IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 3, pp. 530–573. DOI:10.46586/tches.v2024.i3.530-573

Thunderbird: Efficient Homomorphic Evaluation of Symmetric Ciphers in 3GPP by combining two modes of TFHE

Benqiang Wei[1,2], Xianhui Lu[1,2(✉)], Ruida Wang[1,2], Kun Liu[1,2], Zhihao Li[1,2] and Kunpeng Wang[1,2]

[1] Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
{weibenqiang,luxianhui}@iie.ac.cn

Abstract. Hybrid homomorphic encryption (a.k.a., transciphering) can alleviate the ciphertext size expansion inherent to fully homomorphic encryption by integrating a specific symmetric encryption scheme, which requires selected symmetric encryption scheme that can be efficiently evaluated homomorphically. While there has been a recent surge in the development of FHE-friendly ciphers, concerns have arisen regarding their security. A significant challenge for the transciphering community remains the efficient evaluation of symmetric encryption algorithms that have undergone extensive study and standardization.

In this paper, we present an evaluation framework, dubbed Thunderbird, which for the first time presents efficient homomorphic implementations of stream ciphers SNOW 3G and ZUC that are standardized in the 3G Partnership Project (3GPP). Specifically, Thunderbird combines gate bootstrapping mode and leveled evaluation mode of TFHE to cater to various function types within symmetric encryption algorithms. In the gate bootstrapping mode, we propose a variant of the homomorphic full adder that consumes only a single blind rotation, which may be of independent interest. In the leveled evaluation mode, we employ the CMux gate combining with hybrid packing technique to efficiently achieve lookup tables, significantly reducing the need for gate bootstrapping, and adapt the current optimal circuit bootstrapping to expedite the Thunderbird framework. We have implemented the Thunderbird framework in the TFHEpp public library. Experimental results demonstrate that SNOW 3G and ZUC can homomorphically generate a keyword in only 7 seconds and 9.5 seconds, which are *52*× and *32*× faster than the trivial gate bootstrapping mode, respectively. For the homomorphic evaluation of the AES-128 algorithm using Thunderbird, we achieve a speedup of *1.9*× in terms of latency and use less evaluation key compared to the state-of-the-art work.

Keywords: Hybrid homomorphic encryption · TFHE · SNOW 3G · ZUC · AES · Standardized Cipher

1 Introduction

Fully Homomorphic Encryption (FHE) is a cryptographic technique with the remarkable ability to perform computations on encrypted data without requiring decryption. This property makes it useful in privacy-preserving applications such as cloud computing, medical diagnostics and financial analytics. However, current fully homomorphic encryption suffers from two serious drawbacks: first, the application of FHE results in a severe efficiency penalty. Secondly, existing FHE schemes suffer from ciphertext size expansion, where

Received: 2024-01-15 Accepted: 2024-03-15 Published: 2024-07-18

the ciphertext size is several orders of magnitude larger than the corresponding plaintext size [DGH+23]. This expansion is extremely unfriendly to embedded devices with limited bandwidth, memory, and computational power. There are two distinct approaches to address the issue of ciphertext expansion: one is to utilize hybrid homomorphic encryption (HHE) [NLV11], and the other is to leverage LWE encryption combined with efficient LWEs-to-RLWE ciphertext conversion [CDKS21, BCK+23]. In this paper, we concentrate on the HHE that achieves the optimal ciphertext expansion ratio.

1.1 Hybrid Homomorphic Encryption

Hybrid homomorphic encryption, also known as transciphering, was initially proposed by Naehrig et al. [NLV11] to reduce the transmission cost between the client and the cloud by combining FHE and symmetric encryption algorithms. Specifically, instead of directly using FHE to encrypt the data, the client first encrypts the plaintext m using a symmetric encryption (SYM) scheme and then sends the ciphertext $\mathrm{SYM}(m)$ to the cloud server. The cloud server performs homomorphic evaluation on the decryption circuit of SYM and converts $\mathrm{SYM}(m)$ to homomorphic ciphertext $\mathrm{FHE}(m)$, enabling the execution of homomorphic function computation. As a result, the ciphertext expansion ratio can be reduced to 1. Note that the data to be sent also includes the evaluation key, but its size is always much smaller than the homomorphic ciphertext size of m.

Selection of FHE Scheme for HHE Since Gentry's pioneering work in 2009 [Gen09], various practical FHE schemes and their implementations have been proposed, notable schemes include BGV [BGV14], BFV [Bra12, FV12], CKKS [CKKS17] and TFHE [CGGI20]. The BGV/BFV scheme supports leveled homomorphic evaluation and Single Instruction Multiple Data (SIMD) operation, but their parameter sets must be chosen according to the multiplication depth of the target application. In general, bootstrapping, which is used to refresh the noise inside the ciphertext to support further computation, is not recommended to use due to the high computational cost. The CKKS scheme specializes in enabling approximate homomorphic computation of real numbers. The TFHE scheme stands out by offering efficient gate bootstrapping and functional bootstrapping, which allows for the computation of arbitrary functions over finite precision while refreshing the noise. Unlike the BGV/BFV/CKKS schemes, it is free from the circuit multiplication depth limitation and is now widely favored for various applications [CJP21, TCBS23]. There is no doubt that the selection of FHE scheme has a substantial impact on the operational efficiency of the hybrid homomorphic encryption framework. When considered globally, the efficiency performance of different FHE schemes for a given real-world computational task varies significantly.

Selection of Symmetric Encryption Scheme for HHE The HHE execution framework can reduce the transmission bandwidth at the cost of increasing the computational workload of the server compared with the traditional FHE framework. To be more specific, the server is required to homomorphically evaluate the decryption circuit of the symmetric encryption scheme used, this would lead to a non-negligible additional computational overhead on the server side. Consequently, the selection of the symmetric encryption scheme becomes a crucial consideration when applying the HHE framework.

At the beginning, researchers mainly focused on the evaluation of standardized symmetric encryption algorithms, such as the NIST standardized block cipher AES. Various evaluation methods for AES based on different FHE schemes [GHS12, CLT14, DHS16, CHK19], have been continuously proposed. However, AES was not considered suitable for the transciphering framework due to the long evaluation latency caused by the high multiplication depth. Subsequently, some researchers shifted their attention towards evaluating lightweight

ciphers, such as SIMON [LN14] and Prince [DSES14]. In the context of applying hybrid homomorphic encryption, stream ciphers present an appealing choice over block ciphers. This preference arises from the fact that when using stream ciphers, the generation of the keystream is independent of the data to be encrypted or decrypted and the homomorphic keystream can be generated in advance *offline*. Subsequently, when the cloud receives symmetrically encrypted data, it can conveniently perform the transciphering operation by homomorphically XORing it with the pre-computed homomorphic encryption key *online*. Canteaut et al. [CCF+16] proposed the first HHE framework based on stream ciphers and evaluated Trivium using the BGV scheme, which belongs to the eSTREAM portfolio. Bendoukha et al. [BBS21] evaluated Grain128-AEAD, a NIST finalist for lightweight cryptography, based on TFHE. Another line of research on HHE is to design FHE-friendly symmetric encryption ciphers characterized by lower multiplication complexity and multiplication depth. Block cipher LowMC was first designed for MPC and FHE scenarios by Albrecht et al. [ARS+15]. Since then a large number of FHE-friendly ciphers continue to be proposed such as FLIP [MJSC16], FiLIP [MCJS19, HMR20], Elisabeth [CHMS22], Rasta [DEG+18], Masta [HKC+20], Dasta [HL20], Fasta [CIR22] and Pasta [DGH+23].

Indeed, these newly designed ciphers bring substantial improvements to the efficiency of the HHE framework. However, they also pose a distinctive challenge when it comes to security analysis. This challenge primarily stems from the fact that these ciphers often incorporate novel mathematical structures and components. As a result, the time required for thorough security analysis can be extensive, and the path to their deployment in industrial applications or potential standardization remains uncertain. Furthermore, some of these newly developed ciphers have faced security attacks, raising concerns about their reliability. The original design of FLIP was broken by algebraic attack [DLR16]. Liu et al. [LSMI21] proposed an algebraic attack on Rasta and Dasta with a reduced number of rounds. They used linearization techniques to exploit the low-degree expressions of Rasta's inverse nonlinear layer for their attack. Chaghri [AMT22], a new block cipher with lower multiplication depth than AES, was quickly attacked by [LAW+23]. Recently, Rubato [HKL+22] over $\mathbb{Z}_q$ was also found to be insecure [GAH+23] when q is a non-prime number. Gilbert et al. [GBJR23] presented several variants of key-recovery attack on Elisabeth-4 with 128-bit security, which are time-memory tradeoffs. And an algebraic attack on CKKS-friendly cipher HERA [CHK+21] using multiple collisions was proposed by [LKSM23]. Radheshwar et al. [RKMR23] studied differential fault attack against Rasta and FiLIP ciphers. Most of these passwords have been patched afterward, but there is a long way to go to standardize them.

Considering the security challenges associated with newly designed ciphers in the context of HHE, it is a judicious approach to prioritize the efficient evaluation of symmetric cipher algorithms that have already undergone rigorous study and standardization. The 3G Partnership Project (3GPP) has introduced widely adopted confidentiality and integrity algorithms for LTE mobile networks, which include 128-EEA1 and 128-EIA1 based on SNOW 3G, 128-EEA2 and 128-EIA2 based on AES, and 128-EEA3 and 128-EIA3 based on ZUC. It's worth noting that SNOW 3G[1], AES and ZUC have also been selected as the core encryption algorithms for the 5G communication system. However, to the best of our knowledge, except for the AES algorithm, there are no publicly available homomorphic evaluations and related benchmarks for the stream ciphers SNOW 3G and ZUC. Therefore, it is evidently both necessary and urgent to delve into the application of SNOW 3G and ZUC within HHE. Such research endeavors could potentially pave the way for their integration into the HHE framework.

[1]Although there are some attacks on SNOW 3G, such as distinguishing and correlation attacks in [YJM19, GHW23], the complexity of their attacks are far beyond the 128-bit security claimed by SNOW 3G. Only when the key length of SNOW 3G is increased to 256 bits, this result can be considered as an academic attack. Therefore, SNOW 3G is still secure.

1.2 Contributions and Techniques

In this paper, we focus on the efficient evaluation of standardized symmetric ciphers in 3GPP and explore the prospects of their application in the HHE framework. Our contributions and techniques are summarized as follows.

- We present an evaluation framework, dubbed Thunderbird, which combines the existing gate bootstrapping and leveled homomorphic evaluation modes of TFHE, leading to a significant efficiency improvement for the homomorphic evaluation of the stream cipher SNOW 3G and ZUC. In the gate bootstrapping mode, for the addition modulo 2^{32} non-linear function, we propose a variant of the full adder that consumes only one blind rotation operation, which can be of independent interest. To compute Sbox nonlinear functions efficiently, we customize the CMux-based evaluation by combining hybrid packing technique. Particularly, for SNOW 3G, we dramatically reduce the number of gate bootstrapping used by utilizing 8-to-32-bit lookup tables.
- Secondly, circuit bootstrapping (TLWE-to-TRGSW) serves as a crucial bridge within the Thunderbird framework. We conduct a comprehensive analysis and provide an overview of the current state-of-the-art techniques in circuit bootstrapping . Then, we adapt the optimal circuit bootstrapping to accelerate the Thunderbird framework. Our tuning is 1.6× faster than the current publicly available circuit bootstrapping implementation.
- Moreover, we demonstrate the versatility of Thunderbird by extending it to the homomorphic evaluation of the block cipher AES. Specifically, instead of following standard AES implementation, we utilize a LUT-based AES implementation that merges SubBytes, ShiftRows and MixColumns operations into 8-to-32-bit tables, thus enhancing its compatibility with Thunderbird. The advantage of our LUT-based AES evaluation over the state-of-the-art is its reduced consumption of XOR gates and smaller evaluation key size.
- Finally, we implement the Thunderbird framework in the TFHEpp public homomorphic encryption library. The experimental results demonstrate that Thunderbird-based implementations yield impressive performance gains compared to gate bootstrapping evaluation mode. For SNOW 3G, generating a keyword takes only 7 seconds, which is 52 times faster than the gate bootstrapping mode, while for ZUC, it takes only 9.5 seconds, a 32-fold improvement over the gate bootstrapping mode. Regarding AES, when Thunderbird is integrated with the homomorphic gate HomoXOR, the latency for one block takes 110 seconds. Additionally, when Thunderbird is combined with the freeXOR gate, the latency of an AES block is further reduced to 46 seconds, which is about 2 times faster than current state-of-the-art implementation.

