Overlapped Bootstrapping for FHEW/TFHE and Its Application to SHA3

Deokhwa Hong[1], Youngjin Choi[1], Yongwoo Lee[1], and Young-Sik Kim[2]

[1] Inha University, Incheon, Korea.
deokhwa@inha.edu, yj160727@inha.edu, yongwoo@inha.ac.kr
[2] Daegu Gyeongbuk Institute of Science and Technology, Daegu, Korea
ysk@dgist.ac.kr

Abstract. Homomorphic Encryption (HE) enables operations on encrypted data without requiring decryption, thus allowing secure handling of confidential data within smart contracts. Among the known HE schemes, FHEW and TFHE are particularly notable for use in smart contracts due to their lightweight nature and support for arbitrary logical gates. In contrast, other HE schemes often require several gigabytes of keys and are limited to supporting only addition and multiplication. As a result, many researches have been conducted on implementing smart contract functionalities over HE, broadening the potential applications of blockchain technology. However, a significant drawback of the FHEW/TFHE schemes is the need for bootstrapping after the execution of each binary gate. While bootstrapping reduces noise in the ciphertext, it also becomes a performance bottleneck due to its computational complexity.

In this work, we propose an efficient new bootstrapping method for FHEW/TFHE that takes advantage of the flexible scaling factors of encrypted data. The proposed method is particularly beneficial in circuits with consecutive XOR gates. Moreover, we implement Keccak using FHEW/TFHE, as it is one of the most important functions in smart contracts. Our experimental results demonstrate that the proposed method reduces the runtime of Keccak over HE by 42%. Additionally, the proposed method does not require additional keys or parameter sets from the key-generating party and can be adopted by the computing party without need for any extra information.

Keywords. Homomorphic encryption, bootstrapping, cryptographic hash function

1 Introduction

Homomorphic Encryption. Homomorphic encryption (HE) is a cryptographic technique that allows operations to be performed on encrypted data without requiring decryption. Among HE schemes, those based on learning with errors (LWE) are the strongest candidates. However, one limitation of these HE schemes is the accumulation of noise in the ciphertext during operations, thus the message is usually separated from the noise by multiplying it with a large value

known as the scaling factor. Despite this, noise accumulates during the evaluation of homomorphic circuits, increasing the likelihood of decryption failure. To address this, Gentry introduced the bootstrapping technique, which allows for the construction of fully homomorphic encryption (FHE) systems capable of performing an unlimited number of operations [15].

Since Gentry's breakthrough, various FHE schemes have been developed. Notable examples include the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [5] and the Brakerski-Fan-Vercauteren (BFV) scheme [14], both are FHE over integers. Another prominent scheme is CKKS [8], proposed by Cheon et al., which supports FHE over real numbers. Additionally, the FHEW/TFHE and its variants [13, 19, 18, 9, 10] provide efficient FHE for boolean (or larger) gate operations. Various libraries, such as SEAL [21], TFHE-rs [23], and OpenFHE [20], are available, facilitating the development of a wide range of FHE applications.

Applications. FHE has become a foundational cryptographic tool, applied in many fields such as secure multiparty computation and private information retrieval. It is also actively being adopted artificial privacy-preserving intelligence and machine learning [17, 7], as well as in symmetric key cryptosystems like AES usually for the purpose of transciphering [16, 22] and cryptographic hash functions like SHA256, for example, in the Zama bounty program.

While public blockchains offer high integrity and support a wide range of applications via smart contracts, they suffer from a significant limitation: all data stored on the blockchain is publicly accessible. HE, however, allows for computation on encrypted data, enabling smart contracts to process confidential information while preserving privacy. This enhances the utility of smart contracts for handling confidential data.

For example, extensive research is being conducted on fhEVM [11], which aims to execute the Ethereum Virtual Machine (EVM), the environment for running smart contracts, using FHE. In this context, FHEW/TFHE schemes play a crucial role due to their low memory requirements and efficient handling of arbitrary boolean gate operations.

Our Contribution. In this paper, we propose a novel circuit evaluation method for the FHEW/TFHE schemes by flexibly adjusting the message scaling factor during its bootstrapping step. Our contributions are summarized as follows:

1. **Evaluation of symmetric gate with reduced failure probability:** The proposed method is particularly efficient in scenarios where consecutive X(N)OR gate operations are prevalent. Such operations are commonly found in binary circuits, especially in cryptographic hash functions like Keccak, SHA3. Our technique can be applied to a wide range of algorithms and does not require additional key generation, thus imposing no extra burden on the client. In FHEW/TFHE schemes, the first step of gate bootstrapping between two ciphertexts typically involves addition, subtraction, or multiplication by a constant, followed by functional bootstrapping with gate

evaluation. Our method optimizes this process for XOR gates by replacing the initial step of subtracting two noisy ciphertexts and multiplying by a constant with a simpler addition operation. By adjusting the message scaling factor during the functional bootstrapping process, we reduce the noise introduced during these operations, thereby lowering the failure probability of XOR gate operations. Importantly, this adjustment uses only public information and does not require any additional computation.

2. **Overlapped bootstrapping:** Cheon et al. [6] highlighted the relatively high computational failure probability of FHEW/TFHE, which prevents it from achieving IND-CPA$^{\mathrm{D}}$ security (indistinguishability under chosen plaintext attack with decryption oracle). To mitigate this, parameter adjustments were made to reduce the failure probability to negligible levels. Applying our technique in this context further reduces the failure probability without sacrificing performance.
   In response to these challenges, we propose a novel approach that allows multiple XOR gate operations to be performed with a single bootstrapping operation, a technique we refer to as *overlapped bootstrapping*. Empirical results show that the overlapped bootstrapping reduces the runtime of circuits with consecutive XORs a lot, especially, it reduces the runtime of Keccak256 up to 44%.

1.1 Organization

The rest of the paper is organized as follows. Section 2 presents the basics of lattice-based HE, prior FHEW/TFHE bootstrapping methods, and the Keccak algorithm. Our proposed bootstrapping method is detailed in Section 3. In Section 4, we provide implementation results and analyze runtime performance. Finally, we conclude the paper in Section 5 with remarks.

2 Preliminaries

The inner product between two vectors is denoted by $\langle \cdot, \cdot \rangle$, and the multiplication of two polynomials is either denoted by a dot $(\cdot)$ or omitted depending on the context. Let the polynomial ring be $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, where $X^N + 1$ is the $2N$-th cyclotomic polynomial for some power of two N. We denote the residue ring of $\mathcal{R}$ modulo an integer Q as $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$. Elements in $\mathcal{R}_Q$ are represented in bold, such as $\boldsymbol{a} \in \mathcal{R}_Q$, and the i-th coefficient of the element $\boldsymbol{a}$ is denoted by a_i. Vectors are represented by $\vec{v}$, and their i-th element is written as v_i. We use $x \leftarrow \chi$ to indicate that x is sampled from the distribution χ; $x \leftarrow S$ denotes that x is uniformly sampled from a set S.

