Time-Memory Trade-off Algorithms for Homomorphically Evaluating Look-up Table in TFHE

Shintaro Narisada
KDDI Research, Inc.
Saitama, Japan
sh-narisada@kddi.com

Hiroki Okada
KDDI Research, Inc.
Saitama, Japan
The University of Tokyo
Tokyo, Japan
ir-okada@kddi.com

Kazuhide Fukushima
KDDI Research, Inc.
Saitama, Japan
ka-fukushima@kddi.com

Takashi Nishide
University of Tsukuba
Ibaraki, Japan
nishide@risk.tsukuba.ac.jp

ABSTRACT

We propose time-memory trade-off algorithms for evaluating look-up table (LUT) in both the leveled homomorphic encryption (LHE) and fully homomorphic encryption (FHE) modes in TFHE. For an arbitrary n-bit Boolean function, we reduce evaluation time by a factor of $O(n)$ at the expense of an additional memory of "only" $O(2^n)$ as a trade-off: The total asymptotic memory is also $O(2^n)$, which is the same as that of prior works. Our empirical results demonstrate that a 7.8× speedup in runtime is obtained with a 3.8× increase in memory usage for 16-bit Boolean functions in the LHE mode. Additionally, in the FHE mode, we achieve reductions in both runtime and memory usage by factors of 17.9× and 2.5×, respectively, for 8-bit Boolean functions. The core idea is to decompose the function f into sufficiently small subfunctions and leverage the precomputed results for these subfunctions, thereby achieving significant performance improvements at the cost of additional memory.

CCS CONCEPTS

• **Security and privacy** → *Public key encryption.*

KEYWORDS

FHE; CMux Tree; Space-Time Trade-off

1 INTRODUCTION

TFHE (Torus Fully Homomorphic Encryption) [7, 9] is one of the most promising variants of the homomorphic encryption schemes, notable for its fast bootstrapping and applicability to homomorphically evaluate arbitrary non-linear functions. A demanding application of TFHE is a homomorphic encryption compiler [17, 21], which enables secure execution of arbitrary program code or Boolean circuits.

The drawbacks of FHE applications stem from their lower runtime efficiency compared to non-encrypted computations [12]. Specifically, efficiently evaluating a non-structured non-linear function f in FHE is challenging, where only the look-up table (LUT) of f is known to the evaluator. This scenario is common in FHE compilers, where many conditional operations, such as `if` and `switch` operators, are compiled as non-linear functions in TFHE.

Thus far, many prior works have addressed this problem. The CMux tree [13], a binary decision diagram composed of CMux gates, is one of the fastest methods for evaluating arbitrary function in the *leveled homomorphic encryption* (LHE) mode of TFHE. The method is further enhanced by the packing techniques [8].

The tree-based method [14] and the chaining method [5], which leverage programmable bootstrapping, can evaluate arbitrary functions in the *fully homomorphic encryption* (FHE) mode of TFHE, where programmable bootstrapping is central to initializing noise.

The new version of WoP-PBS (without padding programmable bootstrapping) [2] enables fast LUT evaluation in what we call the *hybrid* mode of TFHE, where both circuit bootstrapping and programmable bootstrapping are utilized during computation. If the structure of the function f is known, an efficient homomorphic circuit can be constructed using the method [3] in the FHE mode of TFHE.

1.1 Contribution

Despite advancements, runtime remains a significant bottleneck for applications of TFHE. One approach to address this challenge is to develop time-memory trade-off algorithms, which aim to reduce runtime at the expense of additional memory. In this paper, we provide time-memory trade-off algorithms for evaluating arbitrary Boolean functions in both the LHE and FHE mode of TFHE.

- For the LHE mode, we propose a time-memory trade-off for the CMux tree algorithm, which reduces evaluation time by a factor of $O(n)$ at a cost of additional memory of $O(2^n)$, where n represents the number of input bits to the function. Note that $O(2^n)$ space is already necessary to store the LUT of the function.
- For the FHE mode, we introduce homomorphic evaluation algorithms that leverage the Disjunctive Normal Form (DNF) and Conjunctive Normal Form (CNF) representations of a function f. These algorithms have smaller space complexity than the Binary Decision Diagram (BDD) based approach in terms of the required number of LWE ciphertexts. We achieve time-memory trade-offs of these algorithms that are the same as those in the LHE mode.

- Implementation details and experimental results for the proposed algorithms are provided for 8-bit and 16-bit Boolean functions. We demonstrate a time-memory trade-off with a 7.8× reduction in total runtime and a 3.8× increase in memory usage for 16-in 1-out Boolean functions.

The technical overview of the proposed algorithm is to decompose the LUT of the Boolean function f into sufficiently small tables, then efficiently evaluate each table and merge them to compute the result for the entire LUT. The key idea to reduce the time complexity is to share the results of "common LUTs" to reduce the number of homomorphic operations required. Intuitively, this can be seen as finding a minimum set of substrings that cover the output vector of f. This can be viewed as another way of simplifying a Boolean function beforehand using a Karnaugh map.

For example, assume that the output vector $\mathbf{y}$ of $f_{\mathbf{y}}(x_1, x_2, x_3)$ is 01100110. A minimum set of substrings of length 2 is $\{01, 10\}$, and the set of length 4 is $\{0110\}$. One can then evaluate f by merging the evaluation results for 01, 10 and 0110 while traversing reversely the BDD constructed on the LUT, as illustrated in Figure 1. More details about the idea are explained in Example 4.1.

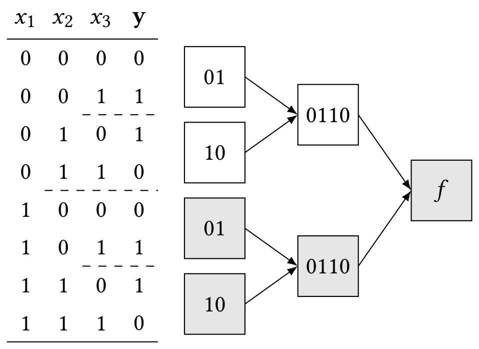

Figure 1: Overview of the proposed algorithm: our algorithm can omit homomorphic operations on gray boxes that share common sub-tables in the LUT, which are separated by dashed lines.

The rest of the paper is organized as follows. Section 2 describes the notation and preliminaries of TFHE. In Section 3, we describe Boolean functions in TFHE and the CMux tree algorithm. Section 4 presents our time-memory trade-off algorithms. We provide empirical results and analysis in Section 5. Section 6 provides concluding remarks.

2 PRELIMINARIES

We summarize the symbols that appeared in this paper in Table 1. The security of TFHE is based on the generalized version of the LWE problem [6, 16].

2.1 Cryptographic Structures

We briefly introduce LWE-based cryptographic structures used for homomorphic computations in TFHE. An LWE ciphertext is the smallest data unit in the TFHE scheme.

Table 1: Notation

Symbol	Description
$\mathbb{A}$	A set
$\mathbb{A}^n$	Set of n-dimensional vectors consisting of elements in $\mathbb{A}$
$\mathbb{A}_q$	The set $\mathbb{A}$ modulo q
$\mathbb{Z}$	Set of integers
$\mathbb{R}$	Set of real numbers
$\mathbb{B} = \mathbb{Z}_2$	Set of binary numbers
$\mathcal{R}$	Set of integer coefficient of polynomials modulo $X^N + 1$, where N is a power of 2
$\mathbf{a}$	A polynomial $\mathbf{a} \in \mathcal{R}$ or a vector $\mathbf{a} \in \mathbb{Z}^n$
a_i	The i-th coefficient or term of $\mathbf{a}$
$\mathbf{a}[i:j]$ $(i < j)$	A vector $(a_i, a_{i+1}, \dots, a_{j-1})$
$\mathbf{a} \parallel \mathbf{b}$	A vector $(a_1, \dots, a_n, b_1, \dots, b_n)$
$a \xleftarrow{\$} \mathbb{X}$	a is uniformly sampled from $\mathbb{X}$
$a \leftarrow \chi$	a is sampled from a distribution χ
N_T	Size of a single T (e.g., T is ciphertext or key)

Definition 2.1 (LWE ciphertext). Given $n, q \in \mathbb{Z}$, the LWE ciphertext of the message $m \in \mathbb{Z}$ is $\mathsf{LWE}(m) := (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$, where $b = \mathbf{a} \cdot \mathbf{s} + m + e$. Additionally, $\mathbf{a} \xleftarrow{\$} \mathbb{Z}_q^n$, $\mathbf{s} \leftarrow \chi$ and $e \leftarrow \chi'$, where χ is a key distribution and χ' is an error distribution.