1.3 Related Work

In Section 1.3.1, we briefly describe the some works about how to improve the evaluation latency of existing ciphers that have been standardized and well-studied. And we shortly discuss the design and performance of FHE-friendly symmetric encryption algorithms with low multiplicative complexity in Section 1.3.2. In Section 1.3.3, we describe an alternative method used to reduce transmission bandwidth for FHE applications without symmetric encryption.

1.3.1 Transciphering Performance using Standardized Cipher

AES stands as a standardized and extensively researched symmetric encryption algorithm, having evolved into one of the most widely used algorithms. Now it serves as a crucial

benchmark in both secure multiparty computing and fully homomorphic encryption research. Gentry et al. [GHS12] introduced the initial homomorphic evaluation based on the BGV scheme, reporting latencies of 4 minutes and 18 minutes without and with bootstrapping, respectively. In other words, each AES block took approximately 6s and 2s under amortization. Recent advancements by Trama et al. [TCBS23] showcased an AES block evaluation time of only 4.2 minutes using functional bootstrapping under a single thread. With the integration of multi-threading, they asserted that homomorphic evaluation of an AES block could be accomplished in a mere 28 seconds. Wei et al. [WWL+23] presented a reduced evaluation latency of AES to 86 seconds under a single thread, employing hybrid packing and circuit bootstrapping techniques. Homomorphic evaluation of AES based on the CKKS scheme was proposed for the first time in [ADE+23] and the authors further demonstrated the advantages of the transciphering framework for deep neural network evaluation, achieving an evaluation latency of 31 minutes using multiple cores under CPU, with an amortization time of 56.7 milliseconds. For the standardized Trivium, which belongs to the eSTREAM portfolio, was initial evaluated by [CCF+16] and further improved by Balenbois et al. [BOS23]. Utilizing functional bootstrapping and parallelization, they achieved a latency of about 1.89 milliseconds per bit using 128 virtual CPUs. Additionally, Grain128a, a NIST lightweight cryptography finalist, garnered attention in the work of Bendoukha [BBS21], who provided a homomorphic evaluation using the TFHE homomorphic compiler. Further, Bendoukha et al. [BCBS23] reduced the running time of warmup circuit of Grain128-AEAD to 3.98 minutes by means of functional bootstrapping,

1.3.2 The Design and Performance of FHE-friendly Ciphers for Transciphering

Since the previous standardized symmetric encryption algorithms were designed with security and efficient implementation in mind, they are not customized for FHE. As a result, the excessive multiplication depths lead them to be not FHE-friendly. The design of FHE-friendly symmetric algorithms has received a lot of attention, and symmetric encryption algorithms with lower multiplication depths and multiplication complexities have been continuously proposed. FLIP stream cipher [MJSC16] combines the advantages of block cipher and stream cipher to make the keystream have constant and small noise during homomorphic evaluation. For the improved FLIP cipher FiLIP over $\mathbb{Z}_2$, initially proposed by Méaux et al. [MCJS19], in [CDPP22] Cong et al. presented the fastest evaluation time of only 2.6ms per bit. For the $\mathbb{Z}_{2^q}$ cipher Elisabeth-4 [CHMS22], a stream cipher tailored for TFHE's functional bootstrapping, an amortization of about 22.8ms per bit is achieved under parallel acceleration. The pursuit of minimizing AND depth and the number of ANDs required per encrypted bit originated with Rasta [DEG+18]. Subsequent studies, including Masta, Dasta, and Fasta, aimed to improve upon Rasta. Recently, $\mathbb{F}_p$ -type cipher Pasta [DGH+23], was designed for BGV/BFV scheme, and their starting points come from concrete application cases, such as matrix-vector multiplication. For the Pasta-3, they gave 128*16 bits transciphering in 9.28s based on the SEAL library [SEA23], equating to approximately 4.5ms per bit.

In Asiacrypt 2021 [CHK+21], the authors presented a Real-to-Finite-field (RtF) hybrid framework to support the CKKS scheme by combining the BFV scheme and proposed the FHE-friendly stream cipher HERA. In Eurocrypt 2022, Ha et al. [HKL+22] proposed a faster Rubato cipher suitable for the RtF framework. HERA and Rubato ciphers, specifically tailored for the CKKS scheme, achieved throughput of 25μs and 18μs per bit, respectively.

1.3.3 Alternative Approach

In addition to utilizing hybrid homomorphic encryption, Chen et al. [CDKS21] proposed an alternative method that can effectively reduce the transmission bandwidth of the client. Briefly, the client encrypts the message using LWE-based symmetric encryption scheme. After receiving the LWE ciphertexts, the cloud converts them to RLWE ciphertext using the LWEs-to-RLWE algorithm and then performs the homomorphic function computation. For all LWE ciphertexts $(\mathbf{a}_i, b_i)$, the random vector part of the i-th LWE ciphertext $\mathbf{a}_i$ can be obtained by a pseudo-random number generator $f : \{0,1\}^* \to \mathbb{Z}_q^N$, with input seed se and index i, *i.e.*, $\mathbf{a}_i = f(se; i)$.

As a result, the communication cost per LWE ciphertext is only $\log q$ bits. The cost of this approach is transferred to the server-side LWEs-to-RLWE ciphertext conversion. Chen et al proposed an efficient method to pack many LWE ciphertexts into one RLWE ciphertext by evaluating trace function. Recently, Bae et al. [BCK+23] proposed more efficient ring packing using MLWE ciphertexts, which significantly outperforms the HERA and Rubato proposals when applied to the conversion of LWEs to CKKS ciphertext.

1.4 Paper organization

The paper is organized as follows. Section 2 gives some preliminaries about the TFHE scheme used in this paper. In Section 3, we first discuss about the evaluation strategy of the standardized algorithm SNOW 3G. Section 4 describes about the optimization of circuit bootstrapping and presents our evaluation framework. Based on our framework, we give efficient evaluation of ZUC and AES in Section 5 and Section 6. Section 7 shows the specific implementation and experimental results. Section 8 concludes this paper.

2 Preliminaries

In this section, we will review and explain some basic concepts throughout this work, especially the building blocks of the TFHE scheme.

2.1 Notations

We denote the security parameter by λ. The Real Torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ is the set of real numbers modulo 1. Note that we define the interval of $\mathbb{T}$ to be $[-1/2, 1/2]$. Let $\mathbb{B} = \mathbb{Z}_2$. Let $\mathbb{A}$ be a set, we denote by $\mathbb{A}_q^n$ the set of vectors with n elements in $\mathbb{A}$ modulo q, and by $\mathbb{A}_N[X]^n$ the set of n vectors with polynomials modulo $(X^N + 1)$, where N is a power of 2. Namely, $\mathbb{Z}_N[X] = \mathbb{Z}[X]/(X^N + 1)$ is integer polynomials modulo $X^N + 1$, $\mathbb{B}_N[X]$ means the polynomials in $\mathbb{Z}_N[X]$ with binary coefficients, and $\mathbb{T}_N[X]$ is the polynomials with coefficients in $\mathbb{T}$.

We use lower-case bold letters and upper-case bold letters to represent the vectors and matrices, respectively. For two vectors $\mathbf{a}$ and $\mathbf{b}$, we denote their inner product by $\langle \mathbf{a}, \mathbf{b} \rangle$. $M_{m,n}(E)$ is $m \times n$ -size matrix with entries in E. Ring elements are indicated using lower-case letters, e.g., $a \in \mathbb{A}_N[X]$. The notation a_i refers to the i-th coefficient of polynomial $a(x)$. We use $x \leftarrow D$ to represent that x itself or coefficients are sampled from the distribution D. We denote by $\boxplus$ the integer modular addition mod 2^{32} and by $\oplus$ the bitwise exclusive OR.

2.2 The TFHE Cryptosystem

The TFHE scheme was proposed by Chillotti et al. [CGGI20] which is built on top of the FHEW scheme [DM15]. There are three different ciphertext types used for efficient bootstrapping of TFHE: TLWE, TRLWE and TRGSW. It is worth emphasizing that

these three ciphertext types can be transformed into each other to cope with different computational tasks. Next, we briefly introduce them as follows.

- **TLWE.** TLWE is the Torus version of the learning with errors (LWE) problem [Reg09], which can be expressed as $(\mathbf{a}, b) \in \mathbb{T}^{n+1}$. Specifically, $b = \langle \mathbf{a}, \mathbf{s} \rangle + m + e$, where $\mathbf{s} \leftarrow \mathbb{B}^n$ is the secret key, $m \in \mathbb{T}$ is the encoded plaintext message and the error e is drawn from a Gaussian distribution with mean 0 and standard deviation σ.

- **TRLWE.** TRLWE is the Torus version of the ring-LWE problem [LPR10] and can be represented as $(a(x), b(x)) \in \mathbb{T}_N[X]^2$. To be specific, $b(x) = a(x)s(x) + m(x) + e(x)$, where $s(x) \in \mathbb{B}_N[X]$ is the secret key, $m(x) \in \mathbb{T}_N[X]$ is the plaintext message polynomial and the error $e(x) \in \mathbb{T}_N[X]$ is a polynomial with random coefficients $e_i \in \mathbb{T}$ sampled from a Gaussian distribution with mean 0 and standard deviation σ.

- **TRGSW.** TRGSW can be seen as a vector composed of TRLWE, *i.e.*, $M_{2\ell,2}(\mathbb{T}_N[X])$. In detail, TRGSW encrypts the message $m \in \mathbb{B}$ into C as follows:

$$C = \begin{pmatrix} a_1(x) & b_1(x) \\ a_2(x) & b_2(x) \\ \vdots & \vdots \\ a_\ell(x) & b_\ell(x) \\ a_{\ell+1}(x) & b_{\ell+1}(x) \\ \vdots & \vdots \\ a_{2\ell}(x) & b_{2\ell}(x) \end{pmatrix} + m \cdot \begin{pmatrix} 1/B_g & 0 \\ 1/B_g^2 & 0 \\ \vdots & \vdots \\ 1/B_g^\ell & 0 \\ 0 & 1/B_g \\ \vdots & \vdots \\ 0 & 1/B_g^\ell \end{pmatrix}$$

  where $(a_i(x), b_i(x))$, for $1 \leq i \leq 2\ell$ is are TRLWE ciphertexts encrypting 0 using the same secret key, B_g denotes the basis of gadget decomposition and ℓ is the length of gadget decomposition. TRGSW ciphertext is used as bootstrapping key in TFHE.

2.3 Building Blocks in TFHE

In this subsection, we will briefly introduce the building blocks in TFHE bootstrapping, such as Sample Extraction, Key Switching, CMux gate, Blind Rotation.

2.3.1 Sample Extraction

Sample Extraction can extract the TLWE ciphertext from the TRLWE ciphertext. Specifically, given a ciphertext $c = (a(x), b(x)) \in \text{TRLWE}(m(x))$ and an index $i \in [0, N-1]$, we can extract the TLWE ciphertext encrypting the i-th coefficient of $m(x)$ without introducing any new noise and we denote this operation by $\text{SampleExtract}_i(c)$. For example, $\text{SampleExtract}_0(c)$ is $\text{TLWE}(m_0) = (a_0, -a_{N-1}, \cdots, -a_1, b_0) \in \mathbb{T}^{N+1}$. This can be simply proved by the decryption function of TRLWE.

2.3.2 Key Switching

- **TLWE-to-T(R)LWE** In [CGGI20], the authors proposed two types of key switching. One is called *Public KeySwitching*, which allows to implement the public function f operation on message while packing multiple TLWE ciphertexts. It is used as one of the building blocks of the TFHE bootstrapping to implement dimension switching. We call it *IdentityKeySwitching* when the function is an identity function. Another is called *Private KeySwitching*, which allows to embed a private function f into the key switching key to implement function computation on the message. We put the details of these two algorithms in Appendix B.1.

