



Leuvenshtein: Efficient FHE-based Edit Distance Computation with Single Bootstrap per Cell

Wouter Legiest[1], Jan-Pieter D'Anvers[1], Bojan Spasic[2], Nam-Luc Tran[2] and Ingrid Verbauwhede[1]

[1] COSIC, KU Leuven
firstname.lastname@esat.kuleuven.be
[2] Society for Worldwide Interbank Financial Telecommunication (Swift)

Abstract. This paper presents a novel approach to calculating the Levenshtein (edit) distance within the framework of Fully Homomorphic Encryption (FHE), specifically targeting third-generation schemes like TFHE. Edit distance computations are essential in applications across finance and genomics, such as DNA sequence alignment. We introduce an optimised algorithm that significantly reduces the cost of edit distance calculations called Leuvenshtein. This algorithm specifically reduces the number of programmable bootstraps (PBS) needed per cell of the calculation, lowering it from approximately 94 operations – required by the conventional Wagner-Fisher algorithm – to just 1. Additionally, we propose an efficient method for performing equality checks on characters, reducing ASCII character comparisons to only 2 PBS operations. Finally, we explore the potential for further performance improvements by utilising preprocessing when one of the input strings is unencrypted. Our Leuvenshtein achieves up to 278× faster performance compared to the best available TFHE implementation and up to 39× faster than an optimised implementation of the Wagner-Fisher algorithm. Moreover, when offline preprocessing is possible due to the presence of one unencrypted input on the server side, an additional 3× speedup can be achieved.

1 Introduction

The past 20 years have seen a major evolution of the global financial system. Financial crises, geopolitical events and economic growth have deeply impacted the direction that banking regulations have taken. One of the major policy shifts is in the direction of increasing transparency and sharing information between financial institutions. For instance, the G20 set targets for cross-border payments [FSB21] formulating objectives for enhancing cost, speed, financial inclusion and transparency in an effort to guarantee efficiency and seamlessness of an interconnected financial system. In line with the PSD2 directive [Cou15], which initiated the open-banking initiative in Europe, the EU has recently proposed the Financial Data Access framework which will grant consumers and SMEs to authorise third parties to access their data held by financial institutions. Information sharing among financial institutions is seen as paramount in the fight against financial

crime and money laundering [MA17], and is also expected to drive GDP gains of major economies [EC22]. In the previously mentioned initiatives, success can only be achieved if trust is built among all the actors. Trust can only be built if security and privacy are guaranteed in the exchange of information. While the directives are clear, the means to achieve a successful implementation are left to the actors proposing the services and products.

Considering the recent legislative proposal to make Euro payments instant [EC22], there is an obligation for payment providers to verify the match between the bank account number and the name of the beneficiary provided by the payer, as well as to alert the payer of possible mistakes or suspected fraud before the payment is made. In such applications, string similarity calculations are ubiquitous to provide robustness against spelling errors [AQA21]. One example is the edit distance, which calculates the minimum number of edits between two given strings.

Recently, technologies enabling computation on encrypted data, namely fully homomorphic encryption (FHE) have become more practical. Informally, FHE is an encryption scheme that enables a data owner to securely outsource computation on their data to an untrusted processing party, whereby the processing party computes over encrypted data and stays oblivious of the data and the computed result. The utility of FHE comes at a performance price, which can sometimes be prohibitive for time-critical applications. However, the recent advances in software [CJL+20] and hardware [GBP+23, vDTV23] implementation of the underlying FHE algorithms show promising performance results, encouraging the practitioners to start including FHE in production.

FHE schemes fall into two main categories: second-generation schemes like BGV [BGV14], BFV [FV12], and CKKS [CKKS17] support parallel computations on batched ciphertexts but have larger ciphertexts and slower bootstrapping. Third-generation schemes like TFHE [CGGI20] prioritise speed with smaller ciphertexts and faster bootstrapping, though they work on small, individual messages and require bootstrapping for nearly every operation. TFHE also allows any function to be applied 'for free' during bootstrapping, making it ideal for fast, logic-based encrypted computations.

In the context of string matching for financial applications, FHE could pose as an important enabler [Max21]. Instead of physically sharing their customer information, parties could compute the desired outcome of the matching operation avoiding data sharing in clear altogether. The institution sending the payment would encrypt the transaction data using a suitable FHE scheme, and send it to a third party which would compute the desired matching score in the encrypted domain and return the encrypted result. This result (and any intermediate variable) can only be decrypted by the payer institution. In the process, according to the principles of FHE, the third party provably does not learn anything about the transaction data, and the institution sending the payment does not learn anything about the customers of the institution receiving the payment.

Another interesting application of approximate string matching is in secure and privacy-preserving DNA analysis. Approximate string matching is essential

in DNA analysis, where genetic sequences must be matched while allowing for slight discrepancies due to mutations. This flexibility is vital for detecting similar sequences that may vary because of natural mutations or sequencing errors.

Cheon et al. [CKL15] provided the first edit distance algorithm in the context of somewhat homomorphic encryption. They develop both equality check and min functions and use this to build up edit distance calculation. They also give a thorough analysis of the homomorphic depth of their solution. Their methods were later generalised by Vanegas et al. [VCA23], elaborated for an MPC context. Later, Aziz et al. [AAM17], Asharov et al. [AHLR18] and Zheng et al. [ZLS+19] proposed an approximation of the edit distance for genome analysis for fully homomorphic encryption. All the above techniques are based on second-generation FHE schemes and are built around arithmetic ciphertexts.

Edit distance calculations for third-generation FHE schemes are less well-researched. Recently, ZAMA showed an edit distance calculation for TFHE as a demonstration of the concrete compiler [Zam22a]. This demonstrator is based on high-level code (i.e., Python) that is transformed to TFHE by the concrete compiler, and the implementation uses a recursive definition of the edit distance.

1.1 Our contribution

In this paper, we develop a new edit distance algorithm for third-generation FHE schemes. We develop a new algorithm adapted to TFHE, which we call Leuvenshtein, and use this to show that the properties of the programmable bootstrapping play very well with edit distance calculations. The main ideas of our implementation are:

1. **Small representations**: We use differential values that represent the differences between intermediate results, instead of working on the intermediate results themselves. This reduces the size of the intermediate variables, leading to a smaller representation and more efficient calculations, significantly reducing the PBS costs. The size of our intermediate representations is small enough to fit in one ciphertext encoding 4 bits.
2. **Re-using the programmable bootstrap**: In each iteration of the Leuvenshtein algorithm we have to produce two output values (i.e., the horizontal and vertical differential values). We show that one can rewrite the equations so that both output values can be computed with the same non-linear parts, which as a result means that they only differ by a (cheap) addition. As the main cost is in the non-linear part, which needs to be done using costly programmable bootstrapping, this technique reduces the calculation cost by roughly half.
3. **Non-linear calculation in only one lookup**: During our calculation we have to compute the minimum of three inputs. A common strategy would be to do two bivariate lookups that each take two inputs. To enable the calculation of the non-linear part in only 1 programmable bootstrap, we propose a denser packing of the inputs. The input to our non-linear part has a total of $3 \times 3 \times 2 = 18$ values, while our 4-bit programmable bootstrap only

allows a 16-value function. By adapting the non-linear function to start and end with zeros, we enable a larger effective lookup that can accommodate the full 18 values, saving another factor two in PBS.

Our resulting Leuvenshtein algorithm[3] requires 94× less PBS compared to a textbook Wagner-Fischer implementation, and 16× compared to a bitsliced implementation (i.e., from Myers [Mye99]), excluding the equality calculations.

Our second contribution is an optimised equality check implementation that uses significantly fewer programmable bootstraps (PBS). This method allows us to encode characters more optimally, reducing the number of ciphertexts required by half. More specifically, using our method, we are able to do an equality test on 7-bit ASCII strings in 2 PBS, instead of the standard 5 PBS as would be used by a standard equality check (as for example implemented in TFHE-rs).

A third contribution looks at preprocessing to reduce the (online) running cost. As our improved edit distance calculation only requires 1 PBS per edit distance, the main cost of the algorithm sits in the equality calculation. In case one of the input strings is unencrypted, we show that one can do a precalculation where each encrypted string is compared to each letter of the alphabet and all the results are stored in a lookup table. During the edit distance calculation one then only has to perform an (unencrypted) lookup to select the relevant equality value. This technique is useful when the encrypted string is known in advance, or when the alphabet is small compared to the string lengths.

Combining these contributions, our implementation of the Levenshtein distance for ASCII inputs achieves a speedup of up to 278× over the best available implementation, and a factor 40× over our own state-of-the-art Wagner-Fisher implementation. In case of one unencrypted input, in some instances a further 3× speedup is possible due to our improved preprocessing.

2 Preliminaries

In this section, we will first introduce the TFHE homomorphic encryption scheme. Then, we will define edit distances and specifically the Levenshtein distance, including the most relevant algorithms to calculate it.

2.1 Notation

For the rest of this work, we will compare two strings $a_{1..m}, b_{1..n}$, with lengths of m and n characters, respectively. All characters of the strings come from an alphabet Σ, and with $|\Sigma|$ we denote the number of characters in the alphabet. The ith character of a string will be denoted as a_i. When the alphabet size is larger than the plaintext size, characters are represented through multiple symbols that are encrypted individually, denoted with $a_i^{(j)}$ for the jth part of the ith character of string a. For example, an 8-bit character can be split into two 4-bit symbols, $a^{(1)}$ and $a^{(2)}$.