RLWE is a ring version of LWE on $\mathcal{R}_q$, which encrypts a vector of a message $\mathbf{m} \in \mathcal{R}_q$.

Definition 2.2 (RLWE ciphertext). The RLWE ciphertext of the message $\mathbf{m} \in \mathcal{R}_q$ is $\mathsf{RLWE}(\mathbf{m}) := (\mathbf{a}, \mathbf{b}) \in \mathcal{R}_q^2$, where $\mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{m} + \mathbf{e}$, $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$, $\mathbf{s} \leftarrow \chi$ and $e_i \leftarrow \chi'$.

The gadget LWE is a vector of LWE ciphertexts, which is used to construct a GSW ciphertext.

Definition 2.3 (Gadget LWE ciphertext). Given a gadget vector $\mathbf{v} = (v_1, \dots, v_\ell) \in \mathbb{Z}^\ell$, the gadget LWE, denoted by Lev is an ℓ-dimensional vector of LWE ciphertexts defined by

$$\mathsf{Lev}(m) := (\mathsf{LWE}(v_1 \cdot m), \dots, \mathsf{LWE}(v_\ell \cdot m)) \in \mathbb{Z}_q^{(n+1)\ell}.$$

GSW is a vector of gadget LWE ciphertexts utilized for multiplying two ciphertexts.

Definition 2.4 (GSW ciphertext). The GSW ciphertext is an $(n+1)$-dimensional vector of the gadget LWE ciphertexts defined by

$$\mathsf{GSW}(m) := (\mathsf{Lev}(-s_1 \cdot m), \dots, \mathsf{Lev}(-s_n \cdot m), \mathsf{Lev}(m)),$$

where s_i is the i-th element of the secret key $\mathbf{s}$.

We can also construct the gadget version of RLWE and the ring version of GSW, which are generalized as GLev and RGSW (or GGSW). For more details, see [15, 18].

2.2 Basic Operations

The LWE ciphertext offers homomorphic addition and scalar multiplication as follows.

Definition 2.5 (Addition). Homomorphic addition of $\mathsf{LWE}(m_1)$ and $\mathsf{LWE}(m_2)$ is written as

$$\mathsf{LWE}(m_1 + m_2) = \mathsf{LWE}(m_1) + \mathsf{LWE}(m_2).$$

Definition 2.6 (Scalar multiplication). Homomorphic scalar multiplication between $\mathsf{LWE}(m)$ and a plaintext $k \in \mathbb{Z}$ is written by

$$\mathsf{LWE}(k \cdot m) = k \cdot \mathsf{LWE}(m).$$

To multiply two ciphertexts in TFHE, an operation called the external product is required.

Definition 2.7 (External Product (XP)). The external product $\boxdot$ is defined by

$$\begin{aligned}\mathsf{GSW}(m_1) \boxdot \mathsf{LWE}(m_2) &:= \sum_{i=1}^{n} (a_i \odot \mathsf{Lev}(-s_i \cdot m_1)) + b \odot \mathsf{Lev}(m_1) \\ &= \mathsf{LWE}(-\mathbf{a} \cdot \mathbf{s} \cdot m_1 + b \cdot m_1) \\ &= \mathsf{LWE}(m_1 \cdot m_2),\end{aligned}$$

where $\mathsf{LWE}(m_2) = (\mathbf{a}, b)$ and $\odot$ is the gadget product defined by $a \odot \mathsf{Lev}(m) := \mathsf{LWE}(a \cdot m)$.

We can also define the XP operation between an RGSW ciphertext and an RLWE ciphertext by $\mathsf{RGSW}(\mathbf{m}_1) \boxdot \mathsf{RLWE}(\mathbf{m}_2) := \mathsf{RLWE}(\mathbf{m}_1 \cdot \mathbf{m}_2)$. More details can be found in [15].

2.3 Bootstrapping and Key Switching

In TFHE, bootstrapping is a crucial procedure used to reset the noise growth in an LWE ciphertext. Furthermore, during this process, it is possible to evaluate an arbitrary function $f : \mathbb{Z}_p \to \mathbb{Z}_p$. This capability is known as programmable bootstrapping.

Definition 2.8 (Programmable bootstrapping (PBS)). Given an LWE ciphertext $\mathsf{LWE}_\mathbf{s}(m)$ under the secret key $\mathbf{s}$, $\mathsf{RLWE}_{\mathbf{s}'}(f)$ encoding the look-up table of f under the new secret key $\mathbf{s}'$, and a bootstrapping key $\mathsf{bsk} := (\mathsf{GSW}_{\mathbf{s}'}(s_1), \ldots, \mathsf{GSW}_{\mathbf{s}'}(s_n))$, PBS outputs a fresh LWE as follows:

$$\mathsf{PBS}(\mathsf{LWE}_\mathbf{s}(m), \mathsf{RLWE}_{\mathbf{s}'}(f), \mathsf{bsk}) := \mathsf{LWE}_{\mathbf{s}'}(f(m)).$$

A limitation of PBS is its performance for a large plaintext modulus p. For example, in `TFHE-rs`, which is the latest TFHE library, the supported plaintext space is $\log_2 p \leq 7$. Since the secret key changes during PBS, the key switching procedure is invoked with a PBS execution to adjust the key length.

Definition 2.9 (Key switching (KS)). Given an LWE ciphertext $\mathsf{LWE}_\mathbf{s}(m)$ under the secret key $\mathbf{s}$ and a key switching key $\mathsf{ksk} := (\mathsf{Lev}_{\mathbf{s}'}(s_1), \ldots, \mathsf{Lev}_{\mathbf{s}'}(s_n))$, KS outputs an LWE:

$$\mathsf{KS}(\mathsf{LWE}_\mathbf{s}(m), \mathsf{ksk}) := \mathsf{LWE}_{\mathbf{s}'}(m).$$

In the FHE mode of TFHE, a common framework for a sequence of homomorphic operations is DP-KS-PBS. This framework begins with DP (dot product) operations, which include additions and scalar multiplications, followed by a KS and a (variant of) PBS operation. For details, refer to [2, 10].

In the LHE mode of TFHE, it is possible to homomorphically evaluate arbitrary Boolean functions or deterministic automata using GSW ciphertexts. Circuit bootstrapping is a key component of the LHE mode, enabling the conversion of LWE to the GSW ciphertext while initializing the noise.

Definition 2.10 (Circuit bootstrapping (CBS)). Given an LWE ciphertext $\mathsf{LWE}_\mathbf{s}(m)$ under the secret key $\mathbf{s}$ and a circuit bootstrapping key cbk, where a cbk consists of a bootstrapping key and multiple key switching keys, CBS outputs a fresh GSW:

$$\mathsf{CBS}(\mathsf{LWE}_\mathbf{s}(m), \mathsf{cbk}) := \mathsf{GSW}_{\mathbf{s}'}(m).$$

Standard CBS consists of ℓ invocations of functional bootstrapping or one $\mathsf{PBSmanyLUT}$ [11], followed by 2ℓ invocations of private key switching. Details about the CBS can be found in [9, 18]. In what follows, we construct homomorphic circuits utilizing the aforementioned operations. For simplicity, we may omit terms of the cryptographic structures when they are implicitly understood.

3 HOMOMORPHICALLY EVALUATING LOOK-UP TABLE

We aim to evaluate a Boolean function with the form $f : \mathbb{B}^n \to \mathbb{B}$ in TFHE. Among the state-of-the-art algorithms for evaluating arbitrary Boolean functions in TFHE [2, 3, 5, 13, 14], we focus on one of the most promising methods called the *CMux tree* [13]. Before describing the algorithm, we introduce several useful properties of Boolean functions.