- **TRLWE-to-TRLWE** Ciphertext switching between RLWEs is an important component in fully homomorphic encryption, and is widely used in schemes constructed based on the RLWE assumption, such as BGV, BFV, CKKS and TFHE. It is mainly used to switch the secret key of ciphertext after certain homomorphic operations, such as ciphertext multiplication, and automorphism on the ciphertext. In essence, the ciphertext switching between RLWEs is a homomorphic decryption phase operation. For details, see Appendix B.2. For the analysis of noise analysis, we refer the readers to [MP21].

Key Switching can be used to pack some TLWE ciphertexts into a TRLWE ciphertext, while Sample Extraction can unpack TLWE from TRLWE ciphertext. They enable the inter-conversion of T(R)LWE and TRLWE ciphertexts.

2.3.3 CMux Gate

CMux gate stands for Controlled MUltipleXer and is the fundamental computational unit in TFHE bootstrapping. Now let us first review the definition of external multiplication:

$$\begin{aligned} \boxdot : \mathrm{TRGSW} \times \mathrm{TRLWE} &\to \mathrm{TRLWE} \\ (A, b) &\longmapsto A \boxdot b = G^{-1}(b) \cdot A, \end{aligned}$$

where G^{-1} is the gadget decomposition. The uniqueness of the noise growth of the external multiplication is the asymmetric growth, allowing the noise of the accumulator to grow linearly.

CMux gate is constructed by external multiplication, which takes two TRLWE ciphertexts d_0, d_1 as input and one TRGSW ciphertext c as selected input, and then outputs one TRLWE ciphertext:

$$\mathrm{CMux}(c, d_1, d_0) = c \boxdot (d_1 - d_0) + d_0.$$

For more details on the external multiplication, we refer the readers to [CGGI20].

2.3.4 Blind Rotation

Blind Rotation (BR) is the central building block in TFHE bootstrapping, which is composed of n CMux gates, where n is the dimension of the TLWE ciphertext to be bootstrapped. Its key idea is that it can blindly rotate the test polynomial using encrypted number, for more details refer to Algorithm 1.

Another important application of blind rotation is lookup table (LUT) by vertical packing, which is introduced in [CGGI20]. Suppose we want to find the corresponding value of x in Table A. We decompose x in a binary way, *i.e.*, $x = \sum_{i=0}^{n-1} x_i \cdot 2^i$, and encrypt each component $C_i = \mathrm{TRGSW}(x_i), i \in [0, n-1]$. Each function value of Table A is encrypted as TRLWE ciphertext by coefficient packing. By calling $\mathrm{BlindRotation}(A, (2^0, 2^1, \cdots, 2^{n-1}, 0), (C_0, \cdots, C_{n-1}))$, the function value we want to find will be moved to the constant term position and then the desired TLWE ciphertext will be obtained using $\mathrm{SampleExtract}_0$ procedure.

2.4 Bootstrapping Types in TFHE

In this subsection, we will briefly introduce several different bootstrap types in the TFHE scheme to give the readers a more intuitive understanding.

Algorithm 1 BlindRotation Algorithm [CGGI20]

Input: A TRLWE ciphertext(or noiseless) $c \in \text{TRLWE}(m(x))$.
Input: Integer vector $(a_1, \cdots, a_n, b) \in \mathbb{Z}_{2N}^{n+1}$
Input: A set of TRGSW ciphertexts C_i encrypting $s_i, i \in [1, n]$
Output: A TRLWE ciphertext of $X^{-\rho} \cdot m(x)$, where $\rho = b - \sum_{i=1}^{n} s_i \cdot a_i \bmod 2N$

```
ACC ← X^{-b} · c
for i = 1 to n do
    ACC ← CMux(C_i, X^{a_i} · ACC, ACC)
end for
return ACC
```

2.4.1 Gate Bootstrapping

Compared with other fully homomorphic encryption schemes, the TFHE scheme is characterized by efficient bootstrapping operation. It supports various homomorphic logic gates such as NOT, AND, OR, XOR, etc. This operation is called *Gate Bootstrapping* (GBS) [CGGI20], which implies that every logic gate is immediately followed by a bootstrapping operation. For ease of representation, in gate bootstrapping, binary messages 0 and 1 are encoded as $-1/8$ and $1/8$ over the torus, respectively. Now assume two TLWE ciphertexts c_1 and c_2, then some homomorphic gates are as follows:

- HomoNOT$(c) = (\mathbf{0}, 1/8)$ - c (no bootstrapping);
- HomoAND$(c_1, c_2) = (\mathbf{0}, -1/8)$+ Bootstrap$(c_1 + c_2)$;
- HomoXOR$(c_1, c_2) = (\mathbf{0}, 1/4)$+Bootstrap$(2(c_1 \pm c_2))$;
- HomoOR$(c_1, c_2) = (\mathbf{0}, 1/8)$+ Bootstrap$(c_1 + c_2)$;
- HomoMUX(c, d_0, d_1) can evaluate $c?d_1 : d_0 = (c \wedge d_1) \oplus ((1 - c) \wedge d_0)$ using two gate bootstrappings and a public key switching. Compared to CMux gate, HomoMUX is more expensive, but its input and output ciphertext are both TLWE ciphertexts.

For the details of Bootstrap operation, we refer the readers to Appendix A.

2.4.2 Functional Bootstrapping and Multi-value Bootstrapping

Functional Bootstrapping (FBS) [BGGJ19] (or called Programmable Bootstrapping [CJP21], PBS) is an extension of gate bootstrapping that supports embedding the value of an arbitrary function into the test polynomial $T(X)$. Therefore, it is a very powerful tool that provides a novel way to evaluate any negacyclic function $f : \mathbb{Z}_t \to \mathbb{Z}_t$ (*i.e.*, it has to satisfy $f(m + t/2) = -f(m)$ for all m due to modulo $X^N + 1$) on the encrypted input. If f is public, then the test polynomial is set as

$$T(X) = \sum_{i=0}^{N-1} f(\left\lfloor \frac{i \cdot t}{2N} \right\rceil) \cdot X^i.$$

Note that $T(X)$ contains some redundancy in order to decode noisy ciphertexts.

Furthermore, multi-value functional bootstrapping (MVBS) is first proposed by Carpov et al. [CIM19], which supports the computation of multiple functions on the same input while consuming only one functional bootstrapping. Firstly, extract a common function tv_0 from all the test polynomial functions (TV_{F_i}), and then blindly rotate the test polynomial v to obtain the ACC using the input TLWE. Finally, multiply $\frac{TV_{F_i}}{tv_0}$ (with a small canonical coefficient) by the accumulator ACC to obtain the result of the function, respectively. We

refer the readers to Algorithm 4 in [CIM19]. In 2021, Chillotti et al. [CLOT21] proposed a novel PBSmanyLUT technique, which can also evaluate many functions simultaneously. Their strategy is to encode all function values entirely into a test polynomial, thus consuming only an blind rotation. Finally, all function results are rotated to the first few coefficients of the test polynomial. We refer the readers to Algorithm 6 in [CLOT21].

2.4.3 Circuit Bootstrapping

In addition to the fully homomorphic evalution (FHE) mode, TFHE also supports a leveled homomorphic evaluation (LHE) mode, which is built from CMux gates. As shown in Section 2.3.3, the external multiplication requires the inputs to be the TRLWE ciphertext and TRGSW ciphertext and the output to be the TRLWE ciphertext. As a result, we cannot arbitrarily combine circuits like gate bootstrapping or functional bootstrapping. The multiplication between two TRLWEs is not defined in TFHE. Fortunately, circuit bootstrapping (CBS) can convert the TLWE to TRGSW ciphertext, thus making the LHE mode of TFHE feasible. However, its running time is much more expensive than gate bootstrapping. In Section 4, we would describe in detail circuit bootstrapping and its optimization.

3 Homomorphic Evaluation of SNOW 3G

3.1 A Short Specification of SNOW 3G

SNOW 3G is the core of the 3GPP standard algorithms EEA2 & EIA2 for data confidentiality and data integrity. It is a stream cipher algorithm for 32-bit word implementations. It consists of a 16-level linear feedback shift register (LFSR) on $\mathbb{F}_{2^{32}}$ and a finite state machine (FSM). The generation of the keystream sequence consists of an initialization process shown in Figure 10 and a keystream generation process, as shown in Figure 1. Here, we mainly focus on the keystream generation stage.

Keystream Generation: After the initialization stage, the FSM is clocked once. The output word of the FSM is discarded. Then the LFSR is clocked once in Keystream Mode. After that n keystream words are produced by repeating the following three steps:
for $t = 1$ to n:

(1) The FSM is clocked and produces a 32-bit output word F;

(2) The next keystream word is computed as $z_t = F \oplus s_0$, for $1 \leq t \leq n$;

(3) Then the LFSR is clocked in Keystream Mode.

3.2 Overall Analysis of Function in SNOW 3G

We now first summarize the function types required to perform SNOW 3G, since they would collectively determine the choice of homomorphic computation method and message encoding of FHE scheme. In the context of FHE, the choice of message space size affects the choice of parameters for the homomorphic encryption scheme. The second thing to note is that the choice of message encoding determines the type of computation supported by the corresponding homomorphic ciphertext. For example, in BGV, BFV, and CKKS schemes, slot encoding is usually used to support SIMD addition and multiplication, but nonlinear functions cannot be evaluated directly and can only be handled by approximate polynomials. The TFHE scheme supports efficient universal gate bootstrapping so that arbitrary functions can be computed by circuit constructed by gate bootstrapping. The

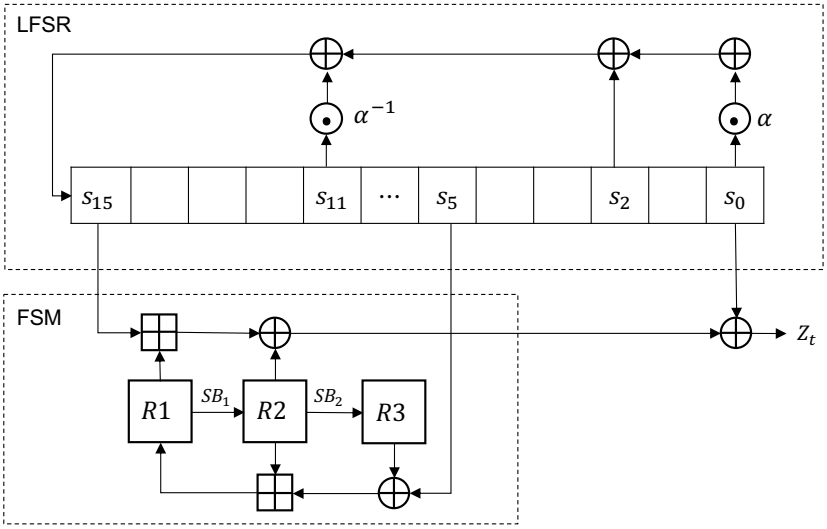

Figure 1: Keystream generation of SNOW 3G.

ability of functional bootstrapping to compute arbitrary functions defined in the message space for free while refreshing the noise has been widely favored [CJP21, TCBS23]. However, as just stated, once the message space is determined, the type of computation is determined, and any function in the homomorphic sense must be converted to the basic type of computation supported by the homomorphic ciphertext. An effective alternative solution is ciphertext conversion [BGG18, LHH+21], *i.e.*, to convert the ciphertext form to make it easier to handle specific functions.

Three types of functions are included in the FSM operation: modular addition $\boxplus$, lookup table SB_1/SB_2 and bitwise XOR, while in the initialization or keystream mode, bit shiftings, bitwise XOR and $\text{MUL}_\alpha/\text{DIV}_\alpha$ functions are needed to update the LFSR. There is no doubt that evaluating these functions based on the gate bootstrapping mode is the most straightforward approach. The computational cost of the evaluation mode based on gate bootstrapping depends on the number of gates consumed by the function computation. It is particularly suitable for operations such as bit shiftings, bitwise XOR due to bit-wise encryption. Next we would give the computation strategy for all functions in SNOW 3G based on gate bootstrapping.

