general and precise way: let $\alpha \in \left[-2^{l-1}, 2^{l-1}\right)$ be such that $a \stackrel{\text{def}}{=} \text{Rep}(\alpha)$. A truncation protocol will compute a share $[\![y]\!]_{2^k}$ taking $[\![a]\!]_{2^k}$ as an input such that $y = \text{Rep}\left(\left\lfloor \frac{\alpha}{2^m} \right\rfloor\right)$, for some $m < l$. But in the case of $\mathbb{Z}_{2^k}$, we need to again state the truncation in terms of another procedure called *logical right shift*, that we will denote by LogShift. Let α be defined as above, let $a \stackrel{\text{def}}{=} \text{Rep}(\alpha)$. Given that $\alpha \in \left[-2^{l-1}, 2^{l-1}\right)$, we can assume that its representation a is an l-bit number, which means that $a \in \left[0, 2^l\right)$. So, the representation of a is as follows:

$$\left(0, 0, \ldots, 0, a_{l-1}, \ldots, a_0\right).$$

After calling $\mathsf{LogShift}_m([\![a]\!]_{2^k})$, we obtain shares of

$$\left(\underbrace{0,0,0,\ldots,0}_{k-l+m \text{ bits}}, \underbrace{a_{l-1},\ldots,a_m}_{l-m \text{ bits}}\right).$$

Once we obtain a protocol for the logical shift, we can use it to compute the truncation according to the following equation:

$$\left\lfloor \frac{a}{2^m} \right\rfloor \equiv \mathsf{LogShift}_m\left(a + 2^{l-1}\right) - 2^{l-m-1} \mod 2^k.$$

Therefore, in the end, we only need a protocol to compute the logical shift. In Protocol 2.8, we can find a complete specification of the protocol Π_{LogShift} to perform this task which is taken from [Esc+20, Figure 9]. There, LT is a binary circuit that takes two parameters in binary representation and outputs the bit 1 if the first parameter is less than the second one, or output 0 otherwise. We will show intuitively how the protocol accomplishes the task at hand. The first difficulty in the design of protocols over rings of the form $\mathbb{Z}_{2^k}$ is that we can not always divide by a power of two mod 2^k as we do in protocols based on fields. The strategy to avoid the use of inverses of powers of two is to perform operations that shift the bits to the right or to the left of the elements involved in the computation. This technique is very common in works that use $\mathbb{Z}_{2^k}$ as an underlying algebraic domain. We will show how these shifts work for Step 1 and the analysis for the rest of the protocol is similar.

To give an intuitive explanation of Π_{LogShift}, let us start by fixing the binary representation of the quantities involved there. The goal in Step 1 is to compute $a \mod 2^m$. Remember that a, the input of the protocol, is an l-bit number, so let the binary representation of a be

$$a = \left(0,\ldots,0,a_{l-1},a_{l-2},\ldots \underbrace{a_{m-1},\ldots,a_0}_{a \mod 2^m}\right).$$

Here, notice that the last m bits of a are the binary representation of $a \mod 2^m$. Now, let the representation of the edaBit r be

$$r = (0,\ldots,0,\ldots,\underbrace{r_{m-1},\ldots,r_0}_{l \text{ bits}}).$$

$$\underbrace{}_{k-l \text{ bits}}$$

When we compute $a + r$, we obtain a representation with the following structure:

$$a + r = \left(\underbrace{\star, \ldots, \star, \star, z}_{k-m \text{ bits}}, \overbrace{c_{m-1}, \ldots, c_0}^{(a+r) \mod 2^m} \right). \tag{2-2}$$

In this sum, the bit positions denoted by stars ($\star$) are values that we are not interested in. Note that when we add the least m significant bits of both a and r, we obtain m-bits plus a possible carry bit that we denote as z. Now, in Step 1a, the parties compute $2^{k-m}(a+r)$. This operation makes a bit-wise left shift in the representation of $a + r$ by $k - m$ bits, obtaining the representation

$$c = 2^{k-m}(a + r) \mod 2^k = (c_{m-1}, \ldots, c_0, 0, \ldots, 0).$$

This is precisely the representation of c. There, we preserve the annotation of the set of bits that are the binary expansion of the value of $(a+r) \mod 2^m$. Then, when we perform the operation $c/2^{k-m}$ as in Step 1d, we are shifting the bits of c to the right by $k-m$ bits obtain the following representation:

$$\frac{c}{2^{k-m}} = \left(0, \ldots, 0, \overbrace{c_{m-1}, \ldots, c_0}^{(a+r) \mod 2^m}\right).$$

The key is to notice that

$$\frac{c}{2^{k-m}} = (a + r) \mod 2^m$$

as we can see in the annotated binary representations. But

$$\begin{aligned} \frac{c}{2^{k-m}} &= (a + r) \mod 2^m \\ &= [(a \mod 2^m) + (r \mod 2^m)] \mod 2^m \\ &= [(a \mod 2^m) + r] \mod 2^m. \end{aligned}$$

The modular reduction in r can be ignored because r is a m-bit number. Here is where the bit z in Equation (2-2) takes place. Notice that adding $a \mod 2^m$ and r can cause an overflow bit denoted by z, but reducing this sum modulo 2^k can delete this value. So, to obtain an expression of $a \mod 2^m$, we need to correct this overflow arithmetically. To do

this, we compute if there is an overflow by computing the LT procedure to obtain the bit v as in Step 1b. By doing this, we obtain that

$$\frac{c}{2^{k-m}} = (a \mod 2^m) + r - 2^m \cdot v.$$

Therefore, we obtain the desired expression shown in Step 1d:

$$(a \mod 2^m) = 2^m \cdot v - r + \frac{c}{2^{k-m}}$$

As we mentioned before, the correctness of Step 2 can be explained in a similar way as before.

Protocol 2.8: Π_{LogShift}

The protocol takes as inputs a share $[\![a]\!]_{2^k}$, where $a \in [0, 2^l)$, and a integer $m < l$ wich is the number of bits to shift. The protocol has access to the $\mathcal{F}_{\mathsf{ABB}}$ functionality. Also, the protocol has access to a pair of edaBits:

- An edaBit of length m where the arithmetic part is $[\![r]\!]_{2^k}$ and his binary representation is $([\![r_{m-1}]\!]_2, \ldots, [\![r_0]\!]_2)$.
- An edaBit of length $l - m$ where the arithmetic part is $[\![r']\!]_{2^k}$ and his binary representation is $\left([\![r'_{l-m-1}]\!]_2, \ldots, [\![r'_0]\!]_2\right)$.

The protocol return $[\![y]\!]_{2^k}$, where $y = \mathsf{LogShift}_m(a)$. In the description of the protocol, if we have a value a r-bit value $x \in \mathbb{Z}_{2^k}$, we denote its bit decomposition by $(x_i)_{i=0}^{r-1}$.

1. The parties compute shares of $a \mod 2^m$ as follows:
   a) The parties call $c \stackrel{\text{def}}{=} \mathsf{Open}\left(2^{k-m} \cdot ([\![a]\!]_{2^k} + [\![r]\!]_{2^k})\right)$.
   b) The parties compute $[\![v]\!]_2 \stackrel{\text{def}}{=} \mathsf{LT}\left((c_i)_{i=k-m+1}^{k}, (r_i)_{i=0}^{m-1}\right)$
   c) Using the functionality $\mathcal{F}_{\mathsf{ABB}}$, the parties convert $[\![v]\!]_2$ into an arithmetic share $[\![v]\!]_{2^k}$.
   d) The parties compute
   $$[\![a \mod 2^m]\!]_{2^k} = 2^m \cdot [\![v]\!]_{2^k} - [\![r]\!]_{2^k} + \frac{c}{2^{k-m}}.$$
2. The parties compute the truncation:
   a) The parties compute $[\![b]\!]_{2^k} \stackrel{\text{def}}{=} [\![a]\!]_{2^k} - [\![a \mod 2]\!]_{2^k}$.
   b) The parties call $d \stackrel{\text{def}}{=} \mathsf{Open}\left(2^{k-l} \cdot ([\![b]\!]_{2^k} + 2^m \cdot [\![r']\!]_{2^k})\right)$.
   c) The parties compute $[\![u]\!]_2 \stackrel{\text{def}}{=} \mathsf{LT}\left((d_i)_{i=k-l+m}^{k-1}, (r'_i)_{i=0}^{l-m-1}\right)$.
   d) Using the functionality $\mathcal{F}_{\mathsf{ABB}}$ the parties convert $[\![u]\!]_2$ into $[\![u]\!]_{2^k}$.
   e) The parties compute
   $$[\![y]\!]_{2^k} = 2^{l-m} \cdot [\![u]\!]_{2^k} - [\![r']\!]_{2^k} + \frac{d}{2^{k-l+m}}.$$

3 A solution to the edit distance problem using secret-sharing

In this chapter, we will show an efficient strategy to compute the edit distance between two chains using multi-party computation protocols. In particular, we will use protocols based on secret-sharing schemes. Although the strategy presented here works for any secret-sharing scheme, we will focus on the design of a strategy that fits better with schemes whose computation domain is $\mathbb{Z}_{2^k}$. To plot the strategy, we will work using the Algorithm 1 as a reference, which is more convenient for our solution as we will see later. To accomplish our goal, we will call the section between Line 1 and Line 10 in Algorithm 1 as the *preamble*. On the other hand, we will call the section between Line 11 and Line 22 as the *arithmetic section*.

Let us show an overview of the strategy. First, we will encode the input of the chains with a binary encoding to compute the preamble efficiently using protocols based on binary computation domains. These protocols have a good performance for computing comparisons which are the main operation performed in the preamble. Once the preamble is computed, we use daBits/edaBits to transform the binary shares from the preamble into arithmetic shares for its use in the arithmetic section. After the share transformation, we optimize the arithmetic section by reducing the number of rounds needed to compute the position $(n+1, m+1)$ of the matrix D. To do this, we will take the advantage of a protocol to compute the minimum of a list of numbers whose number of rounds is logarithmic in the length of the list.

3.1 Preamble computation

As we saw in Chapter 2, the protocols that work on binary domains are very efficient to compute comparisons, because the XOR can be computed with low communication costs. According to Algorithm 1, the main task in the preamble is to compute the matrix t, but the computation of this matrix relies on the equality comparison of two nucleotides. So, the first part of the strategy is to propose a method to compare nucleotides in an efficient

way using a binary domain.

To compare two nucleotides, we will encode the nucleotides of a DNA chain using a binary encoding, namely, the nucleotide A will be encoded as 00, C as 01, G as 10, and T will be encoded as 11. The particular encoding is not relevant here if it is used consistently in the encoding of both chains. So, a nucleotide N will be represented as two binary numbers, that is, $N \stackrel{\text{def}}{=} \langle b_0, b_1 \rangle$, where $b_0, b_1 \in \mathbb{Z}_2$. In terms of secret-sharing, we denote the sharing of the nucleotide $N = \langle b_0, b_1 \rangle$ as $[\![N]\!] \stackrel{\text{def}}{=} \langle [\![b_0]\!]_2, [\![b_1]\!]_2 \rangle$. Extending the XOR operation for binary numbers, we define the XOR operation between two nucleotides $N = \langle b_0, b_1 \rangle$ and $N' = \langle b'_0, b'_1 \rangle$ to be $N \oplus N' \stackrel{\text{def}}{=} \langle b_0 \oplus b'_0, b_1 \oplus b'_1 \rangle$. This XOR operation can be defined for shares in a natural way, that is, $[\![N]\!] \oplus [\![N']\!] \stackrel{\text{def}}{=} \langle [\![b_0]\!]_2 \oplus [\![b'_0]\!]_2, [\![b_1]\!]_2 \oplus [\![b'_1]\!]_2 \rangle$.

With all of the operations defined, we can compare two nucleotides using the XOR operation, because

$$N \oplus N' = 0 \quad \text{if and only if} \quad N = N'. \tag{3-1}$$

Notice that the XOR of two nucleotides is another binary tuple. So, the comparison of two nucleotides can be reduced to the problem of determining when a binary tuple is equal to zero. Let $N = \langle b_0, b_1 \rangle$ be a nucleotide. We define

$$S(N) \stackrel{\text{def}}{=} b_0 \vee b_1.$$

It is important to note that $S(N) = b_0 + b_1 + b_0 b_1$, which involves just one product. With this definition, it holds that

$$S(N) = 0 \quad \text{if and only if} \quad N = 0. \tag{3-2}$$

By combining both Equation (3-1) and Equation (3-2), we can conclude that

$$N = N' \quad \text{if and only if} \quad S(N \oplus N') = 0.$$

Given that the computation domain is $\mathbb{Z}_2$, if $a, b \in \mathbb{Z}_2$, then $a \oplus b = a + b \pmod 2$. So, if $N = \langle b_0, b_1 \rangle$ and $N' = \langle b'_0, b'_1 \rangle$, then

$$\left[\!\left[N \stackrel{?}{=} N' \right]\!\right]_2 = 1 - \left[\left([\![b_0]\!]_2 + [\![b'_0]\!]_2 \right) \vee \left([\![b_1]\!]_2 + [\![b'_1]\!]_2 \right) \right]. \tag{3-3}$$

Here, the notation $N \stackrel{?}{=} N'$ is a bit, which means that $(N \stackrel{?}{=} N') = 1$ if $N = N'$, and $(N \stackrel{?}{=} N') = 0$ if $N \neq N'$. Therefore, we can obtain a boolean share of the assertion $N \stackrel{?}{=} N'$ using only one product as follows:

$$\left[\!\left[N \stackrel{?}{=} N' \right]\!\right]_2 = 1 - \left[\left([\![b_0]\!]_2 + [\![b'_0]\!]_2 \right) + \left([\![b_1]\!]_2 + [\![b'_1]\!]_2 \right) + \left([\![b_0]\!]_2 + [\![b'_0]\!]_2 \right) \left([\![b_1]\!]_2 + [\![b'_1]\!]_2 \right) \right].$$

Using the proposed approach, we can compute the matrix t in the preamble efficiently. A positive consequence of this method is that each position of the matrix t will contain a binary share with the answer to the assertion presented in Equation (3-3). In that way, each position of the matrix can be transformed from a binary share to an arithmetic share to be used in the arithmetic section of the Algorithm 1.