3.1 Decomposing Boolean Function into Boolean Circuit in TFHE

An n-in 1-out Boolean function $f(x_1, x_2, \ldots, x_n) = y$ can be expressed in its Algebraic Normal Form (ANF), which is a combination of variables using the `XOR` ($\oplus$) and `AND` ($\wedge$) operators.

Definition 3.1 (Algebraic Normal Form (ANF)). The ANF of an n-in 1-out Boolean function $f : \{0,1\}^n \mapsto \{0,1\}$ is a logical formula in which each term corresponds to a specific input combination of n variables. The ANF is defined by

$$\begin{aligned}f(x_1, \ldots, x_n) = a_0 \oplus a_1 \wedge x_1 \oplus a_2 \wedge x_2 \oplus \ldots \oplus a_n \wedge x_n \oplus \\ a_{n+1} \wedge x_1 \wedge x_2 \oplus \ldots \oplus a_{\frac{n(n+1)}{2}} \wedge x_{n-1} \wedge x_n \oplus \\ \ldots \oplus a_{2^n-1} \wedge x_1 \wedge x_2 \wedge \ldots \wedge x_n,\end{aligned}$$

where $a_0, \ldots, a_{2^n-1} \in \mathbb{B}$ are the Boolean coefficients and $x_1, \ldots, x_n$ are the Boolean variables.

This ANF can be implemented as a homomorphic Boolean circuit in the FHE mode of TFHE. In the FHE mode, each variable (x_i or a_i) is encrypted and each operation ($\oplus$ or $\wedge$) is homomorphically evaluated by the *gate bootstrapping* [9]. This allows us to evaluate a Boolean function with arbitrary n without increasing the amount of noise in the output ciphertext. However, this approach is expensive in TFHE because it requires $O(n2^n)$ executions of the gate bootstrapping, which is one of the most time-consuming operations in TFHE.

An alternative way to decompose a Boolean function into a Boolean circuit is to express f as the Binary Decision Diagram (BDD). In this paper we describe the BDD using the CMux gate [13]. The CMux gate is a 3-in 1-out Boolean function defined by

$$\mathsf{CMux}(a, b, c) := \begin{cases} a & (c = 0), \\ b & (c = 1). \end{cases}$$

Definition 3.2 (Binary Decision Diagram (BDD)). The BDD of a Boolean function $f : \{0,1\}^n \mapsto \{0,1\}$ is a logical circuit described by

$$f(x_1,\ldots,x_n) = \mathsf{CMux}(\sigma_{1,0},\sigma_{1,1},x_1),$$
$$\sigma_{i,j} = \mathsf{CMux}(\sigma_{i+1,2j},\sigma_{i+1,2j+1},x_{i+1}) \text{ for } 0 \le j \le 2^i - 1,$$
$$\sigma_{n,j} = y_j \text{ for } 0 \le j \le 2^n - 1.$$

where y_j denotes the output of f for the j-th input $\mathbf{x}_j$, i.e., $y_j = f(\mathbf{x}_j)$, where $\mathbf{x}_j = (x_1,\ldots,x_n)$ and $j = \sum_{1\le i\le n} 2^{n-i}x_i$.

The BDD requires $2^n - 1$ CMux executions in total to evaluate a function. However, the noise growth during computation is shown to be still linear with respect to $\sqrt{n}$ in TFHE [7, Lemma 5.5]. Fortunately, the CMux gate can be executed without bootstrapping in TFHE, allowing us to evaluate f more efficiently than ANF in the LHE mode of TFHE.

3.2 CMux Tree in the LHE Mode of TFHE

In the standard CMux tree, we evaluate $f(x_1,\ldots,x_n)$ in a width-first (layer-by-layer) order. From Definition 3.2, we first fill the n-th layer values $\sigma_{n,j}$ for $0 \le j \le 2^n - 1$ by reading the entire output vector, which requires 2^n LWE ciphertexts. Recursively, the i-th layer is computed from the $(i+1)$-th layer by 2^i executions of CMux: $\sigma_{i,j} = \mathsf{CMux}(\sigma_{i+1,2j},\sigma_{i+1,2j+1},x_{i+1})$ for $0 \le j \le 2^i - 1$. Finally, $f(x_1,\ldots,x_n)$ is obtained in $\sigma_{0,0}$.

In the following, we introduce a practical construction of the CMux tree as implemented in [22], where the number of LWE ciphertexts required is $O(n)$ for an arbitrary n-in 1-out function f. The pseudo-code of the CMux tree is shown in Algorithm 1. In this paper we explain the CMux tree without packing techniques, such as vertical packing and horizontal packing [8] to generalize and simplify each process. While our algorithm clearly works with the horizontal packing, some adaptations would likely be required for the vertical packing. We leave this for future work.

A CMux tree is typically used for the homomorphic evaluation of a function f when only a LUT is available for an evaluator. A LUT of f is the set that collects all input-output pairs of f: $\{(\mathbf{x}_j,y_j)\}_{0\le j\le 2^n-1}$. First, we need to store the inputs for the index sol: $\mathbf{x}_{\text{sol}} = (x_1,x_2,\ldots,x_n)$ and all the outputs of f: $\mathbf{y} = (y_0,y_1,\ldots,y_{2^n-1})$, where sol $= \sum_{1\le i\le n} 2^{n-i}x_i$. Each x_i is encrypted as a GSW ciphertext. Each y_i is required to be encrypted as an LWE ciphertext when f is private. Otherwise, we encode y_i in a trivial LWE: $(\mathbf{0},y_i)$. Then, we traverse the CMux tree of depth n from two adjacent leaves (depth n) to the root (depth 0) using a working memory V of length $|V| = 2n$ and an auxiliary vector $\mathbf{t}$ of length $|\mathbf{t}| = n+1$. V_d stores two input LWE ciphertexts $V_{d,0},V_{d,1}$ for a CMux gate at depth $1 \le d \le n-1$, and $V_{0,0}$ stores the solution, i.e., $f(x_1,x_2,\ldots,x_n)$. The variable $t_d \in \{0,1,2\}$ stores a flag value that represents the state of V_d. Specifically,

(1) $t_d = 0$ indicates that both elements of V_d are $\perp$.
(2) $t_d = 1$ indicates that $V_{d,0} \ne \perp$ and $V_{d,1} = \perp$.
(3) $t_d = 2$ indicates that neither element of V_d is $\perp$.

Note that we do not need to encrypt $\mathbf{t}$, as its value does not reveal any information about x_i or y_i. If $t_d = 2$, we can execute a CMux gate by $\mathsf{CMux}(V_{d,0},V_{d,1},x_d)$ and its result is then appropriately stored in either $V_{d-1,0}$ or $V_{d-1,1}$. In TFHE, a CMux gate

Algorithm 1: CMuxTree [13, 22]

```
Input: $x_1, x_2, ..., x_n$, the vector of all the $2^n$ outputs of the
       function $f$: $\mathbf{y} = (y_0, y_1, \ldots, y_{2^n-1})$
Output: $f(x_1, x_2, ..., x_n)$
1  $V \leftarrow \{V_{i,j} \leftarrow \perp \mid 0 \le i \le n-1, j \in \{0,1\}\}$      ▷ Initialize $V$
2  $\mathbf{t} \leftarrow \{t_i = 0 \mid 0 \le i \le n\}$          ▷ Initialize vector $\mathbf{t}$ for flag
3  for $i \leftarrow 0, \ldots, 2^{n-1} - 1$ do
4      $t_n = 2$                       ▷ Depth $n$ (leaf) is ready for CMux
5      for $d \leftarrow n, \ldots, 1$ do
6          if $t_d = 2$ then
7              if $d = n$ then
8                  if $t_{d-1} = 0$ then
9                      $V_{d-1,0} \leftarrow \mathsf{CMux}(y_{2i}, y_{2i+1}, x_d)$
10                 else
11                     $V_{d-1,1} \leftarrow \mathsf{CMux}(y_{2i}, y_{2i+1}, x_d)$
12             else if $t_{d-1} = 0$ then
13                 $V_{d-1,0} \leftarrow \mathsf{CMux}(V_{d,0}, V_{d,1}, x_d)$
14             else
15                 $V_{d-1,1} \leftarrow \mathsf{CMux}(V_{d,0}, V_{d,1}, x_d)$
16             $t_{d-1} = t_{d-1} + 1$              ▷ Compute depth $d-1$
17             $t_d = 0$                             ▷ Reset depth $d$
18         else
19             break                              ▷ Goto next leaf
20 return $V_{0,0}$                          ▷ Return root value of $V$
```

$\mathsf{CMux}(a,b,c)$ is homomorphically evaluated by

$$\mathsf{GSW}(c) \boxdot (\mathsf{LWE}(b) - \mathsf{LWE}(a)) + \mathsf{LWE}(a). \tag{1}$$

Thus, we need to prepare n GSW ciphertexts of x_i as selectors for CMux gates. Finally, we obtain $\mathsf{LWE}(f(x_1,x_2,\ldots,x_n))$ in $V_{0,0}$. The correctness of the algorithm can be shown by inductively proving that each step maintains the desired invariant, leading to the correct evaluation of the function f. The minimum plaintext space is $p = 3$ for the CMux tree, as at least the set of integers $\{-1,0,1\}$ is necessary to store intermediate values to evaluate Eq. (1) for binary inputs $a,b,c \in \{0,1\}$.