2.1 Basic (Ring-)LWE Encryption

Let q and n be positive integers. We define LWE encryption as follows:

$$\mathrm{LWE}_{\vec{s}}(m) = (\vec{a}, b) = (\vec{a}, \langle \vec{a}, \vec{s} \rangle + m + e) \in \mathbb{Z}_q^{n+1},$$

where $m \in \mathbb{Z}_q$ is the message, $\vec{s} \leftarrow \chi_{\text{sk}}$ is the secret key, $\vec{a} \leftarrow \mathbb{Z}_q^n$ is a random vector, and $e \leftarrow \chi_e$ is the noise term. Note that χ_e is typically a discrete Gaussian distribution with variance σ^2 and zero mean, and χ_{sk} is typically chosen as $\mathcal{B} = \{0, 1\}$, $\mathcal{T} = \{-1, 0, 1\}$, or larger secret key sets. We omit the subscript $\vec{s}$ when it is obvious. The decryption of the LWE ciphertext is defined as follows:

$$\text{LWE}^{-1}(\vec{c}, \vec{s}) = b - \langle \vec{a}, \vec{s} \rangle = m + e \approx m,$$

where $\vec{c} = (\vec{a}, b) \in \mathbb{Z}_q^{n+1}$ is the LWE ciphertext. Through this process, we can obtain a message with some errors included.

Ring-LWE (RLWE) encryption is an extension of LWE encryption to $\mathcal{R}_q$, and we define RLWE ciphertext of message $\boldsymbol{m}$ under secret $\boldsymbol{s}$ as follows:

$$\text{RLWE}(m) = (\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a}, \boldsymbol{a} \cdot \boldsymbol{s} + \boldsymbol{m} + \boldsymbol{e}) \in \mathcal{R}_q^2,$$

where $\boldsymbol{m} \in \mathcal{R}_q$ is the message, $\boldsymbol{s} \in \mathcal{R}_q$ is the secret key, $\boldsymbol{a} \in \mathcal{R}_q$ is a polynomial with random coefficients, and $\boldsymbol{e} \in \mathcal{R}_q$ is the noise. Note that N is the degree of $\mathcal{R}_q$, so the number of messages in the RLWE scheme is N. Similar to LWE, we define RLWE decryption as follows:

$$\langle (\boldsymbol{a}, \boldsymbol{b}), (\boldsymbol{s}, 1) \rangle = \boldsymbol{b} - \boldsymbol{a} \cdot \boldsymbol{s} = \boldsymbol{m} + \boldsymbol{e} \approx \boldsymbol{m},$$

where $(\boldsymbol{a}, \boldsymbol{b}) \in \mathcal{R}_q^2$ is an RLWE ciphertext.

2.2 FHEW-like Cryptosystems

FHEW and TFHE are HE schemes that support homomorphic operations on boolean gates. In FHEW/TFHE, the message is defined as $m \in \mathcal{B}$, and the LWE encryption for FHEW/TFHE is defined as follows:

$$\text{LWE}(m) = (\vec{a}, b) = (\vec{a}, \langle \vec{a}, \vec{s} \rangle + m \cdot \Delta + e) \in \mathbb{Z}_q^{n+1},$$

where $\Delta = \frac{q}{4}$ is the scaling factor used for binary messages.

FHEW-like Bootstrapping. Ducas and Micciancio first proposed an HE scheme with bootstrapping that operates in less than a second and uses a small key size, called FHEW [13]. This bootstrapping process consists of three main functions: accumulator initialization, blind rotation, and LWE extraction. The definitions of each function are given as follows.

Accumulator Initialization. In FHEW/TFHE, the accumulator is initialized as $\text{ACC} \leftarrow \text{RLWE}(\boldsymbol{h})$, where the initial polynomial $\boldsymbol{h} = \sum h_i X^i$ is determined by a mapping function $f : \mathbb{Z}_q \rightarrow \mathbb{Z}_p$. Here, p represents the plaintext modulus, and the maping function satisfies the property $f(v + \frac{q}{2}) = -f(v)$ for all possible values $v \in \mathbb{Z}_q$. Details regarding the mapping function are elaborated upon in the blind rotation step. We initialize $h_i = f(-i)$.

Blind Rotation. The blind rotation is a core technique to refresh a high-noise ciphertext. This involves performing a homomorphic multiplication of the monomial X^u on the previously defined accumulator, where $u = -\langle \vec{a}, \vec{s} \rangle$. After this process, the constant term of the accumulator is $f(b - \langle \vec{a}, \vec{s} \rangle) = f(m + e)$, and the result is encrypted in RLWE. To refresh the noise, f is defined as the decryption function. In FHEW/TFHE, different f functions are defined for each boolean gate, and they are called mapping functions. For example, the mapping function f for AND gate is defined as follows:

$$f(m^*) = \begin{cases} \frac{q}{8} & \text{if } \frac{3q}{8} \leq m^* < \frac{7q}{8} \\ -\frac{q}{8} & \text{if } -\frac{q}{8} \leq m^* < \frac{3q}{8}, \end{cases}$$

with $\Delta = \frac{q}{4}$. The definition of the mapping function for all gate operations can be found in Table 1. Note that several techniques perform blind rotation using different approaches: DM[19], CGGI[9, 10], and LMKCDEY[18]. The primary distinction between these techniques lies in the method used to homomorphically multiply the monomial X^u with the accumulator.

Table 1. Boolean Gate Operations and Mappings [19]. Note that in [19], the mapping ranges for the X(N)OR gate are set to $[q/8, 5q/8)$ and $[-3q/8, q/8)$. In this case, it must satisfy $|e| < \frac{q}{8}$; however, if the range is set to $[q/4, 3q/4)$ and $[-q/4, q/4)$, it only needs to satisfy $|e| < \frac{q}{4}$, making it more efficient.

Gate	Computation($\odot$)	maps to $\frac{q}{8}$	maps to $-\frac{q}{8}$
AND	$c_1 + c_2$	$[3q/8, 7q/8)$	$[-q/8, 3q/8)$
NAND	$c_1 + c_2$	$[-q/8, 3q/8)$	$[3q/8, 7q/8)$
OR	$c_1 + c_2$	$[q/8, 5q/8)$	$[-3q/8, q/8)$
NOR	$c_1 + c_2$	$[-3q/8, q/8)$	$[q/8, 5q/8)$
XOR	$2(c_1 - c_2)$	$[q/4, 3q/4)$	$[-q/4, q/4)$
XNOR	$2(c_1 - c_2)$	$[-q/4, q/4)$	$[q/4, 3q/4)$

LWE Extraction. The decrypted message, processed by homomorphic operations, is located in the constant term of the RLWE polynomial after blind rotation. The process of extracting this into LWE is called LWE extraction, and it can be performed without introducing additional noise.

Note that the function f used in the blind rotation step must satisfy $f(v + \frac{q}{2}) = -f(v)$. To meet this condition, we set the value of the function to $\frac{q}{8}$ and $-\frac{q}{8}$. After the blind rotation step and LWE extraction, to correct the scaling factor, we add $\frac{q}{8}$ to the LWE ciphertext, which is the output of LWE extraction.