[3] A Rust implementation is available on https://github.com/KULeuven-COSIC/leuvenshtein and on https://zenodo.org/records/15638825

2.2 TFHE

Fully homomorphic encryption (FHE) enables computations to be carried out on encrypted data. This paper will focus on FHE schemes with programmable bootstrapping, specifically the Torus Fully Homomorphic Encryption (TFHE) scheme [CGGI20]. We will provide a high-level introduction to TFHE; for more details, we refer to [CGGI20, Joy22].

In TFHE, a ciphertext typically holds 1-4 bits of plaintext while allowing for linear operations such as addition, subtraction, and multiplication with a small unencrypted value at a relatively low cost. However, these operations increase the noise in the ciphertexts. Once a certain number of operations have been performed, a noise reduction procedure called bootstrapping becomes necessary. Bootstrapping resets the noise, enabling further computations, but it is significantly more costly than linear operations. It is possible to chain ciphertexts together to encrypt larger integers.

One key advantage of TFHE is its capability to apply any lookup table (LUT) to the ciphertext without incurring any cost during bootstrapping. This process, known as programmable bootstrapping (PBS), facilitates executing highly non-linear functions on encrypted data. An example of such a LUT is shown in Table 1 on the following page.

To illustrate, consider two 2-bit encrypted messages, x and y, both encrypted in a 4-bit plaintext space (Figure 1 on the next page). To compare them, we first compute $x + (y \ll 2) \equiv (x + 4 \cdot y)$, resulting in a 4-bit value. The two least significant bits represent x, and the two most significant bits represent y. Then, we use a lookup table to check if the first 2 bits of the input equal the last 2 bits.

In more detail, the plaintext space is typically divided into message bits (the least significant bits of the plaintext space), carry bits (in the middle), and a padding bit (the most significant bit). The message bits represent plaintext values after encryption or bootstrapping. In contrast, the carry bits, initially zero, are filled after linear operations are performed (e.g. the $x + (y \ll 2)$ operation as described above). The padding bit is typically kept at zero to simplify the application of a LUT during programmable bootstrapping.

In more advanced scenarios, one can make an abstraction of the plaintext and carry space and use the entire plaintext space without the message-carry division. However, in this scenario, one should generally ensure that the padding bit remains zero to ensure proper LUT lookups during programmable bootstrapping.

More specifically, we can create the corresponding Lookup Table (LUT) for any arbitrary function as long as the padding bit is 0. Due to the nature of TFHE calculations, when the padding bit is 1, the lookup result will be the negative of the corresponding input with the padding bit 0. For example, if using a 5-bit plaintext space to evaluate function f through a LUT, and we want to use the entire 5-bit, we must consider that for input values $x = 2^5/2 = 16 < x < 2^5 = 32$, the function will become $f(x) = -f(x - 16)$. This property is due to the

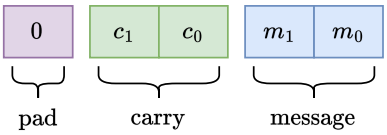

Fig. 1: Subdivision of a 5-bit plaintext in 2-bit message and carry space for a TFHE ciphertext

Table 1: LUT Table for function $f(x) = x - 4$ in a 5-bit plaintext space. The right side can not be chosen, as it is the negative (mod 16) of the left side.

x	Output	x	Output	x	Output	x	Output
0 (0 0000)	12	8 (0 1000)	4	16 (**1** 0000)	4	24 (**1** 1000)	12
1 (0 0001)	13	9 (0 1001)	5	17 (**1** 0001)	3	25 (**1** 1001)	11
2 (0 0010)	14	10 (0 1010)	6	18 (**1** 0010)	2	26 (**1** 1010)	10
3 (0 0011)	15	11 (0 1011)	7	19 (**1** 0011)	1	27 (**1** 1011)	9
4 (0 0100)	0	12 (0 1100)	8	20 (**1** 0100)	0	28 (**1** 1100)	8
5 (0 0101)	1	13 (0 1101)	9	21 (**1** 0101)	15	29 (**1** 1101)	7
6 (0 0110)	2	14 (0 1110)	10	22 (**1** 0110)	14	30 (**1** 1110)	6
7 (0 0111)	3	15 (0 1111)	11	23 (**1** 0111)	13	31 (**1** 1111)	5

negacyclic nature of the polynomials and is inherent in any current FHE scheme with PBS.

Another critical aspect of FHE is that data-dependent branching (e.g., if, while statements) cannot be used due to its confidential nature. To evaluate a branch in FHE, all possible outcomes must be calculated. This means that ideally, programs need to be rewritten to avoid if statements, and if statements cannot be used to skip irrelevant parts of the execution. We will revisit this topic in our discussion of edit distance calculation algorithms.

In the following sections, we will use a parameter set with a plaintext size of 5 bits. In this set, the lowest 4 bits can be freely assigned, while the most significant bit is utilised for padding. This parameter set is commonly used in practice.

2.3 Edit distance

The *edit distance* is a metric used to measure the similarity between two strings by calculating the number of edit operations needed to transform one string into another. It differs from the Hamming distance, which only considers the similarity of corresponding characters. For example, the Hamming distance between `'abcdex'` and `'xabcde'` is six, while the edit distance is two (one insertion and one deletion).

The edit distance exists in various variants where each variant allows a different set of operations: the first two operations, *'insertion'* and *'deletion'*, corre-

spond to the addition or removal of a character. The third operation is *'substitution'*, which involves replacing one character with another. The fourth operation is *'transposition'*, where two adjacent characters swap places. In this work we will focus on the Levenshtein distance, which considers the first three operations.

There are various versions of the edit distance. For example, costs can be assigned to each operation, allowing different weights to be applied. When all operations have a uniform unit cost, it is referred to as *simple edit distance*. If non-unit costs are used, it is called *general edit distance*. In some cases, the goal is to find the exact value below a specific limit, and once that limit is exceeded, the exact value becomes unimportant. An *approximated edit distance* can be used in such scenarios.

The popular Levenshtein distance is often interchangeably used with edit distance. These metrics are popular tools in (financial) fraud detection, DNA sequence comparison, calculating distances between matrix sequences [PHRL19, SAE+08, BDL+19], and spell checkers.

Calculating the edit distance The Levenshtein distance was originally obtained using a recursive definition. This definition was later converted to an executable algorithm, the Wagner-Fischer algorithm, using dynamic programming. Since then, more efficient variations have been proposed, improving time and space complexity. For a comprehensive overview of this plaintext algorithm, refer to the work of Navarro [Nav01].

Advanced algorithms, such as those based on the Four Russians Method [MP80], Suffix trees [Knu73], or filtering [Ukk92], are not suitable for implementation in FHE due to their data-dependent assumptions or alphabet-specific data representations. Additionally, algorithms based on nondeterministic finite automaton (NFA) [Ukk85b] will have the same complexity as the Wagner-Fischer algorithm in the encrypted domain. The representation of the automaton will have the same form as the d-matrix. Therefore, specific FHE-friendly optimisations are needed to speed up the calculations of the edit distance further.

Wagner-Fischer The Wagner-Fischer algorithm [Vin68, WF74] uses dynamic programming to create a distance matrix (or 'd-matrix'). Each element in the matrix represents the edit distance of the corresponding substrings up to that point in the matrix. For instance, $D[i,j] = ed(a_{1..i}, b_{1..j})$ corresponds to the edit distance of the first i characters of string a and the first j characters of string b. Specifically, in the simple edit distance case, each value of the d-matrix is determined by the following equation:

$$D[i,j] = \begin{cases} D[i-1,j-1] & \text{if } a_i = b_j \\ 1 + \min(D[i-1,j], D[i,j-1], D[i-1,j-1]) & \text{otherwise.} \end{cases} \quad (1)$$

To calculate the next value, the algorithm uses three previously computed values. These dependencies make the algorithm difficult to parallelise. The original definition has a time and memory complexity of $\mathcal{O}(n^2)$. A simple optimisation

is to reduce the space complexity to $\mathcal{O}(n)$ by only storing some columns of the d-matrix. Examples of the d-matrix are given in Figure 2.

Since its definition, many variations have been proposed to optimise the calculation. They mostly rely on skipping parts of the calculations based on the alphabet or data-dependent intermediate values [Mye86, Ukk85a]. It is impractical to port these optimisations to the FHE domain, as we do not know the value of intermediate variables and can thus not do any data-dependent optimisations.

		f	r	i	d	a	y
	0	1	2	3	4	5	6
m	1	**1**	2	3	4	5	6
o	2	2	**2**	3	4	5	6
n	3	3	3	**3**	4	5	6
d	4	4	4	4	**3**	4	5
a	5	5	5	5	4	**3**	4
y	6	6	6	6	5	4	**3**

		x	a	b	c
	0	1	2	3	4
a	1	**1**	**1**	2	3
b	2	2	2	**1**	2
c	3	3	3	2	**1**
x	4	3	4	3	**2**

Fig. 2: d-matrix of the (Simple) Edit distances of $d(\texttt{`monday'}, \texttt{`friday'}) = 3$ and $d(\texttt{`abcx'}, \texttt{`xabc'}) = 2$

Myers An alternative approach to Wagner-Fischer was proposed by Myers [Mye99]. This approach targets modern CPUs by rewriting the algorithm in terms of bits and optimising it for this lead. The main idea is to store the differential values (or differences between adjacent horizontal and vertical cells) in the d-matrix instead of absolute distances.