3.1.1 Complexity analysis

Let us analyze the online complexity of the computation of the matrix t in the preamble. Suppose that we have two DNA chains P and Q so that $|P| = n$ and $|Q| = m$. The preamble requires us to compute $n \cdot m$ positions of the matrix t, and to compute each position we need to compute one comparison. So, to compute the matrix, we need to perform $n \cdot m$ comparisons in total.

Suppose that we are using a security model with a passive adversary with one corruption, and we have all the multiplication triples needed. To compute a comparison, we need to perform a multiplication, which needs two invocations to the Open protocol. Therefore, we need to perform $2nm$ invocations of the Open protocol in total. Each invocation of Open needs to send two bits, one for each party. Then, we are sending $4nm$ bits in total to compute the matrix t.

It is relevant to say that the previous estimation does not hold for an active adversary. In that case, the number of bits transmitted is higher due to the additional mechanisms needed to ensure share authentication. Also, we are not considering the particularities of the transmission through the network: on the one hand, the TCP/IP metadata can add more bits to transmit the information; on the other hand, some real-world implementations use packaging strategies to send a group of bits in one package instead of sending one by one. These aspects can change the number of bits sent to the network to compute the preamble but they are challenging to compute and also they are protocol dependent.

It is important to notice that the computation of every position of the matrix t does not have any dependency on other positions, which means that we can compute all the positions of the matrix in parallel. This implies that the computation of the matrix only costs one round.

3.2 Arithmetic section

Once the computation of the matrix t is completed in the preamble, we transform all the positions of the matrix t into arithmetic shares using daBits as shown in Section 2.5. After that process, we obtain a new version of the matrix t whose shares can be used to compute the arithmetic section. To compute such section, we start from the arithmetic shares of t, that is, each party has a share $[\![t(i,j)]\!]_{2^k}$ for each index (i,j). Also, following the Algorithm 1, $D(i,0) = i$, for all $i \in \mathbb{J}_n$, and $D(0,j) = j$, for all $\mathbb{J}_m$. These are the starting points of the secure protocol. Our goal is to compute a share of the bottom-right corner of the matrix D, namely, $[\![D(n,m)]\!]_{2^k}$.

If we take a look into the Algorithm 1 in the arithmetic section, we notice that at each iteration of the loop, we need to compute the minimum between three positions of the matrix D that were already computed in previous iterations. There are well known protocols to compute comparisons between two signed integers based on the computation of the most significant bit of the subtraction of both numbers [Dam+19]. This gives us a direct solution for the arithmetic section and the private computation of the edit distance between the two chains. The inconvenience of taking such approach is the sequential dependency between the positions of the matrix. This dependency prevents us from parallelizing the process, which increases the number of rounds needed to compute the position $(n+1, m+1)$ of the matrix D. Our approach to overcome this limitation is to not compute all of the position of the matrix, instead, we will compute some selected positions. Each of such position will be computed as the minimum of some quantities that depend on more than three positions that were already computed, which requires a logarithmic number of rounds.

The approach used here is based on the ideas presented in [CKL15]. In such reference, they use the strategy mentioned above to compute the edit distance using homomorphic encryption schemes. This thesis will generalize this idea and apply it on secret-sharing protocols. Let P and Q be two DNA chains with lengths n and m respectively. In that case, the matrix D will have $n+1$ rows and $m+1$ columns. According to Algorithm 1,

$$D(i,j) = \min \begin{cases} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+t(i,j) \end{cases}. \tag{3-4}$$

But, if we continue the process one more time with the positions inside the minimum, we obtain that

$$D(i-1,j) = \min \begin{cases} D(i-2,j)+1 \\ D(i-1,j-1)+1 \\ D(i-2,j-1)+t(i-1,j) \end{cases},$$

$$D(i,j-1) = \min \begin{cases} D(i-1,j-1)+1 \\ D(i,j-2)+1 \\ D(i-1,j-2)+t(i,j-1) \end{cases},$$

and

$$D(i-1,j-1) = \min \begin{cases} D(i-2,j-1)+1 \\ D(i-1,j-2)+1 \\ D(i-2,j-2)+t(i-1,j-1) \end{cases}. \tag{3-5}$$

When we take all of this equations and replace them in Equation (3-4), we have that

$$D(i,j) = \min \begin{cases} D(i-2,j)+2 \\ D(i-2,j-1)+t(i-1,j)+1 \\ D(i-2,j-1)+3 \\ D(i-1,j-2)+3 \\ D(i-2,j-2)+t(i-1,j-1)+2 \\ D(i,j-2)+2 \\ D(i-1,j-2)+t(i,j-1)+1 \\ D(i-2,j-1)+t(i,j)+1 \\ D(i-1,j-2)+t(i,j)+1 \\ D(i-2,j-2)+t(i,j)+t(i-1,j-1) \end{cases} \tag{3-6}$$

$$= \min \begin{cases} D(i-2,j)+2 \\ D(i-2,j-1)+t(i-1,j)+1 \\ D(i-2,j-1)+t(i,j)+1 \\ D(i-2,j-2)+t(i-1,j-1)+t(i,j) \\ D(i,j-2)+2 \\ D(i-1,j-2)+t(i,j-1)+1 \\ D(i-1,j-2)+t(i,j)+1 \end{cases}.$$

There are two points worth to mention. First, notice that we compute the minimum over 7 formulas knowing that each term in Equation (3-4) is expanded in at least other 3 formulas. This is because some formulas can be discarded given that all of them are inside a minimum, so we can remove the formulas that are redundant in the sense that their value are greater than some other formula in the group. We will deal with a method to remove the redundant formulas in Chapter 4. The second point to mention is that in Equation (3-6), the position $D(i,j)$ does not depend on the position $D(i-1,j-1)$, but the expression to compute $D(i,j-1)$ and $D(i-1,j)$ do. To avoid computing $D(i-1,j-1)$, we can replace the expression for $D(i-1,j-1)$ from Equation (3-5) into the equations for $D(i,j-1)$ and $D(i-1,j)$, and take the last replacements to put them into Equation (3-4). We can perform this process recursively to write $D(i,j)$ in terms of formulas that include the value of other positions of the matrix. Such positions lie on the border of a rectangle

inside the matrix whose bottom right corner is $D(i,j)$. More specifically and following the notation of [CKL15, Section 4.3], let $\tau \in \mathbb{N}$ be with $\tau > 0$. We define the $(\tau+1)$-box for $D(i,j)$ as the set comprised of the union of the following sets[1]:

$$\mathcal{T} \stackrel{\text{def}}{=} \{D_{i-\tau,j-\tau}, D_{i-\tau,j-\tau+1}, \ldots, D_{i-\tau,j}\},$$
$$\mathcal{B} \stackrel{\text{def}}{=} \{D_{i,j-\tau}, D_{i,j-\tau+1}, \ldots, D_{i,j}\},$$
$$\mathcal{L} \stackrel{\text{def}}{=} \{D_{i-\tau,j-\tau}, D_{i-\tau+1,j-\tau}, \ldots, D_{i,j-\tau}\},$$
$$\mathcal{R} \stackrel{\text{def}}{=} \{D_{i-\tau,j}, D_{i-\tau+1,j}, \ldots, D_{i,j}\}.$$

With this definitions, it holds that not just $D(i,j)$ but all the elements in $\mathcal{B} \cup \mathcal{R}$ can be written as the minimum of formulas that depend on positions in the set $\mathcal{T} \cup \mathcal{L}$. Figure 3-1 shows the positions that belong to the $(\tau+1)$-box. In the image, we highlight the sets $\mathcal{T}$, $\mathcal{R}$, $\mathcal{B}$ and $\mathcal{L}$.

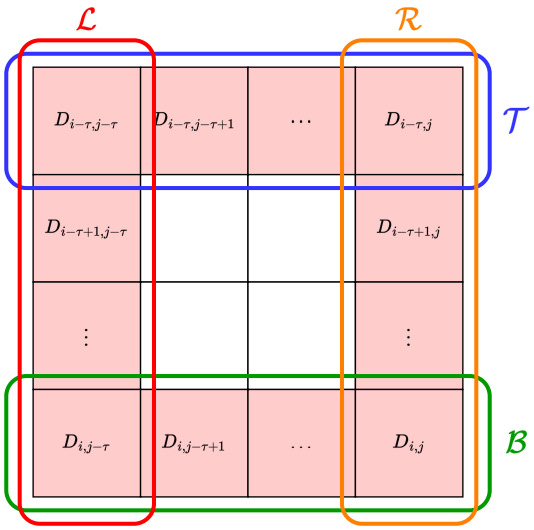


Figure 3-1: Positions that belong to the $(\tau+1)$-box for the position $D(i,j)$.

Continuing with the example for $\tau = 2$ and using the new notation for the borders of the box, we can compute the positions $D(i-1,j)$ and $D(i,j-1)$ as we did for $D(i,j)$ using the following equations in terms of the positions in $\mathcal{T} \cup \mathcal{L}$:

$$D(i-1,j) = \min \begin{cases} D(i-2,j)+1 \\ D(i-2,j-1)+t(i-1,j) \\ D(i-2,j-2)+t(i-1,j-1)+1 \\ D(i-1,j-2)+2 \end{cases}, \tag{3-7}$$

[1] Note on notation: in some situations, we will avoid the parentheses notation and we replace it for subscript notation for the matrices D and t. That is, $D(i,j)$ will be written as $D_{i,j}$ and $t(i,j)$ will be written as $t_{i,j}$.

and

$$D(i,j-1) = \min \begin{cases} D(i,j-2)+1 \\ D(i-1,j-2)+t(i,j-1) \\ D(i-2,j-2)+t(i-1,j-1)+1 \\ D(i-2,j-1)+2 \end{cases}. \tag{3-8}$$

The ideas presented in [CKL15] use just one $(\tau+1)$-box, with $\tau = n$, to compute the edit distance of two chains of the same length by calculating $D(n,n)$. They show that such position of the matrix D can be computed as the minimum of a list of $2^{n+1}-n$ numbers.

Our work goes beyond the results of [CKL15] in the sense that the positions in a $(\tau+1)$-box are a subset of the entire matrix D, so this matrix can be divided in $(\tau+1)$-boxes. Figure 3-2 shows an example of the complete matrix D divided into boxes. The light red positions are the positions that are needed to compute the position $D(n,m)$, and the dark red positions are the most expensive position to compute inside each $(\tau+1)$-box. The first advantage of the subdivision is that we are allowed to compute the edit distance between two DNA chains that do not necessarily have the same length. The second advantage, which we will discuss later, is that the size of the $(\tau+1)$-box induces a trade-off between the number of rounds to compute the edit distance and the data sent during the protocol execution.

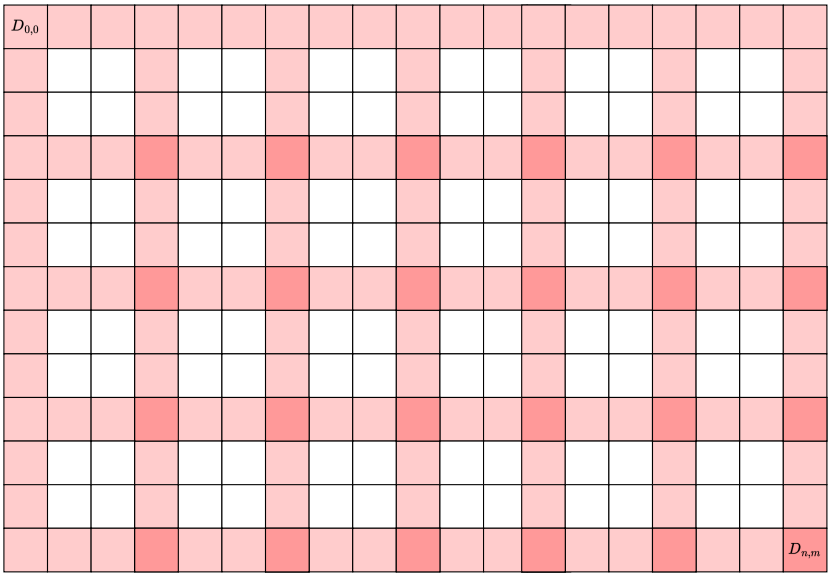


Figure 3-2: A complete matrix D divided in $(\tau+1)$-blocks.

To compute the positions required to calculate the edit distance using secret-sharing pro-

tocols, we use the protocol MIN_q proposed in [Dam+19; Tof07]. This protocol computes the minimum of a list of q numbers in $O(\log_2 q) \cdot (c_r + 2)$ rounds, where c_r is the number of rounds of a comparison, using $q - 1$ comparisons and $2q - 2$ multiplications. As an example, for $\tau = 2$, we can compute securely a share of the position $D(i, j)$ by executing the protocol MIN_7 as follows:

$$[\![D(i,j)]\!]_{2^k} = \text{MIN}_7 \begin{cases} [\![D(i-2,j)]\!]_{2^k} + 2 \\ [\![D(i-2,j-1)]\!]_{2^k} + [\![t(i-1,j)]\!]_{2^k} + 1 \\ [\![D(i-2,j-1)]\!]_{2^k} + [\![t(i,j)]\!]_{2^k} + 1 \\ [\![D(i-2,j-2)]\!]_{2^k} + [\![t(i-1,j-1)]\!]_{2^k} + [\![t(i,j)]\!]_{2^k} \\ [\![D(i,j-2)]\!]_{2^k} + 2 \\ [\![D(i-1,j-2)]\!]_{2^k} + [\![t(i,j-1)]\!]_{2^k} + 1 \\ [\![D(i-1,j-2)]\!] + [\![t(i,j)]\!]_{2^k} + 1 \end{cases} .$$

The shares $[\![D(i-1,j)]\!]_{2^k}$ and $[\![D(i,j-1)]\!]_{2^k}$ can be written similarly following the Equations (3-7) and (3-8) shown before, and using the protocol MIN_4 in both cases.

