Scilight

Pragmatic Cybersecurity
https://www.sciltp.com/journals/pc

Article

Check for updates

Efficient Homomorphic String Search via TFHE †

Shintaro Narisada [1,*], Hiroki Okada [1,2], Takashi Nishide [3] and Kazuhide Fukushima [1]

[1] KDDI Research, Inc., Fujimino-shi 356-8502, Japan
[2] Graduate School of Information Science and Technology, The University of Tokyo, Tokyo 113-8654, Japan
[3] Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba 305-8577, Japan
* Correspondence: sh-narisada@kddi.com
† Extended from a non-archival presentation titled: Efficient Homomorphic String Search via TFHE. Presented at the 31st Australasian Conference on Information Security and Privacy, Perth, WA, Australia, 6–9 July 2026.

How To Cite: Narisada, S.; Okada, H.; Nishide, T.; et al. Efficient Homomorphic String Search via TFHE. *Pragmatic Cybersecurity* **2026**, *1*(2), 13. https://doi.org/10.53941/pc.2026.100013

Received: 9 July 2026
Revised: 13 August 2026
Accepted: 14 August 2026
Published: 21 August 2026

Abstract: We present a method for secure pattern matching over encrypted texts using TFHE. Our approach realizes a fully secure binary search algorithm by leveraging two operational modes of integer-input TFHE. While the BGV-based method of Bonte and Iliashenko (CCSW'20) requires $O(|P| \cdot |T|)$ secure character comparisons to find a pattern P in a text T, our method reduces this to $O(|P| \log |T|)$ comparisons, achieving improved scalability for large texts. As a result, our method can find a pattern of length 100 in an encrypted text containing genomic data of one million characters in less than 5 min, where prior work would require approximately 5 days for the same task. These results highlight the practicality of TFHE and its potential for large-scale secure string search.

Keywords: TFHE; secure search; secure pattern matching

1. Introduction

As cloud computing becomes central to data processing, the demand for privacy-preserving technologies has increased. *Secure search* [1] is a general framework for querying encrypted data stored in the cloud in a way that matches a client's query without revealing any information to the cloud, and has applications including genome sequence analysis, confidential document inspection, and privacy-aware recommendation systems. Among the various secure search primitives, including fully homomorphic encryption (FHE) [1,2], searchable encryption (SE) [3,4], and property preserving encryption (PPE) [5,6], as well as a public key encryption primitive called Stream Encryption supporting Pattern Matching (SEPM) [7–10], we focus on the setting that uses FHE.

FHE enables arbitrary computations over encrypted data and serves as a simple, leakage-resistant framework for secure search. Existing FHE-based secure search can be classified into two categories: (1) The first focuses on minimizing the cost of searching for a match in the cloud (the matching phase), at the expense of preprocessing and postprocessing on the client side; this category is generally referred to as *secure pattern matching* (SPM) [2,11]. In SPM, we regard data as a string, the input data stored in the cloud is called a *text*, whereas the query is called a *pattern*; (2) The second prioritizes minimizing the client time and communication cost, and is referred to as *(setup-free) secure search* [1,12].

Since the bottleneck of secure search lies in the matching phase for large datasets, in this paper we focus on SPM to reduce the matching cost. A representative FHE-based SPM protocol by Bonte and Iliashenko [11] requires $O(|P| \cdot |T|)$ time to search for a pattern P in an encrypted text T.

Several recent studies have considered secure pattern matching under different cryptographic and system models. Bkakria and Izabachène [13] proposed a method using the BFV scheme for encrypted data streams, combining the data fragmentation technique of [8] with homomorphic Hamming-distance computation. CIPHERMATCH [14] uses the BFV scheme and accelerates exhaustive exact matching by using memory-efficient data packing and performing homomorphic additions within an SSD. Together with the method of Bonte and Iliashenko [11], these approaches examine all candidate positions or fragments and thus have matching complexity that is linear in the text length.

Despite these advances, in the FHE-based SPM setting, it remains unclear whether a protocol can exploit a specialized string data structure, such as a suffix array [15] or a suffix tree [16], and achieve $o(|P| \cdot |T|)$ matching complexity.

Another issue of secure search protocols using FHE schemes is their limited scalability to large data sizes, particularly regarding the query length. For example, in the latest methods [11,12], a query is assumed to be stored in a single ciphertext. It remains unclear how to efficiently scale such protocols to support longer queries across multiple ciphertexts.

1.1. Contribution

In this paper, we propose an efficient and scalable SPM protocol for searching a pattern P in an encrypted text T. Our protocol achieves $O(|P| \log |T|)$ time complexity in the matching phase by securely realizing binary search over a suffix array [15] within the TFHE scheme [17,18].

Specifically, our SPM protocol consists of the following three phases:

(1) **Preprocessing Phase**: The client constructs a suffix array of the text and encrypts both the text and the suffix array as secure look-up tables (LUTs), which are then stored in the cloud. For moderate-sized texts, this one-time plaintext preprocessing cost remains manageable. Constructing a suffix array of size $|T|$ requires $O(|T| \log |T|)$ time in plaintext [19], and its practical cost is generally lower than that of the matching phase, in which all computations are performed on ciphertexts. Our protocol assumes that the data owner possesses the plaintext before encryption and has sufficient local computational resources. Therefore, it is not directly applicable to data that is already encrypted and stored in the cloud.

(2) **Matching Phase**: With an encrypted pattern P, the cloud performs binary search on the LUTs to securely obtain a candidate string $T^{(i)}$ using the CMux tree algorithm [18], a core component of TFHE. An efficient string comparison between $T^{(i)}$ and P is then conducted using the tree-PBS algorithm [20], which allows homomorphic computation of the lexicographic order of two strings in $O(|P|)$ time, resulting in a total matching complexity of $O(|P| \log |T|)$. Compared with fragment-based methods for encrypted streams [13], which pack each overlapping fragment into a ciphertext and efficiently evaluate the Hamming distances for multiple candidate positions within the fragment in parallel, our method extracts a substring specified by an encrypted suffix array index and compares it with the pattern character by character.

(3) **Postprocessing Phase**: The client directly receives only the encrypted matching position i, and no postprocessing is required.

1.1.1. Scalability of Query Length

Our protocol does not impose any restriction on the length of the input query, while maintaining its effectiveness. We provide a proof showing how a query can be securely extended to an arbitrary length.

1.1.2. Security

Our protocol preserves the privacy of T, P, and the matching position i against an honest-but-curious cloud. The true text and pattern lengths are hidden within their public padded lengths, which reveal only upper bounds on the true lengths.

1.1.3. Experimental Results

We compare the performance of our method with existing FHE-based SPM protocols and demonstrate better scalability for larger text sizes. Specifically, our method can find a pattern of length 100 in a text of a quarter-billion characters in about 30 min, whereas the previous method [11] took about 10 min to find a length-100 pattern in only 1182 characters.

To ensure reproducibility, all codes are publicly available at https://github.com/kr-isg-projects/tfhe-pattern-match (accessed on 13 August 2026).

2. Preliminaries

2.1. Notation

Vectors are written in bold letters. For a vector $\mathbf{a}$, a_i denotes its i-th component. For $i < j$, we define $\mathbf{a}[i : j] = (a_i, \ldots, a_{j-1})$. The concatenation of two vectors $\mathbf{a}$ and $\mathbf{b}$ is denoted by $\mathbf{a} \,\|\, \mathbf{b}$. The set of integers $\{\ell, \ell + 1, \ldots, r\}$ is denoted by $[\ell, r]$. For a positive integer q, we identify $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ with the representative set $[-q/2, q/2 - 1]$. For a power of two N, let

$$\mathcal{R}_N = \mathbb{Z}[X]/(X^N+1) \qquad \text{and} \qquad \mathcal{R}_{q,N} = \mathbb{Z}_q[X]/(X^N+1).$$

The set $\mathbb{B}$ denotes $\{0,1\}$, and $\mathbb{B}_N[X] = \mathbb{B}[X]/(X^N+1)$.

For a finite set $\mathcal{A}$, we write $x \xleftarrow{\$} \mathcal{A}$ to denote that x is sampled uniformly at random from $\mathcal{A}$. For a distribution χ, we write $x \leftarrow \chi$ to denote that x is sampled from χ. Unless stated otherwise, all logarithms are to the base 2.

2.2. String Search

Let Σ be a finite alphabet, i.e., a finite set of characters. We call $|\Sigma|$ the alphabet size. A sequence of elements of Σ is called a string. Following convention, strings are denoted using non-bold uppercase letters.

Let T be a string called a *text* and P be a string called a *pattern*:

$$T = (t_0, t_1, \ldots, t_{|T|-1}), \quad P = (p_0, p_1, \ldots, p_{|P|-1}), \quad t_j, p_j \in \Sigma,$$

where $|P| \leq |T|$.

One of the most fundamental problems in string search is *pattern matching*, where given T and P, we aim to find a matching position i such that

$$T[i : i+|P|] = P.$$

For $0 \leq i \leq |T| - |P|$, let $T^{(i)}$ be the substring of length $|P|$ starting at position i:

$$T^{(i)} = T[i : i+|P|] = (t_i, t_{i+1}, \ldots, t_{i+|P|-1}),$$

which we call the i-th slice of T. An exhaustive search compares each substring $T^{(i)}$ with P for all $0 \leq i \leq |T|-|P|$, requiring $O(|P| \cdot |T|)$ character comparisons. Although faster pattern-matching algorithms exist in the plaintext setting [21], applying them directly in FHE schemes remains challenging. A key obstacle is the computational constraints of homomorphic operations, such as the limited precision of a single ciphertext, the high cost of non-linear operations over arbitrary precision, and the difficulty of supporting random access without incurring access-pattern leakage.

3. TFHE

We introduce the core components of the TFHE scheme [17,18] and the key homomorphic operations required to realize an efficient FHE-based SPM protocol.

3.1. Cryptographic Structures

We use p and q to denote the plaintext and ciphertext moduli, respectively. We define the scaling factor as $\Delta = q/p$. We define an LWE ciphertext as follows.

Definition 1 (LWE ciphertext)**.** *Given a message $m \in \mathbb{Z}_p$, its LWE ciphertext under the secret key $\mathbf{s} \in \mathbb{B}^n$ is defined as $\mathsf{LWE}_{\mathbf{s}}(\Delta m) := (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$, where $b = \langle \mathbf{a}, \mathbf{s} \rangle + \Delta m + e$. Additionally, $\mathbf{a} \xleftarrow{\$} \mathbb{Z}_q^n$, $\mathbf{s} \leftarrow \chi$, and $e \leftarrow \chi'$, where χ and χ' denote the key and error distributions, respectively.*

RLWE is the ring analogue of LWE over $\mathcal{R}_{q,N}$, which encrypts a polynomial message $M \in \mathcal{R}_{p,N}$.

Definition 2 (RLWE ciphertext)**.** *The RLWE ciphertext of the message $M \in \mathcal{R}_{p,N}$ under the secret key $\mathbf{S} \in (\mathbb{B}_N[X])^k$ is $\mathsf{RLWE}_{\mathbf{S}}(\Delta M) := (\mathbf{A}, B) \in \mathcal{R}_{q,N}^{k+1}$, where $B = \sum_{i=1}^{k} A_i \cdot S_i + \Delta M + E$, $A_i \xleftarrow{\$} \mathcal{R}_{q,N}$, $\mathbf{S} \leftarrow \chi$, and $E \leftarrow \chi'$.*

The (R)Lev ciphertext is a vector of (R)LWE ciphertexts constructed using a gadget vector.

Definition 3 ((R)Lev ciphertext)**.** *Given a gadget vector $\mathbf{v} = (v_1, \ldots, v_\ell) \in \mathbb{Z}^\ell$, the* (R)Lev *ciphertext is an ℓ-dimensional vector of (R)LWE ciphertexts. Specifically,* RLev *is defined by*

$$\mathsf{RLev}_{\mathbf{S}}(M) := (\mathsf{RLWE}_{\mathbf{S}}(v_1 \cdot M), \ldots, \mathsf{RLWE}_{\mathbf{S}}(v_\ell \cdot M)) \in \mathcal{R}_{q,N}^{(k+1)\ell}.$$

Lev is defined in the same manner. Finally, we define the RGSW ciphertext, a key component in TFHE.

Definition 4 (RGSW ciphertext)**.** *Given a message polynomial $M \in \mathcal{R}_{p,N}$, an RGSW ciphertext is defined as a vector of $k+1$ RLev ciphertexts:*

$$\mathsf{RGSW}_{\mathbf{S}}(M) := (\mathsf{RLev}_{\mathbf{S}}(-S_1 \cdot M), \ldots, \mathsf{RLev}_{\mathbf{S}}(-S_{k+1} \cdot M)) \in \mathcal{R}_{q,N}^{(k+1)^2\ell},$$

where $\mathbf{S} = (S_1, \ldots, S_k)$ is the RLWE secret key and $S_{k+1} = -1$.