In simple edit distance, each neighbouring value in the d-matrix can differ by at most one (see Eq. 1). The Myers algorithm takes advantage of this by only representing the horizontal and vertical differences between neighbouring cells in the d-matrix (Figure 3 on the next page). This allows Boolean logic to compute the distance, making it possible to better utilise hardware parallelisation, particularly on CPUs. Note that from these horizontal and vertical differences, it is straightforward to reconstruct any value in the d-matrix by choosing a path from the start to the targeted cell and summing the horizontal and vertical differences along this path.

The core of the algorithm is to store only the neighbour differences using ternary values $\{-1, 0, 1\}$, both in the vertical and horizontal direction:

$$\begin{aligned} \Delta v[i,j] &= D[i,j] - D[i-1,j], \\ \Delta h[i,j] &= D[i,j] - D[i,j-1]. \end{aligned} \tag{2}$$

We can use the following equations to directly calculate the Δv and Δh values. When we examine a cell (i, j), we can define Δv_{out} and Δh_{out} as the values $\Delta v[i, j]$ and $\Delta h[i, j]$. These are the values we aim to compute in this cell. The equations above can then be rephrased in terms of the equality EQ

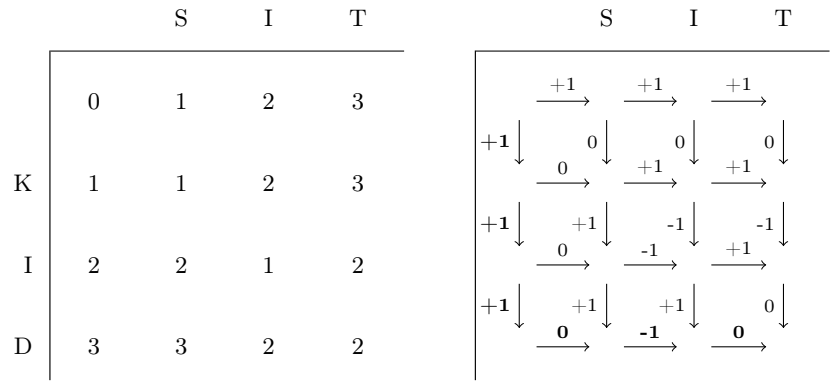

Fig. 3: Edit distance calculation of $d(\texttt{'KID'}, \texttt{'SIT'}) = 2$ through the Wagner-Fisher algorithm (left), where the absolute distances are calculated; and the Myers algorithm (right), which calculates the relative distances.

between the two relevant characters a_i and b_j, and the previous values of Δv and Δh which we denote as Δv_{in} and Δh_{in}. An overview of the input and output variables is given in Figure 4.

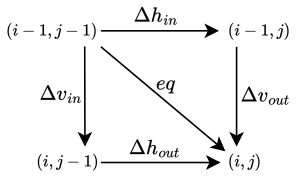

Fig. 4: Myers cell [Mye99]

We can transform Equation 2 to calculate the outputs Δv_{out} and Δh_{out} of a single d-matrix cell, in function of the inputs EQ, Δv_{in}, Δh_{in}:

$$\Delta v_{out} = \min \begin{pmatrix} 1, \\ \Delta v_{in} + 1 - \Delta h_{in}, \\ 1 - \text{EQ} - \Delta h_{in} \end{pmatrix},$$

$$\Delta h_{out} = \min \begin{pmatrix} 1, \\ 1 + \Delta h_{in} - \Delta v_{in}, \\ 1 - \text{EQ} - \Delta v_{in} \end{pmatrix}.$$

Building on this concept, it is possible to transform the entire Wagner-Fischer algorithm to use only Boolean operations and additions. For instance, they define two Boolean values to represent the ternary nature of the delta elements. By leveraging a w bit CPU architecture, the algorithm achieves a time complexity of $\mathcal{O}(\lceil m/w \rceil n)$. However, the Myers algorithm does not necessarily translate well to a TFHE environment, where operations can be performed on multi-bit values. A detailed overview of the Myers algorithm is provided in [Mye99].

3 Encrypted Levenshtein

Edit distance calculation generally involves two phases: equality checking and the main algorithm. The equality checking phase determines character equality between the input strings. The main algorithm then uses these equalities to compute the d-matrix (or differentials in the Myers approach) and determine the edit distance. This section will focus on the main algorithm, while section 4 will focus on the equality checking phase.

3.1 Main algorithm

This section will develop a new edit distance algorithm suited for the encrypted domain. Our algorithm improves the Myers approach detailed in section 2.3 in the FHE case. As a reminder, the main idea of this approach is to calculate the differential values between two nodes in the edit distance calculation, as given in Figure 4.

The Myers approach is designed for CPU-optimised operations, such as Boolean operations and additions, by using bitslicing. This makes operations very efficient on CPU, but they do not necessarily translate to efficient operations in the TFHE domain. Our approach focuses on optimising for FHE in two steps:

First, our approach uses small multivalue operands (typically in the range $\{-1, 0, 1\}$) instead of binary values in Myers. For example, we represent the trinary Δh_{in} and Δv_{in} operators in one operand instead of splitting into binary positive and negative parts, as done by Myers. This limits the number of inputs and outputs that need to be handled and is efficient due to the native multivalue operations in TFHE.

Secondly, we rewrite the cell equations to allow the calculations in only one bootstrap. This means that Δv_{in}, Δh_{in}, and EQ are given as inputs to the PBS, and Δv_{out}, Δh_{out} are extracted as the outputs. To achieve this, we have to optimise the PBS to perform a lookup that is relevant for both the outputs Δv_{out} and Δh_{out}. We then show that we can still perform this lookup in 1 PBS by carefully manipulating the PBS function.

Combining the PBS calculations Using the Myers approach, our algorithm calculates two output values Δv_{out} and Δh_{out} for each cell, as explained in Figure 4. The formulas to compute Δv_{out}, Δh_{out} are shown below:

$$\Delta v_{out} = \min \begin{pmatrix} 1, \\ \Delta v_{in} + 1 - \Delta h_{in}, \\ 1 - \text{EQ} - \Delta h_{in} \end{pmatrix},$$
$$\Delta h_{out} = \min \begin{pmatrix} 1, \\ 1 + \Delta h_{in} - \Delta v_{in}, \\ 1 - \text{EQ} - \Delta v_{in} \end{pmatrix}.$$

The first optimisation is to rewrite the equations to have a similar non-linear operation. We can rewrite both equations to:

$$\begin{aligned} \Delta v_{out} &= \min(-\text{EQ}, \Delta v_{in}, \Delta h_{in}) + (1 - \Delta h_{in}), \\ \Delta h_{out} &= \min(-\text{EQ}, \Delta v_{in}, \Delta h_{in}) + (1 - \Delta v_{in}). \end{aligned} \tag{3}$$

In this form, we can focus on $\min(-\text{EQ}, \Delta v_{in}, \Delta h_{in})$ in the PBS, and perform the $(1 - \Delta h_{in})$ operations using linear computations without bootstrapping at low cost. This optimisation reduces the calculation cost with approximately a factor 2.

Extended lookups A standard approach to calculate the min function in Equation 3 would be to do two bivariate lookups, which would first combine two inputs into the key (key $= \Delta v_{in} + 4 \cdot \Delta h_{in}$), after which a PBS with relevant lookup table is performed on this key to calculate the function $\min(\Delta v_{in}, \Delta h_{in})$. In the second phase, a similar min function is performed between the result of the first min function and EQ. This approach requires two PBS lookups for each cell of the Leuvenshtein calculation.

In this section we will reduce this further to one PBS per cell in the standard Levensthein case, by combining the calculation of both min functions. In this case, we have Δv_{in}, $\Delta h_{in} \in [-1, 0, 1]$ and $\text{EQ} \in [0, 1]$. By combining the inputs to the min functions in a more dense way (e.g. $\Delta v_{in} + 3 \cdot \Delta h_{in} + 9 \cdot \text{EQ}$) one could reduce the input size. However, even in the best case, this entails a $3 \times 3 \times 2 = 18$-entry lookup table for the min operation, while we only have a 16-entry lookup table available.

It is important to note the negacyclic nature of the TFHE-PBS lookup. In TFHE with a 4-bit message (and carry), one can construct any lookup table of 16 values (values 0 to 15). Lookups at values 16 to 31 will result in the negated value of the corresponding value at position $i - 16$. This means that if we can place a zero lookup at position 0, and the value at position 16 is also 0, we can essentially extend the lookup table with one extra value. We can extend this as long as the ith and $(i + 16)^{\text{th}}$ values are both 0.

When we rewrite the formulas for Δv_{out} and Δh_{out} as:

$$\begin{aligned} \Delta v_{out} &= \{1 + \min(-\text{EQ}, \Delta v_{in}, \Delta h_{in})\} - \Delta h_{in}, \\ \Delta h_{out} &= \{1 + \min(-\text{EQ}, \Delta v_{in}, \Delta h_{in})\} - \Delta v_{in}. \end{aligned}$$

The function between brackets returns mostly zeros during the programmable bootstrap. We will denote this value with M, or M_{ij}, denoting the M value in the cell at location i, j. Combining this with the following key for the PBS lookup:

$$(\Delta v_{in} + 1) + 3 \cdot (1 + \Delta h_{in}) + 9 \cdot \text{EQ} \tag{4}$$

results in a lookup table of 18 values, which starts and ends with two zeros. This allows us to fit this lookup table into a 16-value TFHE lookup table, as detailed in Table 2.