Given the MIN_q functionality, we can design a protocol to compute the edit distance. Our proposed method iterates over all the positions in the set $\mathcal{B} \cup \mathcal{R}$ of the $(\tau + 1)$-boxes, computing each position one by one. Such positions can be written in terms of the positions in $\mathcal{T} \cup \mathcal{L}$ of that box as discussed before. So, if we iterate the boxes in a left-to-right and top-to-down way, we will have all the information needed to calculate securely all the positions required. This is assuming that we have all the formulas pre-computed for each position in $\mathcal{B} \cup \mathcal{R}$, which is completely feasible to do in an automated way as we will see in Chapter 4. At the end of the protocol, each party will hold a share $[\![D(n,m)]\!]_{2^k}$, which is the share of the edit distance. It can be revealed using the OPEN protocol for each party to acquire the final result.

3.2.1 Complexity analysis

To measure the complexity of our solution, let us compute the complexity of the naive solution with no grouping of formulas as a baseline. Then, we will compute the complexity of our strategy.

For the naive approach, notice that we need to compute $n \cdot m$ positions of the matrix. Now, to compute the minimum of three numbers, we can use the protocol presented in [Dam+19, Figure 14, Instruction (2)] which requires a constant number of rounds, comparisons, and multiplications. Then, we can compute the edit distance of two chains using $O(nm)$ comparisons, multiplications, and rounds.

Now, let us analyze our proposal. In Chapter 4, we will show a strategy such that the

number of formulas in the minimum computation to calculate one position of a $(\tau+1)$-box is bounded by $O(\tau \cdot 2^{3\tau})$. Remember that the protocol to compute the MIN_q functionality requires $q-1$ comparisons and $2q-2$ multiplications using in $O(\log_2 q) \cdot (c_r + 2)$ rounds according to [Tof07, Section 13.1.1]. If we assume for simplicity that τ divides both m and n, we need to compute nm/τ^2 boxes. Given that we need to compute $2\tau - 1$ positions in each $(\tau+1)$-box, we are required to compute

$$\frac{nm}{\tau^2} \cdot (2\tau - 1)$$

positions from the matrix D in total. However, the interesting part of this solution is that all the positions in $\mathcal{B} \cup \mathcal{R}$ within one box can be computed in parallel because there is no dependency between them. This makes the term $2\tau - 1$ disappear from the round count. Joining all of the results and computations, we conclude that for two DNA chains of lengths n and m, we can compute the edit distance in

$$O\left(\frac{nm}{\tau^2} \cdot (3\tau + \log_2 \tau) \cdot (c_r + 2)\right)$$

rounds,

$$O\left(\frac{nm}{\tau^2} \cdot (2\tau - 1) \cdot (\tau \cdot 2^{3\tau} - 1)\right)$$

comparisons, and

$$O\left(\frac{nm}{\tau^2} \cdot (2\tau - 1) \cdot (\tau \cdot 2^{3\tau+1} - 2)\right)$$

multiplications.

Comparing both the baseline and the complexity of our strategy, we find that our method has an improvement in the number of rounds. Moreover, the higher the τ, the better the improvement. However, the higher the τ, the higher the number of multiplications and comparisons, which increases directly the data sent in the protocol execution. This is a trade-off that should be considered according to the network capabilities in which the protocol is executed. If the network has a high bandwidth, we can increase the τ keeping the data sent in reasonable quantities for the bandwidth, while the number of rounds is reduced. This strategy could reduce the overall execution time of the protocol compared to naive implementation. In Chapter 5, we will experimentally test this trade-off and draw some conclusions about it.

3.3 A protocol to compute the edit distance

A complete specification of the protocol to compare two DNA chains can be found in Algorithm 2. In such specification, we are assuming that τ divides both m and n for

simplicity. If it does not happen, the strategy shown in Section 3.2 can be generalized easily to compute the positions in the matrix D not for a $(\tau+1)$-box but for a set of positions that comprise a rectangle inside D following the exact same steps shown in that section.

There are three sub-routines to highlight. The first routine is the protocol Π_{CMP} that compares the binary secret-shares of two nucleotids in a DNA chain as it was completely specified in Section 3.1. The second routine is Π_{B2A} protocol, which takes a binary share and convert it into a arithmetic share. This can be done using daBits as it was mentioned in Section 2.5. And finally, the GetFormulas which takes a position in $\mathcal{B} \cup \mathcal{R}$ of a $(\tau+1)$-box and returns the set of formulas needed as arguments to the minimum function required compute such position in the matrix D. We need to clarify that this subroutine is performed in the clear. Moreover, all of these formulas can be computed previous to the beginning of the secure protocol execution because they do not need any secret information about the DNA chains. In Chapter 4 we will show the design and analysis of an algorithm for this sub-routine.

Algorithm 2 Protocol for secure edit distance computation.

Input: two binary secret-shared chains $[\![P]\!]_2 = [[\![p_1]\!]_2, \cdots, [\![p_n]\!]_2]$ and $[\![Q]\!]_2 = [[\![q_1]\!]_2, \cdots, [\![q_m]\!]_2]$. The size of the box τ.

Output: an share of the edit distance distance between the chains P and Q

Let t_B be a matrix with dimensions $n \times m$.
for $i = 1$ to n **do**
    for $j = 1$ to n **do**
        $[\![t_B(i,j)]\!]_2 = \Pi_{\mathsf{CMP}}([\![p_i]\!]_2, [\![q_j]\!]_2)$
    end for
end for
Let t be a matrix with dimensions $n \times m$.
for $i = 1$ to n **do**
    for $j = 1$ to n **do**
        $[\![t(i,j)]\!]_{2^k} = \Pi_{\mathsf{B2A}}([\![t_B(i,j)]\!]_2)$
    end for
end for
Let D be a secret-shared matrix with dimensions $(n+1) \times (m+1)$.
for $i = 0$ to n **do**
    $[\![D(i,0)]\!]_{2^k} = \Pi_{\mathsf{Input}}(i)$
end for
for $j = 0$ to m **do**
    $[\![D(0,j)]\!]_{2^k} = \Pi_{\mathsf{Input}}(j)$
end for
for $i = 1$ to n/τ **do**
    for $j = 1$ to m/τ **do**
        Let $B_{i,j}$ the $(\tau+1)$-box indexed with (i,j).
        Let $\mathcal{B}$ the bottom border of $B_{i,j}$.
        Let $\mathcal{R}$ the right border of $B_{i,j}$.
        for each position in $\mathcal{B} \cup \mathcal{R}$ indexed with (l,r) **do**
            $\mathcal{S}_{l,r} = \textsc{GetFormulas}(l,r)$
            $[\![D(l,r)]\!]_{2^k} = \textsc{Min}_{\textsc{Length}(\mathcal{S}_{l,r})} \{[\![F]\!]_{2^k} \mid F \in \mathcal{S}_{l,r}\}$
        end for
    end for
end for
return $[\![D(n,m)]\!]_{2^k}$

4 Automated generation of formulas to compute the edit distance

In Chapter 3, we developed a method to compute the edit distance between two chains using secret-sharing protocols. However, in that chapter, we assume that the formulas inside the minimum function of each position of the $(\tau+1)$-box are given. In this chapter, we will show an automatic method to generate the expressions to calculate all of the positions in $\mathcal{B} \cup \mathcal{R}$ of the $(\tau+1)$-box in terms of the positions in the sets $\mathcal{T} \cup \mathcal{L}$. Our goal is not to just extract the formulas that compute the edit distance correctly. That can be done naively by applying Equation (3-4) recursively. Instead, we want to extract the smallest set of formulas that allows the correct computation of the edit distance by removing the redundant formulas inside the minimum.

Let us explain our goal using a concrete example. Suppose that we divide the matrix D in $(\tau+1)$-boxes with $\tau = 2$. Let us choose an arbitrary box, and suppose that we want to write the position $D(i,j)$ in terms of positions in $\mathcal{T} \cup \mathcal{L}$. Figure 4-1 shows the box that we are using. The goal is to write $D(i,j)$, which is painted with red in the figure, in terms of the positions that are painted with blue. If we apply Equation (3-4) recursively as we explained in Section 3.2 without removing any formula, we get that

$$D(i,j) = \min \begin{cases} D(i-2,j)+2 \\ D(i-2,j-1)+t(i-1,j)+1 \\ D(i-2,j-1)+3 \\ D(i-1,j-2)+3 \\ D(i-2,j-2)+t(i-1,j-1)+2 \\ D(i,j-2)+2 \\ D(i-1,j-2)+t(i,j-1)+1 \\ D(i-2,j-1)+t(i,j)+1 \\ D(i-1,j-2)+t(i,j)+1 \\ D(i-2,j-2)+t(i,j)+t(i-1,j-1) \end{cases} . \tag{4-1}$$

Observe that each formula inside the minimum function is in terms of the positions in $\mathcal{T} \cup \mathcal{L}$ plus some values in the matrix t and a constant. Applying this identity to compute the position $D(i,j)$ ends up in a correct result. However, the problem with this set of

formulas is that it is far from optimal in the number of formulas. Take for example the formulas

$$D(i-2,j-1)+t(i-1,j)+1 \quad \text{and} \quad D(i-2,j-1)+3.$$

Both formulas are inside a minimum, but for any assignment of $D(i-2,j-1)$ and $t(i-1,j)$, it holds that

$$D(i-2,j-1)+t(i-1,j)+1 \leq D(i-2,j-1)+3,$$

because $t(i-1,j) \in \{0,1\}$. This allows us to remove the formula $D(i-2,j-1)+3$ from the set of formulas without affecting the overall result of the minimum function. The same situation occurs with the pair of formulas

$$D(i-2,j-2)+t(i,j)+t(i-1,j-1) \quad \text{and} \quad D(i-2,j-2)+t(i-1,j-1)+2,$$

where we can remove the formula $D(i-2,j-2)+t(i-1,j-1)+2$ without changing the overall result of the minimum.

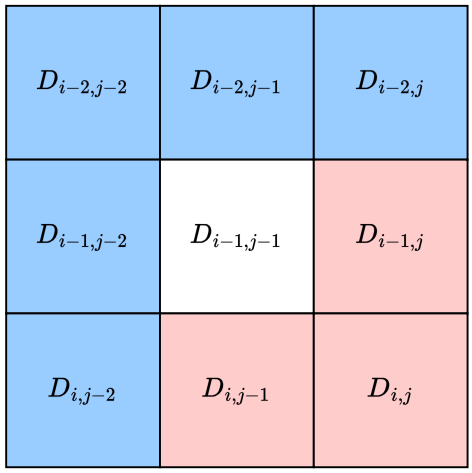


Figure 4-1: $(\tau+1)$-box with $\tau = 2$.

The same problem applies if we want to write $D(i,j-1)$ and $D(i-1,j)$ (and in general, whatever position in $\mathcal{B} \cup \mathcal{R}$) in terms of the formulas that depends on the positions in $\mathcal{T} \cup \mathcal{L}$.

The problem at hand can be expressed as follows: we need to find an algorithmic way to find an expression to compute each position in $\mathcal{B} \cup \mathcal{R}$ in terms of the positions in $\mathcal{T} \cup \mathcal{L}$ in such a way that the number of formulas inside the minimum function is minimal.

The purpose behind the answer to this question is to remove the overhead on the secure computation of the minimum. As we saw in Section 3.2, the computational complexity of the MIN_q protocol increases with q. So, we need to keep the number of formulas at a minimum to improve the performance of the calculation of the edit distance without affecting the final result of the computation.

4.1 The dependency graph

To solve the problem at hand, we will express the problem as a problem over graphs. Let us take a position $D(i,j)$ in the matrix. By considering Equation (3-4), we can say that $D(i,j)$ depends on positions $D(i,j-1)$, $D(i-1,j)$ and $D(i-1,j-1)$. This dependency can be represented in a directed labeled graph with colored edges as in Figure **4-2**. The vertices of the graph are the positions $D(i,j)$, $D(i,j-1)$, $D(i-1,j-1)$ and $D(i-1,j)$. The labels are disposed of as follows: if in the Equation (3-4) the position $D(i,j)$ has a formula $D(i',j')+a$ as an argument of the minimum function, then, the label of the edge going from $D(i',j')$ to $D(i,j)$ will be a. The color of the edge depends on its label: if the label is 1, then the color will be black, otherwise, the color will be red.

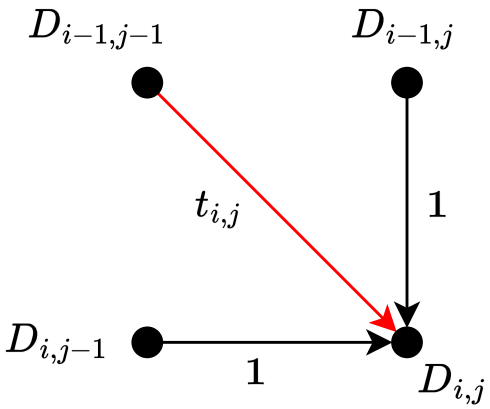


Figure 4-2: Graph with the dependencies for $D(i,j)$.

If we continue the process with the vertices $D(i-1,j-1)$, $D(i-1,j)$ and $D(i-1,j)$ following the same rules, we end up with a graph as in Figure **4-3**. It is important to make clear that the graph is not constructed with multi-edges. So, if an edge arises for a second time in the middle of the construction of the graph, this second edge is ignored.