3.3 Evaluation of Modular Addition $\boxplus$

Modular addition in SNOW 3G means modulo 2^{32}, *i.e.*, given two 32-bit word X and Y, then to compute 32-bit output value $Sum = X + Y \bmod 2^{32}$. This modular addition($\boxplus$)operation can be easily computed using Ripple-Carry Adder (RCA) by discarding the most important bit of result. In detail, given two n-bit integers, X and Y, with their binary representation expressed as $X = [X_0, X_1, \cdots, X_{n-1}], Y = [Y_0, Y_1, \cdots, Y_{n-1}]$, we can build the full adder (FA) according to the following formula:

$$\begin{aligned} Carry_{i+1} &= (X_i \wedge Y_i) \vee ((X_i \oplus Y_i) \wedge Carry_i); \\ Sum_i &= X_i \oplus Y_i \oplus Carry_i, \end{aligned}$$

where $Carry_0 = 0$. Under gate bootstrapping mode, we can evaluate $\wedge$, $\vee$ and $\oplus$ gate by HomoAND, HomoOR and HomoXOR homomorphic gates, respectively. Therefore, this textbook modular addition circuit can be evaluated using $32 * 5 - 3 = 157$ gate bootstrappings, and these gates must be evaluated in serial. How to accelerate the full

adder in the homomorphic context is one of the important challenges for TFHE applications, *i.e.*, to reduce the number of gate bootstrappings consumed.

In fact, FA can be considered as a gate constructed from a 3-input XOR gate (the lowest important bit of sum of 3-input) and a majority gate (the most important bit of sum of 3-input) that share the same inputs. Matsuoka et al. [MHSB21] proposed cryptographic optimization for 3-input gates or multi-output gates using the TFHE scheme, including half adder, full adder and AOI21. In particular, they proposed an optimized full adder: the 2BR Full Adder (Algorithm 3 in [MHSB21]) consuming two blind rotations when the message $\{0,1\}$ is encoded to $\{-\frac{1}{8},\frac{1}{8}\}$ respectively, as Section 2.4.1. Specifically, let CX_i, CY_i and $CCarry_i$ be ciphertexts corresponding to input X_i, Y_i and $Carry_i$. First, we compute $C_{add} = CX_i + CY_i + CCarry_i$, then

- $\text{HomoSum}(CX_i, CY_i, CCarry_i) = \text{Bootstrap}(-2 \cdot C_{add})$;
- $\text{HomoCarry}(CX_i, CY_i, CCarry_i) = \text{Bootstrap}(C_{add})$.

A high-level explanation is as follows. For HomoSum gate, the plaintext phase corresponding to $-2 \cdot C_{add}$ is $\frac{1}{4}$ if the number of ciphertexts corresponding to plaintext with phase $\frac{1}{8}$ is odd, and $-\frac{1}{4}$ otherwise. Therefore, the evaluation of BlindRotation with test polynomial $TV[X] = \sum_{i=0}^{N-1} \frac{1}{8} X^i$ corresponds to the evaluation of HomoSum. For HomoCarry gate, if at least 2 of the 3 input ciphertexts correspond to plaintexts with phase $\frac{1}{8}$, then the plaintext of their sum C_{add} is a positive number, *i.e.*, $\frac{1}{8}$ or $\frac{3}{8}$ and is a negative number, *i.e.*, $-\frac{1}{8}$ or $-\frac{3}{8}$, otherwise. Therefore, the evaluation of BlindRotation with test polynomial $TV[X] = \sum_{i=0}^{N-1} \frac{1}{8} X^i$ corresponds to the evaluation of HomoCarry.

3.3.1 Single-Gate-Bootstrapping for Full Adder

Matsuoka et al. also proposed 1BR Full Adder (Algorithm 4 in [MHSB21]). However, this requires the message $\{0,1\}$ is encoded to $\{-\frac{1}{12},\frac{1}{12}\}$, respectively. They used the multi-value bootstrapping technique to evaluate the two output tables of the full adder, thus consuming only one blind rotation at the cost of increasing the decryption error probability. Here, we present a new simpler and faster method for full adder evaluation without changing the message encoding, which may be of independent interest. We observe that the result of HomoSum can be obtained directly from HomoCarry when the message $\{0,1\}$ is encoded to $\{-\frac{1}{8},\frac{1}{8}\}$:

$$\text{HomoSum}(CX_i, CY_i, CCarry_i) = C_{add} - 2 \cdot \text{HomoCarry}(CX_i, CY_i, CCarry_i).$$

We call it single-gate-bootstrapping for full adder. Therefore, we can also implement the evaluation of the full adder with just one gate bootstrapping (or BR) without changing the message encoding. Notice that compared with HomoCarry, HomoSum would be a relatively noisy ciphertext, which is exactly the result we want for modular addition $\boxplus$. In this way, we need only 32 blind rotations to complete the evaluation of 32-bit modular addition. Theorem 1 demonstrates the correctness of our single-gate-bootstrapping for full adder and the noise growth.

Theorem 1. *Let Bootstrap denote TFHE's Gate Bootstrapping (Algorithm A), $C_{add} = CX_i + CY_i + CCarry_i$, then Bootstrap($C_{add}$) outputs HomoCarry($CX_i, CY_i, CCarry_i$), and C_{add}−2Bootstrap(C_{add}) outputs HomoSum($CX_i, CY_i, CCarry_i$), such that*

- $\sigma^2_{\text{HomoCarry}(CX_i,CY_i,\text{CCarry}_i)} = \sigma^2_{\text{Bootstrap}}$
- $\sigma^2_{\text{HomoSum}(CX_i,CY_i,\text{CCarry}_i)} \leq 3\sigma^2_{fresh} + 2\sigma^2_{\text{Bootstrap}}$

where $\sigma^2_{\text{HomoCarry}(CX_i,CY_i,\text{CCarry}_i)}$, $\sigma^2_{\text{HomoSum}(CX_i,CY_i,\text{CCarry}_i)}$, $\sigma^2_{\text{Bootstrap}}$ denote the error variance of the algorithm outputs, σ^2_{fresh} denote the error variance of the fresh ciphetext.

Proof. The corresponding plaintext phases of C_{add}, Bootstrap(C_{add}) and C_{add}-Bootstrap(C_{add}) are shown in Table 1:

Table 1: Truth Table

X_i	Y_i	$Carry_i$	C_{add}	Bootstrap(C_{add})	$C_{add}-2$Bootstrap(C_{add})
0	0	0	$-\frac{3}{8}$	$-\frac{1}{8}$	$-\frac{1}{8}$
0	0	1	$-\frac{1}{8}$	$-\frac{1}{8}$	$\frac{1}{8}$
0	1	0	$-\frac{1}{8}$	$-\frac{1}{8}$	$\frac{1}{8}$
0	1	1	$\frac{1}{8}$	$\frac{1}{8}$	$-\frac{1}{8}$
1	0	0	$-\frac{1}{8}$	$-\frac{1}{8}$	$\frac{1}{8}$
1	0	1	$\frac{1}{8}$	$\frac{1}{8}$	$-\frac{1}{8}$
1	1	0	$\frac{1}{8}$	$\frac{1}{8}$	$-\frac{1}{8}$
1	1	1	$\frac{3}{8}$	$\frac{1}{8}$	$\frac{1}{8}$

The decoded results of the last two columns are equal to Carry_{i+1} and Sum_i, respectively, thus without decryption failures, Bootstrap(C_{add}) outputs HomoCarry($CX_i, CY_i, CCarry_i$), and $C_{add}-2$Bootstrap(C_{add}) outputs HomoSum($CX_i, CY_i, CCarry_i$). Then, we analyze the error growth.

- $\sigma^2_{\text{HomoCarry}(CX_i,CY_i,\text{CCarry}_i)} = \sigma^2_{\text{Bootstrap}(C_{add})} = \sigma^2_{\text{Bootstrap}}$
- $\sigma^2_{\text{HomoSum}(X_i,Y_i,\text{CCarry}_i)} \leq \sigma^2_{C_{add}} + 2\sigma^2_{\text{Bootstrap}(C_{add})} \leq 3\sigma^2_{fresh} + 2\sigma^2_{\text{Bootstrap}}$

□

3.4 Evaluation of SB_1/SB_2

Nonlinear function lookup tables are heavily used in SNOW 3G. The S-Box SB_1 maps a 32-bit input to a 32-bit output. Specifically, let $w = w_0||w_1||w_2||w_3$ be the 32-bit input and then $SB_1(w) = r_0||r_1||r_2||r_3$, where r_0, r_1, r_2, r_3 are defined as

$$\begin{pmatrix} r_0 & = \text{MULx}(S_R(w_0), 0x1B) \oplus S_R(w_1) \oplus S_R(w_2) \oplus \text{MULx}(S_R(w_3), 0x1B) \oplus S_R(w_3) \\ r_1 & = \text{MULx}(S_R(w_0), 0x1B) \oplus S_R(w_0) \oplus \text{MULx}(S_R(w_1), 0x1B) \oplus S_R(w_2) \oplus S_R(w_3) \\ r_2 & = S_R(w_0) \oplus \text{MULx}(S_R(w_1), 0x1B) \oplus S_R(w_1) \oplus \text{MULx}(S_R(w_2), 0x1B) \oplus S_R(w_3) \\ r_3 & = S_R(w_0) \oplus S_R(w_1) \oplus \text{MULx}(S_R(w_2), 0x1B) \oplus S_R(w_2) \oplus \text{MULx}(S_R(w_3), 0x1B) \end{pmatrix}$$

and S_R is 8-to-8-bit Rijndael S-Box. The S-Box SB_2 also maps a 32-bit input to a 32-bit output as SB_1, except that it uses another Rijndael S-Box S_Q. Therefore, we next illustrate our evaluation method with SB_1 as an example.

We first give the evaluation of S-box S_R by tree-based lookup table algorithm. It must be noted that our input ciphertext is in the form of TLWE, and we choose HomoMUX as the basic gate in our tree lookup table, which ensures that the type of ciphertext remains uniform after evaluating the S-box lookup table, as shown in Algorithm 2.

For subroutine MULx(V, c) function, it is a branching function:

$$\text{MULx}(V, c) = \begin{cases} (V <<_8 1) \oplus c & \text{, if the most significant bit of V equals 1,} \\ V <<_8 1 & \text{, else.} \end{cases}$$

Thanks to the utilization of our bit-wise encryption, we can implement bit extraction and bit shiftings for free and then use HomoMUX gate to achieve result selection according to the most significant bit of V.

Algorithm 2 8-to-8-bit S_R lookup table using HomoMUX gate

Input: $C_{8,256}^{256}$ TLWE ciphertexts encrypting S_R table values
Input: $C_i^{input}, 0 \leq i \leq 7$ TLWE ciphertexts encrypting eight input values
Output: $C_i^{output}, 0 \leq i \leq 7$ TLWE ciphertexts encrypting eight output values
1: **for** $i = 0$ to 7 **do**
2: **for** $j = 0$ to 7 **do**
3: **for** $k = 0$ to $2^{7-j} - 1$ **do**
4: $C_{i,k}^{2^{7-j}} = \text{HomoMUX}(C_j^{input}, C_{i,2k}^{2^{8-j}}, C_{i,2k+1}^{2^{8-j}})$
5: **end for**
6: **end for**
7: **end for**
8: **return** $C^{output} = C_{i,0}^0, 0 \leq i \leq 7$

3.5 Evaluation of MUL_α, DIV_α

MUL_α and DIV_α map 8 bits to 32 bits and are used in initialization mode and keystream mode to update the LFSR as follows:

$$\begin{aligned}\text{MUL}_\alpha(c) =&(\text{MULxPOW}(c, 23, 0xa9)||\text{MULxPOW}(c, 245, 0xa9)\\ &||\text{MULxPOW}(c, 48, 0xa9)||\text{MULxPOW}(c, 239, 0xa9)),\\ \text{DIV}_\alpha(c) =&(\text{MULxPOW}(c, 16, 0xa9)||\text{MULxPOW}(c, 39, 0xa9)\\ &||\text{MULxPOW}(c, 6, 0xa9)||\text{MULxPOW}(c, 64, 0xa9)),\end{aligned}$$

where

$$\text{MULxPOW}(V, i, c) = \begin{cases} V & , \text{if } i = 0, \\ \text{MULx}(\text{MULxPOW}(V, i-1, c), c) & , \text{else.} \end{cases}$$

Obviously, the evaluation of MUL_α and DIV_α are actually the evaluation of MULxPOW, which is a recursive use of MULx function.