3.2.1 *Computational Complexity.*

We describe the time and space complexity of Algorithm 1. In the LHE mode of TFHE, the external product (XP) in the CMux gate is the bottleneck in runtime. Therefore, the time complexity can be evaluated by counting the number of XP operations in the circuit. We also assume that the n GSW ciphertexts of the inputs are precomputed via circuit bootstrapping, as in [7]. Since we require 2^{i-1} XPs from depth i to $i-1$, the total number of XPs in the tree is

$$\sum_{1\le i\le n} 2^{i-1} = 2^n - 1. \tag{2}$$

The space complexity is

$$nN_{\mathsf{GSW}} + (2^n + 2n)N_{\mathsf{LWE}} + n + 1$$

when f is private. In most typical cases, f is a publicly available function. Thus the formula is rewritten as

$$nN_{\mathsf{GSW}} + 2nN_{\mathsf{LWE}} + 2^n + n + 1. \tag{3}$$

In TFHE, the relation $N_{\mathsf{GSW}} \gg N_{\mathsf{LWE}}$ holds. Our objective in this paper is to develop a time-memory trade-off algorithm that reduces the number of CMux executions in the tree (Eq. (2)) while allowing for a moderate increase in space complexity (Eq. (3)).

4 TIME-MEMORY TRADE-OFF ALGORITHMS

This section describes our time-memory trade-off algorithms for evaluating any Boolean function f in both the LHE mode and the FHE mode of TFHE.

4.1 Time-Memory Trade-off CMux Tree in the LHE Mode of TFHE

To the best of our knowledge, the CMux tree is the fastest method for homomorphically evaluating arbitrary functions in the LHE mode of TFHE. In [13], the CMux tree for any public Boolean function f can reduce the number of CMux gates via the Nerode's partitioning algorithm. In this paper, to reduce the number of CMux gates, we introduce a constructive CMux tree algorithm by taking an alternative approach that increases working memory as a trade-off. This approach leverages the number of binary patterns at a depth d of the tree, and can effectively reduce the runtime for evaluating arbitrary public Boolean functions.

The core idea of our algorithm is to decompose the entire function f, whose output vector is $\mathbf{y}$, into sufficiently small subfunctions $f^{(b)}$. Each subfunction $f^{(b)}$ produces an output vector $\mathbf{y}_i^{(b)}$ of length $b = 2^i$:

$$\mathbf{y} = (\mathbf{y}_1^{(b)}, \mathbf{y}_2^{(b)}, \dots, \mathbf{y}_{2^n/b}^{(b)}).$$

A function f with an output vector $\mathbf{y}$ is denoted by $f_{\mathbf{y}}$.

Then, each subfunction $f^{(b)}$ is evaluated based on the inputs $x_1, \dots, x_n$ beforehand and we utilize the results for the final evaluation of f. The number of distinct subfunctions $f^{(b)}$ decomposed from f is upper bounded by:

$$\min(2^b, 2^n/b). \tag{4}$$

In the CMux tree, there is a relationship between b and the depth d $(0 \le d \le n)$ of the tree, where $b = 2^{n-d}$. In addition, $f^{(2b)}$ can be efficiently evaluated using the evaluation results for $f^{(b)}$ and a CMux gate. These properties indicate that we can reduce the number of CMux executions significantly in the CMux tree regardless of the structure of f, especially near the bottom of the tree.

Example 4.1. We aim to evaluate the following 3-bit Boolean function $f_{\mathbf{y}}(x_1, x_2, x_3)$ with $x_1 = 1, x_2 = 0$ and $x_3 = 1$,

x_1	x_2	x_3	$\mathbf{y}$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

where the output vector is $\mathbf{y} = (1, 1, 0, 0, 1, 0, 0, 1)$. The function can be decomposed into two subfunctions: $f_{\mathbf{y}_1}^{(4)}(x_2, x_3)$ and $f_{\mathbf{y}_2}^{(4)}(x_2, x_3)$, where $\mathbf{y} = \mathbf{y}_1 \parallel \mathbf{y}_2$, $\mathbf{y}_1 = (1, 1, 0, 0)$ and $\mathbf{y}_2 = (1, 0, 0, 1)$. These subfunctions are evaluated with (x_2, x_3): $f_{\mathbf{y}_1}^{(4)}(0, 1) = 1$ and $f_{\mathbf{y}_2}^{(4)}(0, 1) = 0$.[1] Finally, $f_{\mathbf{y}}$ is evaluated using the following equality:

$$f_{\mathbf{y}}(x_1, 0, 1) = \mathtt{CMux}(f_{\mathbf{y}_1}^{(4)}(0, 1), f_{\mathbf{y}_2}^{(4)}(0, 1), x_1),$$

by substituting $x_1 = 1$.

4.1.1 *Algorithm Detail.*

The pseudo-code of our time-memory trade-off CMux tree is presented in Algorithm 2. The algorithm proceeds from bottom to top as follows. When the depth is $n - 1$ in the CMux tree, f is decomposed into a combination of four subfunctions: $f_{(0,0)}^{(2)}(x_n) = 0, f_{(0,1)}^{(2)}(x_n) = x_n, f_{(1,0)}^{(2)}(x_n) = 1 - x_n$ and $f_{(1,1)}^{(2)}(x_n) = 1$. These subfunctions are evaluated by a single variable x_n. An encrypted set $\{0, x_n, 1 - x_n, 1\}$ is sufficient to express the evaluation results of all subfunctions with $b = 2$. Thus, 2^{n-1} CMux executions from the naive CMux tree are no longer necessary at depth $n - 1$ with an additional memory of length $4N_{\mathsf{LWE}}$ as a trade-off.

At depth $n-2$, any $f_{\mathbf{y}_1 \parallel \mathbf{y}_2}^{(4)}(x_{n-1}, x_n)$ can be evaluated by $f_{\mathbf{y}_1}^{(2)}(x_n)$ and $f_{\mathbf{y}_2}^{(2)}(x_n)$:

$$f_{\mathbf{y}_1 \parallel \mathbf{y}_2}^{(4)}(x_{n-1}, x_n) = \mathtt{CMux}(f_{\mathbf{y}_1}^{(2)}(x_n), f_{\mathbf{y}_2}^{(2)}(x_n), x_{n-1}).$$

As there are 16 distinct subfunctions with $b = 4$, all subfunctions $f^{(4)}$ can be evaluated by at most 16 CMux gates. Additionally, if $\mathbf{y}_1 = \mathbf{y}_2$, we immediately obtain $f_{\mathbf{y}_1 \parallel \mathbf{y}_2}^{(4)}(x_{n-1}, x_n) = f_{\mathbf{y}_1}^{(2)}(x_n)$. Thus, in total, at most $16 - 4 = 12$ executions of CMux gates and $16N_{\mathsf{LWE}}$ are sufficient to obtain/store all the results, instead of 2^{n-2} CMux executions from the naive CMux tree.

We continue the above procedure until the depth d reaches $n - \ell$, where ℓ represents a parameter indicating a trade-off level for $1 \le \ell \le n$. After that, we switch to Algorithm 1, whose maximal depth is reduced to $n - \ell$. Note that each leaf of the reduced tree refers to the evaluation result of the corresponding subfunction $f^{(b)}(x_{n-\ell+1}, \dots, x_{n-1}, x_n)$ for $b = 2^\ell$. Finally, an LWE ciphertext of $f(x_1, \dots, x_n)$ is obtained in $V_{0,0}$.