3.2. Basic Homomorphic Operations

TFHE supports homomorphic addition and scalar multiplication on (R)LWE ciphertexts. To multiply two ciphertexts, the external product is required.

Definition 5 (External Product (XP))**.** *Let $\overline{\overline{\mathbf{C}}}_1 = \mathsf{RGSW}(M_1)$ and $\mathbf{C}_2 = \mathsf{RLWE}(\Delta M_2) = (A_1, \ldots, A_k, A_{k+1} = B)$. The external product $\boxdot : \mathsf{RGSW} \times \mathsf{RLWE} \to \mathsf{RLWE}$ is defined as*

$$\begin{aligned}\overline{\overline{\mathbf{C}}}_1 \boxdot \mathbf{C}_2 &:= \sum_{i=1}^{k+1} (A_i \odot \mathsf{RLev}(-S_i \cdot M_1)) \\ &= \mathsf{RLWE}\left(\left(\sum_{i=1}^{k+1} A_i \cdot S_i\right) \cdot M_1\right) \\ &= \mathsf{RLWE}(\Delta M_1 \cdot M_2),\end{aligned}$$

where $\odot : \mathcal{R}_{q,N} \times \mathsf{RLev} \to \mathsf{RLWE}$ denotes the gadget product, defined by

$$A \odot \mathsf{RLev}(M) \;:=\; \mathsf{RLWE}(A \cdot M).$$

Using the XP operation together with RLWE addition, a key homomorphic operation, called the Controlled MUX (CMux) gate, can be constructed.

Definition 6 (Controlled MUX gate (CMux) [17,22])**.** *Let $\mathbf{C}_0, \mathbf{C}_1 \in \mathsf{RLWE}$ and let $\overline{\overline{\mathbf{C}}} \in \mathsf{RGSW}$ be an encryption of a bit $b \in \{0, 1\}$. The CMux gate $\mathsf{CMux} : \mathsf{RGSW} \times \mathsf{RLWE} \times \mathsf{RLWE} \to \mathsf{RLWE}$ is defined by*

$$\mathsf{CMux}(\overline{\overline{\mathbf{C}}}, \mathbf{C}_0, \mathbf{C}_1) := \overline{\overline{\mathbf{C}}} \boxdot (\mathbf{C}_1 - \mathbf{C}_0) + \mathbf{C}_0.$$

The CMux gate homomorphically selects an RLWE ciphertext depending on an encrypted control bit b. Using the CMux operation, one can construct blind rotation, which is the core component in the bootstrapping procedure in TFHE.

3.3. Programmable Bootstrapping and Key Switching

In the FHE setting, bootstrapping refers to the procedure of refreshing the noise accumulated during homomorphic computations, thereby enabling the evaluation of circuits of arbitrary depth. Moreover, TFHE allows one to evaluate a look-up table (LUT) of size at most p (generally $p/2$) during the bootstrapping process itself, which is known as programmable bootstrapping (PBS).

In this paper, we describe the generalized variant, PBSmanyLUT [23], which enables the simultaneous evaluation of 2^ν LUTs ($\nu \geq 0$) using a single blind rotation.

Definition 7 (PBSmanyLUT)**.** *The inputs to the* PBSmanyLUT *procedure are $\mathbf{c} = \mathsf{LWE}_{\mathbf{s}}(\Delta m) \in \mathbb{Z}_q^{n+1}$ under the LWE secret key $\mathbf{s} = (s_1, \ldots, s_n)$, $\mathbf{C} = \mathsf{RLWE}_{\mathbf{S}'}(\Delta M_{(f_0,\ldots,f_{2^\nu-1})})$ under the RLWE secret key $\mathbf{S}' = (S'_1, \ldots, S'_k)$, where $M_{(f_0,\ldots,f_{2^\nu-1})}$ encodes the LUTs $f_0, \ldots, f_{2^\nu-1}$ with $f_j : \mathbb{Z}_p \to \mathbb{Z}_p$ for $0 \leq j < 2^\nu$, and $\mathsf{bsk} = (\mathsf{RGSW}_{\mathbf{S}'}(s_i))_{1 \leq i \leq n}$. It outputs refreshed LWE ciphertexts $\mathbf{c}_0, \ldots, \mathbf{c}_{2^\nu-1}$ such that $\mathbf{c}_j = \mathsf{LWE}_{\mathbf{s}'}(\Delta f_j(m)) \in \mathbb{Z}_q^{kN}$, where $\mathbf{s}' \in \mathbb{B}^{kN}$ is the LWE secret key corresponding to $\mathbf{S}'$.*

Note that in this paper, we encrypt the LUTs as (non-trivial) RLWE ciphertexts since the LUTs are private. The PBSmanyLUT procedure consists of the following three steps:

(1) (ModulusSwitch) For the input LWE ciphertext $\mathbf{c} = (a_1, \ldots, a_n, b) \in \mathbb{Z}_q^{n+1}$, compute $\tilde{\mathbf{c}} = (\tilde{a}_1, \ldots, \tilde{a}_n, \tilde{b}) \in \mathbb{Z}_{2N}^{n+1}$, where $\tilde{a}_i = \left\lfloor \frac{a_i \cdot 2N \cdot 2^{-\nu}}{q} \right\rceil \cdot 2^\nu$ and $\tilde{b} = \left\lfloor \frac{b \cdot 2N \cdot 2^{-\nu}}{q} \right\rceil \cdot 2^\nu$.

(2) (BlindRotate) Homomorphically compute $\widetilde{\mathbf{C}} = \mathbf{C} \cdot X^{-\tilde{b}+\sum_{i=1}^{n} \tilde{a}_i s_i}$, where each term $X^{\tilde{a}_i s_i}$ is computed using the CMux operation with the bootstrapping key bsk.

(3) (SampleExtract) Extract the j-th ($0 \leq j < 2^{\nu}$) coefficient of the message polynomial $\Delta f_j(m)$ in $\widetilde{\mathbf{C}}$ as an LWE ciphertext $\mathbf{c}_j = \mathsf{LWE}_{\mathbf{s}'}(\Delta f_j(m))$.

If $n+1 \neq kN$, the LWE dimension changes after the PBS. In practice, we have $n+1 < kN$ to reduce the number of CMux operations in BlindRotate. To change the LWE dimension, LWE-to-LWE key switching (denoted as KeySwitch) is used.

Definition 8 (Key Switching [17])**.** *The inputs to* KeySwitch *are* $\mathbf{c} = \mathsf{LWE}_{\mathbf{s}}(\Delta m)$ *under the secret key* $\mathbf{s} = (s_1, \ldots, s_n)$*, and* $\mathsf{ksk} = (\mathsf{Lev}_{\mathbf{s}'}(s_i))_{1 \leq i \leq n}$*, where* $\mathbf{s}' = (s'_1, \ldots, s'_{n'})$ *is the secret key. It outputs an LWE ciphertext* $\mathbf{c}' = \mathsf{LWE}_{\mathbf{s}'}(\Delta m) \in \mathbb{Z}_q^{n'+1}$.

In TFHE, the Fully Homomorphic Evaluation (FHE) mode consists of PBS, KeySwitch, and basic homomorphic operations (additions and scalar multiplications). To maximize efficiency in FHE mode, KeySwitch is performed immediately before PBS, and this combined operation is denoted by KS-PBS.

Sample extraction converts an RLWE ciphertext into an LWE ciphertext. Conversely, LWE-to-RLWE key switching can convert an LWE ciphertext into an RLWE ciphertext. In [24], an efficient LWE-to-RLWE key switching using the homomorphic trace function is proposed.

Definition 9 (LWE-to-RLWE KS [24])**.** *The inputs to* LWE-to-RLWE-KS *are* $\mathbf{c} = \mathsf{LWE}_{\mathbf{s}}(\Delta m) \in \mathbb{Z}_q^{n+1}$ *under the secret key* $\mathbf{s} = (s_1, \ldots, s_n)$*, and the automorphism keys* $\mathsf{atk}_{d_j} = (\mathsf{RLev}_{\mathbf{S}}(S_i(X^{d_j})))_{1 \leq i \leq k}$ *under the secret key* $\mathbf{S} = (S_1, \ldots, S_k)$ *for* $1 \leq j \leq \log N$*. Here,* $d_j \in \mathbb{Z}_{2N}^{\times}$ *are the specific exponents required to evaluate a trace function* HomTrace*. It outputs* $\mathbf{C} = \mathsf{RLWE}_{\mathbf{S}}(\Delta m) \in \mathcal{R}_{q,N}^{k+1}$.

3.4. Circuit Bootstrapping

The original version of circuit bootstrapping (CBS) converts an LWE ciphertext encrypting a bit $b \in \{0, 1\}$ into a fresh RGSW ciphertext [18], which allows the evaluation of a sequence of CMux gates, referred to as the Leveled Homomorphic Evaluation (LHE) mode of TFHE. For integer-input TFHE, generalized variants of CBS were proposed by Bergerat et al. [25] and later improved by Wang et al. [26].

Definition 10 (CBS for Integer-Input LHE Mode [26])**.** *Given a* $\log p$*-bit message* $m = \sum_{i=0}^{\log p - 1} m_i \cdot 2^i \in \mathbb{Z}_p$ *with* $m_i \in \{0, 1\}$*, the algorithm* CBS *takes as input* $\mathbf{c} = \mathsf{LWE}_{\mathbf{s}}(\Delta m) \in \mathbb{Z}_q^{n+1}$ *under the LWE secret key* $\mathbf{s} = (s_1, \ldots, s_n)$*, and a CBS key* cbk*, which consists of* bsk*,* ksk*,* atk*, and a scheme-switching key. It outputs fresh RGSW ciphertexts* $\left(\overline{\overline{\mathbf{C}}}_i = \mathsf{RGSW}(m_i)\right)_{i=0}^{\log p - 1}$.

In the improved CBS algorithm, we iteratively convert the i-th bit of the message in $\mathbf{c} = \mathsf{LWE}_{\mathbf{s}}(\Delta m)$ into $\overline{\overline{\mathbf{C}}}_i = \mathsf{RGSW}(m_i)$. To do so, we first perform a scalar multiplication $\mathbf{c} \cdot p \cdot 2^{-i-1}$ to lift the bit m_i to the MSB of $\mathbf{c}$, followed by KeySwitch, PBSmanyLUT, HomTrace, and the scheme-switching algorithm. For further details, see Section 5.2 in [26].

3.5. Evaluate High-Precision Functions via Integer-Input FHE/LHE

Combined with the above procedures, we can construct homomorphic algorithms in both the LHE and FHE modes of TFHE to efficiently evaluate high-precision functions. For the FHE mode, we utilize the tree-PBS method proposed by Bourse et al. [20].

Definition 11 (Tree PBS [20] with a Binary Decision Tree)**.** *Let* $d \in \mathbb{N}$*,* $\delta \leq p/2$*, and* $\{\mathbf{c}_j\}_{j=0}^{2^d-1}$ *be* 2^d *LWE ciphertexts, where each* $\mathbf{c}_j = \mathsf{LWE}_{\mathbf{s}}(\Delta m_j)$ *encrypts a message* $m_j \in \mathbb{Z}_\delta$*. Let* $f : \mathbb{Z}_\delta^{2^d} \to \mathbb{Z}_\delta$ *be a function represented by a LUT of size* 2^d.

The treePBS *algorithm homomorphically evaluates a binary decision tree from leaves to root, outputting* $\mathsf{LWE}_{\mathbf{s}}(\Delta f(m_0, \ldots, m_{2^d-1}))$ *at the root. Specifically:*

(1) At depth d*, the* j*-th leaf stores* $\mathbf{c}_j^{(d)} = \mathsf{LWE}_{\mathbf{s}}(\Delta m_j^{(d)})$ *for* $0 \leq j \leq 2^d - 1$.

(2) At depth $0 < i < d$*, the* j*-th node stores* $\mathbf{c}_j^{(i)} = \mathsf{LWE}_{\mathbf{s}}(\Delta m_j^{(i)})$ *for* $0 \leq j \leq 2^i - 1$*, where* $m_j^{(i)} = f_{i,j}(m_{2j}^{(i+1)}, m_{2j+1}^{(i+1)})$ *and* $f_{i,j} : \mathbb{Z}_\delta^2 \to \mathbb{Z}_\delta$ *is a subfunction of* f.

(3) At depth 0*, the root stores* $\mathsf{LWE}_{\mathbf{s}}(\Delta f(m_0, \ldots, m_{2^d-1}))$.

The function f must be decomposable into the subfunctions $f_{i,j}$.

The algorithm essentially requires $2^d - 1$ evaluations of PBS. It can also be generalized to k-ary decision trees for any $k \geq 2$. Using treePBS, the authors in [20] propose an efficient algorithm for homomorphically comparing two large integers.

For the LHE mode, one of the most important algorithms is the CMux tree [17,22], combined with the vertical packing (VP) technique [18], which allows the homomorphic evaluation of arbitrary functions.