RLWE′ and RGSW. The blind rotation generates significant noise because it involves homomorphic multiplication. Therefore, we use variants of RLWE, called RLWE′ and RGSW, which utilize gadget decomposition. Let $\boldsymbol{g} = (g_0, g_1, \ldots, g_{\ell-1}) \in \mathbb{Z}^\ell$ denote the gadget vector, and let $(d_0, \ldots, d_{\ell-1})$ be the gadget decomposition

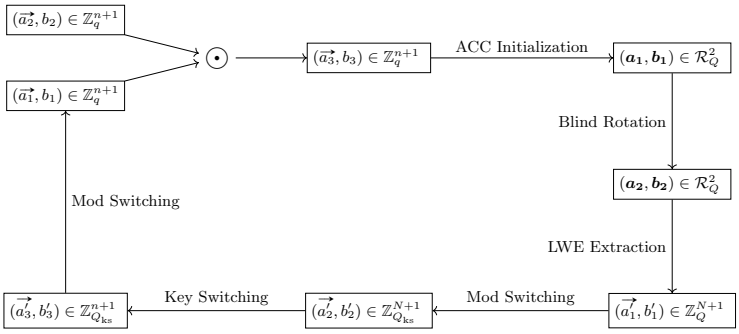


Fig. 1. This figure represents FHEW-like bootstrapping procedure. Note that this procedure is expressed in the same manner as described in [19]. In practice, after LWE extraction, performing the $\odot$ operation followed by mod switching and key switching can further reduce the noise. Here, the $\odot$ operation is a predefined operation based on the gate operation that needs to be performed. Detailed information on this can be found in Table 1.

of d, where $d = \sum_{i=0}^{\ell-1} d_i g_i$ with $B = \lceil Q^{1/\ell} \rceil$ representing the gadget length, $|d_i|_\infty < B$, and B being the base of the gadget decomposition, satisfying $B^\ell \geq Q$.

Now we define RLWE$'$ as follows:

$$\text{RLWE}'(\boldsymbol{m}) = (\text{RLWE}(g_0\boldsymbol{m}), \text{RLWE}(g_1\boldsymbol{m}), \cdots, \text{RLWE}(g_{\ell-1}\boldsymbol{m}));$$

and RGSW encryption of $\boldsymbol{m}$ is defined as follows:

$$\text{RGSW}(\boldsymbol{m}) = (\text{RLWE}'(-\boldsymbol{s} \cdot \boldsymbol{m}), \text{RLWE}'(\boldsymbol{m})).$$

The operations between RLWE and RGSW enable the multiplication of messages to be performed with relatively small noise [13]. We denote this operation as $\text{RLWE} \otimes \text{RGSW}$.

Key switching and Modulus switching. We represent FHEW-like bootstrapping in Figure 1. To understand the FHEW-like bootstrapping procedure, we define two new functions: **ModSwitch** and **KeySwitch**.

ModSwitch. **ModSwitch**, also known as modulus switching, is a function to switch modulus such that $\mathsf{modSwitch} : \mathbb{Z}_Q \to \mathbb{Z}_q$. We define the modulus switching function as $\mathsf{modSwitch}(x) = \left\lfloor \frac{q}{Q} \cdot x \right\rceil$, where $x \in \mathbb{Z}_q$. $\mathsf{modSwitch}$ is naturally expended to RLWE ciphertexts.

KeySwitch. **KeySwitch**, also known as the key switching algorithm, performs a private key change from $\text{LWE}_{\vec{s}'}$ to $\text{LWE}_{\vec{s}}$. To perform this algorithm, one requires a key switching key, denoted as $\mathbf{KSK}_{i,j,v}$, which is defined as $\mathbf{KSK}_{i,j,v} := \text{LWE}_{\vec{s}}(vs_i' B_{\text{ks}}^j)$, where $v \in [0, B_{\text{ks}})$, $i \in [0, N)$, and $j \in [0, d_{\text{ks}})$. As a result, let $(0, b)$ be an encryption under the secret key $\vec{s}'$. We compute $(0, b) -$

$\sum_{i,j} \mathbf{KSK}_{i,j,ai,j}$, after which we obtain an encryption under the secret key $\vec{s}$ with a small additional noise.

DM, CGGI, and LMKCDEY Blind Rotations. There are several differences in these methods performing blind rotation. These differences primarily concern how the monomial X^u is multiplied and the structure of the blind rotation key, brk, which is used for this multiplication with the accumulator. As a result, each technique varies in terms of computational complexity and key size.

DM blind rotation. The blind rotation key used in the DM method can be defined as follows:

$$\mathrm{brk}_{v,i,j} = \left\{ \mathrm{RGSW}\left(X^{vB_r^j s_i}\right) | v \in \mathbb{Z}_{B_r}, j \in [0,d), i \in [0,n) \right\},$$

where B_r is the base used in the decomposition of a_i. In other words, the blind rotation key of the DM method consists of all possible combinations of the decompositions of a_i and the secret key values s_i. The DM method updates the homomorphic accumulator ACC as $\mathrm{ACC} \leftarrow \mathrm{ACC} \otimes \mathrm{brk}_{a_i,i,j}$. It maintains the same computational complexity across different secret key distributions, including binary, ternary, and arbitrary integers.

CGGI blind rotation. The CGGI method uses the blind rotation key in the following form:

$$\mathrm{brk}_{j,u} = \left\{ \mathrm{RGSW}\left(x_{j,u}\right) | \vec{x_j} \in \{0,1\}^{|U|} \text{ such that } \sum_{u \in U} u \cdot x_{j,u} = s_j \right\},$$

where $U \subset \mathbb{Z}_q$. For example, we can define U for a ternary secret key as $U = 1, -1$ and $U = 1$ for a binary secret key. This blind rotation technique can be interpreted as the execution of a homomorphic CMUX operation:

$$\mathrm{ACC} \leftarrow \mathrm{ACC} + (X^{a_i} - 1)(\mathrm{ACC} \otimes \mathrm{brk}_{j,u}),$$

The CGGI method is affected by secret key distributions; it has low computational complexity when used with binary or ternary secret key distributions.

LMKCDEY blind rotation. The LMKCDEY blind rotation leverages automorphism to improve performance. We define the automorphism $\psi_k : \mathcal{R}_q \to \mathcal{R}_q$ as $\psi_k(m(X), \mathsf{ak}_k) = m(X^k)$, where $\mathsf{ak}_k = \mathrm{RLWE}'(-\boldsymbol{s}(X^k))$ is an automorphism key. Note that after applying the automorphism, key switching is required to revert the secret key from $\boldsymbol{s}(X^k)$ back to $\boldsymbol{s}$.

The keys used in blind rotation can be described as follows:

$$\left\{ \mathrm{brk}_i = \mathrm{RGSW}(s^i) | i \in [0,n) \right\}, \left\{ \mathsf{ak}_{-1}, \mathsf{ak}_j | j \in [0,w] \right\},$$

where w is a small integer. It has been demonstrated that this method can be performed efficiently with small constant number of automorphism keys. The blind rotation procedure can be represented as follows:

$$\mathrm{ACC} \leftarrow \psi_{a_i}(\psi_{a_i^{-1}}(\mathrm{ACC}, \mathsf{ak}_{a_i^{-1}}) \otimes \mathrm{brk}_i, \mathsf{ak}_{a_i}).$$

2.3 SHA3 and Keccak algorithm

SHA3 is the most recent member of the Secure Hash Algorithm family of standards by NIST in 2015. It is based on the Keccak algorithm [3], which employs a novel method called sponge construction. We briefly describe the structure of Keccak, and for more comprehensive details, we refer the readers to [3].

The function Keccak-f. The Keccak-f function is the core permutation function of the Keccak algorithm, which serves as the foundation of SHA3. It operates on a fixed-size state and applies a sequence of invertible transformations to produce a pseudo-random permutation. To enhance understanding, key terms describing the state in detail are illustrated in Figure 2.