4.1.2 *Analysis.*

We provide analytical results for Algorithm 2 with a trade-off parameter ℓ. For the time complexity, we again count the number of XP executions during the computation. The time complexity required to compute the evaluation results for all subfunctions $f^{(b)}$ for $b = 2^1, \dots, 2^\ell$ is

$$\sum_{b \in \{2^i\}_{2 \le i \le \ell}} \min(2^b - 2^{b/2}, 2^n/b).$$

The time complexity of the remaining CMux tree is

$$\sum_{1 \le i \le n-\ell} 2^{i-1} = 2^{n-\ell} - 1.$$

[1]Note that, e.g., when $\mathbf{y}_1 = \mathbf{y}_2$, we do not need to evaluate $f_{\mathbf{y}_2}^{(4)}(0, 1)$ and can reuse $f_{\mathbf{y}_1}^{(4)}(0, 1)$, which has already been computed, since $f_{\mathbf{y}_2}^{(4)}(0, 1) = f_{\mathbf{y}_1}^{(4)}(0, 1)$, and this explains the basic idea behind why our approach can lead to lower time complexity compared with the naive CMux tree.

Algorithm 2: Time-memory trade-off of CMuxTree

```
Input: $\ell \in \mathbb{N}$, $x_1, x_2, ..., x_n$, the vector of all the $2^n$ outputs
       of the function $f$: $\mathbf{y}$
Output: $f(x_1, x_2, ..., x_n)$
$V \leftarrow \{V_{i,j} \leftarrow \perp \mid 0 \le i \le n-\ell-1, j \in \{0,1\}\}$   ▷ Initialize $V$
$\mathbf{t} \leftarrow \{t_i = 0 \mid 0 \le i \le n-\ell\}$   ▷ Initialize vector $\mathbf{t}$ for a flag
$f^{(2)}_{(0,0)} \leftarrow 0$   ▷ Evaluation result of $f^{(2)}(x_n) = 0$
$f^{(2)}_{(0,1)} \leftarrow x_n$   ▷ $f^{(2)}(x_n) = x_n$
$f^{(2)}_{(1,0)} \leftarrow 1 - x_n$   ▷ $f^{(2)}(x_n) = 1 - x_n$
$f^{(2)}_{(1,1)} \leftarrow 1$   ▷ $f^{(2)}(x_n) = 1$
for $d \leftarrow n-1, \dots, n-\ell+1$ do
    $b \leftarrow 2^{n-d}$
    foreach subfunction $f^{(2b)}_{\mathbf{y}_1 \| \mathbf{y}_2}$ decomposed from $f$ do
        if $\mathbf{y}_1 = \mathbf{y}_2$ then
            $f^{(2b)}_{\mathbf{y}_1 \| \mathbf{y}_2} \leftarrow f^{(b)}_{\mathbf{y}_1}$
        else
            $f^{(2b)}_{\mathbf{y}_1 \| \mathbf{y}_2} \leftarrow \mathsf{CMux}(f^{(b)}_{\mathbf{y}_1}, f^{(b)}_{\mathbf{y}_2}, x_d)$
    end
for $i \leftarrow 0, \dots, 2^{n-\ell-1} - 1$ do
    $t_{n-\ell} = 2$   ▷ Depth $n-\ell$ is ready for CMux
    for $d \leftarrow n-\ell, \dots, 1$ do
        if $t_d = 2$ then
            if $d = n-\ell$ then
                $\mathbf{y}_1 \leftarrow \mathbf{y}[2^{\ell+1}i : 2^{\ell}(2i+1)]$
                $\mathbf{y}_2 \leftarrow \mathbf{y}[2^{\ell}(2i+1) : 2^{\ell+1}(i+1)]$
                if $t_{d-1} = 0$ then
                    $V_{d-1,0} \leftarrow \mathsf{CMux}(f^{(2^\ell)}_{\mathbf{y}_1}, f^{(2^\ell)}_{\mathbf{y}_2}, x_d)$
                else
                    $V_{d-1,1} \leftarrow \mathsf{CMux}(f^{(2^\ell)}_{\mathbf{y}_1}, f^{(2^\ell)}_{\mathbf{y}_2}, x_d)$
            else if $t_{d-1} = 0$ then
                $V_{d-1,0} \leftarrow \mathsf{CMux}(V_{d,0}, V_{d,1}, x_d)$
            else
                $V_{d-1,1} \leftarrow \mathsf{CMux}(V_{d,0}, V_{d,1}, x_d)$
            $t_{d-1} = t_{d-1} + 1$   ▷ Compute depth $d-1$
            $t_d = 0$   ▷ Reset depth $d$
        else
            break   ▷ Goto next node
return $V_{0,0}$   ▷ Return root value of $V$
```

For space complexity, the memory required to store results of all subfunctions $f^{(b)}$ for $b = 2^1, \dots, 2^\ell$ is

$$\sum_{b \in \{2^i\}_{1 \le i \le \ell}} \min(2^b, 2^n/b) N_{\mathsf{LWE}}.$$

The space complexity of the reduced CMux tree is

$$(n-1)N_{\mathsf{GSW}} + 2(n-\ell)N_{\mathsf{LWE}} + 2^n + n - \ell + 1.$$

If we set $\ell = O(\log n)$, we can reduce the time complexity by a factor of $O(n)$ at the cost of an additional space complexity of $O(2^n)$. This represents a practical time-memory trade-off, as $O(2^n)$ space is already required for the LUT. Experimentally, we show that $\ell = \log(n/2)$ works well to evaluate arbitrary 8-bit and 16-bit Boolean functions.

The noise growth of our algorithm is the same as that of the original CMux tree, i.e., proportional to $\sqrt{n}$. This is because the CMux depth of Algorithm 2 is $O(n)$, which is the same as that of Algorithm 1. Algorithm 2 does not leak any information about the inputs $x_1, x_2, \dots, x_n$ or the output value $f(x_1, x_2, \dots, x_n)$, except for the structure of the function f. The required plaintext space is $p = 3$, which is the same as in Algorithm 1, since $\{0, 1\}$ is necessary to store the evaluation results for subfunctions, $\{-1, 0, 1\}$ is required to evaluate a CMux gate with a single XP execution.

4.2 DNF and CNF in the FHE Mode of TFHE

There are seminal works on evaluating arbitrary Boolean functions in the FHE mode of TFHE [2, 3, 14]. Many of these are based on the BDD, which is similar to the CMux tree. In this study, we introduce alternative algorithms based on the Disjunctive Normal Form (DNF) or Conjunctive Normal Form (CNF) of the function f and its time-memory trade-off variants, which has smaller space complexity in terms of the number of LWE ciphertexts required. First we define the literal of x.

Definition 4.2 (Literal). The literal of a Boolean variable x is defined by

$$x^e := \begin{cases} \overline{x} & (e = 0), \\ x & (e = 1). \end{cases}$$

A DNF or CNF of a Boolean function f can be obtained via Boole's expansion theorem, which is often referred to as the Shannon expansion.

Theorem 4.3 (Boole's expansion theorem). *Let $f(x_1, x_2, \dots, x_n)$ be a Boolean function. Boole's expansion theorem states that a DNF of f can be expressed as follows:*

$$\begin{aligned} f(x_1, \dots, x_n) &= \overline{x_1} \wedge f(0, x_2, \dots, x_n) \vee x_1 \wedge f(1, x_2, \dots, x_n) \\ &= \cdots \\ &= \bigvee_{e_1, \dots, e_n \in \{0,1\}} x_1^{e_1} \wedge \cdots \wedge x_n^{e_n} \wedge f(e_1, \dots, e_n), \end{aligned} \quad (5)$$

where $\vee$ denotes the OR *operation and $\bigvee$ denotes the disjunction of logical terms. A CNF of f can be expressed as follows:*

$$\begin{aligned} f(x_1, \dots, x_n) &= (x_1 \vee f(0, x_2, \dots, x_n)) \wedge (\overline{x_1} \wedge f(1, x_2, \dots, x_n)) \\ &= \cdots \\ &= \bigwedge_{e_1, \dots, e_n \in \{0,1\}} x_1^{\overline{e_1}} \vee \cdots \vee x_n^{\overline{e_n}} \vee f(e_1, \dots, e_n), \end{aligned} \quad (6)$$

where $\bigwedge$ denotes the conjunction of logical terms. $f(e_1, \dots, e_n)$ equals y_i for $i = \sum_{1 \le j \le n} 2^{n-j} e_j$.