Definition 12 (VP CMux Tree [18])**.** *Let $d \geq \log N$, $m = \sum_{i=0}^{d-1} 2^i m_i$, with $m_i \in \{0,1\}$, and let $(\overline{\overline{\mathbf{C}}}_i)_{i=0}^{d-1}$ be RGSW ciphertexts, where each $\overline{\overline{\mathbf{C}}}_i = \mathsf{RGSW}_{\mathbf{S}}(m_i)$ encrypts m_i. Let $f : [0,1)^d \to \mathbb{Z}_p$ be an arbitrary function represented by a LUT of size 2^d.*

The $\mathsf{CMuxTree}_f(m)$ algorithm evaluates a binary decision tree from leaves to root, and outputs at the root:

$$\mathsf{LWE}_{\mathbf{s}}(\Delta f(m_0, \ldots, m_{d-1}))$$

(1) At depth $d' = d - \log N$, the j-th leaf stores $\mathbf{C}_j^{(d')} = \mathsf{RLWE}_{\mathbf{S}}(\Delta M_j^{(d')})$ for $0 \leq j \leq 2^{d'} - 1$, where $\{M_j^{(d')}\}_{j=0}^{2^{d'}-1}$ encode the LUT of f.

(2) At depth $0 < i < d'$, the j-th node stores $\mathbf{C}_j^{(i)} = \mathsf{RLWE}_{\mathbf{S}}(\Delta M_j^{(i)})$ for $0 \leq j \leq 2^i - 1$, where

$$\mathbf{C}_j^{(i)} = \mathsf{CMux}(\overline{\overline{\mathbf{C}}}_{i+\log N}, \mathbf{C}_{2j}^{(i+1)}, \mathbf{C}_{2j+1}^{(i+1)}).$$

(3) At depth 0, perform a $\mathsf{BlindRotate}$ on $\mathbf{C}_0^{(0)}$ using the RGSW ciphertexts $(\overline{\overline{\mathbf{C}}}_i)_{i=0}^{\log N-1}$ to compute $\mathbf{C}_0^{(0)} \cdot X^{-\sum_{i=0}^{\log N-1} 2^i m_i}$. Then, apply $\mathsf{SampleExtract}$ to obtain the final LWE ciphertext.

To evaluate an arbitrary function $F : [0,1)^d \to [0,1)^d$, we iteratively perform the CMuxTree algorithm $\lceil d/\log p \rceil$ times for subfunctions f_j, where $0 \leq j < \lceil d/\log p \rceil$, and each $f_j : [0,1)^d \to \mathbb{Z}_p$ outputs the j-th segment of F's output. It is also possible to encode multiple subfunctions f_j in a single RLWE ciphertext. This reduces the number of CMuxTree executions and is referred to as horizontal packing (Section 5.1 in [18]).

4. Fully Homomorphic String Search

In this section, we first review an FHE-based SPM algorithm as a baseline construction, which requires $O(|P| \cdot |T|)$ PBS executions in the integer-input FHE mode of TFHE. We then propose an improved FHE-based SPM algorithm with complexity $O(|P| \log |T|)$, leveraging both the integer-input FHE mode and the LHE mode of TFHE.

4.1. Protocol

Both the baseline and proposed methods follow the standard SPM protocol [2], with a minor modification to reduce communication costs. The protocol involves three parties: a *user*, who owns the secret text T; a *client*, who owns the secret pattern P; and a *server*, which performs secure computation. A *user* and a *client* may be treated as the same entity. The protocol relies on an FHE scheme, which may be based on either symmetric-key or public-key encryption, as in TFHE [27,28]. Here, $[\![x]\!]$ denotes the ciphertext of x.

(1) **Setup Phase.** The client generates an FHE public key pk and a secret key sk (or pk = sk in the symmetric-key case), and distributes pk to the user.

(2) **Text Preprocessing and Sending Phase.** The user preprocesses the plaintext T (e.g., by sorting or padding sentinel characters to hide its length) to generate D_T, a data structure representing T (if necessary). The user then encrypts D_T using pk and sends $[\![D_T]\!]$ to the server.

(3) **Matching Phase.** (1) The client encrypts the pattern P with padding using pk to obtain $[\![P]\!]$ and sends it to the server; (2) The server executes a function $\mathsf{FindMatch}([\![D_T]\!], [\![P]\!])$, which returns a matching position $[\![i]\!]$, if it exists, where $T[i : i + |P|] = P$; otherwise, it returns a failure symbol $[\![\perp]\!]$ (e.g., $\perp = -1$). The server then sends $[\![i]\!]$ to the client; (3) The client decrypts $[\![i]\!]$ using sk to obtain the matching result i.

Features of the Protocol

The key advantage of the protocol is its confidentiality against the server. The server does not learn the text, pattern, matching position, or their true lengths beyond the padded lengths and other public information. The communication cost between the client and the server is low, since the server only returns a single matching position

i. This is a key difference from the standard SPM protocol [2], in which the server returns a vector of length $|T| - |P| + 1$, where each index has a boolean value indicating a match.

4.2. Baseline Construction Using the FHE Mode of TFHE

4.2.1. Preprocessing and Message Encoding

For a text and a pattern over an alphabet of size $|\Sigma|$, we encode each character as integers $\tilde{t}_i, \tilde{p}_i \in [1, |\Sigma|]$, reserving the value 0 for padding. We then extend these vectors with 0s:

$$\widetilde{\mathbf{T}} = (\tilde{t}_0, \ldots, \tilde{t}_{|T|-1}, 0, \ldots, 0), \quad \widetilde{\mathbf{P}} = (\tilde{p}_0, \ldots, \tilde{p}_{|P|-1}, 0, \ldots, 0).$$

Finally, $\widetilde{\mathbf{T}}$ and $\widetilde{\mathbf{P}}$ are encrypted as vectors of LWE ciphertexts. In the following, we may omit explicit references to the cryptographic structures for simplicity.

4.2.2. Comparing the Slice and Pattern

In FindMatch, we first compute a vector $\mathbf{b}$ representing matching indicators:

$$\mathbf{b} = (b_0, \ldots, b_{|\widetilde{\mathbf{T}}|-|\widetilde{\mathbf{P}}|}), \quad b_i = \begin{cases} 0, & \widetilde{\mathbf{T}}^{(i)} \neq \widetilde{\mathbf{P}}, \\ 1, & \widetilde{\mathbf{T}}^{(i)} = \widetilde{\mathbf{P}}. \end{cases}$$

Since the plaintext modulus p is small in TFHE, directly computing the Hamming distance between two strings homomorphically [2] may produce incorrect results. To address this, we compute b_i using a chain of non-linear Boolean operations. For each slice $\widetilde{\mathbf{T}}^{(i)} = (\tilde{t}_i, \ldots, \tilde{t}_{i+|\widetilde{\mathbf{P}}|-1})$, we set

$$b_i = \bigwedge_{0 \le j \le |\widetilde{\mathbf{P}}|-1} \big(\mathsf{IsZero}(\tilde{t}_{i+j} - \tilde{p}_j) \vee \mathsf{IsZero}(\tilde{p}_j)\big), \quad \mathsf{IsZero}(x) = \begin{cases} 1, & x = 0, \\ 0, & \text{otherwise}. \end{cases}$$

This evaluates to 1 if and only if, for every position j, either $\tilde{t}_{i+j} = \tilde{p}_j$ or $\tilde{p}_j = 0$. The number of PBS operations required to compute $\mathbf{b}$ is $O(|\widetilde{\mathbf{T}}| \cdot |\widetilde{\mathbf{P}}|)$, since each $\wedge$, $\vee$, and evaluation of IsZero requires one PBS.

4.2.3. Obtaining a Matching Index from an Indicator

Given a matching indicator vector $\mathbf{b} = (b_0, \ldots, b_{|\widetilde{\mathbf{T}}|-|\widetilde{\mathbf{P}}|})$, we compute the following recurrence in reverse order:

$$s_{|\widetilde{\mathbf{T}}|-|\widetilde{\mathbf{P}}|} = b_{|\widetilde{\mathbf{T}}|-|\widetilde{\mathbf{P}}|} \cdot (|\widetilde{\mathbf{T}}| - |\widetilde{\mathbf{P}}|), \quad s_i = b_i \cdot (i - s_{i+1}) + s_{i+1}.$$

This recurrence is equivalent to a CMux operation. Since we only have LWE ciphertexts of $\mathbf{b}$, the multiplication $b_i \cdot (i - s_{i+1})$ is performed via PBS operations. To this end, we define a bivariate function ZCMux:

$$s_i = \mathsf{ZCMux}(b_i, i) + \mathsf{ZCMux}(1 - b_i, s_{i+1}), \quad \mathsf{ZCMux}(b, s) = \begin{cases} 0, & b = 0, \\ s, & b = 1. \end{cases}$$

A LUT evaluation of a bivariate function is also possible in the integer-wise TFHE [25]. Here, $\mathsf{ZCMux}(b, s)$ acts like a CMux gate with 0, enabling multiplication of ciphertexts via PBS. Finally, the matching index

$$s_0 = \min\{i \mid \widetilde{\mathbf{T}}^{(i)} = \widetilde{\mathbf{P}}\}$$

is obtained after performing $O(|\widetilde{\mathbf{T}}|)$ PBS evaluations.

4.3. Proposed Construction Using the Hybrid Mode of TFHE

In the baseline construction, equality is checked for every slice $(T^{(i)}, P)$, resulting in a complexity linear in the number of candidate positions $(|T| - |P| + 1)$. To reduce the number of equality checks while preserving obliviousness (i.e., preventing access pattern leakage), we propose combining a binary search over the suffix array [15] with secure LUT evaluation in TFHE [18]. This reduces the number of substring equality checks from $O(|T|)$ to $O(\log |T|)$, while maintaining both obliviousness and confidentiality.

4.3.1. Overview of Our Method

Let $T = (t_0, \ldots, t_{|T|-1})$ be a text. We denote the suffix of T starting at position i by

$$\mathsf{suf}_T(i) := (t_i, t_{i+1}, \ldots, t_{|T|-1}).$$

The suffix array of T is defined as

$$\mathsf{SA}_T := (\mathsf{SA}_T(0), \ldots, \mathsf{SA}_T(|T|-1)),$$

where $\mathsf{SA}_T(j) = i$ indicates that $\mathsf{suf}_T(i)$ is the j-th smallest suffix of T in lexicographic order.

It is known that the suffix array can be constructed in $O(|T| \log |T|)$ integer comparisons by the Larsson–Sadakane method [19]. Since SA_T stores the starting positions of the suffixes of T in lexicographic order, we can perform a binary search over SA_T to determine whether a pattern P occurs in T, which requires $O(\log |T|)$ lexicographic comparisons between P and suffixes of T.

In Algorithm 1, we present a high-level overview of our improved FindMatch function, which uses the suffix array SA_T. Given a text T and pattern P, along with the suffix array SA_T constructed during the preprocessing phase, we repeatedly halve the search region $[\ell, r]$ depending on the result of the Compare function for the two strings $T^{(i)}$ and P, where $i = \mathsf{SA}_T(\lfloor(\ell + r)/2\rfloor)$:

$$\mathsf{Compare}(A, B) = \begin{cases} -1, & A < B \quad (A \text{ is lexicographically smaller than } B), \\ 0, & A = B, \\ 1, & A > B \quad (A \text{ is lexicographically larger than } B). \end{cases}$$

If $T^{(i)} = P$, then i is returned as the matching position.

Figure 1 illustrates the protocol flow of our proposed SPM method based on Algorithm 1.

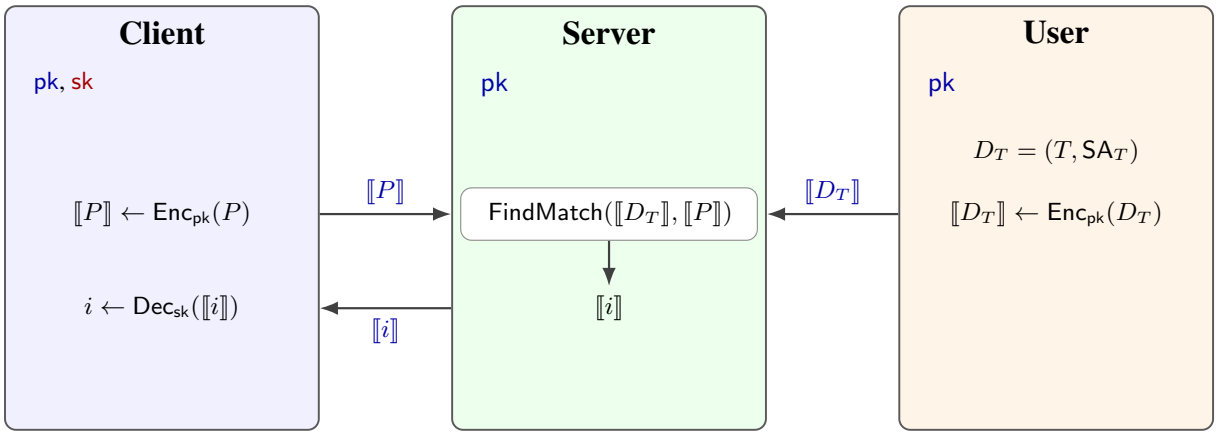


Figure 1. Protocol overview of the proposed secure pattern matching method. Here, T and P denote the text and pattern, respectively, and SA_T denotes the suffix array of T. Encryption and decryption are denoted by $\mathsf{Enc}_{\mathsf{pk}}(\cdot)$ and $\mathsf{Dec}_{\mathsf{sk}}(\cdot)$, respectively.