Table 2: LUT Table for function $\text{LUT}_{\min} : 1 + \min(-\text{EQ}, \Delta v_{in}, \Delta h_{in})$, with $x = (\Delta v_{in} + 1) + 3 \cdot (1 + \Delta h_{in}) + 9 \cdot \text{EQ}$ in a 5-bit plaintext space. Note that the right side is the negative of the left side (mod 16) due to the negacyclic nature of the LUT.

x	Output	x	Output	x	Output	x	Output
0 (0 0000)	0	8 (0 1000)	1	16 (**1** 0000)	0	24 (**1** 1000)	15
1 (0 0001)	0	9 (0 1001)	0	17 (**1** 0001)	0	25 (**1** 1001)	0
2 (0 0010)	0	10 (0 1010)	0	18 (**1** 0010)	0	26 (**1** 1010)	0
3 (0 0011)	0	11 (0 1011)	0	19 (**1** 0011)	0	27 (**1** 1011)	0
4 (0 0100)	1	12 (0 1100)	0	20 (**1** 0100)	15	28 (**1** 1100)	0
5 (0 0101)	1	13 (0 1101)	0	21 (**1** 0101)	15	29 (**1** 1101)	0
6 (0 0110)	0	14 (0 1110)	0	22 (**1** 0110)	0	30 (**1** 1110)	0
7 (0 0111)	1	15 (0 1111)	0	23 (**1** 0111)	15	31 (**1** 1111)	0

Limiting the noise growth Our approach yields the same result as the corresponding plaintext algorithms. However, since the computation is performed under FHE, there remains a small probability of error. This probability is typically kept extremely low for cryptographic security purposes (for instance, below 2^{-64}) by carefully choosing parameters that control noise growth. In this subsection, we demonstrate that, with a few minor adaptations, our algorithm remains within the predefined noise bounds, thereby preserving the intended low failure probability. This failure probability is sufficiently low that it should not be noticeable by end-users.

In the previous discussion, we simplified the operations in two cases: costly non-linear bootstraps and cheap linear operations that do not require a bootstrap. In reality, there is a maximum number of linear operations that can be performed before a bootstrap is needed. In our case the calculation of the key can exceed this threshold when not taken into account.

We will denote the variance of the noise of one ciphertext with ϵ^2_{PBS} (i.e., the noise of a ciphertext at bootstrap time when it has not been combined with another ciphertext). When adding k independent ciphertexts, the noise will be

equivalent to $k \cdot \epsilon_{PBS}^2$, while the multiplication of a ciphertext with k results in a noise equivalent of $k^2 \cdot \epsilon_{PBS}^2$. These come from standard equations for the addition and multiplication of stochastic variables.

For the key calculation in Equation 4, one thus has a noise equivalent of:

$$\epsilon_{key}^2 = \epsilon_{\Delta v_{in}}^2 + 9 \cdot \epsilon_{\Delta h_{in}}^2 + 81 \cdot \epsilon_{\text{EQ}_9}^2.$$

Note that for now, we assume independence between the variables, which we will come back to later in this section.

A first trick to reduce the error is to include the factor 9 at the EQ term: instead of calculating enc(EQ) during the equality checking phase, we adapt the bootstrap to calculate enc($9 \cdot$ EQ). Moreover, this gives us a small speedup since a scalar multiplication is avoided. We will denote the value $9 \cdot$ EQ with EQ_9. This reduces the noise of the key to:

$$\epsilon_{key}^2 = \epsilon_{\Delta v_{in}}^2 + 9 \cdot \epsilon_{\Delta h_{in}}^2 + \epsilon_{\text{EQ}_9}^2 = \epsilon_{\Delta v_{in}}^2 + 9 \cdot \epsilon_{\Delta h_{in}}^2 + \epsilon_{PBS}^2.$$

A second thing to notice is that Δv_{in} and Δh_{in} themselves are calculated recursively. Denoting with $\Delta v_{i,j}$ the Δv_{out} of cell (i, j) (and similarly for Δh_{out}) and $M_{i,j} = D[i, j] - D[i-1, j-1]$, we get the following formulas:

$$\begin{aligned}
\Delta v_{in} &= \Delta v_{i,j-1} \\
&= M_{i,j-1} - \Delta h_{i-1,j-1} \\
&= M_{i,j-1} - M_{i-1,j-1} + \Delta v_{i-1,j-2} \\
&= M_{i,j-1} - \boldsymbol{M_{i-1,j-1}} + M_{i-1,j-2} - \boldsymbol{M_{i-2,j-2}} + \cdots \\
\Delta h_{in} &= M_{i-1,j} - \boldsymbol{M_{i-1,j-1}} + M_{i-2,j-1} - \boldsymbol{M_{i-2,j-2}} + \cdots
\end{aligned}$$

All M are calculated using different inputs, and thus, their respective noise can be considered independent. However, the formulas of Δv_{in} and Δh_{in} have the same negative M terms (in bold in the equation above), which increases the noise in the key:

$$\begin{aligned}
key &= \Delta v_{in} + 3 \cdot \Delta h_{in} + \text{EQ}_9 + Cte \\
&= (M_{i,j-1} - 4 \cdot M_{i-1,j-1} + 3 \cdot M_{i-1,j}) + \\
&\quad (M_{i-1,j-2} - 4 \cdot M_{i-2,j-2} + 3 \cdot M_{i-2,j-1}) + \\
&\quad \cdots + \\
&\quad \text{EQ}_9 + Cte
\end{aligned}$$

or:

$$\epsilon_{key}^2 = N_H \cdot \epsilon_{PBS}^2 + 16 \cdot N_M \cdot \epsilon_{PBS}^2 + 9 \cdot N_L \cdot \epsilon_{PBS}^2 + \epsilon_{PBS}^2$$

with N_H, N_M and N_L the number of respective M terms (i.e., N_H is the number of $M_{i,j-1}$-like terms, N_M the number of $4 \cdot M_{i-1,j-1}$-like terms and N_L the number of $3 \cdot M_{i-1,j}$-like terms).

1	10	10	10	10	10	10
2	15	24	24	24	24	24
2	16	11	20	20	20	20
2	16	12	21	11	20	20
2	16	12	21	12	21	11
2	16	12	21	12	21	12
2	16	12	21	12	21	12
2	16	12	21	12	21	12

Fig. 5: The relative value of the noise in each cell of the calculations. The red line indicates a refresh of the noise is needed using a bootstrap operation. This is for a parameter set that can handle up to 25 additions.

The noise can be easily reduced by changing the equation of the key as:

$$(1 - \Delta v_{in}) + 3 \cdot (1 + \Delta h_{in}) + 9 \cdot \text{EQ}, \tag{5}$$

where Δv_{in} has been negated. This changes the LUT of the bootstrap in Table 2, but the general techniques developed are still valid. As the constant near the $\boldsymbol{M_{i-1,j-1}}$ term is now 2 instead of 4, we have an equivalent noise of:

$$\epsilon^2_{key} = N_H \cdot \epsilon^2_{PBS} + 4 \cdot N_M \cdot \epsilon^2_{PBS} + 9 \cdot N_L \cdot \epsilon^2_{PBS} + \epsilon^2_{PBS}.$$

We now have improved noise equations. To make sure our noise does not surpass the threshold, we have to refresh the noise in the Δv_{in} and Δv_{out} terms just before the noise threshold is exceeded. Figure 5 depicts the noise in the ciphertexts, where the red line indicates where refreshing is needed. This example is for a parameter set that allows a maximum of 25 additions of ciphertexts before bootstrap is needed.

In practice, the standard parameter sets used in TFHE-rs can typically handle more than 4 000 additions before a bootstrap is needed, according to our calculations. As such, only in edit distance calculations with very large words does one have to take these refreshes into account. We have experimentally verified this calculation by running the bootstrap on increasingly noisy ciphertexts and testing when an error occurs.

3.2 Skipping irrelevant cells

When calculating the d-matrix (or equivalently, the horizontal and vertical differences Δh and Δv), certain cells do not influence the final result and thus do not need to be computed.

This can be understood as follows:

- **Maximum Levenshtein Distance**: The maximum possible value of the Levenshtein distance is bounded. In the worst case, it is $\max(m, n)$, meaning that every character in one string needs to be substituted to match the other string.
- **Shortest Path Analogy**: Computing the Levenshtein distance is analogous to finding the shortest path through the d-matrix, where the cost of each move is defined by Equation 1. Horizontal and vertical steps always have a cost of 1, while diagonal steps depend on the value of EQ.
- **Path Cost Bounds**: The minimal cost of traversing from the top-left to the bottom-right corner of the d-matrix is $m + n$ (i.e., m vertical steps and n horizontal steps). This value exceeds the maximum possible Levenshtein distance, which implies that not all cells are relevant to the computation.