We can continue with this process until we reach all the vertices in the border of the $(\tau+1)$-

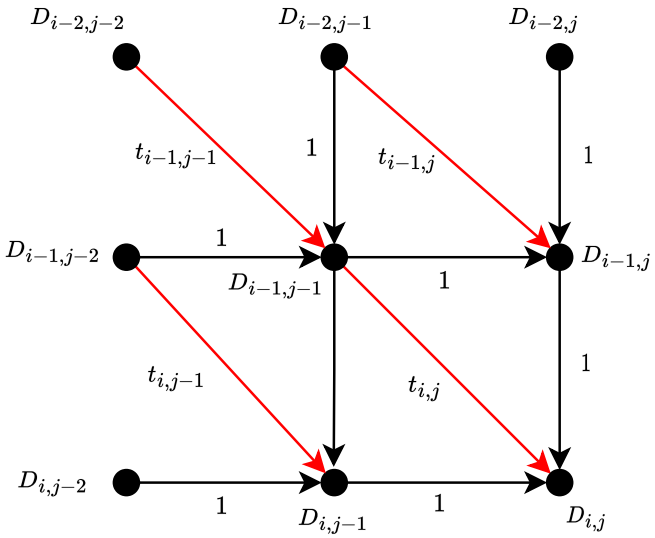


Figure 4-3: Continuation of the expansion in the dependency graph. This graph can also be considered to be the dependency graph for $\tau = 2$.

box. This graph, denoted by G, will be called the *dependency graph*. This abstraction was considered by Ukkonen in [Ukk85, Section 2]. However, we are adding the coloring of the labels as a new element to reach our goal.

4.2 An algorithm for the optimal formula generation

Once the dependency graph G is constructed, we will design an algorithm to generate a set of formulas inside the minimum for each element in $\mathcal{B} \cup \mathcal{R}$ in terms of elements in $\mathcal{T} \cup \mathcal{L}$.

Definition 4.2.1. *Let $V \in \mathcal{B} \cup \mathcal{R}$ and $W \in \mathcal{T} \cup \mathcal{L}$. Let P be a path from W to V. Let us construct the formula $f_P \stackrel{def}{=} W + a$, where a is the sum of all the labels of the edges in the path P. We call f_P the formula induced by P.*

Let us consider $V \in \mathcal{B} \cup \mathcal{R}$. Let us take all the induced formulas f_P, for all the paths P from a point in $\mathcal{T} \cup \mathcal{L}$ to the vertex V. Due to the construction of the dependency graph G, if we put those formulas as arguments in the minimum function, this will be a correct expression to compute the position V of the matrix in terms of elements in $\mathcal{T} \cup \mathcal{L}$. Moreover, we can repeat this process for each $V \in \mathcal{B} \cup \mathcal{R}$. Using a concrete notation, we

can compute the position $V \in \mathcal{B} \cup \mathcal{R}$ in the matrix D as

$V = \min\{f_P \mid P \text{ is a path from } W \text{ to } V \text{ in } G,\ \forall W \in \mathcal{T} \cup \mathcal{L}\}$.

This method gives us a brute-force solution to the problem because we will end up with a non-optimal expression similar to Equation (4-1). However, this approach reduces our problem to the problem of excluding the paths between vertices that induce formulas that do not have any relevance in the computation of the minimum.

Given all the discussion in Section 3.2 about the expansion of the positions in $\mathcal{B} \cup \mathcal{R}$ as the minimum of formulas that are in terms of the positions in $\mathcal{T} \cup \mathcal{L}$, we can state the following proposition.

Proposition 4.2.2. *Let $D(i,j) \in \mathcal{B} \cup \mathcal{R}$ be a position of the matrix D in a $(\tau+1)$-box. Following the Equation (3-4) recursively and without removing any redundant formula in the process, $D(i,j)$ can be written as*

$$D(i,j) = \min_{k=1}^{M}\{f_k\}, \tag{4-2}$$

for some $M \in \mathbb{N}$, where f_k is a formula of the form $f_k = D(i',j') + \sum_{(l,r)} t_{l,r} + b_k$, with $b_k \in \mathbb{Z}$, and $D(i',j') \in \mathcal{T} \cup \mathcal{L}$.

The next definition will be used to relate the concept of formulas induced by paths and the set of formulas inside the minimum function to calculate a position in $\mathcal{B} \cup \mathcal{R}$ belonging to the matrix D.

Definition 4.2.3. *Let $D(i,j) \in \mathcal{B} \cup \mathcal{R}$ and $D(i',j') \in \mathcal{T} \cup \mathcal{L}$ be positions of the matrix D in a $(\tau+1)$-box. According to Proposition 4.2.2, each formula f_k in Equation (4-2) such that*

$$f_k = D(i',j') + \sum_{(l,r)} t_{l,r} + b_k$$

will be called an unrolled formula from $D(i',j')$ to $D(i,j)$. Also, the set of all the formulas with this structure will be called the set of unrolled formulas from $D(i',j')$ to $D(i,j)$.

Now, let us make use of the coloring of the labels in the following definition.

Definition 4.2.4. *Let P and Q be two paths in the dependency graph G. We define $r_{P,Q}$ as the number of red edges in P that are not in Q. On the other hand, we define b_P as the number of black edges in the path P.*

Given this definitions, let us give the first step into the design of an algorithm to extract an optimal set of paths.

Proposition 4.2.5. *Let $\mathcal{P}_{U,W}$ the set of paths that begin in the vertex $U \in \mathcal{T} \cup \mathcal{L}$ and end in the vertex $W \in \mathcal{B} \cup \mathcal{R}$ in the dependency graph G. Let $P \in \mathcal{P}_{U,W}$ be any path. We can remove the formula induced by P from the set of unrolled formulas from U to W without changing the overall value of the minimum, if $r_{Q,P} + b_Q \leq b_P$, for some $Q \in \mathcal{P}_{U,W} \setminus \{P\}$.*

Proof. Let $P \in \mathcal{P}_{U,W}$ be any path, and suppose that there exists some $Q \in \mathcal{P}_{U,W} \setminus \{P\}$ such that $r_{Q,P} + b_Q \leq b_P$. Both paths induce a formula in the set of formulas, so let the formula induced by P be

$$f_P \stackrel{\text{def}}{=} D_U + \sum_i t_i + \sum_{i=1}^{r_{P,Q}} t_i^{(P)} + b_P.$$

The terms denoted by t_i are the red labels shared by both P and Q. Also, the terms denoted by $t_i^{(P)}$ are the red labels that are in P but not in Q. Similarly, let the formula induced by Q be

$$f_Q \stackrel{\text{def}}{=} D_U + \sum_i t_i + \sum_{i=1}^{r_{Q,P}} t_i^{(Q)} + b_Q.$$

As in the previous equation, the terms denoted by $t_i^{(Q)}$ are the terms that are in Q but not in P. In both formulas, D_U is a number in the edit-distance optimization matrix that depends on the endpoint U. Because U is fixed, it is not relevant to say what is the position of D_U in the matrix nor its specific value.

Using the inequality in the hypothesis, it holds that

$$\begin{aligned} f_Q = D_U + \sum_i t_i + \sum_{i=1}^{r_{Q,P}} t_i^{(Q)} + b_Q &\leq D_U + \sum_i t_i + r_{Q,P} + b_Q \\ &\leq D_U + \sum_i t_i + \sum_{i=1}^{r_{P,Q}} t_i^{(P)} + r_{Q,P} + b_Q \\ &\leq D_U + \sum_i t_i + \sum_{i=1}^{r_{P,Q}} t_i^{(P)} + b_P = f_P. \end{aligned}$$

This inequality shows that $f_Q \leq f_P$ for any assignment of the t variables. Which means that we can remove the formula induced by P without changing the overall value of the minimum over the set of unrolled formulas. □

Using Proposition 4.2.5 we can formulate an algorithm which will gives us a subset of the set of unrolled formulas for $W \in \mathcal{B} \cup \mathcal{R}$ without changing its overall value of the minimum. This will allow us to get rid of formulas that are not important in the minimum computation at the moment of executing the MPC protocol. The details of the algorithm are presented in Algorithm 3.

Algorithm 3 Optimal set of paths

Input: a dependency graph G. Two endpoints $U \in \mathcal{T} \cup \mathcal{L}$ and $W \in \mathcal{B} \cup \mathcal{R}$, having an appropriate placement of both endpoints in the graph.

Output: a reduced set of paths $\mathcal{S}$ such that the minimum over all the formulas induced by paths in $\mathcal{S}$ is the same as the minimum over all the formulas induced by paths in $\mathcal{P}_{U,W}$.

```
1: Generate the set P_{U,W}.
2: S ← ∅.
3: for P ∈ P_{U,W} do
4:     r ← True
5:     for Q ∈ P_{U,W} \ {P} do
6:         if r_{Q,P} + b_Q ≤ b_P then
7:             r ← False
8:             break
9:         end if
10:    end for
11:    if r = True then
12:        Append P to S
13:    end if
14: end for
15: return S
```

If we want to generate the expression to compute the position $D(i', j') \in \mathcal{B} \cup \mathcal{R}$ in a $(\tau+1)$-box, we need to run the algorithm letting the vertex $D(i', j')$ fixed, then run it for all of the vertices in $\mathcal{T} \cup \mathcal{L}$, and finally, reunite all of the resulting formulas as arguments of the minimum function.

Another point to take into account is that when we change the starting point of the paths, the induced formulas are not comparable. For example, let $D(i', j'), D(i'', j'') \in \mathcal{T} \cup \mathcal{L}$ and $D(i, j) \in \mathcal{B} \cup \mathcal{R}$. When we extract the formulas to compute $D(i, j)$, we can obtain a pair of formulas as arguments of the minimum function with the following structure:

$D(i', j') + \sum t_{i,j} + k$

and

$D(i'', j'') + \sum t'_{i,j} + w.$

But notice that we have no clue for determining if the first or the second formula is redundant, because it could happen that $D(i', j') \neq D(i'', j'')$. This difference causes that for some pairs of DNA chains as inputs, the first formula could be less or equal to the second one, but for other pairs of inputs, the inverse could happen too. Remember that the formulas are symbolic, which means that the generation of the formulas and the fact that some of them are redundant have nothing to do with the particular values in the matrix D and matrix t for a specific pair of DNA chains. The formulas generated by the algorithm hold no matter what inputs are given to the edit distance protocol. For that reason, we design an algorithm to generate formulas that leaves the starting and end point of the paths fixed.

Before proving the optimality of our algorithm, we need to define formally what we consider an optimal set of paths (which translates directly to an optimal set of formulas). Intuitively, a set of paths $\mathcal{S}$ is optimal if for each path in that set, there exists an assignment of the red variables that makes the corresponding induced formula to be equal to the minimum when it is computed over all the formulas in $\mathcal{S}$. Let us define this notion precisely.

Definition 4.2.6. *(Optimality). Let $U, W \in V(G)$. A set $\mathcal{S} \subseteq \mathcal{P}_{U,W}$ is optimal if, for all $P \in \mathcal{S}$, there exists an assignment of the red variables $(t_{i,j})$ such that, for all $Q \in \mathcal{S} \setminus \{P\}$, it holds that $f_P < f_Q$.*

Note. Symbolically, the Definition 4.2.6 can be described as follows

$$(\forall P \in \mathcal{S})(\exists (t_i) \in \{0,1\}^*)(\forall Q \in \mathcal{S} \setminus \{P\})(f_P < f_Q)$$

So, a set $\mathcal{S} \subseteq \mathcal{P}_{U,W}$ is non-optimal if and only if

$$(\exists P \in \mathcal{S})(\forall (t_i) \in \{0,1\}^*)(\exists Q \in \mathcal{S} \setminus \{P\})(f_P \geq f_Q)$$

Proposition 4.2.7. *(Optimality of Algorithm 3). Let $U, W \in V(G)$ be such that $U \in \mathcal{T} \cup \mathcal{L}$ and $W \in \mathcal{B} \cup \mathcal{R}$. The Algorithm 3 returns an optimal set of paths $\mathcal{S} \subseteq \mathcal{P}_{U,W}$ (in the sense of Definition 4.2.6) such that the minimum over all the formulas induced by paths in $\mathcal{S}$ is the same as the minimum over all the formulas in the set of unrolled formulas from U to W.*

Proof. From Proposition 4.2.5, we know that the algorithm returns a set of paths whose induced formulas does not change the overall value resulting from the minimum function. It remains to show that this set is optimal[1].

[1] We will not consider here the case of $|\mathcal{P}_{U,W}| = 1$. In that case, Algorithm 3 returns the only path that is in $\mathcal{P}_{U,W}$, whose case is trivial. Henceforth, we will consider only the case $|\mathcal{P}_{U,W}| > 1$. Also, the case $\mathcal{P}_{U,W} = \emptyset$ is not considered here due to the definition of optimality.

Let $\mathcal{S} \subseteq \mathcal{P}_{U,W}$ a set of paths returned by Algorithm 3. Let $P \in \mathcal{S}$ an arbitrary path. Let us define an structure for f_P as follows:

$$f_P = D_U + \sum_i t_i^{(P)} + b_P,$$

where $t_i^{(P)}$ are the labels of the red edges in the path P. Let us consider an assignment of the red variables $(t_k) \in \{0,1\}^*$ for the dependency graph G as follows: if the edge labeled as t_k is in the path P, assign $t_k = 0$; if the edge is not in the path P, assign $t_k = 1$. Given this assignment, it holds that

$$f_P = D_U + \sum_i t_i^{(P)} + b_P = D_U + b_P.$$

Now, let us take an arbitrary path $Q \in \mathcal{S} \setminus \{P\}$. Let us assume that the general structure for f_Q is

$$f_Q = D_U + \sum_i t_i + \sum_{i=1}^{r_{Q,P}} t_i^{(Q)} + b_Q.$$

Here, t_i are the labels of the red edges that are in both P and Q simultaneously. On the other hand, $t_i^{(Q)}$ are the labels of the red edges that are in Q but not in P. Given the assignment for the red labels in the graph G fixed above, it holds that

$$f_Q = D_U + \sum_i t_i + \sum_{i=1}^{r_{Q,P}} t_i^{(Q)} + b_Q = D_U + r_{Q,P} + b_Q.$$

Knowing that both P and Q are paths returned by the algorithm, this means that when the paths P and Q were selected in the iterations, the path P was not removed. Therefore, the condition $r_{Q,P} + b_Q \leq b_P$ did not hold. On the contrary, it holds that $r_{Q,P} + b_Q > b_P$. Then, joining all the results, we have that

$$f_Q = D_U + r_{Q,P} + b_Q > D_U + b_P = f_P.$$

This shows that the algorithm is optimal in the sense of Definition 4.2.6. □

4.3 An upper bound for the number of formulas

To compute a bound for the number of formulas generated by our approach, let us consider the graph presented in Figure 4-4. In such a graph, there are l rows of vertices and s columns of vertices. This graph is very similar to the dependency graph in a $(\tau + 1)$-box,

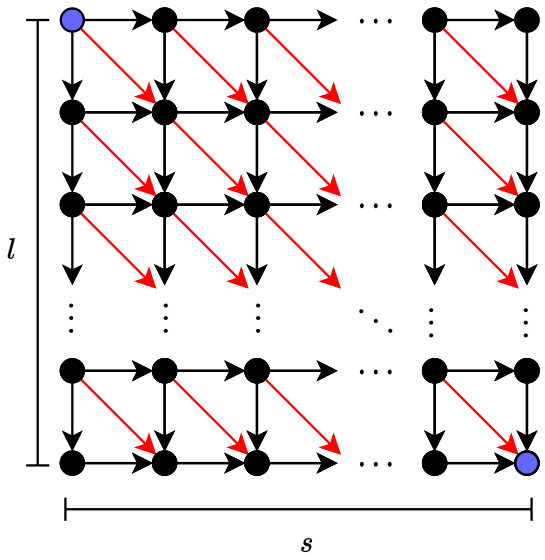


Figure 4-4: Graph used to compute the complexity of our approach.

but this graph has additional edges in the vertices in the border (compare it with Figure 4-3). Not without mention that this graph is not a "square" graph as in Figure 4-3, but a "rectangular" one.