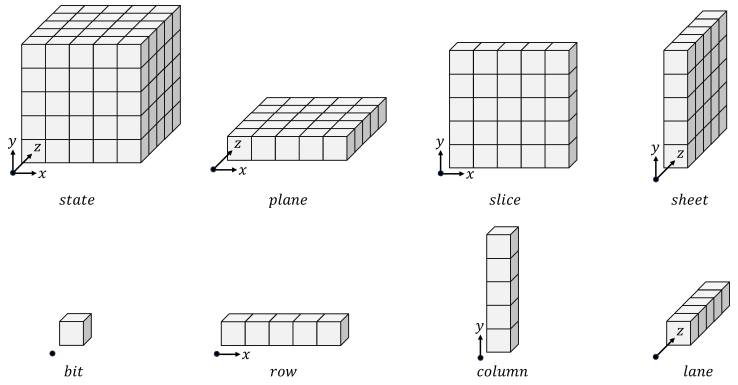


Fig. 2. This figure shows various labels that represent the state, with the x and y axes each having a size of 5x5, and the z axis having a size of 64, resulting in a total size of 1600 bits.

Let b represent the width of the state, where $b = 25 \cdot 2^l$ for some $l = 0, 1, \ldots, 6$. The state can be represented as a three-dimensional array $A[5][5][w]$, where $w = 2^l$ is the lane size. The Keccak-f permutation of b consists of $n_r = 12 + 2l$ rounds. Each round:

1. Takes an input consisting of $b = r + c$ bits, where r and c represent the *bit rate* and *capacity*, respectively.
2. Is composed of five steps: θ, ρ, π, χ, and ι.

Theta(θ) Step. The θ step aims to achieve bit diffusion across the entire state. It can be broken down into the following three steps:

– First, we compute $C[x]$ for $x = 0, 1, 2, 3, 4$:

$$C[x] = \bigoplus_{y=0}^{4} A[x, y],$$

where $A[x, y]$ refers to a w-bit lane in the state, and $\oplus$ denotes the bitwise XOR operation of two w-bit operands.
- Next, we compute $D[x]$ as follows:

$$D[x] = C[(x-1) \bmod 5] \oplus \mathrm{rot}(C[(x+1) \bmod 5], 1),$$

where $\mathrm{rot}(C[], 1)$ denotes a rotation of the operand by 1 bit along the z-axis.
- Finally, we apply $D[x]$ to update the array A:

$$A[x, y] = A[x, y] \oplus D[x].$$

Rho(ρ) and Pi(π) Step. The ρ and π steps work together to ensure diffusion:

- ρ: rotates each lane by a specific offset.
- π: rearranges the positions of lanes within the state.

These operations mix the bits across the entire state, enhancing the overall security of the hash function. This can be represented as a single operation:

$$B[y, (2x + 3y) \bmod 5] = \mathrm{rot}(A[x, y], r[x, y]),$$

where $x, y = 0, 1, 2, 3, 4$ and $r[x, y]$ is a predefined rotation constant for each lane.

Chi(χ) Step. The Chi step introduces non-linearity:

$$A[x, y] = B[x, y] \oplus ((\bar{B}[(x+1) \bmod 5, y]) \wedge B[(x+2) \bmod 5, y]),$$

where $\bar{B}[i, j]$ denotes the bitwise complement of the lane at address $[i, j]$, and $\wedge$ is the bitwise AND operation.

Iota(ι) Step. The ι step breaks symmetry by adding round constants:

$$A'[0, 0] = A[0, 0] \oplus \mathrm{RC}[i_r],$$

where $\mathrm{RC}[i_r]$ is the round constant for round i_r. Note that, since the round constant $\mathrm{RC}[i_r]$ is a known value, the XOR gate operation can be efficiently implemented using a NOT gate.

Sponge Construction. The Keccak hash function utilizes the sponge construction with the Keccak-f permutation. The process consists of two phases:

- **Absorbing phase:** The input message is padded and divided into r-bit blocks, which are then operated XOR into the first r bits of the state, interleaved with applications of Keccak-f.
- **Squeezing phase:** The first r bits of the state are output as blocks, interleaved with applications of Keccak-f, until the desired output length is reached.

The sponge construction is described as:

$$Z = \text{SPONGE}[h, \mathsf{pad}, r](M, d),$$

where h is the Keccak-f permutation, pad is the padding function, r is the bitrate, M is the input message, and d is the desired output length. The padding function pad appends a 1 bit to the message M, followed by as many 0 bits as necessary, and a final 1 bit to ensure the message length is a multiple of r.

3 Overlapped Bootstrapping

In this section, we propose a variant of blind rotation that effectively optimizes circuits with consecutive XOR gate operations. The proposed method exploits the scaling factor of the message in LWE encryption, while in previous works, the scaling factor Δ is rather a fixed value ($q/4$) to determine plaintext space. We take advantage of the fact that the mapping function f can be freely set during blind rotation without incurring additional cost.

3.1 New Blind Rotation Technique

In Table 1, we present the operations and mapping ranges that define gate operations in previous works [19, 20]. The operation to perform an XOR gate is defined as $c_0 \odot c_1 = 2(c_0 - c_1)$. This operation introduces larger noise because it involves multiplication by 2 after subtraction.

We focus on the fact that the output of the XOR gate operation is symmetric[3], and many algorithms, such as SHA3, are preplanned algorithms. In other words, this means that we can know in advance when the XOR gate operation will be executed.

For the symmetric gate operations , by slightly adjusting Δ to $q/2$, the original operation between input LWE ciphertexts, $c_0 \odot c_1 = 2(c_0 - c_1)$, can be modified to $c_0 + c_1$. Unfortunately, we need to revert Δ to $q/4$ for other binary gates, and most circuits utilize various gate operations. To address this, we define a mapping function f_Δ that allows for the free modification of the scaling factor, for example:

$$f_\Delta(m^*) = \begin{cases} \frac{\Delta}{2} & \text{if } \frac{3q}{8} \le m^* < \frac{7q}{8} \\ -\frac{\Delta}{2} & \text{if } -\frac{q}{8} \le m^* < \frac{3q}{8}. \end{cases}$$

The above equation is an example of a mapping function for AND gate. For the blind rotation for any binary gates prior to X(N)OR gate, we perform blind rotation with the mapping function $f_{q/2}$ to set Δ to $q/2$. Then, before a gate other than X(N)OR, we perform blind rotation with the mapping function $f_{q/4}$ to revert Δ to $q/4$. We refer to this technique as *flexible scaling*. Note that

[3] This means that if the number of ones is odd, the output is one; otherwise, the output is zero.

this process does not introduce any extra computational costs, as the mapping function can be freely set by the computing party during the blind rotation process.

Noise Analysis and Failure Probability. FHEW/TFHE will fail to decrypt if the noise does not satisfy $|e| < \frac{q}{8}$ or $|e| < \frac{q}{4}$, where the condition $|e| < \frac{q}{8}$ applies to AND and OR gate, and $|e| < \frac{q}{4}$ applies to the XOR gate. We can use the complementary error function (**erfc**) to calculate the expected failure probability, which is defined as $1 - \mathbf{erf}$, where **erf** denotes the error function. This probability can be expressed as $\mathbf{erfc}\left(\frac{q/8}{\sqrt{2}\sigma_{\text{total}}}\right)$ for AND and OR gates, and $\mathbf{erfc}\left(\frac{q/4}{\sqrt{2}\sigma_{\text{tot}}}\right)$ for the XOR gate. Here, σ_{tot} is the standard deviation of the noise in ciphertext after bootstrapping:

$$\sigma_{\text{tot}}^2 = 2\left(\frac{q^2}{Q_{\text{ks}}^2}\left(\frac{Q_{\text{ks}}^2}{Q^2}\sigma_{\text{ACC}}^2 + \sigma_{\text{MS}_1}^2 + \sigma_{\text{KS}}^2\right) + \sigma_{\text{MS}_2}^2\right),$$

where σ_{ACC}^2 is the noise variance of blind rotation, $\sigma_{\text{MS}_1}^2$ and $\sigma_{\text{MS}_2}^2$ are the noise variance of modulus switching, σ_{KS}^2 is the noise variance of key switching, and Q_{ks} is the key switching modulus. For details regarding noise, we refer to [19] and [13].