These observations reveal that many cells in the d-matrix are unnecessary for determining the final distance. Without loss of generality, consider the case where $m = n$. According to Ukkonen [Ukk85a], if a cell that is k steps away from the main diagonal is on the shortest path, the path has a minimum path cost of $2k$ (consisting of at least k horizontal and k vertical steps). Thus, cells located more than $\lfloor k/2 \rfloor$ steps from the diagonal can be excluded from computation without affecting the accuracy of the result.

In a more extreme situation, one can compute approximate Levenshtein distances, where the result is accurate up to a Levenshtein distance of ℓ, but for larger Levenshtein distances, the output might be wrong (i.e., the output might be larger than expected). In this case, one only has to compute cells that are within $\lceil \ell/2 \rceil$ of the diagonal. This is useful when you want to determine if two strings are approximately equal. Concretely, when $m = n$, we do not need to calculate all the m^2 cells, but we can reduce this to

$$m + 2 \cdot \sum_{i=1}^{\ell} (m - i) = m \cdot (2\ell + 1) - \ell^2 - \ell.$$

Furthermore, this modification precludes the calculation of the edit distance value from the final Δh row. Consequently, the final score is determined by summing all Δv and Δh elements along the diagonal.

3.3 The resulting algorithm

Combining all of the above approaches leads to Algorithm 1 on the following page. This algorithm has as inputs a matrix $\mathbf{EQ_9}$ and parameter ℓ. Matrix $\mathbf{EQ_9}$ will contain the equality information for each character pair. That is, element $[i, j]$ will contain a 9 if $x_i = y_j$ and otherwise will contain 0. Parameter ℓ will denote the approximation level. By assigning $\ell = \lceil \max(m, n)/2 \rceil$, the exact edit distance will be calculated.

To make a fair comparison, we ran our WF and Myers implementations and counted the number of PBS performed. The WF and Myers algorithm would need on average, approximately 94 PBS and 16 PBS to calculate a cell, when

Algorithm 1 Leuvenshtein

function EDIT DISTANCE($\mathbf{EQ_9}, \ell$)
▷ Setup
  $h \leftarrow$ OneMatrix$[0..m, 0..n]$
  $v \leftarrow$ OneMatrix$[0..m, 0..n]$
  LUT_{min}[key] $\leftarrow$ Table 2

▷ Main Algorithm
  for $j \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to m **do**
      if $|i - j| \leq \ell$ **then**
        $key \leftarrow (1 - v[i, j-1]) + 3 \cdot (1 + h[i-1, j]) + \mathbf{EQ_9}[i, j]$
        $min \leftarrow \text{PBS}(key, \text{LUT}_{\text{min}})$
        $v[i, j] \leftarrow min - h[i-1, j]$
        $h[i, j] \leftarrow min - v[i, j-1]$
  return $\Sigma_{i=1}^{m+1} h[i, i] + \Sigma_{i=0}^{n} v[i+1, i]$

Table 3: Overview of the PBS load of the different algorithms for a single cell, using ASCII encoding. The exact number of programmable bootstrap needed for the WF and Myers algorithm is estimated here, as it depends on the exact scenario. More information can be found in section 6

Algorithm	Levenshtein	Improvement factor
WF	94	1×
Myers	16	5.87×
Ours	1	94×

using ASCII encoding. While the size of the other algorithms depends on the character encoding and length of the input strings, our algorithm is independent of these. The cost for our algorithm will remain constant, requiring only one PBS.

4 Equality checking

In the previous section, we described an algorithm that only uses 1 PBS per cell to calculate the Levenshtein distance. For each cell, one also has to calculate the equality between the corresponding characters of the input strings. This typically costs more than 1 PBS per cell, and thus the equality calculation is the most expensive operation of the full Levenshtein algorithm in our case. In this section, we improve the equality calculation in two ways: we introduce a technique that allows doubling the number of plaintext bits in one PBS equality operation, and then we propose a new technique to more efficiently look at larger symbols, notably 7-bit ASCII symbols.

4.1 Doubling the equality PBS size

The standard approach to equality checking involves dividing the binary representation of the input letters into chunks of 2-bit and then pairwise comparing the corresponding bits. This method is also implemented in the software library TFHE-rs [Zam22b]. For instance, when comparing a 2-bit x with a 2-bit y, one computes the key $x + 4 \cdot y$ and then performs a PBS that maps $x = y$ to 1 and all other values to 0. Using the standard parameter size (2-bit plaintext, 2-bit carry), this method can only handle inputs of at most 2 bits.

We present a new method that can handle 4-bit symbols. To compare two (4-bit) chunks x and y, we subtract both values, resulting in a variable with a value between -15 and 15, and a value of 0 if and only if both chunks are the same. As before, this results in more values than the normal 16-value PBS lookup. By choosing the following lookup table for the PBS:

$$\mathrm{LUT}_{\mathrm{EQ}} = \begin{cases} 1 & \text{if } (x - y) = 0 \\ 0 & \text{else,} \end{cases} \tag{6}$$

we can have a 31-value lookup due to the negacyclic property of the lookup and the abundance of 0 values. Note that in our Levenshtein calculation, we sometimes want to calculate EQ_9, for which we adapt the LUT to:

$$\mathrm{LUT}_{\mathrm{EQ}_9} = \begin{cases} 9 & \text{if } (x - y) = 0 \\ 0 & \text{else.} \end{cases} \tag{7}$$

Thus, our new method of equality checking can handle symbols of double size. For typical large integers where the integer is divided into ciphertexts that each contains 2-bit chunks, two 2-bit equality checks can be combined into one by calculating $x = 4 \cdot x^{(2)} + x^{(1)}$ (and similarly for y), thus halving the PBS cost. This also means that an equality check for larger inputs can be done at half of the PBS cost using our method.

4.2 Equality check for large-sized symbols

Larger characters are typically divided into smaller symbols of size t, usually 2 to 4 bits. A sub-equality operation is performed for each pair of chunks, after which the results of these sub-equality operations are combined to produce the final equality result.

The TFHE-rs library provides a two-step algorithm for calculating equality. First, corresponding chunks are compared using the method $x + (y \ll 2)$, as described earlier. This will produce a sub-equality that will compare 2-bit or character data. The ciphertext will contain a one if the two parts are equal, a zero in the other case.

In the second step, the sub-equality results are summed together, in a triangular way, to find the overall equality. All of the sub-equalities are divided into groups of maximally $t-1$ elements. All sub-equalities in a group are summed together and a PBS is applied to the sum to check if the sum reaches its maximum

possible value, i.e. $t-1$. The results are now again grouped into maximally $t-1$ elements, summed together and used in a PBS. This process is repeated until a single ciphertext is obtained, representing the result of the equality check. If all of the sub-equalities of step one consist of a one, the final equality result will also depict a one. Algorithm 2 outlines this approach for characters of at most 30-bit.

For example, when comparing two ASCII characters (7-bit), each character is encoded into four ciphertexts. After computing the sub-equalities, the four outcomes are summed, and a PBS is used to verify if the total sum equals 4.

This method can be further improved using a hybrid approach of our custom equality subcheck and the TFHE-rs combination phase. In this hybrid approach, characters are encoded into consecutive 4-bit chunks. In the first part, subcomponents are computed using our subtraction method. In the second part, the TFHE-rs aggregation step is used to combine the subcomponents efficiently. This reduces the cost of calculation by roughly half.

Algorithm 2 TFHE-rs compare method for characters with maximum size of 30 bit.

Require: Two encoded characters x and y
Require: x, y are split into n encrypted symbols, $n < 16$, each symbol has $t = 2^p$ size plaintext
▷ Equal Part
1: **for** $i \leftarrow 0$ to n **do**
2: $\quad z_0 \leftarrow x^{(i)} + 4 \cdot y^{(i)}$
3: $\quad \text{EQ}^{(i)} \leftarrow \text{PBS}(z_0; \text{LUT}_{\text{EQ}})$

▷ Merging Part
4: Acc $\leftarrow 0$
5: **for** $i \leftarrow 0$ to n **do**
6: $\quad$ Acc $\leftarrow$ Acc $+ \text{EQ}^{(i)}$
7: Acc $\leftarrow \text{PBS}(\text{Acc}, \text{LUT}_{\max})$
8: **return** Acc

4.3 Equality check for medium-sized symbols (e.g., ASCII)

While this combination method of TFHE-rs is efficient for large input symbols, we propose a better method for medium-sized inputs. Specifically, our method outperforms state-of-the-art for 5- to 16-bit inputs, specifically for ASCII inputs. For this explanation, we will assume 7-bit input characters and a 4-bit combined message and carry space.

The first step in our method is to decide on the representation of our characters. We will encode our characters using a 4-bit and a 3-bit symbol, each encrypted in one ciphertext. We will then perform the equality check between the 4-bit symbols as explained in the previous section. The result is a single bit

EQ denoting equality, which is combined with the 3-bit symbol in the following way:

$$2 \cdot (x_i^{(2)} - y_i^{(2)}) + (1 - \text{EQ}^{(1)}).$$

The result of this linear computation is 0 if and only if both the 4-bit symbols and the 3-bit symbols are the same. Thus, we can perform the same PBS lookup as before on this result to calculate the equality of the characters. The formal representation is given in Algorithm 3.