We want to compute an upper bound of the number of formulas that are arguments of the minimum function to calculate the bottom-right corner of a $(\tau + 1)$-box. We will do this by computing the number of all the paths from all the vertices in the top and left border that ends up in the bottom-right corner in Figure 4-4 when $l = s = \tau + 1$. This number will be a correct upper bound because the algorithm takes all the paths from a pair of vertices and removes the redundant ones. Knowing that the bottom-right corner has the highest number of formulas as arguments inside the minimum function, this upper bound will be also bound for the other positions in the set $\mathcal{B} \cup \mathcal{R}$.

This problem reduces to take the graph Figure 4-4 and compute all the paths from the vertex in the top-left corner to the vertex in the bottom-right corner. This can be counted using the Delanoy number, which counts exactly this quantity [Wes20, Definition 1.2.8]. The Delanoy number for a graph with l rows and s columns is

$$\mathcal{D}(l, s) = \sum_{i=0}^{\min\{l,s\}} \binom{l}{i}\binom{s}{i} \cdot 2^i. \tag{4-3}$$

Using Equation (4-3), we can conclude that the upper bound for the number of formulas that are arguments inside the minimum function to calculate the position $D(i, j)$ in a

$(\tau+1)$-box is given by

$$\sum_{k=1}^{\tau}[\mathcal{D}(\tau,\tau-k)+\mathcal{D}(\tau-k,\tau)]+\mathcal{D}(\tau,\tau).$$

Let us break down this equation in pieces. The number $\mathcal{D}(\tau,\tau-k)$ is the number of paths in a graph of τ rows and $\tau-k$ columns. That is the number of paths from a point in $\mathcal{T}\setminus\{D_{i-\tau,j-\tau}\}$ to $D_{i,j}$. Similarly, the number $\mathcal{D}(\tau-k,\tau)$ is the number of paths from a point in $\mathcal{L}\setminus\{D_{i-\tau,j-\tau}\}$ to $D_{i,j}$. And finally, the number $\mathcal{D}(\tau,\tau)$ is the number of paths from $D_{i-\tau,j-\tau}$ to $D_{i,j}$.

Notice that

$$\begin{aligned}
\sum_{k=1}^{\tau}[\mathcal{D}(\tau,\tau-k)+\mathcal{D}(\tau-k,\tau)]+\mathcal{D}(\tau,\tau) &\leq \sum_{k=1}^{\tau}2\mathcal{D}(\tau,\tau)+\mathcal{D}(\tau,\tau)\\
&= 2\tau\mathcal{D}(\tau,\tau)+\mathcal{D}(\tau,\tau)\\
&= (2\tau+1)\mathcal{D}(\tau,\tau)\\
&= (2\tau+1)\sum_{k=0}^{\tau}\binom{\tau}{k}^2\cdot 2^k\\
&\leq (2\tau+1)\cdot 2^{\tau}\cdot\sum_{k=0}^{\tau}\binom{\tau}{k}^2\\
&\leq (2\tau+1)\cdot 2^{\tau}\cdot\left(\sum_{k=0}^{\tau}\binom{\tau}{k}\right)^2\\
&= (2\tau+1)\cdot 2^{\tau}\cdot(2^{\tau})^2\\
&= (2\tau+1)\cdot 2^{3\tau}\\
&= O(\tau\cdot 2^{3\tau}).
\end{aligned}$$

This result allows us to conclude that the number of formulas generated by Algorithm 3 that are arguments of the minimum function used to calculate one position of the set $\mathcal{B}\cup\mathcal{R}$ in a $(\tau+1)$-box is $O(\tau\cdot 2^{3\tau})$.

5 Experiments

In the previous chapters, we explained how to compute the edit distance between two DNA chains and we presented a solution to compute the edit distance based on secret-sharing protocols. The purpose of this chapter is to evaluate the performance of our solution in a practical and experimental way.

To perform the evaluation of our solution, we design four experiments for measuring and comparing different characteristics of the proposed method:

1. We first compare the computation of the preamble in the binary domain with respect to a traditional computation of edit distance where all the computation is done naively in an arithmetic domain.

2. We evaluate how the increasing of τ impacts the performance of our solution in terms of communication and computational complexity.

3. We compare the performance of our proposed solution using secret-sharing protocols with respect to Yao's GC protocols.

4. We evaluate if executing our solution using rings of the form $\mathbb{Z}_{2^k}$ has a better performance compared to executing our protocol using fields of the form $\mathbb{Z}_p$, for a prime p.

These four experiments will give us an empirical idea of the feasibility of our solution and its performance with respect to other techniques commonly used in the literature. As an additional comparison, we make some comments about the performance of our solution compared to computing the edit distance using homomorphic encryption schemes found in previous works.

All the experiments in this section were executed using the MP-SPDZ framework [Kel20]. MP-SPDZ is an MPC framework to easily benchmark multiple protocols in various threat models. In this framework, we can execute protocols of the current state-of-the-art including all the protocols considered in this work. To execute a protocol in MP-SPDZ, the user implements his desired function using the Python 3 programming language with some

additional features from the MP-SPDZ context. The main strength of the framework is that with only one implementation of the function in a high-level language, the user can switch easily between protocols to compute the functionality securely. That means that the user does not need to worry about the particularities of each protocol. He just writes its function in the Python programming language, and the framework is in charge of compiling this high-level source code into the respective circuits to be evaluated securely using the protocols he wants. When a protocol is executed in the framework, one can measure the execution time and the data sent during the protocol execution. Our goal is to use this framework and the metrics mentioned before to evaluate our solution to compute the edit distance.

All the experiments presented in this section were executed in an AWS EC2 instance of type `c6a.4xlarge`[1]. This instance has an AMD EPYC 7R13 Processor with 16 virtual cores and 32 GB of RAM. To measure the impact of the network architecture, we consider two types of network architectures using simulation. The first type does not consider any network limitation, so the speed of communication is very similar to the speed of two separate processes exchanging information in the same machine. The second type is the local area network (LAN) architecture where we simulate a bandwidth of 1.6 gigabytes per second and a latency of 0.3 milliseconds. Given that the executions of the experiments are executed in the same machine, we simulate the LAN restrictions using the `tc`[2] command from the Linux operating system. This command allows us to simulate the restrictions in the communication between processes by choosing the bandwidth and latency that we need. This simulation, allows us to evaluate the impact of the number of rounds and the data sent of each protocol execution. Finally, for all of our experiments, we consider a bit-length of 64, which means that we are allowed to work with 64-bit integers. This amount of bits is enough for the practical uses of our implementation in real-world tasks.

5.1 Performance of the binary computation in the preamble

In Section 3.1, we presented an alternative to compute the matrix t using a binary domain of computation. Then, using daBits, we perform a domain conversion to compute the arithmetic section. The goal of this experiment is to evaluate the performance of the computation of the preamble in a binary domain with respect to a traditional implementation as presented in Algorithm 1 using an arithmetic domain. We need to make clear that we are including the computation of the arithmetic part in the results and it is computed as it

[1] https://aws.amazon.com/ec2/instance-types/c6a/
[2] https://man7.org/linux/man-pages/man8/tc.8.html

is presented in Algorithm 1 for both implementations. In that way, we focus on measuring the improvement just in the computation of the preamble. Given that the preamble is computed in $\mathbb{Z}_2$ in our proposal, the measures of this experiment also include the resources spent in the domain conversion.

For this experiment, we consider two DNA chains of length 1,000. The protocols that we are considering are SPD$\mathbb{Z}_{2^k}$ and its corresponding version with passive security called Semi2^k. The last protocol is similar to SPD$\mathbb{Z}_{2^k}$ but removing all the mechanisms to ensure malicious security like authentication using MACs.

The results of the experiment are presented in Table 5-1. From the results, we can see that our approach reduces the data sent during the protocol execution by approximately 11% for passive security, and a reduction of approximately 22.5% for the case of active security. One of the reasons for this reduction is that when we execute the preamble in an arithmetic domain, we need to use the full bit-length (in our case, 64 bits) to represent only one nucleotide without considering the number of security bits used for the protocol to guarantee privacy and authenticity. Instead, our binary approach just uses 2 bits to represent a nucleotide. This reduction in the representation contributes to the reduction in the transmitted information because the shares that go across the network have fewer bits. Despite this reduction in the data sent, the execution time is not reduced significantly. We can see that the reduction in the execution time varies between 1-8% when we use a LAN. However, this reduction in the execution time can be more significant in network architectures with lower bandwidth where the reduction in the data sent has a bigger effect. In conclusion, although the reduction in the execution time is not significant, the computation of the preamble in a binary domain has benefits considering the data sent alone, which makes it a good choice in networks with low bandwidth.

Network	**Security**	**Data sent [MB] (Naive)**	**Data sent [MB] (Binary)**	**Time [s] (Naive)**	**Time [s] (Binary)**
No limit	Passive	10,509	9,349.2	627.7	480.7
	Active	644,677	499,383	5,351.7	4,215.4
LAN	Passive	10,509	9,349.2	12,224.1	12,087.7
	Active	644,677	499,383	21,785.5	19,919

Table 5-1: Data sent and execution time for the preamble optimization.

5.2 The effect of changing τ

As we saw in Section 3.2, the size of the $(\tau + 1)$-box has repercussions in the number of rounds, the number of multiplications, and the number of comparisons. The larger the τ, the lower the number of rounds, and the larger the number of multiplications and comparisons. This experiment intends to evaluate such trade-off in terms of the data sent and the execution time of the protocol.

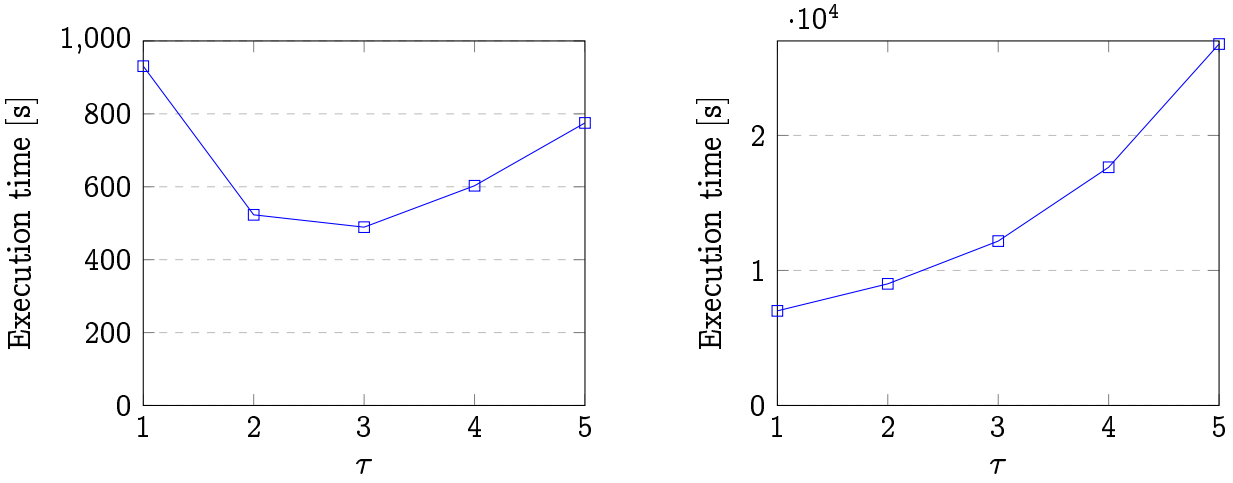
In this experiment, we are considering DNA chains of length 1,020. We evaluate the performance of our solution using protocols that are secure against both passive and active adversaries. For the passive setting, we use the Semi2^k protocol and for the active setting, we use the SPD$\mathbb{Z}_{2^k}$ protocol. We execute each protocol for $\tau \in \{1, 2, 3, 4, 5\}$, and for each value of τ, we measure the data sent and the execution time. For each combination of parameters, we have two types of measures. For the first type, we execute the experiment as described above taking into account both the pre-processing and the online phase. For the second type, we execute the experiment taking into account just the online phase. This allows us to identify the trade-off of our solution in more detail for each stage of the protocol execution.