Algorithm 1: FindMatch (High-Level Overview)

Input: Text T, pattern P, and suffix array SA_T
Output: Matching index i^* such that $T^{(i^*)} = P$ if it exists; $\perp$ otherwise

$\ell \leftarrow 0$, $r \leftarrow |T| - 1$, $c \leftarrow \left\lfloor \frac{\ell + r}{2} \right\rfloor$
while $\ell \leq r$ **do**
  $i \leftarrow \mathsf{SA}_T(c)$
  $s \leftarrow \mathsf{Compare}(T^{(i)}, P)$
  if $s = 0$ **then**
    return $i^* = i$
  Update ℓ, r, c from s
return $\perp$

It is not obvious how to realize Algorithm 1 efficiently in a homomorphic manner, as it includes non-linear

operations, random access, and conditional statements. In the following, we instantiate each procedure in Algorithm 1 using the components of TFHE.

4.3.2. Message Encoding

Our message encoding slightly differs from the baseline method, as we aim to reduce the number of PBSs in the string comparison phase. For a character represented by an integer $t \in \mathbb{Z}_{|\Sigma|}$, we encode it in $\mathbb{Z}_p$ as follows:

$$\tilde{t} = \begin{cases} t+1, & 0 \le t \le |\Sigma|-1, \\ |\Sigma|+1, & \text{(sentinel character '\#')}, \\ -2(|\Sigma|+1), & \text{(wildcard character '}*\text{')}. \end{cases}$$

We avoid using 0 to prevent false positives caused by negation after $\mathsf{BlindRotate}$. The lexicographic order over encoded characters is induced by this integer ordering. Hence the sentinel character '#' is defined to have the largest lexicographic order. The wildcard character '$*$' is chosen outside the range $\{1, \dots, |\Sigma|+1\}$ so that it does not collide with ordinary or sentinel characters under negation.

The padded text $\widetilde{\mathbf{T}}$ is defined as a sequence of encoded characters $\widetilde{\mathbf{T}} = (\tilde{t}_0, \dots, \tilde{t}_{|\widetilde{\mathbf{T}}|-1})$, where $\tilde{t}_j = t_j + 1$ for $0 \le j < |T|$, and $\tilde{t}_j = |\Sigma| + 1$ for $j \ge |T|$. The padded pattern $\widetilde{\mathbf{P}}$ is obtained by appending wildcard symbols '$*$' to the encoded P. Without loss of generality, we assume that $|\widetilde{\mathbf{T}}| \ge N$ is a power of two and $|\widetilde{\mathbf{P}}| \ge N/2$ is a multiple of $N/2$, by appending sufficient padding symbols, for ease of exposition.

4.3.3. Preprocessing

For the padded text $\widetilde{\mathbf{T}} = (\tilde{t}_0, \dots, \tilde{t}_{|\widetilde{\mathbf{T}}|-1})$, the user first constructs the suffix array in plaintext:

$$\mathsf{SA}_{\widetilde{\mathbf{T}}} = (\mathsf{SA}(0), \dots, \mathsf{SA}(|\widetilde{\mathbf{T}}|-1)),$$

where each entry satisfies $\mathsf{SA}(j) \in \mathbb{Z}_{|\widetilde{\mathbf{T}}|}$. As the length of $\widetilde{\mathbf{T}}$ is not limited, we decompose $\mathbb{Z}_{|\widetilde{\mathbf{T}}|}$ into smaller components as $\mathbb{Z}_{|\widetilde{\mathbf{T}}|} \cong \mathbb{Z}_\delta^\beta$, where $\beta = \lceil \log_\delta |\widetilde{\mathbf{T}}| \rceil$ and $\delta \le p$. Accordingly, $\mathsf{SA}_{\widetilde{\mathbf{T}}}$ is decomposed into β arrays $(\mathsf{SA}^{(k)}_{\widetilde{\mathbf{T}}})_{0 \le k < \beta}$, where $\mathsf{SA}^{(k)}_{\widetilde{\mathbf{T}}}$ stores the k-th component in the above decomposition.

Then, $\widetilde{\mathbf{T}}$ and $(\mathsf{SA}^{(k)}_{\widetilde{\mathbf{T}}})_{0 \le k < \beta}$ are encrypted as RLWE ciphertexts, with message polynomials defined as

$$\begin{aligned} M_1^{(j)}(X) &= \tilde{t}_{jN} + \tilde{t}_{jN+1}X + \cdots + \tilde{t}_{(j+1)N-1}X^{N-1}, \\ M_{\mathsf{SA}}^{(j,k)}(X) &= \mathsf{SA}^{(k)}_{\widetilde{\mathbf{T}}}(jN) + \mathsf{SA}^{(k)}_{\widetilde{\mathbf{T}}}(jN+1)X + \cdots + \mathsf{SA}^{(k)}_{\widetilde{\mathbf{T}}}((j+1)N-1)X^{N-1}, \end{aligned}$$

where $0 \le j < |\widetilde{\mathbf{T}}|/N$ and $0 \le k < \beta$.

To ensure correctness, we additionally construct RLWE ciphertexts

$$M_2^{(j)}(X) = \tilde{t}_{jN-\frac{N}{2}} + \tilde{t}_{jN-\frac{N}{2}+1}X + \cdots + \tilde{t}_{(j+1)N-\frac{N}{2}-1}X^{N-1},$$

for $0 \le j < |\widetilde{\mathbf{T}}|/N$, where we define $\tilde{t}_k = |\Sigma| + 1$ for $k < 0$. This polynomial stores $\widetilde{\mathbf{T}}$ right-shifted by $N/2$ positions.

The client encrypts $\widetilde{\mathbf{P}} = (\tilde{p}_0, \dots, \tilde{p}_{|\widetilde{\mathbf{P}}|-1})$ as a vector of LWE ciphertexts, where each LWE ciphertext encrypts a single element $\tilde{p}_i$. In total, the user needs to send $(\beta+2)|\widetilde{\mathbf{T}}|/N$ RLWE ciphertexts, and the client needs to send $|\widetilde{\mathbf{P}}|$ LWE ciphertexts to the server.

4.3.4. Secure LUT Evaluations to Obtain the Slice under the LHE Mode

To securely retrieve the i-th slice $\widetilde{\mathbf{T}}^{(i)}$ of the padded text $\widetilde{\mathbf{T}}$, we perform LUT evaluations over the RLWE ciphertexts. First, we evaluate $i = \mathsf{SA}_{\widetilde{\mathbf{T}}}(c)$ using $\mathsf{CMuxTree}$ [18]. Let $c = \lfloor (|\widetilde{\mathbf{T}}|-1)/2 \rfloor \in \mathbb{Z}_{|\widetilde{\mathbf{T}}|}$. We decompose c as $c = \sum_{k=0}^{\log |\widetilde{\mathbf{T}}|-1} 2^k c_k$, where $c_k \in \{0,1\}$. The server encrypts each bit c_k as an RGSW ciphertext using pk. In the symmetric-key setting, the client encrypts c_k and sends them to the server. Then, $i = \sum_{k=0}^{\beta-1} \delta^k i_k$ is obtained after β $\mathsf{CMuxTree}$ operations over the LUTs of $(\mathsf{SA}^{(k)}_{\widetilde{\mathbf{T}}})_{0 \le k < \beta}$, where $\mathsf{LWE}(\Delta i_k)$ is the output of the k-th execution.

A naive way to homomorphically retrieve $\widetilde{\mathbf{T}}^{(i)}$ as a vector of LWE ciphertexts is to repeatedly execute $\mathsf{CMuxTree}$ over the LUT of $\widetilde{\mathbf{T}}$. However, this approach requires $O(|\widetilde{\mathbf{P}}|)$ executions of CBS and $\mathsf{CMuxTree}$, making it highly inefficient. To address this inefficiency, we propose a method called *negacyclic-free batch extraction*,

which securely retrieves $\widetilde{\mathbf{T}}^{(i)}$ of arbitrary length while reducing the number of CBS and CMuxTree executions to $O(|\widetilde{\mathbf{P}}|/N)$.

Negacyclic-free Batch Extraction (NFBE)

Let $\mathsf{CMuxTree}_L(i)$ denote the CMux tree operation with a LUT L and input i. The key observation is that after performing a BlindRotate on $M_1^{(j)}$ with X^{-k} in $\mathsf{CMuxTree}_{M_1}(i)$, we obtain

$$\begin{aligned} M_1^{(j)}(X)\cdot X^{-k} = \tilde{t}_{jN+k} + \cdots + \tilde{t}_{(j+1)N-1}X^{N-k-1} \\ - \tilde{t}_{jN}X^{N-k} - \cdots - \tilde{t}_{jN+k-1}X^{N-1}, \end{aligned} \tag{1}$$

where $j = \lfloor i/N \rfloor$ and $k = i \bmod N$. Equation (1) shows that, as long as $|\widetilde{\mathbf{P}}| \le N-k$, we can obtain $\widetilde{\mathbf{T}}^{(i)}$ by extracting the first $|\widetilde{\mathbf{P}}|$ coefficients of $M_1^{(j)}(X)\cdot X^{-k}$ using SampleExtract exactly $|\widetilde{\mathbf{P}}|$ times. In particular, this condition holds whenever $|\widetilde{\mathbf{P}}| \le N/2$ and $0 \le k < N/2$.

Let $\mathsf{CMuxTree}'$ denote the CMux tree operation without applying SampleExtract. Then we obtain the following property.

Property 1. *For M_1, let $k = i \bmod N$. Then the coefficient vector of the polynomial output by $\mathsf{CMuxTree}'_{M_1}(i)$ is given by*

$$\mathsf{CMuxTree}'_{M_1}(i) = \widetilde{\mathbf{T}}[i : i+N-k] \,\|\, -\widetilde{\mathbf{T}}[i-k : i]. \tag{2}$$

For M_2, the coefficient vector of the polynomial output by $\mathsf{CMuxTree}'_{M_2}(i)$ is given by

$$\mathsf{CMuxTree}'_{M_2}(i) = \widetilde{\mathbf{T}}[i-N/2 : i+N/2-k] \,\|\, -\widetilde{\mathbf{T}}[i-N/2-k : i-N/2]. \tag{3}$$

From Property 1, we obtain two equivalent coefficient vectors whose rotation amounts differ exactly by $N/2$, namely $\mathbf{v}_1 = \mathsf{CMuxTree}'_{M_1}(i)$ and $\mathbf{v}_2 = \mathsf{CMuxTree}'_{M_2}(i+N/2)$. This guarantees that $\widetilde{\mathbf{T}}^{(i)}$ appears as a prefix of either $\mathbf{v}_1$ or $\mathbf{v}_2$.

Lemma 1 (NFBE for Fixed-Length)**.** *Let $\mathbf{v}_1 = \mathsf{CMuxTree}'_{M_1}(i)$ and $\mathbf{v}_2 = \mathsf{CMuxTree}'_{M_2}(i+N/2)$. For $|\widetilde{\mathbf{P}}| \le N/2$, the length-$|\widetilde{\mathbf{P}}|$ prefix of either $\mathbf{v}_1$ or $\mathbf{v}_2$ coincides with $\widetilde{\mathbf{T}}^{(i)}$.*

Proof. From Property 1, we have $\mathbf{v}_1 = \widetilde{\mathbf{T}}[i : i+N-k] \,\|\, -\widetilde{\mathbf{T}}[i-k : i]$ and $\mathbf{v}_2 = \widetilde{\mathbf{T}}[i : i+N-k'] \,\|\, -\widetilde{\mathbf{T}}[i-k' : i]$, where $k = i \bmod N$ and $k' = (i+N/2) \bmod N$.

Since k' differs from k exactly by $N/2$ modulo N, one of k or k' is at most $N/2$. Assume without loss of generality that $k \le N/2$. Then the first $N-k$ coefficients of $\mathbf{v}_1$ coincide with $\widetilde{\mathbf{T}}[i : i+N-k]$ and contain no negacyclic part. Because $|\widetilde{\mathbf{P}}| \le N/2 \le N-k$, the length-$|\widetilde{\mathbf{P}}|$ prefix of $\mathbf{v}_1$ equals $\widetilde{\mathbf{T}}^{(i)}$. The other case is identical with $\mathbf{v}_2$. □

From Lemma 1, this result can be extended to patterns $\widetilde{\mathbf{P}}$ of *arbitrary* length.