Algorithm 3 Our equality check for 7-bit characters, split into 4 and 3-bit

Require: Two ASCII encoded characters x and y
Require: x, y are split into 4-bit symbols $(x^{(1)}, y^{(1)})$ and 3-bit symbols $(x^{(2)}, y^{(2)})$
1: $z_0 \leftarrow x^{(1)} - y^{(1)}$ ▷ first 4 bits of the character
2: $\text{EQ}^{(1)} \leftarrow \text{PBS}(z_0; \text{LUT}_{\text{EQ}})$
3: $z_1 \leftarrow 2 \cdot (x^{(2)} - y^{(2)}) + (1 - \text{EQ}^{(1)})$ ▷ last 3 bits of the character
4: $\text{EQ} \leftarrow \text{PBS}(z_1; \text{LUT}_{\text{EQ}})$
5: **return** EQ

The result is that we can perform an ASCII equality check with two PBS, while other methods would need five lookups when using state-of-the-art 2-bit equality techniques or three lookups when using our 4-bit equality technique.

4.4 Conclusion

To calculate the equality between two characters, there are three options: TFHE-rs, our own, or the combined approach. All three approaches demonstrate different PBS behaviour. Figure 6 denotes the PBS loads as a function of the input size. From this analysis, we can see that our method is the best for characters of up to 16-bit, which is ideal for an ASCII usecase. For larger characters, the combined method performs better. Furthermore, our method reduces the memory footprint of the encrypted character by up to a factor of two.

5 Preprocessing

Even with reduced cost for the equality check as discussed in the previous section, the equality checking is typically still more expensive than the main algorithm. More specifically the main algorithm uses 1 PBS per cell, while the equality calculation costs C PBS calculations per cell with C a constant depending on the input characters. In this section we will show that in the specific case that one of the strings is unencrypted, we can perform a preprocessing to speed up the equality calculations, making their total cost linear in the input size (i.e. $C \cdot |S| \cdot m$, for m the size of the encrypted input).

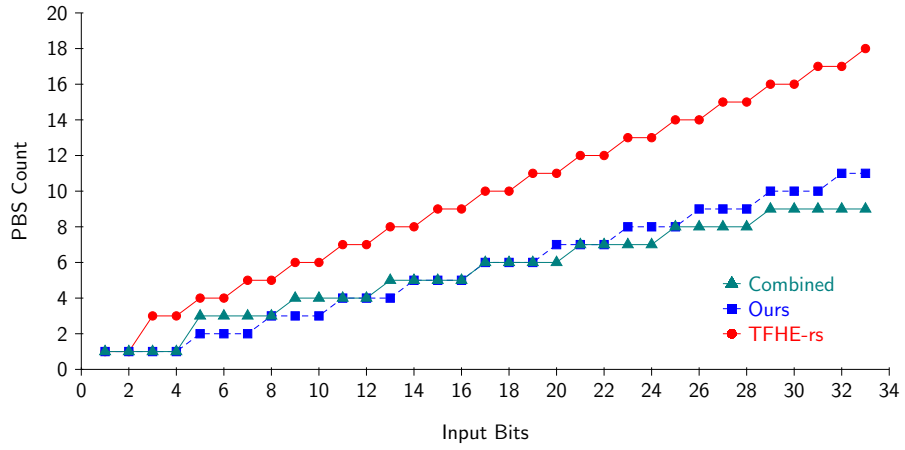

Fig. 6: Comparison of the PBS count for the three equality calculation techniques.

The idea of this preprocessing is to precompute the output of the equality for each possible character of the alphabet for the encrypted input (which has cost $|\Sigma| \cdot m$), similar to a technique proposed by Myers [Mye99] to allow efficient bitsliced implementations. During the equality check one then can perform a simple lookup using the unencrypted input character to find the result of the equality, which does not require any PBS.

More specifically, during the preprocessing phase, we check the equality of each character in the encrypted string with every possible character in the alphabet. The results are then stored in an encrypted table and when the edit distance is calculated, we can use the unencrypted data to obtain the encrypted equality information.

For instance, if the word 'abbey' is the encrypted string and only encodes lowercase letters, we would store 5×26 elements, as shown in Table 4. Note that it is also possible to store $\text{EQ}_9 = \text{enc}(9)$ and can therefore be used directly in the key calculation.

The preprocessing step has a cost of $|\Sigma| \cdot m$, where m is the length of the encrypted string and $|\Sigma|$ is the size of the character set. This is specifically useful in the case of matching DNA sequences, we would only need to store the 4 characters. Moreover, if during encryption we know we will only deal with a specific subset of characters $S \subset \Sigma$, we can reduce the cost to $S \cdot m$ by constructing a table only for the characters in S. As an example, this could be the case with names where characters like Y and Q might not occur in the unencrypted query string.

A standard edit distance calculation requires $m \cdot n$ equality operations. Therefore, if $|\Sigma| < n$ (or $S < n$), this optimisation becomes advantageous. For example, with full ASCII, the method becomes more efficient for $n > 128$. For

Table 4: Preprocessed storage of the word 'abbey' using only lowercase letters.

character	1	2	3	4	5
a	**enc(1)**	enc(0)	enc(0)	enc(0)	enc(0)
b	enc(0)	**enc(1)**	**enc(1)**	enc(0)	enc(0)
c	enc(0)	enc(0)	enc(0)	enc(0)	enc(0)
d	enc(0)	enc(0)	enc(0)	enc(0)	enc(0)
e	enc(0)	enc(0)	enc(0)	**enc(1)**	enc(0)
...	...	...	...	...	...
y	enc(0)	enc(0)	enc(0)	enc(0)	**enc(1)**
...	...	...	...	...	...

lowercase letters, it applies when $n > 26$, and for DNA sequences, it becomes beneficial when $n > 4$.

This approach is particularly useful when matching a plaintext string against a database. In the case where the database is unencrypted and the query is encrypted, one needs to do the query only one time and the result can be used for multiple lookups. If the database is encrypted and the query is not encrypted, one can preprocessing each encrypted string in the database in an offline preparation phase. This means that future plaintext equality checks can be done without the need for additional PBS. Once a new plaintext string needs to be matched, one only needs to perform a lookup in the table.

6 Results

In this section, we will compare our Leuvenshtein algorithm to the state-of-the-art in FHE edit distance calculations. A challenge in analysing the efficiency of our improvements is that there is only one implementation of the Levenshtein distance available in TFHE, which is an implementation that is used as a demonstrator for the concrete compiler [Zam22a]. To have more comparison points, we implemented standard versions of the Wagner-Fischer and Myers algorithms using the TFHE-rs library. Both algorithms rely on standard 2-bit message and 2-bit carry ciphertexts as fundamental building blocks. In our experiments, we used ASCII encoding to encrypt each character. The Wagner-Fischer and Myers algorithms are implemented using the state-of-the-art equality check techniques as available in the TFHE-rs library.

Our algorithm uses both our improved Levenshtein (algorithm 1) and our improved equality check (algorithm 3). A complete overview of the algorithm is given in Algorithm 4 on page 32. We discern four versions of our algorithm in our experiments:

- **Leuvenshtein Exact**: Full calculation of all the cells in the edit distance algorithm.

- **Leuvenshtein Exact Skipping**: Calculation of all the cells that are relevant for the end result, but skipping irrelevant cells as discussed in subsection 3.2.
- **Leuvenshtein Approx. $\ell = n/4$**: Calculation of the approximate edit distance that is accurate for distances lower than $m/4$ as discussed in subsection 3.2. To simplify the discussion we assume $m = n$ for the approximate results.
- **Leuvenshtein Approx. $\ell = 10$**: Calculation of the approximate edit distance that is accurate for distances lower than 10 as discussed in subsection 3.2.

6.1 Counting the bootstraps

In this section, we analyse the theoretical cost of the algorithms, focusing on the number of bootstraps required. Due to the insertion of auxiliary PBS operations for message-carry maintenance in the TFHE-rs library, determining the exact number of PBS used in theory is not trivial. We experimentally determined the average number of PBS per cell in the WF and Myers algorithms by running the program and counting the PBS. For the Leuvenshtein algorithm, we derive a first-order estimate based on the bootstraps incurred by non-linear operations within each cell, excluding incidental bookkeeping bootstraps introduced for noise management.

Table 5: Overview of the PBS load of the different algorithm to match an n-element with an m-element string. The preprocessing and Levenshtein column denote the operations per cell, the last column is a first-order approximation of the total cost over all cells.

Algorithm	Preprocessing	Levenshtein	Total
WF	5	28	$33 \cdot mn$
Myers	5	13	$18 \cdot mn$
Leuvenshtein Exact	2	1	$3 \cdot mn$
Leuvenshtein Exact Skipping	2	1	$\approx 2.25 \cdot mn$
Leuvenshtein Approx. $\ell = m/4$	2	1	$\approx 6 \cdot \ell \cdot n$
Leuvenshtein Approx. $\ell = 10$	2	1	$\approx 60 \cdot n$

From Table 5 one can see that our algorithm significantly reduces the number of required bootstraps to compute a cell in the encrypted domain. Firstly, we need only 2.5× fewer PBS for equality calculations. Secondly, we achieve a reduction of 94× and 16× less PBS compared to the Wagner-Fischer and Myers algorithms, respectively. Overall, for the full distance calculation, our best exact method demonstrates a 44× reduction in the number of PBS required over the Wagner-Fischer algorithm and an 9.33× improvement over the Myers algorithm.