Table 5-2 shows the results of the experiment whose measures include both the pre-processing and the online phase, and the network has no limitations. Also, in Figure 5-1, we present a graphical representation of information in the table. In the case of passive security, we can see that if we choose $\tau = 2$ or $\tau = 3$, there is an improvement in the execution time with respect to $\tau = 1$. However, if we choose a value of τ greater than 3, the execution time starts to increase. This is because, according to our theoretical analysis, if we increase τ, we obtain a lower number of rounds, however, the number of multiplications and comparisons increase exponentially. But remember that to compute multiplications and comparisons we need to generate multiplication triples and edaBits, and both of them require communication and resources. Therefore, we can find a trade-off between the round reduction, and the correlated randomness needed in the protocol. So, there is a moment where the round reduction is not enough to reduce the overall execution time because the computational resources spent in the comparisons and multiplications begin to have a bigger impact. In the case of active security, there is an increasing execution time because the mechanisms to guarantee security against malicious adversaries put an additional overhead to the communication and computation, and such mechanisms are particularly widely applied in the pre-processing stage.

To give a more detailed idea of the computational resources spent in the protocol, Table 5-3 shows the result of the experiment considering just the online phase and without any network limitation. In Figure 5-2, we present its corresponding graphical representation

Pre-processing and online - Without network limit			
Security	τ	**Data sent [MB]**	**Execution time[s]**
Passive	1	9,727.3	930.8
	2	10,129.9	523.0
	3	13,669.8	489.1
	4	21,222.9	602.7
	5	35,486.9	775.1
Active	1	519,759	7,007.5
	2	768,422	9,009.1
	3	1.13×10^6	12,175.7
	4	1.79×10^6	17,643.2
	5	3.02×10^6	26,769.6

Table 5-2: Effect of changing τ considering both the pre-processing and the online phase using a network with no limitations.



(a) Passive security.

(b) Active security.

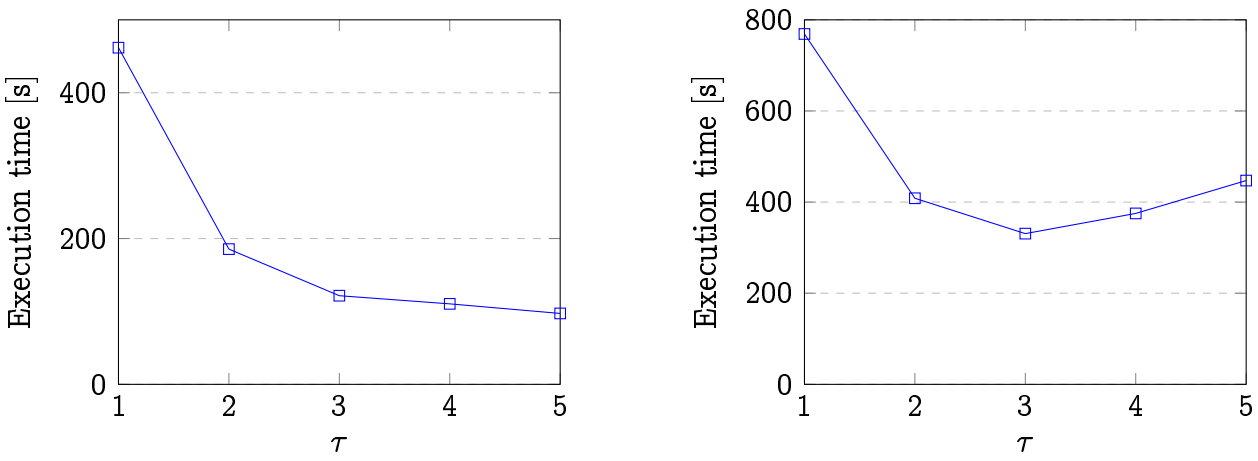
Figure 5-1: Effect of the box size for the experiment without network limits considering both the preprocessing and online phases.

for the execution time as the value of τ changes. Interpreting the results from this table, we observe that for passive security, the execution time is decreasing. Also, we can see that as long as the τ increases, the change in the execution time becomes smaller. This behavior is because the round reduction of our proposal has a noticeable effect in the online phase, however, increasing τ also increases the number of multiplications and comparisons which consumes resources in the online phase. Notice that the minimal number of rounds is reached when $\tau = \min\{n, m\}$, where n and m are the lengths of the chains, obtaining the biggest $(\tau+1)$-box. However, choosing this τ also makes the protocol reach the largest number of multiplications and comparisons, which increases the complexity of the overall protocol. The case of active security shows a very different result. In that case, we observe that the execution time starts to increase for $\tau > 3$. Contrary to the passive security case, to guarantee active security, the online phase has additional work given by the MAC checking. According to the specifications of the SPD$\mathbb{Z}_{2^k}$ protocol, when the protocol computes a multiplication, some values that are secret-shared are opened, but given that the adversary is active, these values need to be checked for correctness using the MAC batch checking. However, the computational complexity of this checking process increases with the number of multiplications, because the match checking needs to compute a linear combination with as many terms as values that need to be checked. According to our estimations, the number of multiplications increases exponentially as τ does. Considering that the communication is very fast given that there is no limitation to the network and that the batch checking has a low communication complexity, we conclude that the increasing trend in the graph is explained by the computational complexity induced by the online processing of the multiplications.

Online phase - Without network limit			
Security	τ	**Data sent [MB]**	**Execution time [s]**
Passive	1	4,020.1	461.8
	2	1,579.3	185.5
	3	1,012.7	121.6
	4	942.6	110.4
	5	1,115.6	97.3
Active	1	795.9	768.9
	2	967.6	408.4
	3	1.37×10^3	330.8
	4	2.15×10^3	375.1
	5	3.61×10^3	447.2

Table 5-3: Effect of changing τ considering just the online phase using a network with no limitations.

Now, given that MPC protocols, and in particular the secret sharing schemes, are sensitive



(a) Passive security.

(b) Active security.

Figure 5-2: Effect of the box size for the experiment without network limits and considering just the online phase.

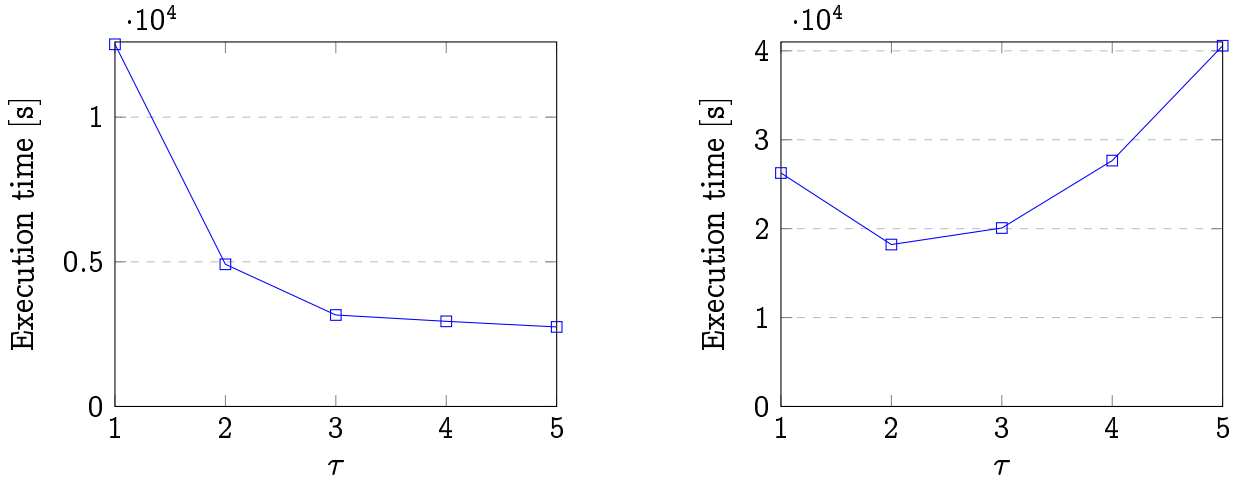
to the conditions imposed by the network, we perform the experiments simulating a LAN network as it was discussed at the beginning of this chapter. Table 5-4 shows the results of the experiments considering both the pre-processing and online phases using a LAN. The corresponding graphical representation of the execution time as τ increases is shown in Figure 5-3. For the case of passive security, we notice that the execution time is decreasing for the values of τ considered in the experiments, contrary to the effect that we saw in Figure 5-2 where there are no limitations to the network. This behavior allows us to conclude that, for the passive case, the complexity associated with the number of rounds is predominant, and the decreasing trend in the graph is explained by the decreasing of the number of rounds as τ increases. Notice that in this experiment, we are considering also the pre-processing phase, where the communication and computational complexity increases as τ does. This increasing in the complexity is shown in the curve which is flattening as τ increases.

On the other hand, for the case of active security, we observe the effect of the trade-off between the round reduction and the number of multiplications and comparisons is more clear. For $\tau = 2$ the execution time shows an improvement because the saving in the number of rounds compensates for the complexity induced by the multiplications and comparisons. However, for $\tau \geq 3$, the exponential increase of the multiplications and comparisons makes the execution time increase.

Finally, Table 5-5 presents the results of the experiment taking into account just the online phase using a LAN. This case is interesting because as τ increases in the considered values, the execution time decreases for both passive and active security, contrary to the other

Pre-processing and online - LAN			
Security	τ	**Data sent [MB]**	**Execution time [s]**
Passive	1	9,727.3	12,518.4
	2	10,129.9	4,914.1
	3	13,669.8	3,161.6
	4	21,222.9	2,942.7
	5	35,486.9	2,749.0
Active	1	519,759	26,257.6
	2	768,422	18,210.2
	3	1.13×10^6	20,071.6
	4	1.79×10^6	27,654.4
	5	3.02×10^6	40,573.4

Table 5-4: Effect of changing τ considering both the pre-processing and online phase using a LAN.



(a) Passive security.

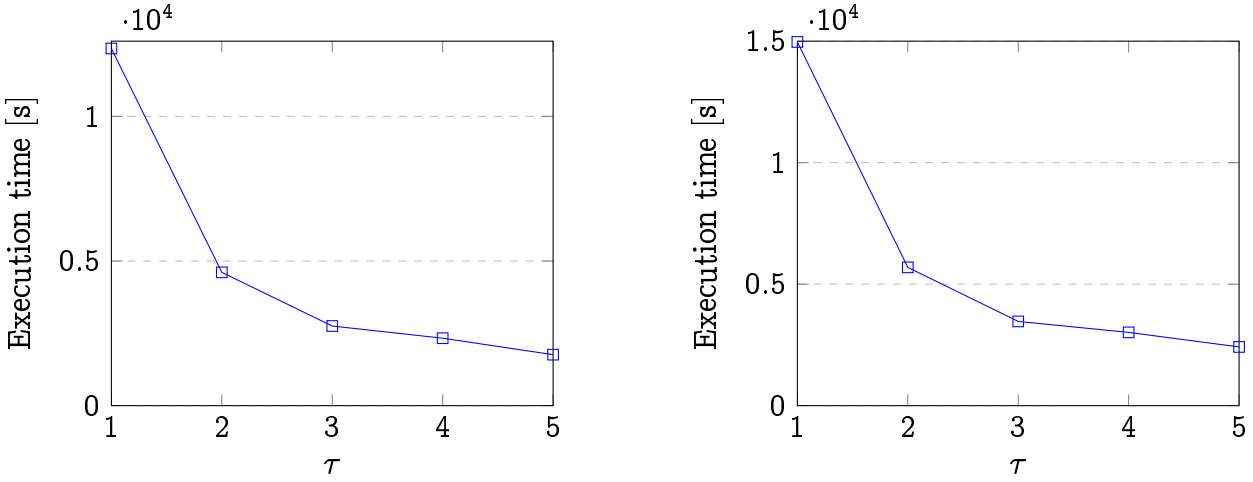
(b) Active security.

Figure 5-3: Effect of the box size for the experiment using LAN and considering both the pre-processing and online phase.

cases where at least one of the situations, active or passive, showed an eventual dominance of the number of comparisons and multiplications in the execution time. This means that, in this case for both active and passive security, the decreasing of rounds has a higher impact on the execution time due to the presence of the latency.

Online phase - LAN			
Security	τ	**Data sent [MB]**	**Time [s]**
Passive	1	4,020.1	12,346.2
	2	1,579.3	4,608.9
	3	1012.7	2,753.7
	4	942.6	2,332.6
	5	1,115.6	1,763.9
Active	1	795.9	14,967.3
	2	967.6	5,692.0
	3	1.37×10^3	3,467.0
	4	2.15×10^3	3,015.8
	5	3.61×10^3	2,417.6

Table 5-5: Effect of the box size for the experiment using LAN and considering just the online phase.



(a) Passive security.

(b) Active security.

Figure 5-4: Effect of the box size for the experiment using LAN and considering just the online phase.

Looking at the experiments globally, in an intuitive way, we can think of the execution time as a function that is the sum of other two functions that overlaps: the execution time induced by the number of rounds and the execution time induced by the number of comparisons and multiplications. The predominance of one of both functions will depend

on the computational context in which the protocol is executed, namely, the local computational power, the network architecture, and its limitations, the value of τ chosen, the length of the chains, etc. However, according to our estimations, the number of rounds decreases as an inverse linear function of τ while the number of multiplications increases exponentially as a function of τ. This means that in any case and for a value of τ large enough, the execution time will begin to increase. It could be possible that for the first values of τ the execution time decreases, but after a certain point, the computational and communication complexity induced by the multiplications will take more importance and will predominate in the execution time making it increase. The moment at which this increase begins will depend on the characteristics of the context as we explained before. Although this trade-off is present, the experiments reveal a positive impact of our approach given that the majority of the experiment shows at least one value of τ in which the execution time is better than the one where $\tau = 1$ (which corresponds to the traditional approach of the Wagner-Fischer algorithm). The key is to find the appropriate value of τ in which the saving in the number of rounds compensates for the complexity induced by the comparisons and multiplications.

There is one case that does not show improvement which is shown in Table 5-2 for a malicious adversary. Here, the execution time is always increasing which appears to be a negative aspect of our proposed approach. However, the case of no limitations in the network is not a realistic scenario in the practical use of MPC. In practical scenarios, there will be a latency and a bandwidth involved and it tends to have constraints similar to the LAN conditions proposed in this work.