3.6 Discussion of Several evaluation Methods for Lookup Table

In the previous subsections, we evaluate all function operations using the gate bootstrapping mode. For S-box S_R, we use HomoMUX as the basic gate shown in Algorithm 2. An 8-to-8-bit S-box table lookup evaluation would consume $8*(128+64+32+16+8+4+2+1) = 2040$ HomoMUX gates, equivalent to 4080 gate bootstrappings. If we estimate that each gate bootstrapping consumes 10ms, an 8-to-8-bit S-box lookup table would consume up to 40 seconds.

Essentially, functional bootstrapping can be viewed as lookup table operation which encodes all function values into test polynomial. As such, it can directly support S-box evaluation. For example, in [CHMS22], the authors directly designed stream cipher algorithm Elisabeth which is friendly to functional bootstrapping. Recently, Trama et al. [TCBS23] analyzed the cost of AES evaluation based on functional bootstrapping of TFHE and achieved the lowest evaluation latency with parallel processing of a single block taking only 28 seconds. Functional bootstrapping appears to be the preferred choice. However, a crucial consideration is whether the other types of function computations are also suitable for functional bootstrapping. In the evaluation of AES, they had to transform the multiplication operation in MixColumns and the XOR operation in AddRoundKey into lookup tables because of the message encoding within the ciphertext. Basic operations like bit shifting and bit XOR would become bottlenecks in the overall evaluation process. Consequently, when dealing with multiple types of function computations simultaneously,

evaluations relying on functional bootstrapping can still incur significant computational costs. Therefore, we do not consider implementation based on functional bootstrapping, and we resort to leveled evaluation mode, where CMux gate is used as the basic gate to compose the overall evaluation circuit. Each CMux gate takes about 34μs [CGGI20] and is much faster than HomoMUX built by gate bootstrapping.

3.7 Optimization of S-box S_R Evaluation via LHE Mode

In [CGGI20], the authors proposed two methods of horizontal packing and vertical packing for arbitrary lookup tables. For security reasons, the dimension of polynomial rings is generally large, such as $N = 1024, 2048$, etc. Therefore, roughly speaking, horizontal packing is suitable for the table with much larger columns than rows, so that we can pack each row of the table into TRLWE ciphertext (coefficient packing), and then use several CMux gates together to pick out the TRLWE ciphertext where the desired result is located. Vertical packing is suitable for the table where the number of rows is much larger than the columns, so we can pack each column as TRLWE ciphertext and then use blind rotation to implement the lookup table, as mentioned in Section 2.3.4.

Currently, for TFHE scheme, the dimension of polynomial ring is usually set to $N = 1024$. For 8-to-8-bit S-box evaluation, either using horizontal or vertical packing will result in underutilization of polynomial packing. As a result, we achieve an efficient evaluation of S_R by combining horizontal and vertical packing shown in Algorithm 3.

Algorithm 3 The evaluation of 8-to-8-bit S-box S_R using hybrid packing

Input: Eight TRGSW ciphertexts $C_0, \cdots, C_7$ encrypting the input bits
Input: Two TRLWE ciphertexts T_0 and T_1 used to pack S_R table
Output: Eight TLWE ciphertexts $c_0, \cdots, c_7$ encrypting the output bits
1: ACC $\leftarrow$ CMux(C_7, T_1, T_0)
2: BlindRotation(ACC,$(8 * 2^0, \cdots, 8 * 2^6, 0), (C_0, \cdots, C_6)$)
3: **for** $i = 0$ to 7 **do**
4: $\quad c'_i$ = SampleExtract$_i$(ACC)
5: $\quad c_i$ = IdentityKeySwitching(c'_i)
6: **end for**
7: **return** $c_0, \cdots, c_7$

Based on the table characteristics of S_R, we only need to pack it to $8 * 2^8/N = 2$ TRLWE ciphertexts, *i.e.*, T_0 and T_1. Of course, since an S-box is generally public, it can be encrypted as noiseless ciphertexts. In line 1 of Algorithm 3, we first pick out the TRLWE ciphertext where the result is located using the TRGSW ciphertext C_7 of the most significant bit of the input. Then after using the remaining TRGSW ciphertext C_i, for $0 \leq i \leq 6$, we can move all the results to the first 8 coefficients of the plaintext polynomial based on the BlindRotation algorithm (line 2). Note that the second parameter setting in the BlindRotation of Algorithm 3 contains a factor of 8. The reason for this is that the output of S_R is 8-bit, *i.e.*, every 8 bits are treated as a block when the table values are packed. Finally we obtain the resulting TLWE ciphertext using the SampleExtract and the IdentityKeySwitching algorithm (lines 4-5). In total, we only need 8 CMux gates, which is much more efficient than Algorithm 2.

One point that cannot be overlooked in Algorithm 3 is efficient CMux gate requires that the ciphertext of the selector bit must be a TRGSW ciphertext. In bootstrapping operation of TFHE scheme, the secret key of TLWE is encrypted in advance as TRGSW ciphertexts so that the decryption circuit can be evaluated in a leveled mode. However, our computational task is more complicated, before and after evaluating the S-box, we have to ensure that the ciphertext is in the form of TLWE to implement other functions

based on gate bootstrapping mode. Therefore, when evaluating the lookup tables, we have to resort to the circuit bootstrapping to transform the TLWE ciphertexts of the input bits into the TRGSW ciphertexts for Algorithm 3.

4 Bridge of FHE and LHE mode: Circuit Bootstrapping

Circuit bootstrapping can be seen as a bridge to compose the circuit in the TFHE leveled evaluation mode. Next we first review circuit bootstrapping and its recent development in Section 4.1. Then in Section 4.2, we present our adjustments to circuit bootstrapping to further improve the overall efficiency of our evaluation framework, which can be of independent interest.

4.1 The State-of-the-art of Circuit Bootstrapping

Chillotti et al. [CGGI17] firstly proposed circuit bootstrapping. They note that each line of TRGSW encrypting message $m \in \mathbb{B}$ under the secret key $S = (-s(x), 1)$ is a TRLWE and their corresponding messages are $m \cdot S_i \cdot \frac{1}{B_g^j}$, for $1 \leq i \leq 2, 1 \leq j \leq \ell$, where ℓ is the decomposition length and B_g is the decomposition basis. Thus the idea of circuit bootstrapping is to reconstruct all TRLWE ciphertexts in TRGSW using initial TLWE ciphertext. In their setup, the parameters are set to three levels in order to control the noise: Level 0, Level 1 and Level 2. Higher levels mean larger parameters and more tolerable noise. Circuit bootstrapping(TLWE-to-TRGSW) consists of two main steps:

(1) Functional Bootstrapping: The functional bootstrapping from Level 0 to Level 2 takes as input TLWE(m) to compute TLWE(mB_g^{-j}), $j \in [1, \ell]$;

(2) TLWE-to-TRLWE Ciphertext Conversion: Convert TLWE(mB_g^{-j}) to TRLWE(mB_g^{-j}) from Level 2 to Level 1 using PublicKeySwitching and TRLWE($-m \cdot s \cdot B_g^{-j}$) using PrivateKeySwitching, respectively.

Chillotti et al. provided a proof-of-concept implementation[1] and claimed that circuit bootstrapping takes 137 milliseconds, about 10 times the cost of gate bootstrapping. With 110-bit security parameters, the first step of the functional bootstrapping operation accounts for 70% of the running time, and the second step of the ciphertext conversion accounts for the remaining 30%. Therefore, how to improve the efficiency of circuit bootstrapping is one of the major challenges in the research area of TFHE.

4.1.1 Improvement of the First Step

In [CJP21], the authors observed that in the first step of TLWE-to-TRGSW, all TLWE ciphertexts TLWE(mB_g^{-j}), $j \in [1, \ell]$ are computed from the same input TLWE(m), and thus this step can be further optimized by using the PBSmanyLUT technique, which is a simpler method and has better noise control than multi-value bootstrapping [CIM19]. In this way, the number of functional bootstrapping in the first step is reduced from ℓ to 1, which greatly improves the efficiency of circuit bootstrapping. For more details of PBSmanyLUT technique, we refer readers to Lemma 4 in [CJP21]. To the best of our knowledge, TFHEpp[2] is the first publicly available homomorphic cryptographic library to implement PBSmanyLUT. In [GBA22], the authors provided a faster implementation using AVX-512 acceleration in the MOSFHET library[3].

[1] https://github.com/tfhe/experimental-tfhe
[2] https://github.com/virtualsecureplatform/TFHEpp
[3] https://github.com/antoniocgj/MOSFHET

4.1.2 Improvement of the Second Step

As discussed above, after the optimization of PBSmanyLUT, the computational cost of the second step has been slightly higher than the first step. In 2019, Chen et al. [CCR19] proposed an improved TRLWE to TRGSW method in order to reduce the communication cost between the client and the server, mainly originating from the TRGSW ciphertexts used for the ORAM eviction process. Their observation lies in the fact that the second ℓ rows of TRGSW ciphertexts are TRLWE(mB_g^{-j}), which is independent of the secret key s, while the first ℓ rows are TRLWE$(-m \cdot s \cdot B_g^{-j})$, which is related to the secret key s, and this part of the ciphertexts can be generated by external multiplication of TRLWE(mB_g^{-j}) and TRGSW$(-s)$ for $j = 1, \cdots, \ell$. Ultimately, these TRLWE ciphertexts collectively constitute the TRGSW ciphertext that encrypts the message m. The reason for the speedup in this approach is that a single external multiplication is considerably less costly than a single PrivateKeySwitching operation.

Further optimizations were found in the MOSFHET library, but the authors are not currently updating these optimizations in [GBA22]. It is described as follows: instead of relying on expensive PrivateKeySwitching or external multiplication, they managed to construct the first ℓ rows TRLWE ciphertexts of TRGSW directly by applying RLWE key switching to the second ℓ rows TRLWE ciphertexts of TRGSW. The operation was later named *EvalSquareMult* in [KLD+23]. The cost of a single key switching operation between RLWEs is roughly half that of an external multiplication, which makes this approach outperform the method used by Chen et al. [CCR19]. Additionally, the key size required for key switching between RLWEs is also smaller than that proposed by Chen et al. Below we briefly describe the *EvalSquareMult* algorithm: given a keyswitching key $sqk = \text{KeySwitchGen}(s, s^2)$ and $c = (a, b) \in \text{TRLWE}(mB_g^{-j})$, EvalSquareMult$(c, sqk)$ performs the following two steps:

- Decompose the first part $a(x)$ into some small polynomials $\sum_{i=1}^{\ell} a_i(x)B_g^i$,
- Compute $(b, 0) + (\sum a_i(x) \cdot sqk[0]_i, \sum a_i(x) \cdot sqk[1]_i) = \text{TRLWE}(-s \cdot m \cdot B_g^{-j})$, as desired.

4.2 Adjustment of Circuit Bootstrapping for Our Evaluation Framework

As discussed above, the best way to implement circuit bootstrapping currently consists of two optimizations: one is the *PBSmanyLUT* technique to accelerate functional bootstrapping of the first step, and the second is *PublicKeySwitching* and *EvalSquareMul* to implement ciphertext conversion. However, we notice that even though the above operations have been implemented in the MOSFHET library, they still run very slowly, the main reason is that the second step of PublicKeySwitching and EvalSquareMult are executed at Level 2 where large parameters are used.

Recall that here, our aim is to use the TRGSW ciphertext as a selector bit for implementing the S-box lookup table through the CMux gate. To achieve this, we made adjustments to the currently optimized circuit bootstrapping process, specifically by moving the ciphertext transformation to Level 1. While this does lead to an increase in the number of polynomials required to pack S-box function values, the impact on the leveled evaluation is minimal. The advantage of this approach is that smaller parameters will result in faster PublicKeySwitching and EvalSquareMult operations. Figure 2 illustrates formally the adjustments we made to the circuit bootstrapping to accelerate the overall computational efficiency. And we provide pseudo-code for our circuit bootstrapping in Algorithm 4. In Section 7.2.2, we present a detailed efficiency comparison of circuit bootstrapping.