Lemma 2 (NFBE for Arbitrary-Length)**.** *Let $\widetilde{\mathbf{T}}^{(i)}$ be of arbitrary length $\frac{mN}{2}$ for $m \in \mathbb{Z}_{>0}$, and partition $\widetilde{\mathbf{T}}^{(i)}$ into consecutive blocks of size $N/2$:*

$$\widetilde{\mathbf{T}}^{(i)} = \widetilde{\mathbf{T}}_0 \,\|\, \widetilde{\mathbf{T}}_1 \,\|\, \ldots \,\|\, \widetilde{\mathbf{T}}_{m-1}.$$

Let $\mathbf{v}_1^{(i)} = \mathsf{CMuxTree}'_{M_1}(i)$ and $\mathbf{v}_2^{(i)} = \mathsf{CMuxTree}'_{M_2}(i)$. Then the blocks of $\widetilde{\mathbf{T}}^{(i)}$ appear in either of the following alternating patterns:

(1) The length-$N/2$ prefix of $\mathbf{v}_1^{(i+\ell N)}$ coincides with $\widetilde{\mathbf{T}}_{2\ell}$ for all $\ell = 0, \ldots, \lfloor (m-1)/2 \rfloor$, and the length-$N/2$ prefix of $\mathbf{v}_2^{(i+(\ell+1)N)}$ coincides with $\widetilde{\mathbf{T}}_{2\ell+1}$ for all $\ell = 0, \ldots, \lfloor (m-2)/2 \rfloor$.

(2) The roles of $\mathbf{v}_1$ and $\mathbf{v}_2$ are swapped: the length-$N/2$ prefix of $\mathbf{v}_2^{(i+\ell N+N/2)}$ coincides with $\widetilde{\mathbf{T}}_{2\ell}$, and the length-$N/2$ prefix of $\mathbf{v}_1^{(i+\ell N+N/2)}$ coincides with $\widetilde{\mathbf{T}}_{2\ell+1}$, for the same ranges of ℓ.

Proof. We prove Case (1), where $k = i \bmod N$ satisfies $k < N/2$. Let $i' = i + \ell N$. Since $i' \bmod N = k$ for all $\ell \ge 0$, Property 1 gives

$$\mathbf{v}_1^{(i')} = \widetilde{\mathbf{T}}[i' : i' + N - k] \,\|\, -\widetilde{\mathbf{T}}[i' - k : i'].$$

Because $k < N/2$, we have $N/2 \leq N - k$. Hence the length-$N/2$ prefix of $\mathbf{v}_1^{(i')}$ equals

$$\widetilde{\mathbf{T}}[i' : i' + N/2] = \widetilde{\mathbf{T}}_{2\ell}.$$

Moreover, since $(i' + N) \bmod N = k$ for all $\ell \geq 0$, applying Property 1 again yields

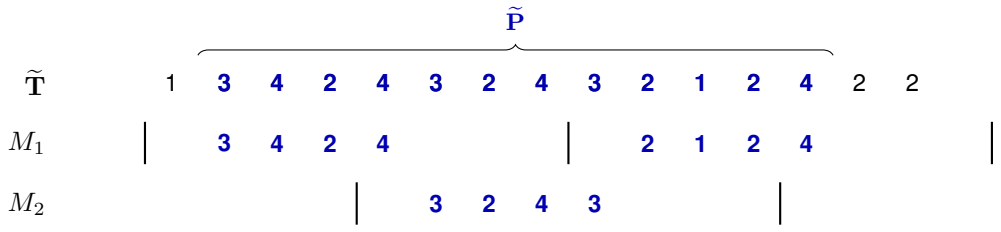
$$\mathbf{v}_2^{(i'+N)} = \widetilde{\mathbf{T}}[i' + N/2 : i' + N/2 + (N - k)] \,\|\, -\widetilde{\mathbf{T}}[i' + N/2 + (N - k) : i' + N/2 + N].$$

Because $k < N/2$, the length-$N/2$ prefix of $\mathbf{v}_2^{(i'+N)}$ equals

$$\widetilde{\mathbf{T}}[i' + N/2 : i' + N] = \widetilde{\mathbf{T}}_{2\ell+1}.$$

Thus, the blocks $\widetilde{\mathbf{T}}_{2\ell}$ and $\widetilde{\mathbf{T}}_{2\ell+1}$ appear alternately in the prefixes of $\mathbf{v}_1$ and $\mathbf{v}_2$. The other case ($k \geq N/2$) is symmetric. □

Figure 2 illustrates how NFBE reconstructs an arbitrary-length pattern by alternately extracting length-$N/2$ blocks from the two shifted message-polynomial representations.



Each length-$N/2$ block of $\widetilde{\mathbf{P}}$ appears in either M_1 or M_2 ($N = 8$ in this example).

Figure 2. Illustration of NFBE. The two overlapping representations of message polynomials, M_1 and M_2, cover the encoded text $\widetilde{\mathbf{T}}$ with a relative shift of $N/2$ positions. After evaluating $\mathsf{CMuxTree}'$ on either M_1 or M_2, we extract consecutive length-$N/2$ blocks of the encoded pattern $\widetilde{\mathbf{P}} = 342432432124$ from $\widetilde{\mathbf{T}}$. For the first block, the rotation amount for M_1 is $k = 1$; therefore, the length-$N/2$ prefix of the rotated polynomial contains no negacyclic terms. In contrast, the corresponding rotation amount for M_2 is $k' = (k + N/2) \bmod N = 5$, and its length-$N/2$ prefix contains negacyclic terms. By advancing the extraction position by $N/2$ and alternating the source between M_1, M_2, and M_1, NFBE extracts the subsequent valid length-$N/2$ blocks with the same rotation amount $k = 1$.

We construct two LWE vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ by applying $\mathsf{SampleExtract}$ to the vectors $\mathbf{v}_1, \mathbf{v}_2$ of Cases (1) and (2) in Lemma 2, extracting the relevant prefixes. By construction, one of them coincides with $\widetilde{\mathbf{T}}^{(i)}$.

NFBE and Data Fragmentation

NFBE is related to data fragmentation for encrypted streams [8, 13], as both use shifted and overlapping representations of the text. However, fragment-based methods choose the fragment size so that the entire pattern is contained in at least one fragment. In NFBE, the entire pattern does not need to fit into a single extracted RLWE substring, since a longer substring can be obtained by concatenating subsequently extracted blocks as a vector of LWE ciphertexts. The drawback compared with fragment-based methods is that our method compares the two resulting LWE vectors character by character, whereas BFV-based fragment methods exploit polynomial packing to evaluate Hamming distances for multiple candidate positions in parallel.

4.3.5. Compare Slice and Pattern under the FHE Mode

Given two vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ (encrypted as LWE ciphertexts), we homomorphically compute $\mathsf{Compare}(\mathbf{u}_1, \widetilde{\mathbf{P}})$ and $\mathsf{Compare}(\mathbf{u}_2, \widetilde{\mathbf{P}})$. We show that treePBS [20] can be adapted to construct the homomorphic Compare function.

Lemma 3. *For two vectors* $\mathbf{u} = (u_0, \ldots, u_{|\widetilde{\mathbf{P}}|-1})$ *and* $\widetilde{\mathbf{P}} = (p_0, \ldots, p_{|\widetilde{\mathbf{P}}|-1})$*, where* $\mathbf{u} \in \{\mathbf{u}_1, \mathbf{u}_2\}$*, we can evaluate* $\mathsf{Compare}(\mathbf{u}, \widetilde{\mathbf{P}})$ *with* $O(|\widetilde{\mathbf{P}}|)$ PBS *executions in a* treePBS*-like fashion.*

Proof. At the leaves, we apply the character-wise comparison function

$$g(x) = \begin{cases} -1, & -|\Sigma| \le x < 0, \\ 0, & x = 0 \text{ or } x \ge |\Sigma| + 1, \\ 1, & 0 < x \le |\Sigma|, \end{cases}$$

to each difference $x_i = u_i - p_i$. Considering the effects of negacyclic rotation and special characters, we have

$$u_i \in [-(|\Sigma| + 1), -1] \cup [1, |\Sigma| + 1], \quad p_i \in \{-2(|\Sigma| + 1)\} \cup [1, |\Sigma|].$$

If $p_i = -2(|\Sigma| + 1)$, then $x_i \ge |\Sigma| + 1$ so that $g(x_i) = 0$, as desired. If $u_i = |\Sigma| + 1$, then $0 < x_i \le |\Sigma|$ and $g(x_i) = 1$, as desired. If $u_i < 0$, then $x_i < 0$. The remaining case corresponds to standard characters $u_i, p_i \in [1, |\Sigma|]$, which is treated as in [20].

Computing g requires $|\widetilde{\mathbf{P}}|$ PBS executions. Each value $g(x_i) \in \{-1, 0, 1\}$ represents the sign of $u_i - p_i$. We then apply a treePBS, as in [20], to propagate the first non-zero value toward the root. This requires $2|\widetilde{\mathbf{P}}| - 1$ PBS executions in total. Hence, the statement holds. □

Now we can compute $\mathsf{Compare}(\widetilde{\mathbf{T}}^{(i)}, \widetilde{\mathbf{P}})$ from Lemma 3 using a single PBS as follows.

Lemma 4. *Given* $s_1 = \mathsf{Compare}(\mathbf{u}_1, \widetilde{\mathbf{P}})$ *and* $s_2 = \mathsf{Compare}(\mathbf{u}_2, \widetilde{\mathbf{P}})$*, we can compute* $\mathsf{Compare}(\widetilde{\mathbf{T}}^{(i)}, \widetilde{\mathbf{P}})$ *with a single* PBS *execution.*

Proof. Consider a function $s = h(s_1, s_2) : \{-1, 0, 1\}^2 \to \{-1, 0, 1\}$, and assume without loss of generality that $s_1 \le s_2$. Since at least one of the two results is correct, we have $h(i, i) = i$ for all $i \in \{-1, 0, 1\}$.

For an input pair $(-1, s)$, the output must be s, because the value -1 can only arise from replacing a positive character $u_i > 0$ with a negative placeholder $\tilde{u}_i < 0$. In this case, if $u_i - p_i \ge 0$, then $\tilde{u}_i - p_i < 0$, and thus the sign of comparison is determined by the index i.

Hence, the pair $(0, 1)$ cannot occur, and the function h correctly determines the sign of $\mathsf{Compare}(\widetilde{\mathbf{T}}^{(i)}, \widetilde{\mathbf{P}})$. □

4.3.6. Updating Variables in the Hybrid Mode

Given an LWE ciphertext encrypting $s = \mathsf{Compare}(\widetilde{\mathbf{T}}^{(i)}, \widetilde{\mathbf{P}})$, we update the encrypted variables ℓ, r, c using $O(\beta)$ CMux and PBS operations. The update rule is as follows:

$$\ell' \leftarrow \begin{cases} c + 1 & \text{if } s = -1, \\ \ell & \text{otherwise,} \end{cases} \quad r' \leftarrow \begin{cases} c - 1 & \text{if } s = 1, \\ r & \text{otherwise,} \end{cases} \quad \text{and} \quad c' \leftarrow \left\lfloor \frac{\ell' + r'}{2} \right\rfloor,$$

which halves the search interval $[\ell, r]$ depending on the sign of s.

In TFHE, conditional statements are evaluated using CMux gates, where the control bit is given as an RGSW ciphertext. To obtain the three branch indicators from s, we extract indicator bits via a CBS operation. More precisely, we redefine the function h in Lemma 4 as

$$s = h(s_1, s_2) : \{-1, 0, 1\}^2 \to \{1, 2, 4\}.$$

Then the flags lt, eq and gt (corresponding to $s = -1$, $s = 0$, and $s = 1$, respectively) are obtained by extracting the individual bits of s.

For example, ℓ is updated as $\mathbf{C}_{\ell'}^{(j)} = \mathsf{CMux}(\overline{\overline{\mathbf{C}}}_{\mathsf{lt}}, \mathbf{C}_{\ell}^{(j)}, \mathbf{C}_{c+1}^{(j)})$, where $\overline{\overline{\mathbf{C}}}_{\mathsf{lt}}$ is the RGSW encryption of lt, and $\mathbf{C}_{c+1}^{(j)}$ (resp. $\mathbf{C}_{\ell}^{(j)}$) denotes the RLWE ciphertext encrypting the j-th decomposition component of $c + 1$ (resp. ℓ). To enable the CMux operation, we convert an LWE ciphertext into an RLWE ciphertext via LWE-to-RLWE-KS, and convert back to LWE via SampleExtract.