For XOR gate in previous works, since the operation is defined as $2(c_0 - c_1)$ using the conventional method, the total variance is defined as follows:

$$\sigma_{\text{tot-XOR}}^2 = 8\left(\frac{q^2}{Q_{\text{ks}}^2}\left(\frac{Q_{\text{ks}}^2}{Q^2}\sigma_{\text{ACC}}^2 + \sigma_{\text{MS}_1}^2 + \sigma_{\text{KS}}^2\right) + \sigma_{\text{MS}_2}^2\right).$$

Since 8 is multiplied, the variance itself is large, but as long as $|e| < \frac{q}{4}$ is satisfied, the failure probability is the same as for other gate operations.

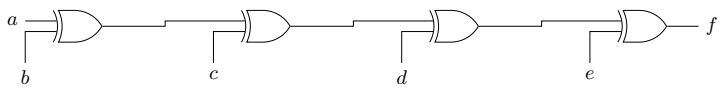
We propose to modify Δ to $q/2$, and then the $\odot$ is changed to $c_0 + c_1$, thus the noise variance is reduced to σ_{tot}^2. However, the failure condition is reduced to $|e| < \frac{q}{4}$ for X(N)OR, and thus has less failure probability.

We can delay the RLWE to LWE conversion, as suggested in [18], and can minimize noise as:

$$\sigma^{*2}{}_{\text{tot}} = \left(\frac{q^2}{Q_{\text{ks}}^2}\left(2\frac{Q_{\text{ks}}^2}{Q^2}\sigma_{\text{ACC}}^2 + \sigma_{\text{MS}_1}^2 + \sigma_{\text{KS}}^2\right) + \sigma_{\text{MS}_2}^2\right).$$

Overlapped Operations and Flexible Scaling. Previously, most HE libraries, including OpenFHE, used parameters with a manageable but not particularly convenient failure probability. However, Cheon et al. demonstrated in [6] that the failure probability, **FP**, must be negligibly small, and since then, the parameters have been revised.

In the homomorphic evaluation of cryptographic hash functions like SHA3, it is crucial to minimize the failure probability, as even a single bit of failure can trigger the avalanche effect, drastically altering the final result and leading to



$$\text{Prior} : f \longleftarrow \mathbf{FBS}(2(e - \mathbf{FBS}(2(d - \mathbf{FBS}(2(c - \mathbf{FBS}(2(a-b)))))))) \quad \text{where } \Delta = \frac{q}{4}$$

$$\text{Overlapped} : f \longleftarrow \mathbf{FBS}(a+b+c+d+e) \quad \text{where } \Delta = \frac{q}{2}$$

Fig. 3. This figure illustrates how sequential XOR gate operations are processed by the prior technique and by our proposed overlapped bootstrapping, respectively. Here, **FBS** refers to the bootstrapping function, which consumes a significant amount of computational resources. When performing four XOR gate operations, the prior technique executes the $2(c_1 - c_2)$ calculation followed by bootstrapping after each gate operation. In contrast, overlapped bootstrapping performs the calculation using $(c_1 + c_2)$, resulting in relatively less noise, allowing it to operate with only one bootstrapping.

unreliable outcomes. This is especially important when using parameters that have relatively high failure probabilities (e.g., 2^{-40}) in the past.

The proposed technique reduces the failure probability of the X(N)OR gate from $\mathbf{erfc}\left(\frac{q/4}{\sqrt{2}\sigma_{\text{tot-XOR}}}\right)$ to $\mathbf{erfc}\left(\frac{q/4}{\sqrt{2}\sigma_{\text{tot}}}\right)$ (or to $\mathbf{erfc}\left(\frac{q/4}{\sqrt{2}\sigma^*_{\text{tot}}}\right)$). When the failure probability is already sufficiently low, further reduction becomes unnecessary.

Interpreting this situation from another perspective, it means that since the failure probability has been reduced beyond what is necessary, there is no need to perform bootstrapping every time an XOR gate operation is executed. We exploit here that addition is equivalent to XOR when $\Delta = q/2$, thus functional bootstrapping is not required for gate evaluation, but only for noise reduction. The proposed technique secures more noise margin by modifying Δ, thus a functional bootstrapping can be performed in a lazy manner.

In particular, when XOR gates are performed consecutively, we first set $\Delta = q/2$. Instead of performing bootstrapping after each XOR gate operation, we replace a sequence of XOR gate operations with additions and perform bootstrapping in one step. We refer to this technique as *overlapped bootstrapping*. The overlapped bootstrapping procedure is illustrated in Figure 3. In this figure, *four nested additions followed by a single bootstrapping* with $\Delta = q/2$ replaces four individual bootstrapping operations with $\Delta = q/4$.

We apply flexible scaling in this process, although with a slight variation. Before performing an XOR gate, we adjust Δ to $q/2$. Before other gates, we use functional bootstrapping to scale $q/4$. While performing consecutive X(N)OR gates but if the noise meets a certain bound, we do functional bootstrapping with $q/2$ scaling to reduce the noise. We present a simple example of flexible scaling in Figure 4.

The XOR gate is simply replaced by an addition in overlapped bootstrapping. Hence, the variance noise introduced by the evaluation of XOR of n ciphertexts is given as follows:

$$\sigma^{*2}_{\text{tot};n} = \left(\frac{q^2}{Q^2_{\text{ks}}}\left(n\frac{Q^2_{\text{ks}}}{Q^2}\sigma^2_{\text{ACC}} + \sigma^2_{\text{MS}_1} + \sigma^2_{\text{KS}}\right) + \sigma^2_{\text{MS}_2}\right).$$

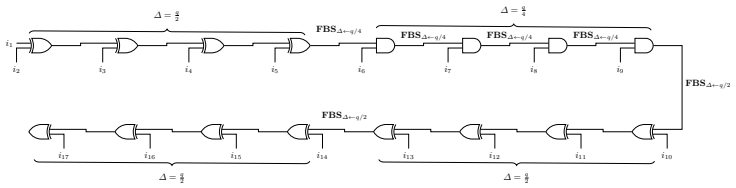


Fig. 4. This figure shows how the flexible scaling is performed. Here, where $\mathbf{FBS}_{\Delta\leftarrow\frac{q}{2}}$ (and $\mathbf{FBS}_{\Delta\leftarrow\frac{q}{4}}$) refers to bootstrapping that uses a mapping function to set Δ to $\frac{q}{2}$ (and $\frac{q}{4}$). Note that to use overlapped bootstrapping, Δ must be $\frac{q}{2}$ when performing the X(N)OR gate, and Δ must be $\frac{q}{4}$ when performing the other gates.

4 Implementation of Keccak with Overlapped Bootstrapping

In this section, we describe the implementation of the SHA3 algorithm (Keccak256, also known as SHAKE128) using FHEW/TFHE and discuss the computational speed benefits that can be achieved through overlapped bootstrapping. A detailed implementation of the SHA3 algorithm using FHEW/TFHE is provided in Appendix A.