5.3 Comparison between garbled circuits and secret-sharing

As we mentioned in Section 1.1, one of the methods commonly used to evaluate the edit distance securely in MPC is garbled circuits (GC). In this experiment, we will compare the performance of our implementation using garbled circuits and using secret-sharing schemes with domain in $\mathbb{Z}_{2^k}$.

In this experiment, due to the memory and processing constraints of the AWS EC2 instance that we were using, we are taking both DNA chains with a length of 210 nucleotides. As in the previous experiments, we are also using a bit-length of 64 and we consider both a network with no limitations and a LAN. Here we are using two types of implementations: for garbled circuits, we are using $\tau = 1$, and for the secret-sharing schemes we are using $\tau = 3$. This difference between the values of τ is because the garbled circuits have a

better performance when $\tau = 1$. In a preliminary exploration, we tried to execute the experiments using values of $\tau > 1$, but the performance got worse as the value of τ increased. The reason for this is that, as τ increases, the number of formulas inside the minimum function increases exponentially, which means that the number of addition gates increases too. Remember that for GC-based protocols, the integer additions are expensive because they need to be expressed as binary circuits composed of several AND gates that depends on the number of bits used in the integer representation. This produces that the methods based on GCs need to garble a higher number of tables which increases the number of computations and the amount of data that needs to be sent through the network. Moreover, the more recent GC-based methods have a constant number of rounds, and in such case, our method which is designed to reduce the number of rounds have no benefit. This means that using a $\tau > 1$ in our algorithm with a GC-based protocol on top only adds more computational complexity to the computation and does not have any benefit.

In Table 5-6, we present the results of the experiment. Notice that when the network has no limitations, the protocols based on secret-sharing perform better than GC-based protocols. In particular, notice that the secret-sharing schemes have approximately 57-99% fewer data sent than the GC-based protocols. The reason for this is that the edit distance solution has a high number of arithmetic operations, which means that the number of gates that the GC-based methods need to garble and send through the network is high. Although the difference in the data sent is significant, it is a well-known fact that the drawback of GC-based protocols is the amount of data sent, but modern implementations have constant-round communication. On the contrary, the secret-sharing schemes have a high amount of communication rounds, but the data sent and the amount of local computations is lower compared to GC-based protocols.

Network	Security	Protocol	Time [s]	Data sent [MB]
No network limit	Passive	Yao's GC	9.9	1,370.3
		Semi2^k	8.4	581.0
	Active	BMR-MASCOT	25,072.5	8.32×10^6
		SPD$\mathbb{Z}_{2^k}$	3.56×10^2	47,865.5
LAN	Passive	Yao's GC	10.4	1,370.3
		Semi2^k	146.9	581.0
	Active	BMR-MASCOT	3.72×10^4	8.32×10^6
		SPD$\mathbb{Z}_{2^k}$	9.91×10^2	47,865.5

Table 5-6: Comparison between GC and secret-sharing schemes.

If we look at the results using LAN, the results are different from the previous ones for the passive case. These results show that the execution time of Yao's GC protocol is lower than the Semi2^k protocol by an order of magnitude. This turnaround is due to the presence

of latency. Given that the number of rounds in Yao's GC is very low, the latency does not have a major impact on the execution time. On the other hand, the Semi2^k protocol has a higher number of rounds, and therefore the latency has a major contribution to the execution time.

Notice that both in the network with no limit and in LAN, the performance of BMR is two orders of magnitude slower than the secret-sharing schemes. The reason for this is that the Wagner-Fischer algorithm has a heavy arithmetic component, namely, the arithmetic section. In such section, we require to perform mostly additions and comparisons between 64-bit integer numbers. The comparisons are tasks that can be done efficiently in GC-based protocols. But addition operations require the computation of AND gates. In the case of BMR, the underlying MPC protocol to compute the offline phase is MASCOT [KOS16], which uses MACs to ensure security against malicious adversaries. So, garbling an AND gate needs to compute multiplications with the assistance of the MASCOT protocol which requires the generation of multiplication triples and puts an additional overhead. This overhead produced by the additions does not appear in secret-sharing schemes because the additions do not need communication. More interestingly, additions do not heavily hurt the performance of the garbling process in Yao's GC because it does not need any communication between parties.

In conclusion, using our approach in a realistic context like a LAN in the presence of an active adversary, it is recommended to use protocols based on secret-sharing which perform significantly better given the high amount of arithmetic operations that need to be computed in the Wagner-Fischer algorithm.

5.4 Comparison between protocols in $\mathbb{Z}_p$ and $\mathbb{Z}_{2^k}$

At the beginning of this work, we presented a research question asking if the protocols based on rings of the form $\mathbb{Z}_{2^k}$ are a good choice among the protocols based on secret-sharing schemes. Our hypothesis is that protocols in $\mathbb{Z}_{2^k}$ are the best suited for computing the edit distance given that the specification of these protocols is more compatible with the current CPU architectures as mentioned in [Dam+19]. Also, given that the Wagner-Fischer algorithm requires a high number of comparisons, we expect these operations to be performed faster than secret-sharing schemes based on fields due to the same reasons expressed above. The goal of this experiment is to empirically verify our hypothesis by comparing secret sharing schemes based on fields with protocols based on rings.

For this experiment, we are considering DNA chains of length 1,020 and we are using our implementation with $\tau = 3$. For the case of rings, we are using SPD$\mathbb{Z}_{2^k}$ which has active

security, and Semi2^k as its corresponding passive secure version. For the case of fields, we are using the protocol MASCOT [KOS16] to guarantee active security and Semi as its corresponding passive secure version. The idea behind Semi is similar to Semi2^k so that Semi is the protocol that results from MASCOT after removing all the mechanisms that guarantee active security. The idea of this experiment is to compute the edit distance using the same implementation for both passive and active security, using both with no network limitations and simulating a LAN. It is important to mention that both implementations use daBits and edaBits. Also, in this experiment, we will consider both the pre-processing and the online phase for each execution of the protocols.

The results of this experiment are presented in Table **5-7**. Notice that except for the case of passive security with no network limitations, the protocols based on rings have a better performance not only in the execution time but in the data sent. The reason behind these results is that protocols in $\mathbb{Z}_{2^k}$ have an advantage using the bit representation. To show this advantage, we will mention two concrete examples. The first example is the daBit checking presented in [Esc+20, Figure 16]. In Step 1(c), for the case of protocols in $\mathbb{Z}_p$, it is needed to get shares of random bits of the form $[\![c_i]\!]_p$ which need to be generated in the pre-processing phase. To check one daBit it is needed to generate s of such random bits, where s is the security parameter of the protocol. On the contrary, this same step does not need any random bit generation for the case of $\mathbb{Z}_{2^k}$. This generation of random bit produces an additional overhead in the pre-processing phase for protocols in $\mathbb{Z}_p$ with respect to protocols in $\mathbb{Z}_{2^k}$. Another example where the protocols in $\mathbb{Z}_{2^k}$ have an advantage is in the edaBit generation. According to the original proposal presented in [Esc+20, Figure 3] in Step 4, when the ring is $\mathbb{Z}_{2^k}$ we do not need to convert the binary shares beyond the k-th position into arithmetic shares to avoid the overflow modulo 2^k, instead we can discard such bits. On the contrary, for the case of $\mathbb{Z}_p$, we need to subtract the excess in the case of an overflow and to do this, we need to convert the bits shares beyond the k-th position from binary shares into arithmetic shares, which requires one daBit per bit conversion. This later advantage of protocols in $\mathbb{Z}_{2^k}$ is of high importance because, as we have seen before, the edaBits are needed to perform comparisons, and the number of comparisons increases with τ, which means that increasing the value of τ will increase the number of edaBits that need to be generated, which add an overhead on the protocols based in $\mathbb{Z}_p$ given the reasons mentioned above.

These experiments along with the theoretical interpretation show that, for the edit distance computation, it is more efficient to consider protocols in $\mathbb{Z}_{2^k}$ instead of protocols in $\mathbb{Z}_p$ due to the presence of a high number of comparisons. As we have seen, the protocols in $\mathbb{Z}_{2^k}$ have an advantage given his design which reduces the number of computations needed to perform such comparisons.

Network	Security	Protocol	Data sent [MB]	Time [s]
No network limit	Passive	Semi2^k	13,669.8	489.1
		Semi	25,393.7	340.6
	Active	SPD$\mathbb{Z}_{2^k}$	1.13×10^6	12,175.7
		MASCOT	3.22×10^6	26,382.9
LAN	Passive	Semi2^k	13,669.8	3,161.6
		Semi	25,393.7	3,786.0
	Active	SPD$\mathbb{Z}_{2^k}$	1.13×10^6	20,071.6
		MASCOT	3.22×10^6	43,811.4

Table 5-7: Comparison between protocols using fields with protocols using rings.

5.5 Comparing our solution with protocols based on homomorphic encryption

As we have seen in Section 1.1, the edit distance problem has been computed securely using homomorphic encryption (HE). The purpose of this section is to make a performance comparison using the results obtained from the previous experiments with the experimental results found in [CKL15] and [Zhe+19].

The first work that we will take into account is the work of Cheon et al. presented in [CKL15]. They consider experiments with DNA chains with length 8 and using a security parameter of 80 bits. According to the results in Table 6 in that paper, their method spends 27.54 seconds in the key generation and 16.45 seconds in the encryption process. In their report, they state that it takes 27.5 seconds to obtain the edit distance using the Halevi-Shoup library (HElib) [HS20] along with the techniques presented in [GHS12]. In our case, considering both the pre-processing and the online phase, using $\tau = 2$ and a LAN, we can obtain the edit distance of chains with length 8 in 0.47 seconds using Semi2^k protocol for passive security and in 16.27 seconds using the SPD$\mathbb{Z}_{2^k}$ protocol for active security. Additionally, Cheon et al. compute some estimations of the performance for their method for DNA chains with lengths up to 100. For the case of DNA chains of length 100 and a security parameter of 62 bits, they estimate that the edit distance can be computed in 1 day and 5 hours. In our case, we saw that using a LAN and the protocol SPD$\mathbb{Z}_{2^k}$, our method can compute the edit distance in 16.51 minutes. These results show that our MPC approach has a better performance than the reported results presented by Cheon et al. In terms of security, a security parameter of 62 bits for HE is not adequate for the current standards of cryptography. Meanwhile, the implementation of the protocol SPD$\mathbb{Z}_{2^k}$ uses 64 bits for statistical security, which is enough to reach a good level of security.

The work of [Zhe+19] also computes the edit distance using HE, however, they allow one of the parties to learn the chain of the other party. This fact allows one of the parties to compute the edit distance between blocks of DNA chains in the clear. Instead, we guarantee privacy for the DNA chains for both parties. Also, our approach does not reveal any information about the DNA chains in the intermediate steps of the computation. This difference between both approaches implies also a difference between the running times where the approach of Zheng et al. is faster than ours. Zheng et al. report that the data owner spends between 5-30 milliseconds approximately to encrypt the genomic data. Also, they report that the cloud server, which is in charge of computing the homomorphic operations, has a runtime for edit distance query between 80-95 milliseconds approximately. our implementation using passive security and a LAN takes 11.92 seconds. This shows that our approach is slower but this is completely expected due to the differences in the security models.

6 Conclusions

In this work, we presented an approach to computing the edit distance securely via the Wagner-Fischer algorithm using protocols based on secret-sharing schemes. The proposed method divides the edit distance computation into two parts: the first part is computed in a binary domain using the protocol Tinier, and the second part is computed in an arithmetic domain using the protocol SPD$\mathbb{Z}_{2^k}$. The conversion between the two domains is done using daBits, and the comparisons required to compute the arithmetic part are computed using edaBits.

For the first part of the computation, we presented an equality test of nucleotides based on binary operations. The experiments reveal that this modification reduces the data sent in the protocol execution by 11% for passive security and 22.5% for active security compared to a naive approach where the nucleotides are encoded as integers. Although the improvement in the data sent is significant, the improvement in the execution time varies between 1-8% in a LAN.

For the arithmetic part, we break down the computation of the edit distance matrix into the computation of the border of sub-boxes inside the matrix whose side length is $\tau + 1$. Instead of computing each position of the border as a minimum of three numbers as in the traditional iteration in the Wagner-Fischer algorithm, we compute them by calculating the minimum of a longer list of numbers. We found that the number of rounds is inversely proportional to τ, but the number of multiplications and comparisons increases exponentially with respect to τ. This means that large values of τ increase the overall execution time of the protocol exponentially. However, for small values of τ we can reach a reduction in the execution time of approximately 78% for passive security and 64% for active security for certain settings compared to a traditional implementation of the Wagner-Fischer algorithm.

As a part of the optimization in the arithmetic section, we use graph theory techniques to design an algorithm to build lists of integer numbers that are arguments of the minimum functions to compute one sub-box. Also, we prove that our algorithm returns the correct list of integers and that it returns the list with minimal length to compute the edit distance correctly. Finally, we analyze the complexity of our algorithm using Delanoy numbers

concluding that our method generates lists of length $O(\tau \cdot 2^{3\tau})$.

We compared our solution with other techniques like garbled circuits (GC) and homomorphic encryption (HE). The results show that using LAN, Yao's GC has an execution time with an order of magnitude less than the passive protocol based on secret-sharing. However, for the case of active security, the secret-sharing schemes have two orders of magnitude less in both the execution time and the data sent compared to using the corresponding active secure protocol based on GC. For the case of HE, when we compare the results of [CKL15] with our implementation using a LAN, they estimate that their method computes the edit distance in 1 day and 5 hours for chains of length 100, while our method using active security takes only 16.51 minutes.

We designed an experiment to compare our proposal using secret-sharing protocols whose computation domain is $\mathbb{Z}_{2^k}$ with protocols whose underlying domain are fields of the form $\mathbb{Z}_p$, for p prime. The experiment reveals that for a LAN, our implementation performs the best in protocols based on rings. When there are no network limitations, our implementation performs best only in active security.