After updating ℓ and r, the value c is recomputed using LWE additions, followed by the evaluation of the function $x \mapsto \lfloor x/2 \rfloor$. This requires one PBS for the right shift and another for carry propagation. Finally, the resulting LWE ciphertexts corresponding to c are converted into RGSW ciphertexts via CBS for use in the next iteration.

4.3.7. Analysis of the Algorithm

We now analyze the correctness, complexity, and security of our method. The following results establish correctness and complexity.

Definition 13 (Correctness)**.** *For a text T and a pattern P, let*

$$\mathcal{M}(T,P) = \{\, i \mid 0 \le i \le |T| - |P|,\ T[i : i + |P|] = P \,\}.$$

Let D_T and $\widetilde{\mathbf{P}}$ denote the encoded and padded representations of T and P defined above, where D_T includes $\widetilde{\mathbf{T}}$ and $\mathsf{SA}_{\widetilde{\mathbf{T}}}$. The protocol is correct if decrypting

$$\mathsf{FindMatch}([\![D_T]\!], [\![\widetilde{\mathbf{P}}]\!])$$

returns an element of $\mathcal{M}(T,P)$ when $\mathcal{M}(T,P) \neq \emptyset$, and returns $\perp$ otherwise, except with decryption failure probability at most ρ.

Theorem 1 (Correctness)**.** *The homomorphic implementation of Algorithm 1 satisfies the above correctness definition.*

Proof. By construction of the encoding and padding, $\mathsf{Compare}(\widetilde{\mathbf{T}}^{(i)}, \widetilde{\mathbf{P}}) = 0$ if and only if $T[i : i + |P|] = P$ for $0 \le i \le |T| - |P|$. In Algorithm 1, Lines 3–4 follow directly from Lemmas 2—4, which establish the correctness of the $\mathsf{Compare}$ function. Therefore, in each iteration, the algorithm correctly retrieves $i = \mathsf{SA}_{\widetilde{\mathbf{T}}}(c)$ and obtains the lexicographic relation between $\widetilde{\mathbf{T}}^{(i)}$ and $\widetilde{\mathbf{P}}$.

Line 7 follows from the hybrid update procedure described above and therefore preserves correctness. If $\widetilde{\mathbf{T}}^{(i)} < \widetilde{\mathbf{P}}$ or $\widetilde{\mathbf{T}}^{(i)} > \widetilde{\mathbf{P}}$, the interval $[\ell, r]$ is updated according to the standard binary search over the suffix array. Thus, if a matching suffix exists, at least one matching suffix remains in the updated interval. Algorithm 1 terminates when equality is detected or when the search interval becomes empty. In the homomorphic implementation, however, the circuit does not terminate early, since the iteration at which a match is found must remain hidden. If equality is detected, the search interval remains unchanged for the remaining iterations.

After $\log|\widetilde{\mathbf{T}}|$ iterations, we determine the final index $i = \mathsf{SA}_{\widetilde{\mathbf{T}}}(\lfloor(\ell+r)/2\rfloor)$. We then check whether $\widetilde{\mathbf{T}}^{(i)} = \widetilde{\mathbf{P}}$. This check can be performed by verifying $s = 0$ and returning

$$\mathsf{CMux}(\overline{\overline{\mathbf{C}}}_{\mathsf{eq}}, \mathbf{C}_{\perp}^{(j)}, \mathbf{C}_{i}^{(j)}),$$

where $\overline{\overline{\mathbf{C}}}_{\mathsf{eq}}$ is the RGSW encryption of eq, and $\mathbf{C}_{\perp}^{(j)}$ (resp. $\mathbf{C}_{i}^{(j)}$) denotes the RLWE ciphertext encrypting the j-th decomposition component of $\perp$ (resp. i). Hence, after decryption, the algorithm returns an element of $\mathcal{M}(T,P)$ if $\mathcal{M}(T,P) \neq \emptyset$, and returns $\perp$ otherwise. □

Theorem 2 (Complexity)**.** *Given a text $\widetilde{\mathbf{T}}$ and a pattern $\widetilde{\mathbf{P}}$, the homomorphic implementation of Algorithm 1 performs $O\Big(\frac{\beta|\widetilde{\mathbf{P}}|}{N}\log|\widetilde{\mathbf{T}}|\Big)$ CBS executions, $O\Big(\Big(\beta + \frac{|\widetilde{\mathbf{P}}|}{N}\Big)\log|\widetilde{\mathbf{T}}|\Big)$ $\mathsf{CMuxTree}$ evaluations, $O\Big(\Big(|\widetilde{\mathbf{P}}| + \frac{\beta|\widetilde{\mathbf{P}}|}{N}\Big)\log|\widetilde{\mathbf{T}}|\Big)$ PBS executions, $O\Big(\beta\log|\widetilde{\mathbf{T}}|\Big)$ additional CMux and $\mathsf{LWE\text{-}to\text{-}RLWE\text{-}KS}$ operations, and $O\Big((|\widetilde{\mathbf{P}}| + \beta)\log|\widetilde{\mathbf{T}}|\Big)$ $\mathsf{SampleExtract}$ operations, where $\beta = \lceil\log_\delta|\widetilde{\mathbf{T}}|\rceil$ is the number of decomposition components used to represent an index in $\mathbb{Z}_{|\widetilde{\mathbf{T}}|}$.*

Proof. Suppose that the server holds the LUTs M_1 and M_2 of $\widetilde{\mathbf{T}}$ as $2|\widetilde{\mathbf{T}}|/N$ RLWE ciphertexts and the LUTs of $\mathsf{SA}_{\widetilde{\mathbf{T}}}$ as $\beta|\widetilde{\mathbf{T}}|/N$ RLWE ciphertexts. In Line 1, ℓ, r, and c are each represented using β LWE ciphertexts. In addition, the bit decomposition of c is represented using $\log|\widetilde{\mathbf{T}}|$ RGSW ciphertexts for the $\mathsf{CMuxTree}$ evaluations. This initialization is performed only once.

In Line 3, we execute β $\mathsf{CMuxTree}$ evaluations to retrieve $i = \mathsf{SA}_{\widetilde{\mathbf{T}}}(c)$ as β LWE ciphertexts. To extract a substring of length $|\widetilde{\mathbf{P}}|$, NFBE uses $O(|\widetilde{\mathbf{P}}|/N)$ indices obtained by adding public multiples of $N/2$ to i. Since each index consists of β decomposition components, generating these indices requires $O(\beta|\widetilde{\mathbf{P}}|/N)$ PBS executions. Their bit decompositions are then converted into RGSW ciphertexts using $O(\beta|\widetilde{\mathbf{P}}|/N)$ CBS executions.

The resulting RGSW ciphertexts are used to evaluate the LUTs M_1 and M_2. This requires $O(|\widetilde{\mathbf{P}}|/N)$ $\mathsf{CMuxTree}$ evaluations. Applying $\mathsf{SampleExtract}$ to their outputs produces the two candidate LWE vectors and

requires $O(|\widetilde{\mathbf{P}}|)$ SampleExtract operations.

In Line 4, the two candidate vectors are compared with $\widetilde{\mathbf{P}}$. By Lemma 3, each comparison requires $O(|\widetilde{\mathbf{P}}|)$ PBS executions. Lemma 4 combines the two comparison results using one additional PBS execution. Thus, the comparison stage requires $O(|\widetilde{\mathbf{P}}|)$ PBS executions.

In Line 7, the comparison result is converted into RGSW branch indicators using $O(1)$ CBS executions. Updating each of the β components of ℓ and r requires $O(\beta)$ CMux, LWE-to-RLWE-KS, SampleExtract, and PBS operations. Computing $c = \lfloor(\ell + r)/2\rfloor$ also requires $O(\beta)$ PBS executions. Finally, the β components of c are converted into $\log|\widetilde{\mathbf{T}}|$ RGSW ciphertexts for the next iteration using $O(\beta)$ CBS executions.

Consequently, one iteration requires $O(\beta|\widetilde{\mathbf{P}}|/N)$ CBS executions, $O(\beta + |\widetilde{\mathbf{P}}|/N)$ CMuxTree evaluations, $O(|\widetilde{\mathbf{P}}|+\beta|\widetilde{\mathbf{P}}|/N)$ PBS executions, $O(\beta)$ CMux and LWE-to-RLWE-KS operations, and $O(|\widetilde{\mathbf{P}}|+\beta)$ SampleExtract operations. Since the binary search consists of $O(\log|\widetilde{\mathbf{T}}|)$ iterations, the stated total operation counts follow. □

These bounds represent the total operation counts. The critical path of each iteration consists of the suffix array lookup, the generation of the indices used by NFBE, the NFBE evaluations, the string comparison, and the update of the binary search state. The CMuxTree evaluations for different decomposition components and different substring blocks can be performed in parallel. Likewise, the PBS evaluations at the same level of the tree-based comparison can be performed in parallel. Therefore, counting the sequential levels of PBS, CBS, and CMux operations, the critical path depth of one iteration is

$$O\left(\underbrace{\log(|\widetilde{\mathbf{T}}|/N)}_{\mathsf{CMuxTree}} + \underbrace{\beta}_{\text{index arithmetic and state update}} + \underbrace{\log|\widetilde{\mathbf{P}}|}_{\mathsf{treePBS}}\right),$$

and the total critical path depth is

$$O\left(\left(\underbrace{\log(|\widetilde{\mathbf{T}}|/N)}_{\mathsf{CMuxTree}} + \underbrace{\beta}_{\text{index arithmetic and state update}} + \underbrace{\log|\widetilde{\mathbf{P}}|}_{\mathsf{treePBS}}\right)\log|\widetilde{\mathbf{T}}|\right).$$

For the correctness and parameter analysis, we upper bound the noise variance accumulated in our circuit. To this end, we first recall the noise growth of the basic building blocks used in our construction. We then analyze the longest noise accumulation path in each of the four main stages: the suffix array lookup and NFBE evaluations using CMuxTree, the string comparison using treePBS, the hybrid state update, and the conversion of the updated index into RGSW ciphertexts using CBS.

Definition 14 (Noise variance of KeySwitch [26])**.** *Let* $\mathsf{V}_{\mathsf{ksk}}$ *denote the noise variance of the key switching key* ksk*. Then,* KeySwitch *outputs an LWE ciphertext with noise variance* V_{ks} *satisfying*

$$\mathsf{V}_{\mathsf{ks}} \le kN\left(\frac{q^2 - B^{2\ell}}{24B^{2\ell}} + \frac{1}{12}\right) + k\ell N\left(\frac{B^2+2}{12}\right)\mathsf{V}_{\mathsf{ksk}}.$$

Definition 15 (Noise variance of LWE-to-RLWE-KS [24])**.** *Let* $\mathsf{V}_{\mathsf{atk}}$ *denote the noise variance of the automorphism key* atk*. Then,* LWE-to-RLWE-KS *outputs an RLWE ciphertext with noise variance* V_{tr} *satisfying*

$$\mathsf{V}_{\mathsf{tr}} \le \left(\frac{N^2-1}{3}\right)\mathsf{V}_{\mathsf{atk}}.$$

Definition 16 (Noise variance of CMux [26])**.** *Let* $\mathsf{V}_{\mathsf{rgsw}}$ *denote the noise variance of the input RGSW ciphertext. Then,* CMux *outputs an RLWE ciphertext with noise variance* $\mathsf{V}_{\mathsf{cmux}}$ *satisfying*

$$\mathsf{V}_{\mathsf{cmux}} \le (1+kN)\left(\frac{q^2 - B^{2\ell}}{24B^{2\ell}} + \frac{1}{12}\right) + (k+1)\ell N\left(\frac{B^2+2}{12}\right)\mathsf{V}_{\mathsf{rgsw}}.$$

We now combine these bounds to obtain an overall bound on the noise growth in the full circuit.

Theorem 3 (Noise Variance)**.** *Let* $\mathsf{V}_{\mathsf{circ}}$ *be the maximum noise variance of any (R)LWE ciphertext in the circuit. Then we have*

$$\mathsf{V}_{\mathsf{circ}} \leq \max\left(2 \cdot \mathsf{V}_{\mathsf{lwe}} + \mathsf{V}_{\mathsf{tr}} + \mathsf{V}_{\mathsf{cmux}} + \mathsf{V}_{\mathsf{ks}},\ \frac{p^2}{2} \cdot \mathsf{V}_{\mathsf{lwe}} + \mathsf{V}_{\mathsf{ks}},\ \log\left(\frac{|\widetilde{\mathbf{T}}|}{N}\right) \cdot \mathsf{V}_{\mathsf{cmux}}\right)$$

where $\mathsf{V}_{\mathsf{lwe}}$ *denotes the error variance of an LWE ciphertext,* V_{ks} *denotes the* KeySwitch *output error variance,* V_{tr} *denotes the* LWE-to-RLWE-KS *output error variance [24], and* $\mathsf{V}_{\mathsf{cmux}}$ *denotes the* CMux *output error variance [26].*

Proof. We identify the longest path in our circuit, i.e., a subcircuit between two noise-initializing procedures: PBS, CBS, or BlindRotate. For PBS, the subcircuit corresponds to the computation of $\ell' + r'$ from ℓ and r. It starts with LWE-to-RLWE-KS applied to ℓ and r, followed by a CMux and a SampleExtract operation, and an addition of two LWE ciphertexts, followed by a KS-PBS to propagate the carry of $\ell' + r'$. Moreover, SampleExtract only extracts an LWE sample from an RLWE ciphertext and does not increase its noise variance. Thus, the noise variance is bounded by $2 \cdot \mathsf{V}_{\mathsf{lwe}} + \mathsf{V}_{\mathsf{tr}} + \mathsf{V}_{\mathsf{cmux}} + \mathsf{V}_{\mathsf{ks}}$.