Number of Nested Additions. One of the key factors that must first be determined to utilize the overlapped bootstrapping we propose is the number of additions to be performed in a nested manner. As presented in [6], the failure probability must be negligibly small for IND-CPA$^{\mathrm{D}}$ security. To safely use overlapped bootstrapping, it is essential to thoroughly understand the configuration of the system being used and to appropriately determine the number of nested additions.

In the case of SHA3, boolean gate operations are performed only in the θ, χ, and ι steps. However, the round constants used in the ι step are public values, and thus, the XOR operation can be replaced by a NOT operation, reducing the number of bootstrapping operations (detailed in Appendix A). Therefore, the steps that need practical consideration are the θ and χ steps. The θ step can be expressed by the following equations:

$$C[i] \longleftarrow H[i] \oplus H[i+1] \oplus H[i+2] \oplus H[i+3] \oplus H[i+4], \tag{1}$$

$$D[i] \longleftarrow C[i-1] \oplus \mathrm{rotate\_left}(C[i+1], 1),$$

$$H'[i] \longleftarrow \underbrace{H[i] \oplus D[i]}_{\mathbf{FBS}_{\Delta\leftarrow\frac{q}{4}}}. \tag{2}$$

The above process is performed independently for each lane. Assume H are fresh ciphertexts with scale $q/2$, then, if all the $\oplus$ operation is replaced with addition, the noise is maximal in (2) as variance ${\sigma^*}^2_{\mathrm{tot};11}$. Since an AND gate operation is planned to be performed in the χ step after the θ step, bootstrapping must be performed after the θ step (2) to adjust Δ to $q/4$.

Next, the χ step can be expressed by the following equations:

$$A[i] \longleftarrow H[i] \times 2, \tag{3}$$

$$H'[i] \longleftarrow A[i] \oplus (\underbrace{H[i+2] \wedge \overline{H[i+1]}}_{\mathbf{FBS}_{\Delta \leftarrow \frac{q}{2}}}) \tag{4}$$

$$H \longleftarrow \mathbf{FBS}_{\Delta \leftarrow \frac{q}{2}}(H'). \tag{5}$$

Note that the reason for multiplying state H by two in (3) is that Δ has to be set to $q/2$, in order to perform the XOR gate operation in (4). The AND gate operation in (4) should be done by $\mathbf{FPS}_{\Delta \leftarrow \frac{q}{2}}$ for the next XOR; then, the noise variance is given as ${\sigma^*}^2_{\text{tot};5}$. We perform $\mathbf{FBS}_{\Delta \leftarrow \frac{q}{2}}$ in (5) to reduce the noise before the next θ step, then the noise is maximal in (2) as ${\sigma^*}^2_{\text{tot};11}$, we refer this as $model_{11}$. For further reduction of runtime, $\mathbf{FBS}_{\Delta \leftarrow \frac{q}{2}}$ in (5) can also be ignored. Then, the noise in (2) is increase to ${\sigma^*}^2_{\text{tot};55}$, we call it $model_{55}$.

4.1 Implementation Results

We implemented Keccak256 and compared the cases where overlapped bootstrapping is used and not used, analyzing each step in detail. For reference, we compare performance based on cases where overlapped bootstrapping is used, particularly $model_{55}$. Note that $model_{55}$ already has a sufficiently low failure probability, so $model_{11}$ is not considered in this subsection.

Table 2. This table presents the details of **LPF_STD128**. **FP** represents the failure probability, with $\mathbf{FP}^*_{model_{11}}$ and $\mathbf{FP}^*_{model_{55}}$ denotes the failure probability of XOR gate with $model_{11}$ and $model_{55}$, respectively. Note that * refers to the failure probability in the case where the $\odot$ operation is performed after LWE extraction.

	n	q	N	$\log_2 Q$	$\log_2 Q_{ks}$	B_g	B_{ks}	B_r	**FP**	$\mathbf{FP}^*_{model_{11}}$	$\mathbf{FP}^*_{model_{55}}$
LPF_STD128	556	2048	1024	27	15	2^7	2^6	2^6	2^{-144}	2^{-483}	2^{-140}

Our implementation[4] uses OpenFHE library v1.2.0, and the parameters were set using the **LPF_STD128** provided by the OpenFHE. Details about **LPF_STD128** can be found in Table 2. Our evaluation environment was configured with an Intel(R) Core(TM) i9-14900K processor, 64GB RAM, and Ubuntu 22.04.2 LTS. The code was compiled with clang++ 14, using the CMake flags NATIVE_SIZE=32 for ciphertext modulo less than 31 bits, and WITH_OMP=OFF, which means single thread. We use DM blind rotation method for the bootstrapping [13], but other methods like CGGI [9] and LMKCDEY [18] can also be applied.

[4] https://github.com/HONGDUCK/SHA3-with-FHE

Table 3. Step-by-step performance comparison before and after applying overlapped bootstrapping, as well as before and after replacing XOR gate operations with NOT gate operations. For the θ and χ steps, the runtime represents the average time per execution, while for the ι step, it reflects the time required for a single round function. The ρ and π steps are excluded, as they involve only bit rotations.

	θ (avg.)	χ (avg.)	ι	Total
Benchmark	194.20s	186.37s	123.96s	9327.93s
$model_{55}$	108.43s	108.28s	9.66×10^{-5}s	5201.04s
(Improvements)	(44.2%)	(41.9%)	(-)	(44.2%)

Runtime per Steps The performance results for the θ and χ steps, both before and after applying overlapped bootstrapping, as well as the performance results before and after replacing the XOR gate operations with NOT gate operations in the ι step, can be found in Table 3.

It should be noted that the proof-of-concept implementation in Table 3 is relatively slow because we use the DM blind rotation method, which is known to be slower, and the experiment was conducted on a single-threaded CPU. Faster bootstrapping methods, such as CGGI and LMKCDEY [9, 18], as well as faster implementations using NTRU-based HE [4], would significantly improve the runtime. The proposed technique can be easily applied to these variants of the FHEW scheme. Additionally, leveraging GPU or other hardware acceleration techniques could further enhance performance [12].

Runtime per Rounds The runtime of the Keccak256 algorithm is proportional to the input data size. The number of rounds is determined by the input length, and this process cannot be parallelized. Therefore, the runtime of the Keccak256 can only be improved by efficiently designing each round function.

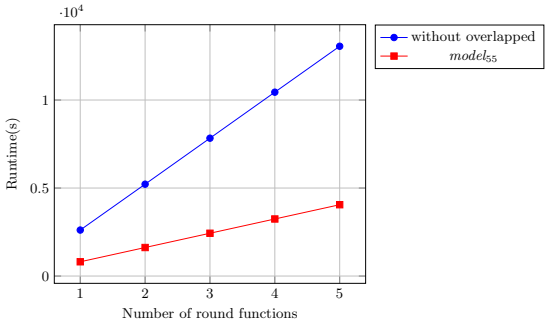


Fig. 5. This figure shows the runtime for processing large input data, with and without applying overlapped bootstrapping. A single round function consists of a total of 24 rounds, with each round performing θ, π, ρ, χ, and ι once.

Figure 5 shows the difference in runtime when processing large input data, with and without applying overlapped bootstrapping. This runtime was measured using 32 threads for multi-threading. When overlapped bootstrapping is applied, the runtime of a single round function decreases, making it more efficient than without overlapped bootstrapping when processing large input data.