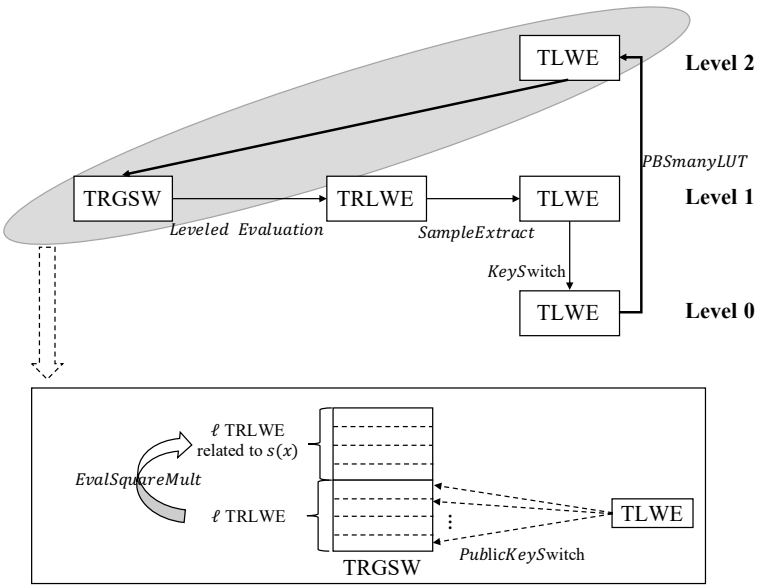

Figure 2: The figure represents a schematic of the execution of our adaption to circuit bootstrapping (TLWE-to-TRGSW). The arrows indicate the operations that can be performed within each level, and how to move from one level to another. The diagram in the box below shows in detail how the TLWE ciphertext is used to construct all the TRLWE ciphertexts inside the TRGSW.

Algorithm 4 Faster circuit bootstrapping combining PBSmanyLUT and EvalSquareMult

Input: a Level 0 TLWE ciphertext: $ct \in \mathrm{TLWE}(m)$

Input: a test polynomial $P(X) = \sum_{i=0}^{\frac{N}{2^\rho}-1} \sum_{j=0}^{2^\rho-1} \cdot \frac{1}{2B_g^j} X^{2^\rho \cdot i + j}$ encoded all function values for PBSmanyLUT, where $\rho = \lceil \log_2(\ell) \rceil$, B_g is the basis of gadget decomposition.

Input: a bootstrapping key from Level 0 to Level 2: bsk

Input: a Level 1 PublicKeySwitching key 1: $pubks$

Input: a Level 1 EvalSquareMult key 1: sqk

Output: a Level 1 TRGSW ciphertext $C \in \mathrm{TRGSW(m)}$

1: $\{\mathrm{ct}_i\}_{i\in[1,\ell]} \leftarrow \mathrm{PBSmanyLUT}\left(ct, bsk, P(X) \cdot X^{N/2^{\rho+1}}, 1, 0, \rho\right)$
2: **for** $j = 1$ to ℓ **do**
3: $\quad \mathrm{ct}'_j = \mathrm{ct}_j + \left(\mathbf{0}, \frac{1}{2B_g^j}\right)$
4: $\quad c^{j+\ell} \leftarrow \mathrm{PublicKeySwitching}(\mathrm{ct}'_j, pubks)$ // the second ℓ rows
5: $\quad c^j \leftarrow \mathrm{EvalSquareMult}(c^{j+\ell}, sqk)$ // the first ℓ rows constructed by the second ℓ rows
6: **end for**
7: **return** $C = (c^j)_{,1\le j\le 2\ell}$

Error Analysis. We propose Theorem 2 to measure the error growth of Algorithm 4.

Theorem 2. *Let n, N denote the dimension of TLWE and TRLWE ciphtext on Level 1, B, ℓ, ϵ denote the gadget decomposition length, base and error of the Level 1 ciphertext, B_{ks}, ℓ_{ks}, ϵ_{ks} denote the gadget decomposition parameters of the PublicKeySwitching algorithm, the same variables with underlines/bars for Level 0 and Level 2 parameters, respectively. Then the error variance of the output C of Algorithm 4 is bounded by*

$$\sigma_C^2 \le \frac{1}{3}\underline{n}\bar{\ell}\bar{N}\bar{B}^2\sigma_{fresh}^2 + \frac{2}{3}\underline{n}(\bar{N}+1)\bar{\epsilon}^2 + \frac{1}{12}\bar{N}\ell_{\mathsf{ks}}B_{ks}^2\sigma_{fresh}^2 + \frac{\bar{N}}{6}\epsilon_{\mathsf{ks}}^2 + \frac{1}{12}N\ell B^2\sigma_{fresh}^2 + \frac{N^2}{12}\epsilon^2,$$

where σ^2_{fresh} is the error variance of the fresh RLWE ciphertext, which be used as the evaluation key.

Proof. The adjustment to circuit bootstrapping algorithm mainly consists of 3 steps: PBSmanyLUT, PublicKeySwitching and EvalSquareMult. We then analyze the noise growth step by step.

At line 1, according to Theorem 4 in [CLOT21], we have:

$$\sigma^2_{PBSmanyLUT}\& \leq \frac{1}{3}\underline{n}\bar{\ell}\bar{N}\bar{B}^2\sigma^2_{fresh} + \frac{2}{3}\underline{n}(\bar{N}+1)\bar{\epsilon}^2$$

At line 4, following Theorem 2 in [CLOT21], the result of the public key switching has an error bound

$$\sigma^2_{PublicKeySwitching} \leq \sigma^2_{PBSmanyLUT} + \frac{1}{12}\bar{N}\ell_{\mathsf{ks}}B^2_{ks}\sigma^2_{fresh} + \frac{\bar{N}}{6}\epsilon^2_{\mathsf{ks}},$$

At line 5, we call EvalSquareMult algorithm on the previous result, then

$$\sigma^2_{EvalSquareMult} \leq \sigma^2_{PublicKeySwitching} + \frac{1}{12}N\ell B^2\sigma^2_{fresh} + \frac{N^2}{12}\epsilon^2.$$

In conclusion, the total error variance of the output of Algorithm 4 is bounded by

$$\sigma^2_C \leq \frac{1}{3}\underline{n}\bar{\ell}\bar{N}\bar{B}^2\sigma^2_{fresh} + \frac{2}{3}\underline{n}(\bar{N}+1)\bar{\epsilon}^2 + \frac{1}{12}\bar{N}\ell_{\mathsf{ks}}B^2_{ks}\sigma^2_{fresh} + \frac{\bar{N}}{6}\epsilon^2_{\mathsf{ks}} + \frac{1}{12}N\ell B^2\sigma^2_{fresh} + \frac{N^2}{12}\epsilon^2.$$

□

4.3 Putting it together

To summarize the above, we propose a hybrid evaluation mode, dubbed Thunderbird, which combines the FHE mode using gate bootstrapping with the LHE mode employing CMux. To the best of our knowledge, this is the first utilization of a combination of these two modes in a practical application.

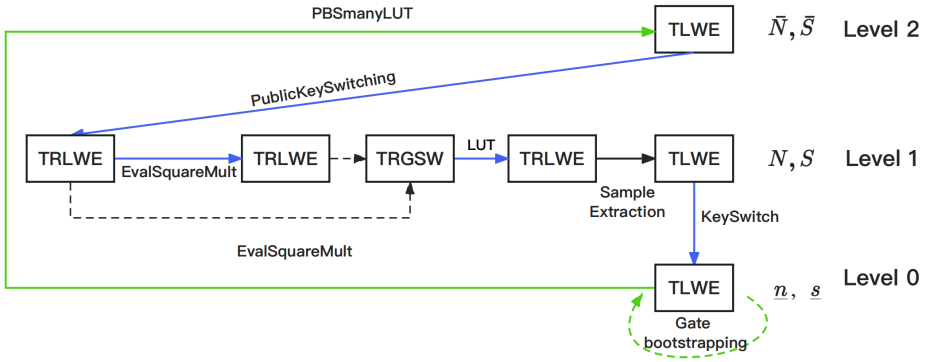

Figure 3: Thunderbird: an evaluation mode combining gate bootstrapping and circuit bootstrapping, which also shows the conversion of all ciphertext types in TFHE. The green arrow means that the noise would be refreshed, and the blue arrow represents that the operation would increase the noise.

Figure 3 provides a detailed representation of our evaluation mode for the ciphers in 3GPP. At Level 0, we perform evaluations of various basic functions that are conducive to gate bootstrapping, including operations like bit shiftings, bit XOR, and modular addition, as outlined in Section 3.2. In Level 2, we perform the first step of circuit bootstrapping:

PBSmanyLUT. At Level 1, we accomplish the TLWE-to-TRLWE ciphertext conversion for constructing TRGSW using PublicKeySwitching and EvalSquareMult. Then, we use the CMux gate to evaluate the S-box lookup table. Finally, with the SampleExtract and IdentityKeySwitching algorithms, we can move back to Level 0 and continue to execute operations that are favorable for gate bootstrapping mode.

4.4 Further Optimization of SB_1/SB_2, MUL_α and DIV_α Evaluation

For 32-to-32-bit S-box SB_1/SB_2 in SNOW 3G, it is composed of 8-to-8-bit S_R, MULx function and a large number of XOR gates. Although we give an efficient leveled evaluation of S_R in Section 3.7, the MULx function and a large number of HomoXOR gates still cause a huge computational overhead, so further optimization of the SB_1/SB_2 evaluation is still necessary. We observe that $SB_1(w)$ and $SB_2(w)$ can be also computed by using lookup tables. In detail, $SB_1(w)$ and $SB_2(w)$, where $w = w_0||w_1||w_2||w_3$, are computed by

$$SB_1(w) = S1\_T0(w3) \oplus S1\_T1(w2) \oplus S1\_T2(w1) \oplus S1\_T3(w0),$$
$$SB_2(w) = S2\_T0(w3) \oplus S2\_T1(w2) \oplus S2\_T2(w1) \oplus S2\_T3(w0),$$

where $S1\_T0, S1\_T1, S1\_T2, S1\_T3, S2\_T0, S2\_T1, S2\_T2$ and $S2\_T3$ are 8-to-32-bit tables. Similarly, the MUL_α and DIV_α functions both map 8 bits to 32 bits, therefore, we can also evaluate them by lookup table efficiently rather than gate bootstrapping. Algorithm 5 presents our efficient leveled evaluation of the 8-to-32-bit table with $S1\_T0$ as an example.

Algorithm 5 Efficient evaluation of $S1\_T0$ (8-to-32-bit) table.

Input: Eight TRGSW ciphertexts $C_0, \cdots, C_7$ encrypting the input bits
Input: Eight TRLWE ciphertexts T_i, for $0 \le i \le 7$, used to pack $S1\_T0$ table
Output: 32 TLWE ciphertexts c_i, for $0 \le i \le 31$, encrypting the output bits
1: **for** $i = 0$ to 3 **do**
2: $\quad Temp_i = \text{CMux}(C_5, T_{2i+1}, T_{2i})$
3: **end for**
4: **for** $i = 0$ to 1 **do**
5: $\quad Tmp_i = \text{CMux}(C_6, Temp_{2i+1}, Temp_{2i})$
6: **end for**
7: $\text{ACC} \leftarrow \text{CMux}(C_7, Tmp_1, Tmp_0)$
8: $\text{BlindRotation}(\text{ACC}, (32 * 2^0, \cdots, 32 * 2^4, 0), (C_0, \cdots, C_4))$
9: **for** $i = 0$ to 31 **do**
10: $\quad c'_i = \text{SampleExtract}_i(\text{ACC})$
11: $\quad c_i = \text{IdentityKeySwitching}(c'_i)$
12: **end for**
13: **return** $c_0, \cdots, c_{31}$

For the 8-to-32-bit table, we need to pack $32 * 2^8/1024 = 8$ TRLWE ciphertexts. We first use TRGSW ciphertexts of the most significant three bits to select the TRLWE ciphertext where the result is located (lines 1-7 of Algorithm 5), and then use the remaining five TRGSW ciphertexts to move the 32 desired output result to the first 32 coefficients of the plaintext polynomial by calling BlindRotation (line 8 of Algorithm 5). Note that since the output of $S1\_T0$ table is 32 bits, the second parameter setting in the BlindRotation algorithm must be adjusted to a multiple of 32 accordingly. Finally, we use the SampleExtract to extract the TLWE ciphertexts of the first 32 coefficients, which is exactly what we want, and IdentityKeySwitching back to Level 0.

Compared to Algorithm 3, the number of CMux gates in Algorithm 5 increases from 8 to 12, but the increased cost is almost negligible. The most important reason for the

speedup of Algorithm 5 is that it ensures that the number of circuit bootstrappings does not increase, while eliminating a large number of gate bootstrappings required by the MULxPow function which recursively calls MULx function.