This work shows that, for the case of edit distance, secret-sharing schemes have a competitive performance to compute functionalities that require bit-wise operations compared to Yao's GC, maintaining its good performance in arithmetic operations and the possibility to perform mixed circuit computations efficiently. Moreover, this work shows that secret-sharing schemes are more efficient than GC for computing the edit distance when considering active adversaries. Considering the findings in this work, we can answer the Research Questions 1 and 2 affirmatively.

As a result of this work, we identify two research problems for future work. The first one is to find an appropriate value of τ to obtain the best overall execution time given parameters of the environment in which the protocol is executed like bandwidth, latency, chain lengths, and the local computational power. The second problem is to generalize the graph theory techniques presented in Section 4.1 to other dynamic programming problems. This could improve the performance of a secure implementation using MPC compared to a naive implementation of the dynamic programming algorithm as it is used when the inputs are in plaintext.

Finally, the main results of this work were published in the International Conference on Cryptology and Information Security in Latin America, LATINCRYPT 2023 [VCA23]. In that work, we modified the bit length in the experiments to 16 bits instead of 64 bits to reduce the amount of data sent in the protocol execution which also improves the overall running time of the protocol. Specifically, for a LAN architecture, the best choice of τ has a reduction in the execution time of 81% for passive security and 54%

for active security compared to a baseline implementation using $\tau = 1$. Comparing the secret-sharing-based solutions with garbled circuit solutions, the experiments show that the former sends between 67% to 99% less data than the latter. Moreover, the comparison between HE-based solutions and secret-sharing-based solutions shows that in the latter, the edit distance can be computed in 96.69 seconds in LAN using the SPD$\mathbb{Z}_{2^k}$ protocol, which increases the breach in the running time presented in Section 5.5 of this work.

Bibliography

[AAM17] Md Momin Al Aziz, Dima Alhadidi, and Noman Mohammed. "Secure approximation of edit distance on genomic data". In: *BMC Medical Genomics* 10.2 (July 2017), p. 41. ISSN: 1755-8794. DOI: 10.1186/s12920-017-0279-9. URL: https://doi.org/10.1186/s12920-017-0279-9.

[Aly+19] Abdelrahaman Aly et al. "Zaphod: Efficiently Combining LSSS and Garbled Circuits in SCALE". In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC'19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 33–44. ISBN: 9781450368292. DOI: 10.1145/3338469.3358943. URL: https://doi.org/10.1145/3338469.3358943.

[Ash+18] Gilad Asharov et al. "Privacy-Preserving Search of Similar Patients in Genomic Data". In: *Proceedings on Privacy Enhancing Technologies* 2018.4 (Aug. 2018), pp. 104–124. DOI: 10.1515/popets-2018-0034. URL: https://doi.org/10.1515/popets-2018-0034.

[BCP03] Emmanuel Bresson, Dario Catalano, and David Pointcheval. "A Simple Public-Key Cryptosystem with a Double Trapdoor Decryption Mechanism and Its Applications". In: *Advances in Cryptology - ASIACRYPT 2003*. Ed. by Chi-Sung Laih. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 37–54. ISBN: 978-3-540-40061-5.

[BHR12] 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. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 784–796. ISBN: 9781450316514. DOI: 10.1145/2382196.2382279. URL: https://doi.org/10.1145/2382196.2382279.

[BMR90] D. Beaver, S. Micali, and P. Rogaway. "The Round Complexity of Secure Protocols". In: *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*. STOC '90. Baltimore, Maryland, USA: Association for Computing Machinery, 1990, pp. 503–513. ISBN: 0897913612. DOI: 10.1145/100216.100287. URL: https://doi.org/10.1145/100216.100287.

[BR93] Mihir Bellare and Phillip Rogaway. “Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols”. In: *Proceedings of the 1st ACM Conference on Computer and Communications Security*. CCS '93. Fairfax, Virginia, USA: Association for Computing Machinery, 1993, pp. 62–73. ISBN: 0897916298. DOI: 10.1145/168588.168596. URL: https://doi.org/10.1145/168588.168596.

[BWY21] Bonnie Berger, Michael S. Waterman, and Yun William Yu. “Levenshtein Distance, Sequence Comparison and Biological Database Search”. In: *IEEE Transactions on Information Theory* 67.6 (2021), pp. 3287–3294. DOI: 10.1109/TIT.2020.2996543.

[CDN15] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015. DOI: 10.1017/CBO9781107337756.

[CKL15] Jung Hee Cheon, Miran Kim, and Kristin Lauter. “Homomorphic Computation of Edit Distance”. In: *Financial Cryptography and Data Security*. Ed. by Michael Brenner et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 194–212. ISBN: 978-3-662-48051-9.

[Cra+18] Ronald Cramer et al. “SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority”. In: *Advances in Cryptology – CRYPTO 2018*. Ed. by Hovav Shacham and Alexandra Boldyreva. Cham: Springer International Publishing, 2018, pp. 769–798. ISBN: 978-3-319-96881-0.

[Dam+12] Ivan Damgård et al. “Multiparty Computation from Somewhat Homomorphic Encryption”. In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 643–662. ISBN: 978-3-642-32009-5.

[Dam+19] Ivan Damgård et al. “New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1102–1120. DOI: 10.1109/SP.2019.00078.

[DEK21] Anders Dalskov, Daniel Escudero, and Marcel Keller. “Fantastic Four: Honest-Majority Four-Party Secure Computation With Malicious Security.” In: *USENIX Security Symposium*. 2021, pp. 2183–2200.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. “ABY - A framework for efficient mixed-protocol secure two-party computation”. In: *Network and Distributed System Security Symposium*. 2015.

[DZ13] Ivan Damgård and Sarah Zakarias. “Constant-Overhead Secure Computation of Boolean Circuits using Preprocessing”. In: *Theory of Cryptography*. Ed. by Amit Sahai. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 621–641. ISBN: 978-3-642-36594-2.

[DZ16] Tamara Dugan and Xukai Zou. “A Survey of Secure Multiparty Computation Protocols for Privacy Preserving Genetic Tests”. In: *2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*. 2016, pp. 173–182. DOI: 10.1109/CHASE.2016.71.

[EKR18] David Evans, Vladimir Kolesnikov, and Mike Rosulek. “A Pragmatic Introduction to Secure Multi-Party Computation”. In: *Foundations and Trends in Privacy and Security* 2.2-3 (2018), pp. 70–246. ISSN: 2474-1558. DOI: 10.1561/3300000019. URL: http://dx.doi.org/10.1561/3300000019.

[EN14] Yaniv Erlich and Arvind Narayanan. “Routes for breaching and protecting genetic privacy”. In: *Nature Reviews Genetics* 15.6 (June 2014), pp. 409–421. ISSN: 1471-0064. DOI: 10.1038/nrg3723. URL: https://doi.org/10.1038/nrg3723.

[Esc+20] Daniel Escudero et al. “Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits”. In: *Advances in Cryptology – CRYPTO 2020*. Ed. by Daniele Micciancio and Thomas Ristenpart. Cham: Springer International Publishing, 2020, pp. 823–852. ISBN: 978-3-030-56880-1.

[Esc21] Daniel Escudero. “Multiparty Computation over $\mathbb{Z}/2^k\mathbb{Z}$”. University of Aarhus, Nov. 2021.

[Fre+15] Tore Kasper Frederiksen et al. “A Unified Approach to MPC with Preprocessing Using OT”. In: *Advances in Cryptology – ASIACRYPT 2015*. Ed. by Tetsu Iwata and Jung Hee Cheon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 711–735. ISBN: 978-3-662-48797-6.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. “Homomorphic Evaluation of the AES Circuit”. In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 850–867. ISBN: 978-3-642-32009-5.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. “How to Play ANY Mental Game”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. STOC ’87. New York, New York, USA: Association for Computing Machinery, 1987, pp. 218–229. ISBN: 0897912217. DOI: 10.1145/28395.28420. URL: https://doi.org/10.1145/28395.28420.

[HS20] Shai Halevi and Victor Shoup. *Design and implementation of HElib: a homomorphic encryption library*. Cryptology ePrint Archive, Paper 2020/1481. https://eprint.iacr.org/2020/1481. 2020. URL: https://eprint.iacr.org/2020/1481.

[IUS09] International University in Germany, Universiteit Technische Eindhoven, and SAP AG. *Secure Supply Chain Management*. Tech. rep. 2009.

[JKS08] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. "Towards Practical Privacy for Genomic Computation". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 2008, pp. 216–230. DOI: 10.1109/SP.2008.34.

[Kel20] Marcel Keller. "MP-SPDZ: A Versatile Framework for Multi-Party Computation". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020. DOI: 10.1145/3372297.3417872. URL: https://doi.org/10.1145/3372297.3417872.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. 2nd. Chapman & Hall/CRC, 2014. ISBN: 1466570261.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 830–842. ISBN: 9781450341394. DOI: 10.1145/2976749.2978357. URL: https://doi.org/10.1145/2976749.2978357.

[KY18] Marcel Keller and Avishay Yanai. "Efficient Maliciously Secure Multiparty Computation for RAM". In: *EUROCRYPT (3)*. Springer, 2018, pp. 91–124. DOI: 10.1007/978-3-319-78372-7_4.

[Lin+19] Yehuda Lindell et al. "Efficient Constant-Round Multi-party Computation Combining BMR and SPDZ". In: *Journal of Cryptology* 32.3 (July 2019), pp. 1026–1069. ISSN: 1432-1378. DOI: 10.1007/s00145-019-09322-2. URL: https://doi.org/10.1007/s00145-019-09322-2.

[LOS14] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. "Dishonest Majority Multi-Party Computation for Binary Circuits". In: *Advances in Cryptology – CRYPTO 2014*. Ed. by Juan A. Garay and Rosario Gennaro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 495–512. ISBN: 978-3-662-44381-1.

[Nie+12] Jesper Buus Nielsen et al. "A New Approach to Practical Active-Secure Two-Party Computation". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 681–700. ISBN: 978-3-642-32009-5.

[Nie07] Jesper Buus Nielsen. *Extending Oblivious Transfers Efficiently - How to get Robustness Almost for Free*. Cryptology ePrint Archive, Paper 2007/215. https://eprint.iacr.org/2007/215. 2007. URL: https://eprint.iacr.org/2007/215.

[Oes+21] Marie Oestreich et al. "Privacy considerations for sharing genomics data". en. In: *EXCLI Journal* (2021), pp. 1243–1260. DOI: 10.17179/EXCLI2021-4002.

[Oha20] Satsuya Ohata. "Recent Advances in Practical Secure Multi-Party Computation". In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E103A (10 Oct. 2020), pp. 1134–1141.

[RS10] Shantanu Rane and Wei Sun. "Privacy preserving string comparisons based on Levenshtein distance". In: *2010 IEEE International Workshop on Information Forensics and Security*. 2010, pp. 1–6. DOI: 10.1109/WIFS.2010.5711449.

[RW19] Dragos Rotaru and Tim Wood. "MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security". In: *Progress in Cryptology – INDOCRYPT 2019*. Ed. by Feng Hao, Sushmita Ruj, and Sourav Sen Gupta. Cham: Springer International Publishing, 2019, pp. 227–249. ISBN: 978-3-030-35423-7.

[ST19] Thomas Schneider and Oleksandr Tkachenko. "EPISODE: Efficient Privacy-PreservIng Similar Sequence Queries on Outsourced Genomic DatabasEs". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Asia CCS '19. Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 315–327. ISBN: 9781450367523. DOI: 10.1145/3321705.3329800. URL: https://doi.org/10.1145/3321705.3329800.

[SW81] T.F. Smith and M.S. Waterman. "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147.1 (1981), pp. 195–197. ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(81)90087-5. URL: https://www.sciencedirect.com/science/article/pii/0022283681900875.

[Tof07] Tomas Toft. "Primitives and Applications for Multi-party Computation". University of Aarhus, Mar. 2007, pp. 101–105.

[Ukk85] Esko Ukkonen. "Algorithms for approximate string matching". In: *Information and Control* 64.1 (1985). International Conference on Foundations of Computation Theory, pp. 100–118. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(85)80046-2. URL: https://www.sciencedirect.com/science/article/pii/S0019995885800462.

[VCA23] Hernán Vanegas, Daniel Cabarcas, and Diego F. Aranha. "Privacy-Preserving Edit Distance Computation Using Secret-Sharing Two-Party Computation". In: *Progress in Cryptology – LATINCRYPT 2023*. Ed. by Abdelrahaman Aly and Mehdi Tibouchi. Cham: Springer Nature Switzerland, 2023, pp. 67–86. ISBN: 978-3-031-44469-2.

[Wes20] Douglas B West. *Combinatorial mathematics*. Cambridge University Press, 2020.

[WF74] Robert A. Wagner and Michael J. Fischer. "The String-to-String Correction Problem". In: *J. ACM* 21.1 (June 1974), pp. 168–173. ISSN: 0004-5411. DOI: 10.1145/321796.321811. URL: https://doi.org/10.1145/321796.321811.

[Yao82] Andrew C. Yao. "Protocols for secure computations". In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 1982, pp. 160–164. DOI: 10.1109/SFCS.1982.38.

[Yao86] Andrew Chi-Chih Yao. "How to generate and exchange secrets". In: *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. 1986, pp. 162–167. DOI: 10.1109/SFCS.1986.25.

[ZH22] Ruiyu Zhu and Yan Huang. "Efficient and Precise Secure Generalized Edit Distance and Beyond". In: *IEEE Transactions on Dependable and Secure Computing* 19.1 (2022), pp. 579–590. DOI: 10.1109/TDSC.2020.2984219.

[Zha+19] Chuan Zhao et al. "Secure Multi-Party Computation: Theory, practice and applications". In: *Information Sciences* 476 (2019), pp. 357–372. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2018.10.024. URL: https://www.sciencedirect.com/science/article/pii/S0020025518308338.

[Zhe+19] Yandong Zheng et al. "Efficient and Privacy-Preserving Edit Distance Query Over Encrypted Genomic Data". In: *2019 11th International Conference on Wireless Communications and Signal Processing (WCSP)*. 2019, pp. 1–6. DOI: 10.1109/WCSP.2019.8927885.