For the DNF of f, Eq. (5) can be homomorphically constructed in the LHE mode of TFHE by applying an XP operation for $\wedge$ and a homomorphic addition for $\vee$. This approach works because at most one conjunction term $(x_1^{e_1} \wedge \cdots \wedge x_n^{e_n} \wedge f(e_1, \dots, e_n))$ is equal to 1, and the computation is performed with $p = 2$. However, for large n, the decryption may fail due to noise growth. For the CNF of f, the above approach does not work, and a bootstrapping technique is required to perform non-linear operations.

With integer-wise TFHE, both Eq. (5) and Eq. (6) can be realized as a homomorphic circuit in the FHE mode of TFHE at a reasonable cost. The idea is to compute one conjunction (disjunction) term and perform a summation with previous calculations in a single PBS, which can be seen as an application of the *chaining method* [5]. The concrete construction of our algorithm is presented in Algorithm 3.

Algorithm 3: ShannonExp

```
Input: $x_1, x_2, ..., x_n$, the vector of all the $2^n$ outputs of the
       function $f$: $\mathbf{y}$
Output: $f(x_1, x_2, ..., x_n)$
1 $s \in \{0,1\}$
    ▷ Initialize the first state $s_0$ depending on DNF or CNF
2 for $(e_1, \dots, e_n) \in \{0,1\}^n$ do
3     $t \leftarrow f(e_1, e_2, \dots, e_n)$          ▷ $y_i$ with $i = \sum_{1 \le j \le n} 2^{n-j} e_j$
4     for $j \leftarrow 1, \dots, n$ do
5         $t \leftarrow t + x_j^{k(e_j)}$                        ▷ Hom. addition
6     $s \leftarrow (n+1)s$                     ▷ Hom. scalar multiplication
7     $s \leftarrow g(s+t)$          ▷ Programmable bootstrapping with $g$
8 return $s$                              ▷ Return the last state $s_{2^n}$
```

4.2.1 *Algorithm Detail.*

Algorithm 3 provides a framework for evaluating a function f in either of DNF or CNF representation. We aim to construct a chain of 2^n subcircuits C_i ($1 \le i \le 2^n$), where the plaintext space is commonly p, such that the inputs of C_i are literals of inputs and s_{i-1}, where s_{i-1} is an output of the previous subcircuit C_{i-1} (with $s_0 = \{0,1\}$), and C_i outputs s_i, ensuring that $s_{2^n} = f(x_1, \dots, x_n)$. A subcircuit C_i involves only homomorphic additions, homomorphic scalar multiplications, and a singe PBS. Such a subcircuit can be efficiently constructed using the method proposed in [3]. Each subcircuit is described by

$$s_i \leftarrow g((n+1)s_{i-1} + x_1^{k(e_1)} + \cdots + x_n^{k(e_n)} + f(e_1, \dots, e_n)),$$

where $k(e_i)$ is e_i when DNF is used and $\overline{e_i}$ when CNF is used. For DNF, $s_0 = 0$ and the non-linear function g in Algorithm 3 is defined as the following g_{DNF},

$$g_{\mathsf{DNF}}(x) := \begin{cases} 0 & (0 \le x \le n), \\ 1 & (n+1 \le x \le 2n+2). \end{cases}$$

For CNF, $s_0 = 1$ and the non-linear function g in Algorithm 3 is defined as the following g_{CNF},

$$g_{\mathsf{CNF}}(x) := \begin{cases} 0 & (0 \le x \le n+1), \\ 1 & (n+2 \le x \le 2n+2). \end{cases}$$

Since each subcircuit is executed in series, the required space complexity is the space needed to execute a single subcircuit C_i.

4.2.2 *Analysis.*

The time complexity of Algorithm 3 is 2^n, which is determined by the number of PBS executions. In the FHE mode of TFHE, a bootstrapping key (BKS) and a key switching key (KSK) are required to execute a PBS. The space complexity is

$$N_{\mathsf{BSK}} + N_{\mathsf{KSK}} + N_{\mathsf{RLWE}} + 3N_{\mathsf{LWE}} + 2^n. \tag{7}$$

An RLWE ciphertext is required to store the LUT of g. An additional LWE ciphertext is necessary in the KS-PBS TFHE to store a key-switched LWE during KS and PBS. The required plaintext space is $p = 4(n+1)+2$ when we use an even p, as it is necessary to double the domain of the non-negacyclic function g. If we choose an odd p, the plaintext space can be reduced by tweaking several modules in TFHE [3, Section 6]. The maximum failure probability of the circuit is measured just before the execution of a PBS.

4.3 Time-Memory Trade-off Algorithms for DNF and CNF

In what follows, we explain the time-memory trade-off algorithm of the DNF variant. The CNF variant can also be obtained in a similar manner. The pseudo-code is provided in Algorithm 4.

Algorithm 4: Time-memory trade-off of ShannonExp

```
Input: $\ell \in \mathbb{N}$, $x_1, x_2, ..., x_n$, the vector of all the $2^n$ outputs
       of the function $f$: $\mathbf{y}$
Output: $f(x_1, x_2, ..., x_n)$
1 $s \in \{0,1\}$
    ▷ Initialize the first state $s_0$ depending on DNF or CNF
2 $f^{(2)}_{(0,0)} \leftarrow 0$
3 $f^{(2)}_{(0,1)} \leftarrow x_n$
4 $f^{(2)}_{(1,0)} \leftarrow 1 - x_n$
5 $f^{(2)}_{(1,1)} \leftarrow 1$
6 for $d \leftarrow n-1, \dots, n-\ell+1$ do
7     $b \leftarrow 2^{n-d}$
8     foreach subfunction $f^{(2b)}_{\mathbf{y}_1 \| \mathbf{y}_2}$ decomposed from $f$ do
9         if $\mathbf{y}_1 = \mathbf{y}_2$ then
10            $f^{(2b)}_{\mathbf{y}_1 \| \mathbf{y}_2} \leftarrow f^{(b)}_{\mathbf{y}_1}$
11        else
12            $f^{(2b)}_{\mathbf{y}_1 \| \mathbf{y}_2} \leftarrow h(f^{(b)}_{\mathbf{y}_1} + 3f^{(b)}_{\mathbf{y}_2} + 2x_d)$      ▷ PBS with $h$
13    end
14 for $(e_1, \dots, e_{n-\ell}) \in \{0,1\}^n$ do
15    $\mathbf{y}_1 \leftarrow \mathbf{y}[2^\ell \sum_i 2^{n-\ell-i} e_i : 2^\ell((\sum_i 2^{n-\ell-i} e_i) + 1)]$
16    $t \leftarrow f^{(2^\ell)}_{\mathbf{y}_1}$                  ▷ $f(e_1, \dots, e_{n-\ell}, x_{n-\ell+1}, \dots x_n)$
17    for $j \leftarrow 1, \dots, n-\ell$ do
18        $t \leftarrow t + x_j^{k(e_j)}$                           ▷ Hom. addition
19    $s \leftarrow (n-\ell+1)s$              ▷ Hom. scalar multiplication
20    $s \leftarrow g(s+t)$                                  ▷ PBS with $g$
21 return $s$                         ▷ Return the last state $s_{2^{n-\ell}}$
```

4.3.1 *Algorithm Detail.*

For an integer parameter ℓ, we consider the $(n-\ell)$-level of the Shannon expansion:

$$f(x_1, \dots, x_n) = \bigvee_{e_1, \dots, e_{n-\ell} \in \{0,1\}} x_1^{e_1} \wedge \cdots \wedge x_{n-\ell}^{e_{n-\ell}} \wedge f(e_1, \dots, e_{n-\ell}, x_{n-\ell+1}, \dots, x_n). \tag{8}$$

Eq. (8) has $2^{n-\ell}$ conjunction terms. If one has all the evaluation results for $f(e_1, \ldots, e_{n-\ell}, x_{n-\ell+1}, \ldots, x_n)$, Eq. (8) can be efficiently computed via Algorithm 3. Evaluating $f(e_1, \ldots, e_{n-\ell}, x_{n-\ell+1}, \ldots, x_n)$ is equivalent to evaluating a subfunction $f^{(b)}(x_{n-\ell+1}, \ldots, x_n)$ of f, where both functions share the common output vector $\mathbf{y}$ of length $b = 2^\ell$. Thus, our strategy is to compute evaluation results for all subfunctions $f^{(b)}$ $(2^1 \leq b \leq 2^\ell)$ in the FHE mode.