5 Homomorphic Evaluation of ZUC

ZUC is a word-oriented stream cipher like SNOW 3G. It takes as input a 128-bit initial key and a 128-bit initial vector (IV) and outputs a keystream consisting of 32 bits, which can be used for encryption/decryption. The ZUC is executed in two stages: the initialization stage shown in Figure 11 and the working stage (*i.e.*, keystream generation) shown in Figure 4. In the first phase, the initialization of the key/IV is performed to update the LFSR, *i.e.*, the cipher is clocked without generating an output. In the second phase, it generates a 32-bit word for each clock pulse.

The ZUC has three logic layers. The top layer is a linear feedback shift register (LFSR) containing 16 stages. Unlike the SNOW 3G, the LFSR of ZUC has 16 31-bit cells, and each cell s_i, for $0 \leq i \leq 15$, is taken from $GF(2^{31}-1)$. The middle layer is used for bit-reorganization, and the bottom layer is a nonlinear function F. For more details, please refer to Appendix D. Here we focus on the keystream generation of ZUC.

5.1 Keystream Generation of ZUC

After the Initialization stage, the algorithm enters the working stage. During the working stage, the algorithm performs the following operations once:

- BitReconstruction();
- $W = F(X_0, X_1, X_2)$, discard W;
- LFSRWithworkMode().

Lastly, the algorithm enters the keystream generation phase, each iteration performs the following operations to generate a 32-bit word Z as output:

1. BitReorganization();
2. $Z = F(X_0, X_1, X_2) \oplus X_3$;
3. LFSRWithworkMode().

5.2 The Linear Feedback Shift Register (LFSR)

First let's look at the LFSR. LFSR has two running modes: initialization mode and working mode, which are used in the initialization phase and work phase, respectively. The core operation of these two modes is the following equation:

$$v = 2^{15} \cdot s_{15} + 2^{17} \cdot s_{13} + 2^{21} \cdot s_{10} + 2^{20} \cdot s_4 + (1 + 2^8) \cdot s_0 \bmod (2^{31} - 1) \quad (*)$$

Unlike SNOW 3G, the functions in Equation$(*)$ consists of modular multiplication and modular addition over $GF(2^{31}-1)$. How to efficiently compute modulo $GF(2^{31}-1)$ in the homomorphic context is a challenge. We propose the following solution idea based on the gate bootstrapping mode.

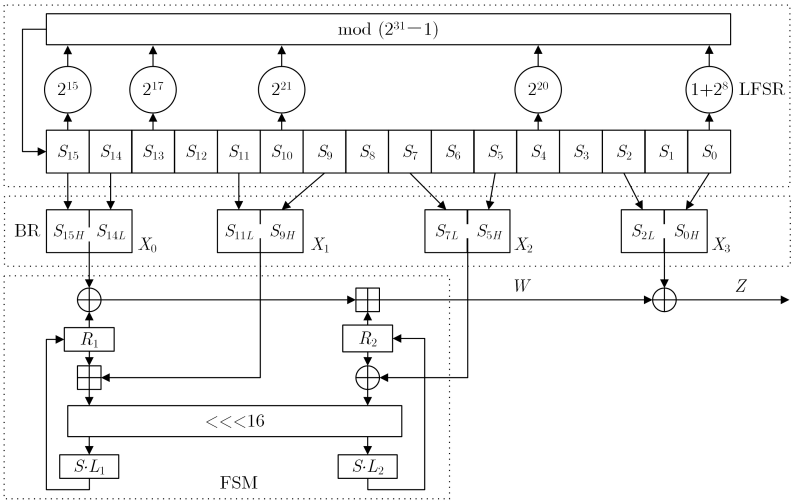

Figure 4: Keystream generation of ZUC. The picture is taken from [Tea21]

5.2.1 Evaluation of Modular Multiplication over $GF(2^{31}-1)$

Updating the LFSR involves multiplying a 31-bit string s by 2^i over $GF(2^{31}-1)$, which can be achieved by a cyclic shifting s by i bits to the left:

$$a \cdot 2^k \bmod (2^{31}-1) = a \lll_{31} k \bmod (2^{31}-1).$$

Therefore, the evaluation of this modular multiplication is free due to the bit-wise encryption.

5.2.2 Evaluation of the Modular Addition over $GF(2^{31}-1)$

For two elements a, b over $GF(2^{31}-1)$, the modular addition of $c = a + b \bmod (2^{31}-1)$ operation can be computed using the following two steps:

$$\begin{aligned}&(1)\ v = a \boxplus b, \text{where } v \text{ is a 32-bit value;}\\ &(2)\ c = (v\ \&\ 0x7fffffff) \boxplus (v \gg 31).\end{aligned}$$

Firstly, we compute the addition of a and b using the 32-bit adder. Since $(v\ \&\ 0x7fffffff)$ is the least significant 31 bits of 32-bit v and $(v \gg 31)$ is the most significant bit of 32-bit c, both of them are directly obtained in gate bootstrapping mode. In total, the evaluation of modular addition over $GF(2^{31}-1)$ just needs two $\boxplus$ gates.

Notice that in the Equation(*), we need to perform successive additions. If each addition is followed by a modulo operation, 6 modular additions are required for one update v, *i.e.*, 12 $\boxplus$ adders. In order to optimize this special modular addition operation, we design a new modular addition strategy based on tree structure: we use 31-bit adder, 32-bit adder and 33-bit adder sequentially to realize arithmetic addition, and finally execute only one modular addition $GF(2^{31}-1)$, thus this first-add-then-modulo operation method consumes only 6 $\boxplus$ gates in total. Figure 5 illustrates this evaluation idea.

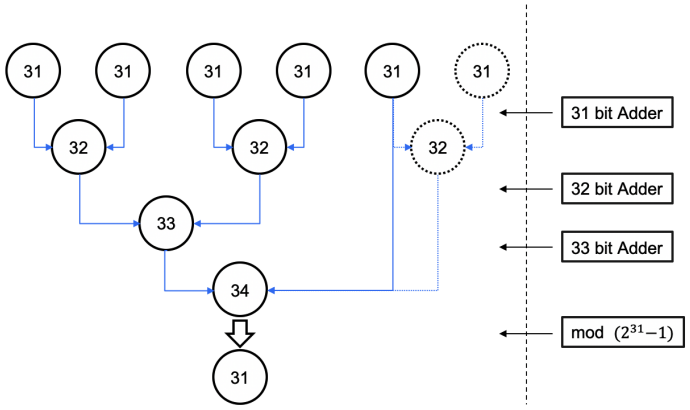

Figure 5: Optimization of modular addition in initialization mode/working mode.

One thing that must be noted is that our proposed single-gate-bootstrapping for full adder is not suitable for LFSR in ZUC. The reason is that its update requires the addition of the previous cells, which would cause the accumulation of noise, so we choose to use 2BR Full Adder algorithm to evaluate $\boxplus$ operation.

5.3 BitReorganization and Nonlinear Function F

BitReorganization extracts 128 bits from the cells of the LFSR to form four 32-bit words, the first three of which will be used by the nonlinear function F, while the last word will be involved in generating the keystream. This operation can be evaluated for free due to bit-wise encryption.

The nonlinear function F is used to update the two 32-bit memory cells R_1 and R_2. The F function contains modular addition $\boxplus$, S-box and linear transformations L, which can all be evaluated using the method presented in Section 3.

6 Homomorphic Evaluation of AES via Thunderbird

6.1 Specification of AES-128

AES-128 is a variant of AES with a 128-bit key that operates on a 128-bit plaintext message (16 bytes), which is represented as a state matrix. The AES encryption process consists of multiple rounds (in this case, 10 rounds for AES-128), and each round consists of four operations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. In detail,

- **SubBytes**: It transforms each element of state matrix non-linearly using 8-bit-to-8-bit S-box. SubBytes is the only nonlinear function operation of AES.
- **ShiftRows**: In a 4×4 state matrix, the first row is held still and the second row's elements are shifted cyclically to the left by one byte. Similarly, the elements of the third row are cycled two bytes to the left, and the elements of the fourth row are cycled three bytes to the left.
- **MixColumns**: A linear mixing operation that operates on the state matrix column-by-column.
- **AddRoundKey:** It refers to performing bitwise XOR of state matrix with the current round key which are generated by the key schedule in advance.

As shown in Figure 6, these operations are repeatedly applied to the state matrix to achieve encryption. Note that there is one more AddRoundKey operation before the start of the first round. In the last round, no MixColumns operation is performed.

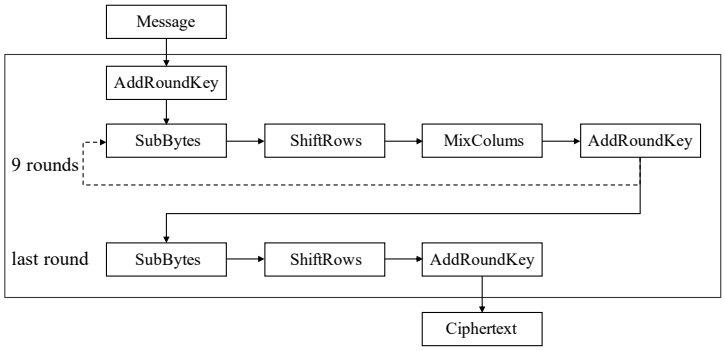

Figure 6: The original encryption process for AES.

We note that Rijndael's designers also proposed a fast lookup table (LUT) method for AES implementation, which we present in Figure 7. The core idea of the lookup table method is to merge the SubBytes, ShiftRows and MixColumns three operations into 8-to-32-bit lookup tables, which are commonly known as T-Box or T-Table. The encryption process is 4 8-to-32-bit tables (T_e) and the decryption process is 4 8-to-32-bit tables (T_d). Each round of AES is generated by 16 lookup tables. For more details we refer readers to [DR13].

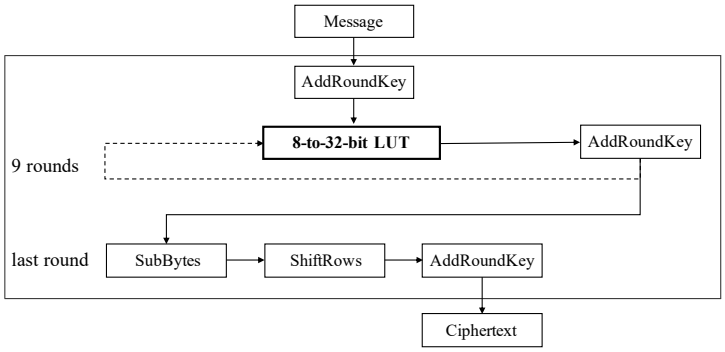

Figure 7: AES-128 encryption implementation using 8-to-32-bit lookup table.

6.2 Homomorphic Evaluation of LUT-based AES

In this subsection, we show that AES can be also efficiently evaluated via the Thunderbird framework. The AES encryption algorithm is designed based on Substitution-Permutation Networks (SPN). Unlike SNOW and ZUC, the only remaining function type in the AES algorithm, besides lookup tables, is the simple XOR operation. Recently, Wei et al. [WWL+23] proposed an AES homomorphic evaluation using circuit bootstrapping for the first time. In detail, they homomorphically evaluate the standard implementation steps of AES. For SubBytes, they evaluate 8-bit-to-8-bit Sbox lookup table computation using CMux gates; for MixColumns, they convert all scalar multiplications to XOR and bit shiftings operations. For example, suppose that a byte (from the lowest bit to the highest bit) is represented as $(b_0b_1b_2b_3b_4b_5b_6b_7)$, then

$$b_0b_1b_2b_3b_4b_5b_6b_7 \times 02 = b_7b_0b_1b_2b_3b_4b_5b_6 \oplus 0b_70b_7b_7000.$$

In addition, by encoding the message $\{0,1\}$ as $\{0,1/2\}$ over Torus, XOR operation can be performed for free, i.e., freeXOR. It is worth noting that since the message space is the entire Torus, in order to satisfy the negacyclic property ($X^{i+N} = -X^i \bmod X^N+1$) required by the test polynomial in PBSmanyLUT, the test polynomial should be modified to

$$P(X) = \sum_{i=0}^{\frac{N}{2^\rho \cdot 2}-1} \sum_{j=0}^{2^\rho-1} (-1) \cdot \frac{1}{2B_g^j} X^{2^\rho \cdot i + j} + \sum_{i=\frac{N}{2^\rho \cdot 2}}^{\frac{N}{2^\rho}-1} \sum_{j=0}^{2^\rho-1} \frac{1}{2B_g^j} X^{2^\rho \cdot i + j}, \text{ where } \rho = \lceil \log_2 \ell \rceil.$$

For details of the proof we refer the readers to Lemma 4 in [CLOT21].