6.2 Implementation results

We implemented the Wagner-Fischer, Myers, and our custom algorithm using Rust v1.80.0, TFHE-rs v0.7.2 [Zam22b] and concrete compiler v2.8.1 [Zam22a] on an Ubuntu 22.04 system. All experiments were conducted on a dual AMD EPYC 9174F 16-Core Processor (for a total of 64 threads). The parameter set used for the Wagner-Fischer and Myers implementations are 2-bit message and 2-bit carry ciphertext which is the most standard choice., For the Leuvenshtein implementation, we opted not to use the carry space, instead employing 4-bit message 0-bit carry parameter set. This implementation choice makes the implementation easier, but has little effect on the performance. All timing results represent the full calculation of the edit distance, including both preprocessing and the main algorithm. Each experiment was conducted using the sequential implementation of the shortint API. We will discuss parallelisation options for these algorithms in next section.

Table 6: Latency results in seconds of the calculation of the edit distance, using ASCII encoding, in the encrypted domain.

	$m = 8$		$m = 100$		$m = 256$	
	Seq.		Seq.		Seq.	
[CKL15][a]	27.54					
[Zam22a]	241.10	1×	12h 36m	1×	6d 21h[b]	1×
WF	77.59	3.1×	3h 24m	3.7×	22h 19m	7×
Myers	17.81	14×	38m 12s	20×	4h 7m	40×
Lvs Exact	2.83	85×	7m 19s	103×	48m 23s	205×
Lvs Exact Skip	**2.28**	106×	**5m 35s**	135×	**35m 41s**	278×
Lvs Apx. $\ell = \frac{m}{4}$	1.50	161×	3m 19s	228×	20m 52s	475×
Lvs Apx. $\ell = 10$			1m 26s	527×	3m 50s	2588×

[a] Using the DGHV scheme, an 80-bit security level and other hardware.
[b] Extrapolated based on 24300 cell calculations.

The results in this table demonstrate that our Leuvenshtein algorithm significantly outperforms state-of-the-art TFHE algorithms. Specifically, we achieve up to 278× speedup over the best available algorithm, and 39× speedup over our Wagner-Fisher implementation.

While we have included the numbers for edit distance for second generation FHE, these results are run with different hardware and lower security settings, and are therefore not suitable for a fair comparison.

Additionally, skipping irrelevant cells as discussed in subsection 3.2 is an efficient way to optimise the algorithm by 25% for exact calculations, and limiting the accuracy for large edit distances can give another significant efficiency

improvement. These results clearly show that our algorithm consistently outperforms existing approaches across all scenarios.

Parallelism: Previous results were achieved using the sequential TFHE-rs parameter set. Greater efficiency can be attained by developing parallel implementations. While a comprehensive investigation into parallelisation is left for future work, we will provide some preliminary insights and suggestions here.

In general one can think of three levels at which to parallelise:

- *Parallelisation over operations*: Some operations require multiple PBS operations that can be performed in parallel. This is the case for for example additions or a min function over larger integers (as used in Wagner-Fisher) or Boolean operations on multiple inputs at the same time (as used in Myers). TFHE-rs has a parallel implementation available for these operations. This strategy can not be used for our method, as we only have 1 PBS per cell of the alogrithm.
- *Parallelisation over cells*: The edit distance algorithm allows for parallelisation across multiple cells that are independent of one another. Specifically, cells equidistant from the first cell can be computed simultaneously. This approach enables efficient parallel computation for the majority of the cells, with only the first and last cells experiencing limited parallelisation. For large string sizes, this strategy achieves near-complete parallelisation, significantly enhancing execution efficiency. The implementation of this parallelisation method is deferred to future work.
- *Batch inputs matching*: In case of multiple edit distance calculations that need to be performed at the same time, one can easily parallelise the calculation by performing all lookups in parallel. This is especially relevant to parallellise the workload for database lookup scenarios. Table 7 shows a comparison of the algorithms under parallel computations.

Execution using the TFHE-rs parallel API shows only a modest impact on execution time (approximately 2.4×), highlighting the need to explore alternative sources of parallelism. In contrast, our batched inputs significantly accelerate computation, achieving near-optimal speedup of 32×, which aligns with full core utilisation of our CPU. This demonstrates that batching is an effective strategy for maximising parallelism and fully utilising server resources. However, for single lookups, alternative parallelisation approaches, such as parallelisation over cells, will need to be explored.

In certain cases, such as the Myers approach, the optimal 32× speed-up is not fully realised. This is due to memory management factors, including cache saturation and reliance on RAM, which could be addressed with careful memory handling.

Preprocessing: In section 5 we discussed preprocessing when one of the strings is not encrypted. Table 8 compares execution of the algorithm with and without

Table 7: Latency in seconds of parallelised implementation using 64 threads each calculating one Levenshtein distance. The first results of WF and Myers in the table only use the parallel TFHE-rs API, the following results use both the parallel API and batched inputs. Relative speedups compare with sequential results from Table 6.

	$m = 8$		$m = 100$		$m = 256$	
	Lat.		Lat.		Lat.	
WF (TFHE-rs parallel ops.)	32.28	2.4×	1h 29m	2.4×	9h 8m	2.4×
Myers (TFHE-rs parallel ops.)	12.00	1.5×	15m 28s	2.5×	1h 41m	2.4×
WF (batch inputs)	2.60	29.8×	6m 30s	31.4×	42m 52s	31.2×
Myers (batch inputs)	0.91	19.5×	2m 45s	13.9×	18m 30s	13.4×
Lvs Exact	0.15	18.9×	14.7	29.9×	1m 35	30.5×
Lvs Exact Skip	0.14	16.3×	11.3	29.6×	1m 12	29.7×
Lvs Exact $\ell = \frac{m}{4}$	0.11	13.6×	6.88	28.9×	43.7	28.6×
Lvs Exact $\ell = 10$	-	-	3.39	25.4×	10.4	22.1×

Table 8: Latency results for the case of one unencrypted string, both the latency of the building up the preprocessing table and main algorithm are given. All results in seconds. The relative speedup is given for the main calculation.

	$n = 8$			$n = 100$			$n = 256$		
	pre	main		pre	main		pre	main	
Lvs Exact - No Prep.	-	2.83	1×	-	439	1×	-	2903	1×
Lvs Exact	29.6	0.93	3×	369	146	3×	942	950	3×
Lvs Exact Skip - No Prep.	-	2.28	1×	-	335	1×	-	2141	1×
Lvs Exact Skip	29.5	0.75	3×	367	110	3×	946	717	3×
Lvs Exact $\ell = \frac{m}{4}$ - No Prep.	-	1.5	1×	-	199	1×	-	1252	1×
Lvs Exact $\ell = \frac{m}{4}$	29.6	0.49	3×	368	65	3×	946	420	3×
Lvs Exact $\ell = 10$ - No Prep.	-			-	86	1×	-	230	1×
Lvs Exact $\ell = 10$				370	29	3×	941	77	3×

preprocessing, based on the scenarios outlined above. All of the calculations are done using our Leuvenshtein algorithm.

From this table we can see that preprocessing reduces the computation with a factor 3×, as can be expected from the fact that character equality costs 2 PBS per cell while the Levenshtein calculation costs 1 PBS per cell. Note that the preprocessing numbers can be improved more if some ASCII characters are not used.

In general, one can discern three scenarios where preprocessing is useful:

- *Single Lookup*: In this scenario, a speedup is achievable when the alphabet size is smaller than the string length (i.e., $|\Sigma| < n$), as discussed in section 5. From the table, one can observe this effect for $n = 256$, where the total cost of preprocessing plus the main algorithm is lower than the cost without preprocessing.
- *Encrypted query against a large unencrypted database*: In this scenario, preprocessing needs to be performed only once. After that, only the main algorithm is executed. For large datasets, this approach clearly demonstrates a speedup approaching a factor of 3×.
- *Unencrypted query against an encrypted database:* Here, the database can undergo a one-time preprocessing step, either in plaintext or in the encrypted domain. Following this, only the main computation cost is incurred, resulting in a similar speedup of 3×.

6.3 Comparison with MPC implementations

In Multi-Party Computation (MPC) settings, a trade-off exists between computation time and bandwidth. Both FHE and MPC enable privacy-preserving computations, but directly comparing their specific security guarantees is challenging due to their differing threat models and trust assumptions. MPC categorises adversarial models into two main types: passive MPC, also called the *honest-but-curious* model, which operates under the assumption that compromised parties will adhere to the protocol while trying to infer confidential information. On the other hand, active MPC is designed to withstand adversaries who may deviate from the protocol in arbitrary ways to compromise correctness or obtain sensitive data.

Vanegas et al. [VCA23] implemented edit distance calculation in an MPC setting, optimising their implementation for DNA strings (i.e., 2-bit characters). They considered an passive and active security model with LAN network delays for both Garbled Circuit (GC) and secret-sharing schemes operating over a $\mathbb{Z}_{2^k}$ domain.

Table 9: Evaluation of the edit distance calculations using LAN setting MPC and FHE solutions for matching a 210-length DNA string. All tested on an `c6a.4xlarge` AWS EC2 instance.