A subfunction $f^{(2b)}_{\mathbf{y}_1 \| \mathbf{y}_2}(x_{n-\ell+1}, \ldots, x_n)$ is evaluated by a DNF from $f^{(b)}_{\mathbf{y}_1}(x_{n-\ell+2}, \ldots, x_n)$ and $f^{(b)}_{\mathbf{y}_2}(x_{n-\ell+2}, \ldots, x_n)$ as follows:

$$f^{(2b)}_{\mathbf{y}_1 \| \mathbf{y}_2}(x_{n-\ell+1}, \ldots, x_n) = x_{n-\ell+1} \wedge f^{(b)}_{\mathbf{y}_1}(x_{n-\ell+2}, \ldots, x_n) \vee \overline{x_{n-\ell+1}} \wedge f^{(b)}_{\mathbf{y}_2}(x_{n-\ell+2}, \ldots, x_n) \quad (9)$$

In the FHE mode of TFHE, Eq. (9) can be implemented by a single PBS with the plaintext space $p = 7$ by utilizing the circuit construction method [3]:

$$f^{(2b)}_{\mathbf{y}_1 \| \mathbf{y}_2}(x_{n-\ell+1}, \ldots, x_n) = h(f^{(b)}_{\mathbf{y}_1} + 3f^{(b)}_{\mathbf{y}_2} + 2x_{n-\ell+1}), \quad (10)$$

where a non-linear function h is defined by

$$h(x) := \begin{cases} 0 & (x = 0, 1, 2, 5), \\ 1 & (x = 3, 4, 6). \end{cases}$$

Similar to the LHE mode, all subfunctions with length $b \leq 2^\ell$ are evaluated by recursively applying Eq. (10).

If we have a sufficiently large plaintext space, $f^{(2^\ell)}_{\mathbf{y}_1}(x_{n-\ell+1}, \ldots, x_n)$ can be directly evaluated with a single PBS, and $p = 2^{\ell-1} + 1$ as follows:

$$f^{(2^\ell)}_{\mathbf{y}_1}(x_{n-\ell+1}, \ldots, x_n) = h_{\mathbf{y}_1}\left(\sum_{n-\ell+1 \leq i \leq n} 2^{n-i} x_i\right), \quad (11)$$

where the LUT of $h_{\mathbf{y}_1}$ is defined by the output vector $\mathbf{y}_1$.

4.3.2 *Analysis.*

The required number of PBS executions to construct all subfunctions using Eq. (10) is exactly the same as the number of XP executions in the LHE mode, which is

$$\sum_{b \in \{2^i\}_{2 \leq i \leq \ell}} \min(2^b - 2^{b/2}, 2^n/b).$$

If we use Eq. (11), the time complexity is reduced to $2^{n-\ell}$. The time complexity of the remaining DNF circuit is $2^{n-\ell}$. For space complexity, we obtain

$$N_{\mathsf{BSK}} + N_{\mathsf{KSK}} + N_{\mathsf{RLWE}} + \left(\sum_b \min(2^b, 2^n/b) + 3\right) N_{\mathsf{LWE}} + 2^n,$$

if Eq. (10) is implemented. At least one RLWE ciphertext is required to store LUTs for PBS operations. Adopting Eq. (11) results in the same space complexity as Eq. (7), since we can reuse a single RLWE ciphertext to store the LUTs of $h_{\mathbf{y}_1}$ and g. However, variations from Algorithm 3 could occur in cryptographic parameters as a result of changes in the plaintext space p.

A practical time-memory trade-off for the circuit is achieved by setting $\ell = O(\log n)$ in both cases. This adjustment reduces time by a factor of $O(n)$ and increases space to $O(2^n)$. Our experiments show that applying Eq. (11) with $\ell = 4$ to the circuit results in the maximal speedup for 8-bit Boolean functions.

5 EXPERIMENTS

We implemented these methods by forking the `tfhe-rs` library [22] version 0.2.4 (commit `cb1a95`). All experiments were conducted on a desktop PC equipped with an AMD Ryzen 9 3900 processor. The cryptographic parameters we used are listed in Table 2. These parameters are derived from the `tfhe-rs` library. The security levels are obtained from the `lattice-estimator`[2] [1].

5.1 LHE Mode

We compare the performance between Algorithm 1 (CMuxTree) and Algorithm 2 (CMuxTreeTMTO) for 8-in 1-out Boolean functions. In this experiment, we evaluate randomly generated 100 Boolean functions and measure their mean values. We use a trade-off parameter $\ell = \log(n/2) = 2$, which minimizes the runtime. Additionally, we utilize RLWE and RGSW (GGSW) ciphertexts instead of LWE and GSW ciphertexts, as this version of `tfhe-rs` supports fast XP operations using the Fast Fourier Transform for these types of ciphertexts. For the cryptographic parameters, we use the `TFHE-lib` parameter set, which is derived from the `tfhe-rs` library. The results are shown in Table 3.

In our environment, a single XP execution consumes approximately 10 microseconds, which accounts for the dominant part of the total runtime. To calculate the total memory usage, we set $N_{\mathsf{GGSW}} = 32,768$ bytes and $N_{\mathsf{RLWE}} = 16,384$ bytes. Typically, 8 bytes are necessary for a single plaintext data unit. Consequently, we achieved a time-memory trade-off with a 2.8 × reduction in total runtime and 1.4 × increase in memory usage.

We also conducted the same experiment for relatively large 16-bit Boolean functions.

For this experiment, we used a trade-off parameter $\ell = \log(n/2) = 3$ and the same parameter set. The results are shown in Table 4.

The number of XP executions is reduced from 2^{16} to $2^2 + 2^4 + 2^8 + 2^{13}$, resulting in a 7.8× speed-up in runtime. The required memory capacity is increased to approximately 3.8×, which also indicates that the time-memory trade-off is not a trivial exchange between time and space complexity. We present a graphical representation of the time-memory trade-off for 16-bit Boolean functions in Figure 2. It indicates that $\ell = 3$ is a favorable choice for both time and memory.

5.2 FHE Mode

In the FHE mode, we compare Algorithm 3 (Shannon) and Algorithm 4 (ShannonTMTO) for 8-bit Boolean functions. The cryptographic parameter set we used for Algorithm 3 is `TFHE-m6`. For Algorithm 4, we compare the two variants using Eq. (10) (version 1) and Eq. (11) (version 2). Version 1 uses $\ell = \log n = 3$ and `TFHE-m6` to compare the performance between Algorithm 3 and Algorithm 4 under the same conditions. Version 2 uses $\ell = 4$ and `TFHE-m5`, whose plaintext space is smaller than that of Algorithm 3 due to the larger parameter ℓ. The results are presented in Table 5.

A single PBS execution takes 53 milliseconds with the `TFHE-m5` parameter set and 119 milliseconds with the `TFHE-m6` parameter set. These durations are significantly slower than the XP operation in the LHE mode, making the PBS the bottleneck in the entire computation. We measured $N_{\mathsf{BSK}} = 117$ megabytes and $N_{\mathsf{KSK}} = 134$

[2] https://github.com/malb/lattice-estimator

Table 2: Cryptographic parameters

Name	Bit Security	p	LWE dim.	RLWE dim.	ℓ_{PBS}	$\log_2(B_{\mathrm{PBS}})$	ℓ_{KS}	$\log_2(B_{\mathrm{KS}})$	σ^2 for LWE	σ^2 for RLWE
TFHE-lib	128	4	830	1024	1	23	5	3	2^{-39}	2^{-103}
TFHE-m5	128	16	808	4096	1	22	3	5	2^{-39}	2^{-103}
TFHE-m6	128	32	875	8192	1	22	6	3	2^{-41}	2^{-124}

Table 3: Results for 8-in 1-out Boolean functions.