5 Conclusion

We proposed a flexible scaling method to improve the runtime and reduce noise in symmetric gates such as XOR and XNOR. By utilizing this technique, we set the scaling factor Δ to $q/2$ for X(N)OR gate operations. For other gates, Δ is reverted to $q/4$ without introducing any overhead. Since most circuits use a combination of different gate operations, maintaining consistent scaling factors is essential for proper functionality. Fortunately, the computing party typically knows the structure of the circuit in advance, which is a fundamental assumption of HE compilers.

We also introduced overlapped bootstrapping as a method to enhance the runtime of the SHA3 algorithm over HE. This technique is effective not only for secure hash functions but also for scenarios involving consecutive XOR gate operations, as it reduces the number of bootstrapping operations required.

Many circuits, including symmetric key algorithms such as AES, rely on consecutive XOR gate operations. Even for a single XOR gate operation, using flexible scaling can significantly reduce the failure probability. By adopting this method, both runtime and failure probability across various applications can be optimized, providing valuable strategies for compiler design in HE systems.

Future work will focus on applying the flexible scaling technique and overlapped bootstrapping to other key functions in smart contracts, with the aim of improving the efficiency of processing confidential data in blockchain environments. Another interesting direction for future research is exploring how our flexible scaling technique can be applied in both batched bootstrapping and HE schemes that utilize the SIMD property [2, 8, 1]. This could enhance bootstrapping efficiency, particularly in low-depth and wide circuits with consecutive XOR gates.

References

1. Bae, Y., Cheon, J.H., Kim, J., Stehlé, D.: Bootstrapping bits with ckks. In: Joye, M., Leander, G. (eds.) Advances in Cryptology – EUROCRYPT 2024. pp. 94–123. Springer Nature Switzerland, Cham (2024)
2. Bae, Y., Kim, J., Stehlé, D., Suvanto, E.: Bootstrapping small integers with ckks. In: Chung, K.M., Sasaki, Y. (eds.) Advances in Cryptology – ASIACRYPT 2024. pp. 330–360. Springer Nature Singapore, Singapore (2025)
3. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology – EUROCRYPT 2013. pp. 313–314. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

4. Bonte, C., Iliashenko, I., Park, J., Pereira, H.V.L., Smart, N.P.: Final: Faster fhe instantiated with NTRU and LWE. In: Advances in Cryptology – ASIACRYPT 2022. pp. 188–215 (2022)
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012. pp. 309–325 (2012). https://doi.org/10.1145/2090236.2090262, https://doi.org/10.1145/2090236.2090262
6. Cheon, J.H., Choe, H., Passelègue, A., Stehlé, D., Suvanto, E.: Attacks against the INDCPA-D security of exact fhe schemes. In: Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, CCS 2024 (2024)
7. Cheon, J.H., Kang, M., Kim, T., Jung, J., Yeo, Y.: High-throughput deep convolutional neural networks on fully homomorphic encryption using channel-by-channel packing. Cryptology ePrint Archive, Paper 2023/632 (2023), https://eprint.iacr.org/2023/632
8. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology – ASIACRYPT 2017. pp. 409–437. Springer (2017)
9. Chillotti, I., Gama, N., Georgieva, M., Izabachene, M.: Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In: Advances in Cryptology – ASIACRYPT 2017. pp. 377–408. Springer (2017)
10. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: Fast fully homomorphic encryption over the torus. Journal of Cryptology pp. 34–91 (2020)
11. Dahl, M., Danjou, C., Demmler, D., Frederiksen, T., Ivanov, P., Joye, M., Rotaru, D., Smart, N., Thibault, L.T.: fhEVM: Confidential evm smart contracts using fully homomorphic encryption (2023), https://github.com/zama-ai/fhevm/raw/main/fhevm-whitepaper.pdf, white Paper
12. Dai, W., Sunar, B.: cuhe: A homomorphic encryption accelerator library. In: Pasalic, E., Knudsen, L.R. (eds.) Cryptography and Information Security in the Balkans. pp. 169–186. Springer International Publishing, Cham (2016)
13. Ducas, L., Micciancio, D.: FHEW: bootstrapping homomorphic encryption in less than a second. In: Advances in Cryptology - EUROCRYPT. pp. 617–640. Springer (2015)
14. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. p. 144 (2012)
15. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM Symposium on Theory of Computing. pp. 169–178 (2009)
16. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Advances in Cryptology – CRYPTO 2012. pp. 850–867. Springer (2012)
17. Lee, J.W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.S., et al.: Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. IEEE Access **10**, 30039–30054 (2022)
18. Lee, Y., Micciancio, D., Kim, A., Choi, R., Deryabin, M., Eom, J., Yoo, D.: Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In: Advances in Cryptology – EUROCRYPT 2023. pp. 227–256. Springer (2023)
19. Micciancio, D., Polyakov, Y.: Bootstrapping in FHEW-like cryptosystems. In: WAHC'21. pp. 17–28 (2021)
20. OpenFHE: Open-Source Fully Homomorphic Encryption Library. https://github.com/openfheorg/openfhe-development (2022)

21. Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL (Jan 2023), microsoft Research, Redmond, WA.
22. Trama, D., Clet, P., Boudguiga, A., Sirdey, R.: A homomorphic AES evaluation in less than 30 seconds by means of TFHE. In: Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 79–90 (2023)
23. Zama: TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data (2022), https://github.com/zama-ai/tfhe-rs

A Implementation of SHA3 Algorithm

This section describes the detailed implementation of the SHA3 algorithm using FHEW/TFHE. To aid understanding, the definition of the state used in the algorithm described later is provided in Figure 6.

13	14	10	11	12
8	9	5	6	7
3	4	0	1	2
23	24	20	21	22
18	19	15	16	17

Fig. 6. This figure represents the definition of a state. A single state consists of 25 lanes, and each lane is made up of 64 bits, in other words, 64 LWE ciphertexts. Each number within the 5×5 matrix indicates the position of each of the 25 lanes.

A.1 Basic Functions.

Note that the 1600-bit state used in Keccak256 is represented as a vector of LWE ciphertexts with a length of 1600. This allows the rotation operation included in the θ step, as well as the π and ρ steps, to be performed easily.

We have detailed the θ step in Algorithm 1. The laneXOR function mentioned in the algorithm takes two lanes as input and performs a bit-wise XOR operation. If overlapped bootstrapping is not used, it requires 64 bootstrapping operations to complete. When using overlapped bootstrapping, the number of bootstrapping operations can be reduced because bootstrapping is performed after a suitable number of additions. More details on this will be described later.

As previously mentioned, the state is implemented as a vector, making the π and ρ steps easy to implement. We have described the π step and ρ step in Algorithms 2 and 3, respectively. The rotate_left function within the algorithm takes a lane and an integer as inputs and performs a left shift by the amount specified by the input integer.

We have described the χ step in Algorithm 4. The laneAND function used here, similar to the laneXOR function, takes two lanes as input and performs a bit-wise AND operation. The same applies to laneNOT, which can be performed without bootstrapping.