Situation	Security	Algorithm	Lat. [s]	Data sent [MB]
MPC [VCA23]	Passive	Yao's GC	2.7	345.8
		Semi2^k	103.0	113.1
	Active	BMR-MASCOT	9, 034.0	2.09 x 106
		SPD$\mathbb{Z}_{2^k}$	368.5	14,893.8
FHE	-	Lvs Exact Skip	1820.8	1.38

For a fair comparison, we evaluated our algorithm using DNA strings of 210 characters in length on the same AWS EC2 instance of type `c6a.4xlarge`[4]. Specifically, only one PBS was utilised for the equality calculation, leaving the remainder of the algorithm unchanged. In addition, our solution easily generalises to arbitrary character sets and is not restricted to domain-specific inputs.

As shown in Table 9, compared to the passive MPC case, our solution is 674× slower but requires 251× less data to be sent. Compared to the active case, our solution is only 4.94× slower but requires 10 792× less data.

In other experiments, Vanegas et al. [VCA23] calculate the edit distance for DNA strings of length 1020. All implementations considered (plaintext, FHE, and MPC) exhibit quadratic cost growth as the input string lengths increase. This trend is evident in our results for string lengths between 100 and 256 and is expected to hold for even larger input sizes. Consequently, the cost scaling between FHE and MPC remains independent of input length, meaning that the ratio of their computational costs remains approximately constant, regardless of how long the input strings are.

It is also worth noting that the approximate algorithm demonstrates significantly better performance due to its linear complexity. However, since this linear behaviour is consistent across all implementations (plaintext, FHE, and MPC), it does not affect the relative cost comparison between them.

7 Conclusion

This paper introduces a novel method for efficiently computing the edit distance on encrypted data within the TFHE framework. Our first contribution demonstrates how to streamline edit distance calculations by employing a compact ternary representation, reusing programmable bootstrapping (PBS) results, and computing a three-input minimum function in a single lookup. The resulting Leuvenshtein algorithm achieves a 94× reduction in the number of PBS operations compared to traditional approaches.

Our second contribution enhances equality checks, particularly for medium-sized inputs. For ASCII encoding, we reduced the lookup cost from 5 PBS operations in the state-of-the-art to just 2 PBS operations.

Finally, we introduced a preprocessing stage that precalculates equality checks when one of the inputs is unencrypted, enabling an additional speedup. For ASCII inputs, this approach achieves up to a 3× improvement.

Our implementation results demonstrate that the Leuvenshtein algorithm delivers speedups of up to 278× over the best available implementation, underscoring its efficiency. These optimisations significantly advance the practicality of encrypted edit distance computations, reducing computational overhead and enhancing scalability for real-world applications.

[4] https://aws.amazon.com/ec2/instance-types/c6a/

Acknowledgements

This work was supported in part by the Horizon 2020 ERC Advanced Grant (101020005 Belfort) and the CyberSecurity Research Flanders with reference number VOEWICS02. Wouter Legiest is funded by FWO (Research Foundation – Flanders) as Strategic Basic (SB) PhD fellow (project number 1S57125N).



References

[AAM17] Md Momin Al Aziz, Dima Alhadidi, and Noman Mohammed. Secure approximation of edit distance on genomic data. *BMC Med. Genomics*, 10(S2), July 2017.

[AHLR18] Gilad Asharov, Shai Halevi, Yehuda Lindell, and Tal Rabin. Privacy-preserving search of similar patients in genomic data. *Proceedings on Privacy Enhancing Technologies*, 2018(4):104–124, October 2018.

[AQA21] Mohannad Alkhalili, Mahmoud H Qutqut, and Fadi Almasalha. Investigation of applying machine learning for watch-list filtering in anti-money laundering. *iEEE Access*, 9:18481–18496, 2021.

[BDL+19] Jasper Beernaerts, Ellen Debever, Matthieu Lenoir, Bernard De Baets, and Nico Van de Weghe. A method based on the levenshtein distance metric for the comparison of multiple movement patterns described by matrix sequences of different length. *Expert Systems with Applications*, 115:373–385, 2019.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *J. Cryptol.*, 33(1):34–91, 2020.

[CJL+20] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Concrete: Concrete operates on ciphertexts rapidly by extending tfhe. In *WAHC 2020-8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, 2020.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[CKL15] Jung Hee Cheon, Miran Kim, and Kristin E. Lauter. Homomorphic computation of edit distance. In *Financial Cryptography Workshops*, volume 8976 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2015.

[Cou15] Council of European Union. Consolidated text: Directive (eu) 2015/2366 of the european parliament and of the council of 25 november 2015 on payment services in the internal market, amending directives 2002/65/ec, 2009/110/ec and 2013/36/eu and regulation (eu) no 1093/2010, and repealing directive 2007/64/ec (text with eea relevance). *OJ*, L 337:35, 2015.

[EC22] EC. Legislative proposal on instant payments. *European Commission*, 2022.

[FSB21] FSB. G20 targets for enhancing cross-border payments. *Financial Stability Board, Bank for International Settlements*, 2021.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

[GBP+23] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios D. Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. BASALISC: programmable hardware accelerator for BGV fully homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(4):32–57, 2023.

[Joy22] Marc Joye. SoK: Fully homomorphic encryption over the [discretized] torus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):661–692, 2022.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[MA17] Nick J Maxwell and David Artingstall. The role of financial information-sharing partnerships in the disruption of crime. *Royal United Services Institute for Defence and Security Studies, FFIS*, 2017.

[Max21] Nick J Maxwell. Innovation and discussion paper: Case studies of the use of privacy preserving analysis to tackle financial crime. *Royal United Services Institute for Defence and Security Studies, FFIS*, 2021.

[MP80] William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[Mye86] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[Mye99] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, 1999.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[PHRL19] Alexander Payne, Nadine Holmes, Vardhman K. Rakyan, and Matthew Loose. Bulkvis: a graphical viewer for oxford nanopore bulk FAST5 files. *Bioinform.*, 35(13):2193–2198, 2019.

[SAE+08] Zhan Su, Byung-Ryul Ahn, Ki-Yol Eom, Min-Koo Kang, Jin-Pyung Kim, and Moon-Kyun Kim. Plagiarism detection using the levenshtein distance and smith-waterman algorithm. In *2008 3rd International Conference on Innovative Computing Information and Control*, pages 569–569, 2008.

[Ukk85a] Esko Ukkonen. Algorithms for approximate string matching. *Inf. Control.*, 64(1-3):100–118, 1985.

[Ukk85b] Esko Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985.

[Ukk92] Esko Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, 1992.

[VCA23] Hernán Vanegas, Daniel Cabarcas, and Diego F. Aranha. Privacy-preserving edit distance computation using secret-sharing two-party computation. In *LATINCRYPT*, volume 14168 of *Lecture Notes in Computer Science*, pages 67–86. Springer, 2023.

[vDTV23] Michiel van Beirendonck, Jan-Pieter D'Anvers, Furkan Turan, and Ingrid Verbauwhede. FPT: A fixed-point accelerator for torus fully homomorphic encryption. In *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 741–755. ACM Press, November 2023.

[Vin68] T. K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4:52–57, 1968.

[WF74] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[Zam22a] Zama. Concrete: TFHE Compiler that converts python programs into FHE equivalent, 2022. `https://github.com/zama-ai/concrete`.

[Zam22b] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. `https://github.com/zama-ai/tfhe-rs`.

[ZLS+19] Yandong Zheng, Rongxing Lu, Jun Shao, Yonggang Zhang, and Hui Zhu. Efficient and privacy-preserving edit distance query over encrypted genomic data. In *2019 11th International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, 2019.

8 Our complete algorithm

Algorithm 4 Complete ASCII-based Encrypted Levenshtein

function EDIT DISTANCE($x_{1..m}, y_{1..n}, \ell$)
▷ Setup
  $h \leftarrow$ OneMatrix$[0..m, 0..n]$
  $v \leftarrow$ OneMatrix$[0..m, 0..n]$
  LUT-$\text{EQ}_9 \leftarrow [9, 0, 0, \dots]$
  LUT-EQ $\leftarrow [1, 0, 0, \dots]$
  LUT-min[key] $\leftarrow$ Table 2
  for $i \leftarrow 1$ to m **do**
    $v[i, 0] \leftarrow 1$
  for $j \leftarrow 1$ to n **do**
    $h[0, j] \leftarrow 1$

▷ Main Algorithm
  for $j \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to m **do**
      if $|i - j| < \ell$ **then**
        $z_1 \leftarrow x_i^{(1)} - y_j^{(1)}$
        $\text{EQ}_1 \leftarrow \text{PBS}(z_1; \text{LUT-EQ})$
        $\text{EQ}_1 \leftarrow 1 - \text{EQ}_1$
        $z_2 \leftarrow x_i^{(2)} - y_j^{(2)}$
        $z_2 \leftarrow 2 \cdot z_2 + \text{EQ}_1$
        $\text{EQ}_9 \leftarrow \text{PBS}(z_2; \text{LUT-EQ}_9)$
        $H_{in} \leftarrow h[i - 1, j] + 1$
        $V_{in} \leftarrow v[i, j - 1] + 1$
        $key \leftarrow (1 - v[i, j - 1]) + 3 \cdot (1 + h[i - 1, j]) + \text{EQ}_9[i, j]$
        $min \leftarrow \text{PBS}(key, \text{LUT-min})$
        $v[i, j] \leftarrow min - h[i - 1, j]$
        $h[i, j] \leftarrow min - v[i, j - 1]$
  return $m + \Sigma_{i=0}^{n} h[m, i]$