Algorithm	# of XPs	Time [us]	Space [KB]
CMuxTree	255	2,797	**526**
CMuxTreeTMTO	**79**	**993**	756

Table 4: Results for 16-in 1-out Boolean functions.

Algorithm	# of XPs	Time [ms]	Space [MB]
CMuxTree	65,535	690	**1.6**
CMuxTreeTMTO	**8,443**	**89**	6.0

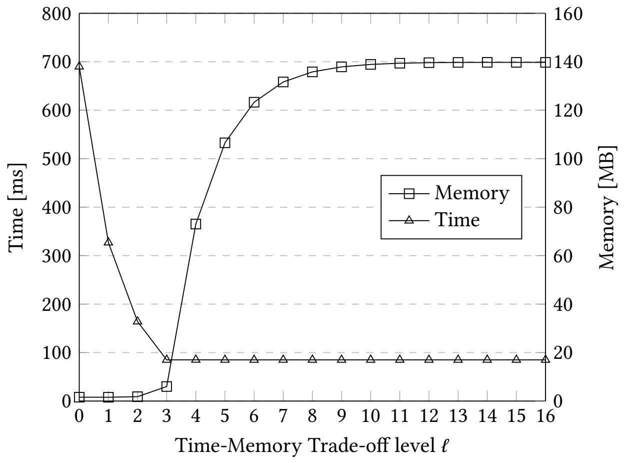

Figure 2: Time-Memory Trade-off for 16-in 1-out functions.

Table 5: Results for 8-in 1-out Boolean functions.

Algorithm	# of PBSs	Time [s]	Space [MB]
Shannon	256	30.5	637.6
ShannonTMTO v1	78	9.3	637.7
ShannonTMTO v2	**32**	**1.7**	**251.7**

megabytes for TFHE-m5, $N_{\mathsf{BSK}} = 235$ megabytes and $N_{\mathsf{KSK}} = 403$ megabytes for TFHE-m6, respectively.

The time-memory trade-off algorithm using Eq. (10) results in less than a 1% increase in memory, while the runtime is approximately 3.3× faster. This is because the space complexity required for a BSK and a KSK is the dominant part of the total memory usage.

The time-memory trade-off algorithm, which uses Eq. (11) with $\ell = 4$, effectively balances the time complexity required for computing all subfunctions and merging the results, resulting in 17.9× speedups compared with the original algorithm. Additionally, the space complexity is reduced to approximately 40%. This reduction occurs because the required number of dimensions for cryptographic structures decreases as the plaintext space is reduced.

We acknowledge that there are more effective methods for evaluating 8-bit Boolean functions in the FHE mode, such as directly evaluating the entire function with a single 8-bit PBS, using the tree-PBS [14], the bootstrapped CMux tree [2], or NTRU-based schemes [4, 20]. The objective of this paper is to demonstrate time-memory trade-offs for TFHE in the FHE mode. Identifying the fastest method or exploring time-memory trade-offs for other methods will be left as future work.

A key advantage of function evaluation in the FHE mode is that it eliminates the need for costly CBS executions to convert LWE ciphertexts to GSW ciphertexts[3]. However, the FHE mode incurs additional costs due to the execution of PBS operations. In practice, hybrid-mode algorithms, such as WoP-PBS [2], which combine the benefits of both LHE and FHE modes, can be more effective. Further exploration into function evaluation using the hybrid-mode TFHE remains open for future research.

6 CONCLUSION

In this paper, we presented time-memory trade-off algorithms for evaluating arbitrary Boolean functions in both the LHE and FHE modes of TFHE. The concrete verification of these trade-offs using the latest TFHE library was conducted. Future research includes the development of time-memory trade-off algorithms for evaluating functions in the hybrid-mode TFHE. Further acceleration is expected by searching more appropriate cryptographic parameters adapted for our schemes, or by combining our method with existing state-of-the-art techniques, such as vertical/horizontal packing and multi-value bootstrapping.

Acknowledgments. T. Nishide was supported in part by The Telecommunications Advancement Foundation and JSPS KAKENHI Grant Number 23K18459.

REFERENCES

[1] Martin R Albrecht, Benjamin R Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W Postlethwaite, Fernando Virdia, and Thomas Wunderer. 2018. Estimate all the {LWE, NTRU} schemes!. In *International Conference on Security and Cryptography for Networks*. Springer, 351–367.

[2] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2023. Parameter optimization and larger precision for (T) FHE. *Journal of Cryptology* 36, 3 (2023), 28.

[3] Nicolas Bon, David Pointcheval, and Matthieu Rivain. 2024. Optimized Homomorphic Evaluation of Boolean Functions. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)* 2024, 1 (2024). https://eprint.iacr.org/2023/1589 To appear.

[3]The latest implementation of the CBS [19] has execution times comparable to those of PBS.

[4] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder VL Pereira, and Nigel P Smart. 2022. FINAL: faster FHE instantiated with NTRU and LWE. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 188–215.

[5] Florian Bourse, Olivier Sanders, and Jacques Traoré. 2020. Improved Secure Integer Comparison via Homomorphic Encryption. In *Topics in Cryptology – CT-RSA 2020*. 391–416.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*. 309–325. https://doi.org/10.1145/2090236.2090262

[7] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology – ASIACRYPT 2016*. 3–33.

[8] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2017. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 377–408.

[9] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology* 33, 1 (2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x

[10] Ilaria Chillotti, Marc Joye, and Pascal Paillier. 2021. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings 5*. Springer, 1–19.

[11] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2021. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III 27*. Springer, 670–699.

[12] Thi Van Thao Doan, Mohamed-Lamine Messai, Gérald Gavin, and Jérôme Darmont. 2023. A survey on implementations of homomorphic encryption schemes. *The Journal of Supercomputing* 79, 13 (2023), 15098–15139. https://doi.org/10.1007/s11227-023-05233-z

[13] Nicolas Gama, Malika Izabachène, Phong Q Nguyen, and Xiang Xie. 2016. Structural lattice reduction: generalized worst-case to average-case reductions and homomorphic cryptosystems. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 528–558.

[14] Antonio Guimaraes, Edson Borin, and Diego F Aranha. 2021. Revisiting the functional bootstrap in TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, 2 (Feb. 2021), 229–253. https://doi.org/10.46586/tches.v2021.i2.229-253

[15] Marc Joye. 2022. SoK: Fully Homomorphic Encryption over the [Discretized] Torus. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022, 4 (Aug. 2022), 661–692. https://doi.org/10.46586/tches.v2022.i4.661-692

[16] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On ideal lattices and learning with errors over rings. In *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*. Springer, 1–23.

[17] Kotaro Matsuoka, Ryotaro Banno, Naoki Matsumoto, Takashi Sato, and Song Bian. 2021. Virtual Secure Platform: A Five-Stage Pipeline Processor over TFHE. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 4007–4024. https://www.usenix.org/conference/usenixsecurity21/presentation/matsuoka

[18] Ruida Wang, Yundi Wen, Zhihao Li, Xianhui Lu, Benqiang Wei, Kun Liu, and Kunpeng Wang. 2024. Circuit Bootstrapping: Faster and Smaller. In *Advances in Cryptology – EUROCRYPT 2024*, Marc Joye and Gregor Leander (Eds.). Springer Nature Switzerland, Cham, 342–372.

[19] Benqiang Wei, Xianhui Lu, Ruida Wang, Kun Liu, Zhihao Li, and Kunpeng Wang. 2024. Thunderbird: Efficient Homomorphic Evaluation of Symmetric Ciphers in 3GPP by combining two modes of TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024, 3 (2024), 530–573.

[20] Binwu Xiang, Jiang Zhang, Yi Deng, Yiran Dai, and Dengguo Feng. 2023. Fast blind rotation for bootstrapping FHEs. In *Annual International Cryptology Conference*. Springer, 3–36.

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

[22] Zama. 2022. 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.