As mentioned above, the round constants used in the ι step are well-known values. Therefore, the original XOR operation can be converted to a NOT operation. This reduces the number of bootstrapping operations. The ι step is detailed in Algorithm 5, and one of the inputs to the algorithm, the round constant position, indicates the location where the NOT gate operation is performed based

Algorithm 1: θ step

Input: Current state H
Output: New state H'

State C, D, H'
for $i = 0; i < 5; i = i + 1$ **do**

$$C[i] \longleftarrow \text{laneXOR}(H[i], \text{laneXOR}(H[i+5], \text{laneXOR}(H[i+10], \text{laneXOR}(H[i+15], H[i+20]))))$$

for $i = 0; i < 5; i = i + 1$ **do**

$$D[i] \longleftarrow \text{laneXOR}(C[(i-1) \mod 5], \text{rotate\_left}(C[(i+5) \mod 5], 1))$$

for $i = 0; i < 5; i = i + 1$ **do**

$$H'[i] \longleftarrow \text{laneXOR}(H[i], D[i])$$
$$H'[i+5] \longleftarrow \text{laneXOR}(H[i+5], D[i])$$
$$H'[i+10] \longleftarrow \text{laneXOR}(H[i+10], D[i])$$
$$H'[i+15] \longleftarrow \text{laneXOR}(H[i+15], D[i])$$
$$H'[i+20] \longleftarrow \text{laneXOR}(H[i+20], D[i])$$

return H'

on the round constant. The round constant position can be found in detail in Table 4.

Algorithm 2: π step

Input: Current state H
Output: New state H'

State H'
$H'[01] \longleftarrow H[06], H'[06] \longleftarrow H[09], H'[09] \longleftarrow H[22], H'[22] \longleftarrow H[14]$
$H'[14] \longleftarrow H[20], H'[20] \longleftarrow H[02], H'[02] \longleftarrow H[12], H'[12] \longleftarrow H[13]$
$H'[13] \longleftarrow H[19], H'[19] \longleftarrow H[23], H'[23] \longleftarrow H[15], H'[15] \longleftarrow H[04]$
$H'[04] \longleftarrow H[24], H'[24] \longleftarrow H[21], H'[21] \longleftarrow H[08], H'[08] \longleftarrow H[16]$
$H'[16] \longleftarrow H[05], H'[05] \longleftarrow H[03], H'[03] \longleftarrow H[18], H'[18] \longleftarrow H[17]$
$H'[17] \longleftarrow H[11], H'[11] \longleftarrow H[07], H'[07] \longleftarrow H[10], H'[10] \longleftarrow H[01]$
return H'

Algorithm 3: ρ step

Input: Current state H
Output: New state H'

State H'
$H'[01] \longleftarrow \text{rotate\_left}(H[01], 01),\ H'[02] \longleftarrow \text{rotate\_left}(H[02], 62)$
$H'[03] \longleftarrow \text{rotate\_left}(H[03], 28),\ H'[04] \longleftarrow \text{rotate\_left}(H[04], 27)$
$H'[05] \longleftarrow \text{rotate\_left}(H[05], 36),\ H'[06] \longleftarrow \text{rotate\_left}(H[06], 44)$
$H'[07] \longleftarrow \text{rotate\_left}(H[07], 06),\ H'[08] \longleftarrow \text{rotate\_left}(H[08], 55)$
$H'[09] \longleftarrow \text{rotate\_left}(H[09], 20),\ H'[10] \longleftarrow \text{rotate\_left}(H[10], 03)$
$H'[11] \longleftarrow \text{rotate\_left}(H[11], 10),\ H'[12] \longleftarrow \text{rotate\_left}(H[12], 43)$
$H'[13] \longleftarrow \text{rotate\_left}(H[13], 25),\ H'[14] \longleftarrow \text{rotate\_left}(H[14], 39)$
$H'[15] \longleftarrow \text{rotate\_left}(H[15], 41),\ H'[16] \longleftarrow \text{rotate\_left}(H[16], 45)$
$H'[17] \longleftarrow \text{rotate\_left}(H[17], 15),\ H'[18] \longleftarrow \text{rotate\_left}(H[18], 21)$
$H'[19] \longleftarrow \text{rotate\_left}(H[19], 08),\ H'[20] \longleftarrow \text{rotate\_left}(H[20], 18)$
$H'[21] \longleftarrow \text{rotate\_left}(H[21], 02),\ H'[22] \longleftarrow \text{rotate\_left}(H[22], 61)$
$H'[23] \longleftarrow \text{rotate\_left}(H[23], 56),\ H'[24] \longleftarrow \text{rotate\_left}(H[24], 14)$
return H'

Algorithm 4: χ step

Input: Current state H
Output: New state H'

Lane a_0, a_1, a_2, a_3, a_4
Lane A_0, A_1
State H'
for $i = 0; i < 25; i = i + 5$ **do**
  $a_0 \longleftarrow H[i+0], a_1 \longleftarrow H[i+1]$
  $a_2 \longleftarrow H[i+2], a_3 \longleftarrow H[i+3]$, $a_4 \longleftarrow H[i+4]$
  $A_0 \longleftarrow H[i+0], A_1 \longleftarrow H[i+1]$
  $H'[0+i] \longleftarrow \text{laneXOR}(a_0, \text{laneAND}(H[2+i], \text{laneNOT}(A_1)))$
  $H'[1+i] \longleftarrow \text{laneXOR}(a_1, \text{laneAND}(H[3+i], \text{laneNOT}(H[i+2])))$
  $H'[2+i] \longleftarrow \text{laneXOR}(a_2, \text{laneAND}(H[4+i], \text{laneNOT}(H[i+3])))$
  $H'[3+i] \longleftarrow \text{laneXOR}(a_3, \text{laneAND}(A_0, \text{laneNOT}(H[i+4])))$
  $H'[4+i] \longleftarrow \text{laneXOR}(a_4, \text{laneAND}(A_1, \text{laneNOT}(A_0)))$
return H'

Algorithm 5: ι step

Input: Current state H, Round constant position RC_p, round r
Output: New state H'

for $i = 0; i < RC_p.size(); i = i + 1$ **do**
  $H'[0][RC_p[r][i]] \longleftarrow \text{NOT}(H[0][RC_p[r][i]])$
return H'

Table 4. Round constant and NOT operation positions

RC[0]	0x0000000000000001	63	RC[12]	0x000000008000808b	32, 48, 56, 60, 62, 63
RC[1]	0x0000000000008082	48, 56, 62	RC[13]	0x800000000000008b	0, 56, 60, 62, 63
RC[2]	0x800000000000808a	0, 48, 56, 60, 62	RC[14]	0x8000000000008089	0, 48, 56, 60, 63
RC[3]	0x8000000080008000	0, 32, 48	RC[15]	0x8000000000008003	0, 48, 62, 63
RC[4]	0x000000000000808b	48, 56, 60, 62, 63	RC[16]	0x8000000000008002	0, 48, 62
RC[5]	0x0000000080000001	32, 63	RC[17]	0x8000000000000080	0, 56
RC[6]	0x8000000080008081	0, 32, 48, 56, 63	RC[18]	0x000000000000800a	48, 60, 62
RC[7]	0x8000000000008009	0, 48, 60, 63	RC[19]	0x800000008000000a	0, 32, 60, 62
RC[8]	0x000000000000008a	56, 60, 62	RC[20]	0x8000000080008081	0, 32, 48, 56, 63
RC[9]	0x0000000000000088	56, 60	RC[21]	0x8000000000008080	0, 48, 56
RC[10]	0x0000000080008009	32, 48, 60, 63	RC[22]	0x0000000080000001	32, 63
RC[11]	0x000000008000000a	32, 60, 62	RC[23]	0x8000000080008008	0, 32, 48, 60