For CBS, the longest path corresponds to the computation of c'. It starts with two additions of LWE ciphertexts and ends with a CBS. As described in Definition 10, integer-wise CBS requires a scalar multiplication by (at most) $p/2$ and a KeySwitch before applying PBSmanyLUT. Thus, the resulting noise variance is bounded by $2 \cdot \left(\frac{p}{2}\right)^2 \cdot \mathsf{V}_{\mathsf{lwe}} + \mathsf{V}_{\mathsf{ks}} = \frac{p^2}{2} \cdot \mathsf{V}_{\mathsf{lwe}} + \mathsf{V}_{\mathsf{ks}}$, which is similar to the result in [26].

Finally, we consider the CMuxTree. It starts with CMux operations of depth $\log\left(\frac{|\widetilde{\mathbf{T}}|}{N}\right)$ and ends with BlindRotate. Since BlindRotate with message bits initializes noise without requiring ModulusSwitch, the accumulated noise variance is bounded by $\log\left(\frac{|\widetilde{\mathbf{T}}|}{N}\right) \cdot \mathsf{V}_{\mathsf{cmux}}$.

The same bound applies to the CMuxTree evaluations used in NFBE. The evaluations for different substring blocks are independent, and therefore increasing $|\widetilde{\mathbf{P}}|$ increases the number of evaluations but does not increase the noise accumulated along a single path.

For the string comparison, each leaf first computes the difference between two LWE ciphertexts and then applies a PBS. The intermediate nodes of treePBS also apply PBS to the outputs of the preceding level. Since each PBS refreshes the ciphertext noise, the $O(\log |\widetilde{\mathbf{P}}|)$ levels of treePBS increase the sequential depth but do not cause the noise variance to accumulate across all levels.

Finally, the noise bounds above do not accumulate over the $O(\log |\widetilde{\mathbf{T}}|)$ binary search iterations. Before the ciphertexts representing ℓ, r, and c are used in the next iteration, they are refreshed by a PBS, CBS, or BlindRotate operation. Therefore, $\mathsf{V}_{\mathsf{circ}}$ is determined by the maximum variance within one of the paths analyzed above, rather than by the sum of the variances over all binary search iterations. □

In the next section, using Theorem 3, we show how to choose a concrete parameter set for our circuit that guarantees both λ-bit security and a circuit failure probability at most ρ.

Finally, we analyze the privacy of our protocol against an honest-but-curious server.

Definition 17 (Security)**.** *Let* $\widetilde{\mathbf{T}}_b$ *and* $\widetilde{\mathbf{P}}_b$ *denote the encoded and padded representations of an input pair* (T_b, P_b) *for* $b \in \{0, 1\}$*. For a server* $\mathcal{S}$*, its view* $\mathsf{View}_{\mathcal{S}}(\widetilde{\mathbf{T}}_b, \widetilde{\mathbf{P}}_b)$ *consists of the public parameters, evaluation keys, ciphertext dimensions, padded lengths* $|\widetilde{\mathbf{T}}_b|$ *and* $|\widetilde{\mathbf{P}}_b|$*, the encrypted inputs, all intermediate ciphertexts, and the encrypted matching result. For any two input pairs* (T_0, P_0) *and* (T_1, P_1) *satisfying*

$$|\widetilde{\mathbf{T}}_0| = |\widetilde{\mathbf{T}}_1| \quad \textit{and} \quad |\widetilde{\mathbf{P}}_0| = |\widetilde{\mathbf{P}}_1|,$$

the protocol is secure if

$$\mathsf{View}_{\mathcal{S}}(\widetilde{\mathbf{T}}_0, \widetilde{\mathbf{P}}_0) \approx_c \mathsf{View}_{\mathcal{S}}(\widetilde{\mathbf{T}}_1, \widetilde{\mathbf{P}}_1),$$

where $\approx_c$ *denotes computational indistinguishability.*

Theorem 4 (Security)**.** *Assuming the IND-CPA security of the underlying TFHE scheme in the presence of its evaluation keys, the proposed protocol is secure against an honest-but-curious server.*

Proof. Consider two input pairs (T_0, P_0) and (T_1, P_1) whose encoded and padded representations have the same lengths. By the IND-CPA security of TFHE, the encrypted inputs corresponding to the two input pairs are computationally indistinguishable.

All intermediate ciphertexts and the encrypted matching result are obtained by applying the same public homomorphic evaluation algorithm to these encrypted inputs. In particular, the conditional updates are evaluated by

CMux using branch flags encrypted as RGSW ciphertexts, and the LUT evaluations for $\mathsf{SA}_{\widetilde{\mathbf{T}}_b}$ and $\widetilde{\mathbf{T}}_b$, which are encrypted as RLWE ciphertexts, are performed obliviously by CMuxTree. Since computational indistinguishability is preserved under probabilistic polynomial-time computation, the resulting server views are also computationally indistinguishable.

Moreover, the number and structure of the evaluated operations depend only on the padded lengths $|\widetilde{\mathbf{T}}_b|$ and $|\widetilde{\mathbf{P}}_b|$. These padded lengths reveal only upper bounds on the true lengths $|T_b|$ and $|P_b|$. Therefore,

$$\mathsf{View}_{\mathcal{S}}(\widetilde{\mathbf{T}}_0, \widetilde{\mathbf{P}}_0) \approx_c \mathsf{View}_{\mathcal{S}}(\widetilde{\mathbf{T}}_1, \widetilde{\mathbf{P}}_1).$$

□

5. Results

We present experimental results for our method, including comparisons with the state-of-the-art SPM method [11] implemented using the BGV scheme, as well as a baseline method implemented using TFHE [29]. We implemented our method based on the `TFHE-rs` library [29] (version 0.5.3, commit `0ce0567`) and an open-source library from [26] for the improved integer-wise CBS https://github.com/KAIST-CryptLab/refined-tfhe-lhe/ (accessed on 13 August 2026). All experiments were conducted on a desktop PC featuring an AMD Ryzen 9 3900 processor and 64 GB of RAM.

5.1. Parameter Selection

In Table 1, we present our TFHE parameter set for $\lambda = 128$-bit security and failure probability $\rho = 2^{-60}$, where $(B_{\mathsf{T}}, \ell_{\mathsf{T}})$ denotes the decomposition base and level for T. b_{tr} denotes the split-FFT base used in HomTrace in the improved CBS [26]. ν_{cbs} denotes the number of LUTs in PBSmanyLUT used in the improved CBS. To derive the parameter set, we use the following theorem from [23].

Table 1. Our TFHE parameter set with $\lambda = 128$ and $\rho = 2^{-60}$.

p	q	n	N	k	ℓ_{ks}	B_{ks}	ℓ_{pbs}	B_{pbs}	ℓ_{tr}	B_{tr}	b_{tr}	ℓ_{ss}	B_{ss}	ℓ_{cbs}	B_{cbs}	ν_{cbs}
2^6	2^{64}	808	2^{12}	1	8	2^2	3	2^{11}	4	2^{12}	2^{40}	4	2^{10}	4	2^5	2

Theorem 5 ([23]). *Suppose that* PBSmanyLUT *takes an input LWE ciphertext with the scaling factor* Δ_{in} *and noise variance* V_{in}. *Then it outputs LWE ciphertexts* $\mathbf{c}_j$ *encrypting* $\Delta f_j(m)$ *for* $j = 0, \ldots, 2^\nu - 1$ *if and only if*

$$\mathsf{V}_{\mathsf{in}} < \frac{\Delta_{\mathsf{in}}^2}{4\Gamma^2} - \frac{q^2}{12w^2} + \frac{1}{12} - \frac{nq^2}{24w^2} - \frac{n}{48},$$

where Γ *depends on the probability of correctness defined as* $P = \mathrm{erf}\left(\frac{\Gamma}{\sqrt{2}}\right)$, *and* $w = 2N \cdot 2^{-\nu}$.

We then search for parameters such that $\mathsf{V}_{\mathsf{circ}} < \mathsf{V}_{\mathsf{in}}$ with failure probability $1 - P = \rho$, and for an (R)LWE security parameter set that achieves λ-bit security as estimated by the `lattice-estimator` https://github.com/malb/lattice-estimator (accessed on 13 August 2026) [30].

To encode a character from Σ into $\mathbb{Z}_p$, we fix $|\Sigma| = 2^2$ for simplicity and encode each character (e.g., `A`, `C`, `G`, `T`, the four nitrogenous bases in DNA) as an element of $[1, 2^2]$. To encode an integer in $\mathbb{Z}_{|\widetilde{\mathbf{T}}|}$ into $\mathbb{Z}_p$, we decompose it into 4-bit components and encode them in the least significant bits (LSBs) of each component in $\mathbb{Z}_p$. The remaining 2 most significant bits (MSBs) of each component are left unused to provide sufficient headroom: one bit for carries during additions and one bit for padding to support arbitrary LUT evaluations of length $p/2$.

5.2. Performance Comparison

Table 2 compares runtime and memory usage of the FindMatch function for a pattern of length $|\widetilde{\mathbf{P}}| = 100$ across different methods. Because the compared implementations use different alphabet sizes, parameter sets, and computational settings, the reported runtimes should not be interpreted as a strict head-to-head comparison. For the proposed method, we set $|\widetilde{\mathbf{T}}| = 2^{12} = N$, since vertical packing is supported for a LUT of length $\geq N$ in the version of `TFHE-rs` we used.

For the TFHE-based method, we measured the runtime of the find function for encrypted ASCII characters, implemented in the latest `TFHE-rs` library (version 1.5.0) [29] using the default parameters.

For the BGV-based method [11], we measured runtime using their open-source library (https://github.com/iliailia/he_pattern_matching (accessed on 13 August 2026)) with the parameter set (Set 7.12) reported in [11]. Here, $|\widetilde{\mathbf{T}}| = 1080$ corresponds to the number of SIMD slots in the BGV scheme, where a single ciphertext contains 1080 slots and thus can store 1080 characters.

Table 2. Measured runtime and memory usage across different SPM methods.

Method	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{P}\rvert$	$\lvert\Sigma\rvert$	λ	ρ	Time [s]	Memory [MB]
Proposed (Section 4.3)	4096	100	2^2	128	2^{-60}	100.9	625.6
TFHE-based [29] (Section 4.2)	1080	100	2^8	128	2^{-128}	1820.8	293.5
BGV-based [11]	1080	100	2^{32}	150	2^{-34}	488.9	714.3

We also evaluated the plaintext preprocessing overhead required to construct the suffix array of T on the client side. For randomly generated DNA texts of lengths 2^{20} and 2^{24}, the suffix array construction took 0.6 seconds and 35.4 seconds, respectively. The corresponding peak memory usage was 40.9 MB and 607.2 MB, respectively. These results show that the preprocessing cost remains manageable for moderate-sized texts, but becomes increasingly demanding as the text size grows. Therefore, our current implementation assumes that the data owner has sufficient local computational resources and is not intended for highly resource-constrained clients.

From Table 2, the proposed method achieves faster runtime than existing methods, even for a text length that is four times larger. We note, however, that the alphabet size $|\Sigma|$ in our experiments is smaller than other methods; therefore, the absolute runtime advantage should be interpreted with this in mind. In the next section, we study scalability and analyze how runtime and memory change as each parameter ($|\widetilde{\mathbf{T}}|$, $|\widetilde{\mathbf{P}}|$, and $|\Sigma|$) increases.

5.3. Scalability Analysis

Table 3 shows the runtime and memory usage of the proposed method when scaling the text length $|\widetilde{\mathbf{T}}|$ with a fixed $|\widetilde{\mathbf{P}}| = 100$ (left), and when scaling the pattern length $|\widetilde{\mathbf{P}}|$ with a fixed $|\widetilde{\mathbf{T}}| = 2^{16}$ (right).

Table 3. Runtime and memory usage of the proposed method when scaling the text length $|\mathcal{T}|$ (left part) and the pattern length $|\mathcal{P}|$ (right part).

$\lvert\mathcal{T}\rvert$	Time [s]	Memory [MB]	$\lvert\mathcal{P}\rvert$	Time [s]	Memory [MB]
2^{12}	100.9	625.6	100	165.5	632.7
2^{14}	138.2	625.9	200	199.6	642.4
2^{16}	166.2	632.6	400	269.9	669.1
2^{18}	211.4	681.1	600	319.3	704.4
2^{20}	247.5	826.8	800	394.4	731.4
2^{22}	315.3	1461.9	1000	457.6	755.7
2^{24}	415.1	3928.2			
2^{26}	773.7	15,350.8			
2^{28}	2055.8	59,388.2			

Since our method requires $O(|\widetilde{\mathbf{P}}| \log |\widetilde{\mathbf{T}}|)$ PBS executions, the runtime increases linearly in $\log |\widetilde{\mathbf{T}}|$ with respect to the text size in practice. As shown in the left table, even when the text length increases from 2^{12} to 2^{28}, the runtime grows moderately from 1.7 min to 34 min, confirming the logarithmic dependency on $|\widetilde{\mathbf{T}}|$. In contrast, the method of [11], whose complexity grows linearly in $|\widetilde{\mathbf{T}}|$, would be expected to require about five days for $|\widetilde{\mathbf{T}}| = 2^{20}$ under comparable settings.

The right table shows that the runtime grows approximately linearly in $|\widetilde{\mathbf{P}}|$, which is consistent with the $O(|\widetilde{\mathbf{P}}|)$ factor in the complexity. Memory usage increases gradually in both cases; however, for large $|\widetilde{\mathbf{T}}|$, the size of the LUTs becomes the dominant component of the space complexity.

To support a larger alphabet of size $|\Sigma|^2$, our current approach essentially requires doubling both the text length and the pattern length. In general, representing an alphabet of size $|\Sigma'|$ over a base alphabet of size $|\Sigma|$ requires $\lceil \log_{|\Sigma|} |\Sigma'| \rceil$ encoded characters per original character. For example, when a base alphabet of size $|\Sigma| = 2^2$ is used to represent the ASCII alphabet of size $|\Sigma'| = 2^7$, each ASCII character is represented by four encoded characters. For the encrypted pattern, this corresponds to four LWE ciphertexts per ASCII character. This extension preserves the structure of our protocol, but increases both runtime and memory consumption accordingly. In particular,

doubling $|\widetilde{\mathbf{P}}|$ directly increases the runtime, and the restriction on $|\Sigma|$ due to the small plaintext modulus p in TFHE remains a bottleneck compared to integer-based FHE schemes such as BGV, which allow a much larger plaintext modulus. Designing a more efficient encoding for large alphabets is left for future work.

6. Conclusions

In this article, we proposed an FHE-based secure pattern matching (SPM) algorithm based on TFHE and evaluated its practical performance. Our experimental results demonstrate that the proposed method achieves faster runtime than existing approaches based on TFHE and BGV, even for larger text sizes, while maintaining comparable memory usage. Furthermore, the scalability analysis confirms that the runtime grows logarithmically with respect to the text length $|\widetilde{\mathbf{T}}|$ and linearly with respect to the pattern length $|\widetilde{\mathbf{P}}|$, which is consistent with the theoretical complexity of $O(|\widetilde{\mathbf{P}}| \log |\widetilde{\mathbf{T}}|)$.

On the other hand, our approach has a limitation in handling large alphabets due to the small plaintext modulus in TFHE. This introduces a trade-off between efficiency and alphabet size compared to integer-based FHE schemes such as BGV. Improving scalability with respect to the alphabet size, as well as designing an FHE-based SPM algorithm with $o(|\widetilde{\mathbf{P}}| \log |\widetilde{\mathbf{T}}|)$ complexity in the matching phase, remains an important direction for future work.

Author Contributions

S.N.: conceptualization, data curation, formal analysis, investigation, methodology, software, validation, writing—original draft preparation; H.O.: conceptualization, writing—reviewing and editing; T.N.: conceptualization, project administration, supervision, writing—reviewing and editing; K.F.: funding acquisition, project administration, supervision. All authors have read and agreed to the published version of the manuscript.

Funding

This work was supported by JST K Program, grant number JPMJKP24U2, Japan.

Institutional Review Board Statement

Not applicable.

Informed Consent Statement

Not applicable.

Data Availability Statement

The code and data supporting the findings of this study are available in the public repository: https://github.com/kr-isg-projects/tfhe-pattern-match.

Acknowledgments

We thank the anonymous reviewers for their valuable comments, which helped improve this manuscript.

Conflicts of Interest

The authors declare no conflicts of interest.

Use of AI and AI-Assisted Technologies

During the preparation of this work, the authors used ChatGPT (OpenAI) to assist with English language editing and proofreading. After using this tool, the authors carefully reviewed and revised the content as necessary and take full responsibility for the content of the published article.

References

1. Akavia, A.; Feldman, D.; Shaul, H. Secure Search on Encrypted Data via Multi-Ring Sketch. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 985–1001.
2. Yasuda, M.; Shimoyama, T.; Kogure, J.; et al. Secure Pattern Matching Using Somewhat Homomorphic Encryption. In Proceedings of the 2013 ACM Cloud Computing Security Workshop (CCSW'13), Berlin, Germany, 4 November 2013; pp. 65–76. https://doi.org/10.1145/2517488.2517497.

3. Hahn, F.; Loza, N.; Kerschbaum, F. Practical and Secure Substring Search. In Proceedings of the SIGMOD/PODS'18: International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 163–176. https://doi.org/10.1145/3183713.3183754.
4. Mainardi, N.; Barenghi, A.; Pelosi, G. Privacy Preserving Substring Search Protocol with Polylogarithmic Communication Cost. In Proceedings of the 35th Annual Computer Security Applications Conference, San Juan, Puerto Rico, 9–13 December 2019; pp. 297–312. https://doi.org/10.1145/3359789.3359842.
5. Boldyreva, A.; Chenette, N.; Lee, Y.; et al. Order-Preserving Symmetric Encryption. In *Advances in Cryptology—EUROCRYPT 2009, Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, 26–30 April 2009*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 224–241. https://doi.org/10.1007/978-3-642-01001-9_13.
6. Pandey, O.; Rouselakis, Y. Property Preserving Symmetric Encryption. In *Advances in Cryptology—EUROCRYPT 2012, Proceedings of the 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, 15–19 April 2012*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 375–391. https://doi.org/10.1007/978-3-642-29011-4_23.
7. Desmoulins, N.; Fouque, P.A.; Onete, C.; et al. Pattern Matching on Encrypted Streams. In *Advances in Cryptology—ASIACRYPT 2018, Proceedings of the 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, 2–6 December 2018*; Springer: Cham, Switzerland, 2018; pp. 121–148. https://doi.org/10.1007/978-3-030-03326-2_5.
8. Bkakria, A.; Cuppens, N.; Cuppens, F. Privacy-Preserving Pattern Matching on Encrypted Data. In *Advances in Cryptology—ASIACRYPT 2020, Proceedings of the 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, 7–11 December 2020*; Springer: Cham, Switzerland, 2020; pp. 191–220. https://doi.org/10.1007/978-3-030-64834-3_7.
9. Bouscatié, É.; Castagnos, G.; Sanders, O. Public Key Encryption with Flexible Pattern Matching. In *Advances in Cryptology—ASIACRYPT 2021, Proceedings of the 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021*; Springer: Cham, Switzerland, 2021; pp. 342–370.
10. Bouscatié, É.; Castagnos, G.; Sanders, O. Pattern Matching in Encrypted Stream from Inner Product Encryption. In *Public-Key Cryptography—PKC 2023, Proceedings of the 26th IACR International Conference on Practice and Theory of Public-Key Cryptography, Atlanta, GA, USA, 7–10 May 2023*; Springer: Cham, Switzerland, 2023; pp. 774–801. https://doi.org/10.1007/978-3-031-31368-4_27.
11. Bonte, C.; Iliashenko, I. Homomorphic String Search with Constant Multiplicative Depth. In Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop, Online, 9 November 2020; pp. 105–117. https://doi.org/10.1145/3411495.3421361.
12. Choi, S.G.; Dachman-Soled, D.; Gordon, S.D.; et al. Compressed Oblivious Encoding for Homomorphically Encrypted Search. In Proceedings of the CCS'21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Online, 15–19 November 2021; pp. 2277–2291. https://doi.org/10.1145/3460120.3484792.
13. Bkakria, A.; Izabachène, M. Efficient Post-Quantum Pattern Matching on Encrypted Data. *IACR Commun. Cryptol.* **2024**, *1*. https://doi.org/10.62056/a09qxrxqi.
14. Kabra, M.; Nadig, R.; Gupta, H.; et al. CIPHERMATCH: Accelerating Homomorphic Encryption-Based String Matching via Memory-Efficient Data Packing and in-Flash Processing. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, Rotterdam, The Netherlands, 30 March–3 April 2025; pp. 111–130. https://doi.org/10.1145/3676641.3716251.
15. Manber, U.; Myers, G. Suffix Arrays: A New Method for on-Line String Searches. *SIAM J. Comput.* **1993**, *22*, 935–948. https://doi.org/10.1137/0222058.
16. Weiner, P. Linear Pattern Matching Algorithms. In Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973), Iowa City, IA, USA, 15–17 October 1973; pp. 1–11. https://doi.org/10.1109/swat.1973.13.
17. Chillotti, I.; Gama, N.; Georgieva, M.; et al. Faster Fully Homomorphic Encryption: Bootstrapping in Less than 0.1 Seconds. In *Advances in Cryptology—ASIACRYPT 2016, Proceedings of the 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, 4–8 December 2016*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 3–33. https://doi.org/10.1007/978-3-662-53887-6_1.
18. Chillotti, I.; Gama, N.; Georgieva, M.; et al. TFHE: Fast Fully Homomorphic Encryption over the Torus. *J. Cryptol.* **2020**, *33*, 34–91. https://doi.org/10.1007/s00145-019-09319-x.
19. Larsson, N.J.; Sadakane, K. Faster Suffix Sorting. *Theor. Comput. Sci.* **2007**, *387*, 258–272. https://doi.org/10.1016/j.tcs.2007.07.017.
20. Bourse, F.; Sanders, O.; Traoré, J. Improved Secure Integer Comparison via Homomorphic Encryption. In Proceedings of the Topics in Cryptology—CT-RSA 2020: The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, 24–28 February 2020; pp. 391–416.
21. Knuth, D.E.; Morris, J.H., Jr; Pratt, V.R. Fast Pattern Matching in Strings. *SIAM J. Comput.* **1977**, *6*, 323–350. https://doi.org/10.1137/0206024.
22. Gama, N.; Izabachène, M.; Nguyen, P.Q.; et al. Structural Lattice Reduction: Generalized Worst-Case to Average-Case

Reductions and Homomorphic Cryptosystems. In *Advances in Cryptology—EUROCRYPT 2016, Proceedings of the 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, 8–12 May 2016*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 528–558.

23. Chillotti, I.; Ligier, D.; Orfila, J.B.; et al. Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE. In *Advances in Cryptology—ASIACRYPT 2021, Proceedings of the 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, 6–10 December 2021*; Springer: Cham, Switzerland, 2021; pp. 670–699.
24. Chen, H.; Dai, W.; Kim, M.; et al. Efficient Homomorphic Conversion between (Ring) LWE Ciphertexts. In *Applied Cryptography and Network Security, Proceedings of the 19th International Conference, ACNS 2021, Kamakura, Japan, 21–24 June 2021*; Springer: Cham, Switzerland, 2021; pp. 460–479. https://doi.org/10.1007/978-3-030-78372-3_18.
25. Bergerat, L.; Boudi, A.; Bourgerie, Q.; et al. Parameter Optimization and Larger Precision for (T) FHE. *J. Cryptol.* **2023**, *36*, 28. https://doi.org/10.1007/s00145-023-09463-5.
26. Wang, R.; Ha, J.; Shen, X.; et al. Refined TFHE Leveled Homomorphic Evaluation and Its Application. In Proceedings of the CCS'25: ACM SIGSAC Conference on Computer and Communications Security, Taipei, Taiwan, 13–17 October 2025; pp. 2309–2323. https://doi.org/10.1145/3719027.3744873.
27. Joye, M. SoK: Fully Homomorphic Encryption over the [Discretized] Torus. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2022**, *2022*, 661–692. https://doi.org/10.46586/tches.v2022.i4.661-692.
28. Joye, M. TFHE Public-Key Encryption Revisited. In Proceedings of the CT-RSA 2024: Cryptographers' Track at the RSA Conference 2024, San Francisco, CA, USA, 6–9 May 2024; pp. 277–291.
29. Zama TFHE-Rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics over Encrypted Data. Available online: https://github.com/zama-ai/tfhe-rs (accessed on 13 August 2026).
30. Albrecht, M.R.; Curtis, B.R.; Deo, A.; et al. Estimate All the {LWE, NTRU} Schemes! In Proceedings of the International Conference on Security and Cryptography for Networks, Amalfi, Italy, 5–7 September 2018; pp. 351–367. https://doi.org/10.1007/978-3-319-98113-0_19.