Different from [WWL+23], we evaluate the LUT-based implementation of AES, which achieves lower evaluation latency. Our efficiency improvement comes from two tricks: Firstly, by evaluating the 8-to-32-bit table, the lookup table results are rotated to the first 32 coefficients of the corresponding plaintext polynomial of TRLWE, so that we can perform all the next XOR operations (including XOR with the round key) directly on resulting TRLWE ciphertexts on Level 1 rather than TLWE ciphertexts on Level 0. This has the advantage of requiring fewer XOR operations per round compared to [WWL+23]. In the homomorphic context this reduces the accumulation of noise within the ciphertext after each round of updates, ultimately resulting in a relatively low decryption failure rate for homomorphic ciphertexts. In particular, for symmetric key encrypted by homomorphic encryption as evaluation key, the client can reduce the online transmission ciphertext size by sequentially packing the round keys as follows:

$$\underbrace{m_0 + \cdots + m_{127}X^{127}}_{rk_0} + \underbrace{m_{128}X^{128} + \cdots + m_{255}X^{255}}_{rk_1} + \underbrace{\cdots + m_{N-1}X^{N-1}}_{rk_{\frac{N}{128}-1}}$$

When the server requires the corresponding round key, it efficiently obtains them by rotating these TRLWE ciphertexts encrypting the round key. This rotation involves multiplying it by X^{-i} to position the desired round key at the first 32 coefficients. Secondly, as in Section 4.4, we can utilize the CMux gate to give an efficient 8-to-32-bit lookup table implementation. Although 8-to-32-bit table outputs more bits than 8-to-8-bit table, in our evaluation framework, the number of inputs is the bottleneck of the overall evaluation because it determines the number of circuit bootstrappings. Therefore, the overall evaluation efficiency of AES would be accelerated thanks to the optimization of the second step of circuit bootstrapping in Section 4.2. Figure 8 illustrates the efficient evaluation process of LUT-based AES.

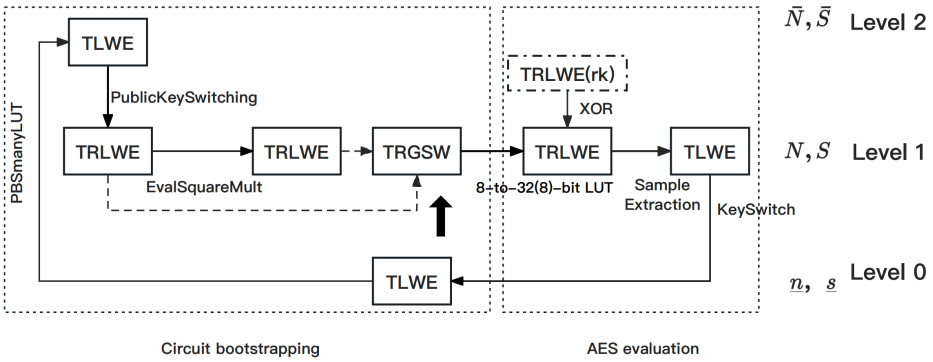

Figure 8: Efficient homomorphic evaluation of AES via LHE mode

Remark 1. Considering the purpose of the hybrid homomorphic encryption framework, i.e., the conversion of AES ciphertexts to TLWE ciphertexts to support homomorphic

computation of functions, although the above mentioned framework in Figure 8 supports efficient AES homomorphic evaluation, this can only support a limited number of function operations due to the message encoding $\{0,1\} \to \{0,1/2\}$. If we want to support more circuits using gate bootstrapping, we can adopt the Thunderbird framework construction, i.e., implement XOR computation using HomoXOR at Level 0, the disadvantage of which is that it will reduce the efficiency of transciphering due to the use of a large number of gate bootstrappings.

Remark 2. Compared to BGV-based method [GHS12], our framework gets rid of the limitation of circuit depth. Therefore, even AES-192 and AES-256 versions, can be also evaluated efficiently using our evaluation mode. This provides the possibility of real-world applications of AES for transciphering.

7 Implementation and Performance

In this section, we present our experimental results and efficiency analysis. Circuit bootstrapping was first implemented as a proof of concept[1], however, it is not compatible with the original TFHE library[2]. Matsuoka et al. [MBM+21] presented the Virtual Security Platform to implement a multi-opcode generic sequential processor on Fully Homomorphic Encryption (FHE) for Secure Multiparty Computing (SMPC). And they implemented TFHEpp library, a Scratch C++ implementation of TFHE on CPU. It supports a variety of homomorphic gate operations, in addition to efficient circuit bootstrapping. Therefore, we choose to test the efficiency and accuracy of our proposed evaluation framework in the TFHEpp library.

Our test environment is Intel(R) Core(TM) i5-11500 CPU @ 2.70GHz and 32 GB of RAM, running the Ubuntu 20.04 operating system.

7.1 Parameter Sets

Notice that our evaluation framework Thunderbird combining gate bootstrapping and circuit bootstrapping spans three levels, where the parameters of Level 0 and Level 1 are involved in gate bootstrapping and the parameters of Level 2 are used to serve for circuit bootstrapping, which requires larger parameters for accommodating noise, *e.g.*, $\bar{Q} = 2^{64}$ for torus representation on Level 2. We choose experimental parameter sets with a security level λ at least 128-bit, which are taken from TFHEpp library, as shown in Table 2.

7.2 Performance Results and Analysis

In this subsection, we will give the experimental results based on the above parameter sets. It is important to emphasize that all our tests use only **a single core**.

7.2.1 Performance of Full Adder

We first give the some benchmarks for testing full adder on TFHEpp library based on the above parameters in Table 3. Each operation is tested 1000 times and then the average time is taken.

7.2.2 Performance of Ciphertext Conversion

We first summarize several implementation methods for T(R)LWE-to-TRGSW ciphertext conversion, presented in Table 4. Notice that the goal of TRLWE-to-TRGSW is to reduce

[1] https://github.com/tfhe/experimental-tfhe
[2] https://github.com/tfhe/tfhe

Table 2: Parameter sets on different levels used for the Thunderbird framework.

Level	Parameter Sets
Level 0	Dimension of TLWE: $\underline{n} = 635$ Standard deviation of noise: $\underline{\alpha} = 2^{-15}$
Level 1	Dimension of ring polynomial for TRLWE: $N = 1024$ Length of gadget decomposition for TRGSW: $\ell = 3$ Basis of gadget decomposition for TRGSW: $B_g = 2^6$ Standard deviation of noise: $\alpha = 2^{-25}$
Level 2	Dimension of ring polynomial for TRLWE: $\bar{N} = 2048$ Length of gadget decomposition for TRGSW: $\bar{\ell} = 4$ Basis of gadget decomposition for TRGSW: $\bar{B_g} = 2^9$ Standard deviation of noise : $\bar{\alpha} = 2^{-44}$
Level 1 $\rightarrow$ 0	Length of the digit decomposition during key switching: $t = 7$ Basis of the digit decomposition during key switching: $B_g = 2^2$
Level 2 $\rightarrow$ 1	Length of the digit decomposition during key switching: $\bar{t} = 10$ Basis of the digit decomposition during key switching: $\bar{B_g} = 2^3$

Table 3: Comparison of message encoding and running time of the state-of-the-art and our variant of full adder based on TFHE.

Method	2BR FA [MHSB21]	1BR FA [MHSB21]	Ours
Message encoding	$\{0,1\} \rightarrow \{-\frac{1}{8}, \frac{1}{8}\}$	$\{0,1\} \rightarrow \{-\frac{1}{12}, \frac{1}{12}\}$	$\{0,1\} \rightarrow \{-\frac{1}{8}, \frac{1}{8}\}$
Time [ms]	18	9	9

the transmission cost to the client due to the large ciphertext size of TRGSW. In this paper, we focus on circuit bootstrapping, i.e., TLWE-to-TRGSW ciphertext conversion. Based on the parameter set in Table 2, we test the execution time of circuit bootstrapping under different approaches, and present the relative time of functional bootstrapping with TLWE-to-TRLWE ciphertext conversion in Figure 9.

Table 4: Comparison of the previous T(R)LWE-to-TRGSW method with our fine-tuning.

	Ciphertext conversion	Method
Chillotti et al. [CGGI17]	TLWE-to-TRGSW	ℓ FBS + ℓ PublicKS + ℓ PrivateKS
Chen et al. [CCR19]	T**R**LWE-to-TRGSW	Trace function evaluation+ External Multiplication
Chillotti et al. [CJP21]	TLWE-to-TRGSW	PBSmanyLUT + ℓ PublicKS + ℓ PrivateKS
Guimarães et al. [GBA22]	TLWE-to-TRGSW	PBSmanyLUT + ℓ PublicKS(Level 2) + ℓ EvalSquareMult(Level 2)
Kim et al. [KLD+23]	T**R**LWE-to-TRGSW	Trace function evaluation+ + EvalSquareMult
Ours	TLWE-to-TRGSW	PBSmanyLUT + ℓ PublicKS(Level 1) + ℓ EvalSquareMult(Level 1)

Experimental results show that our circuit bootstrapping can be reduced to 36ms,

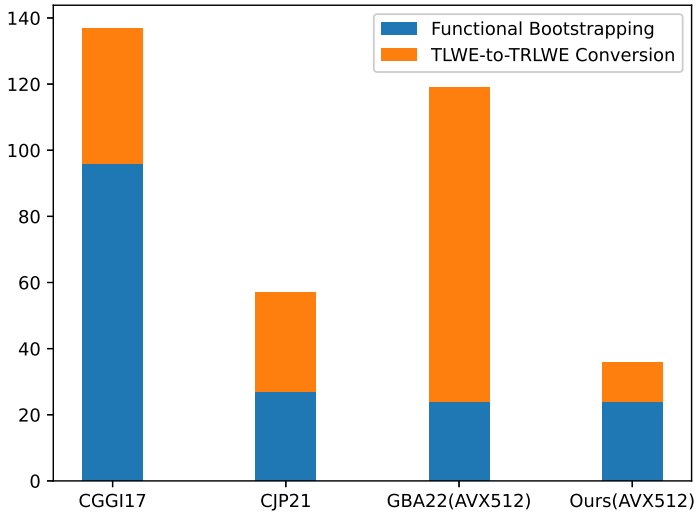

Figure 9: The detailed proportion of each operation in the previous circuit bootstrapping and our adaption to circuit bootstrapping.

where the first step, that is, the functional bootstrapping over Level 2 takes about 24ms, and the second step, that is, ciphertext conversion consumes about 12ms. This greatly accelerates our evaluation framework.

7.2.3 Benchmarks of Transciphering using Standardized Ciphers in 3GPP

We now first present the test benchmarks of two standardized stream ciphers SNOW 3G and ZUC when used in transciphering framework, as shown in Table 5.

Table 5: Running time [second] for stream cipher SNOW 3G and ZUC initialization and keystream generation of per block based on TFHE.

	SNOW 3G		ZUC	
stage / mode	Initialization	KeyStreamGen	Initialization	KeyStreamGen
Gate boostrapping	12160	367	9581	301
Thunderbird	217 (56×)	7 (52×)	332 (28×)	9.5 (32×)

Performance Analysis Now let's explain the data in Table 5. In fact, we are mainly concerned with the keystream generation time, which can of course be computed *offline* in the hybrid homomorphic encryption framework using stream ciphers. Since the two algorithms SNOW 3G and ZUC are implemented for the first time, we first give a simple implementation based on gate bootstrapping mode. In gate bootstrapping mode, the keystream generation time mainly comes from S-box evaluation and linear feedback shift register. Although TFHE supports efficient gate bootstrapping, it still causes huge latency due to the high usage of a large number of gates. In Thunderbird, we implement an CMux-based efficient lookup table by combining with hybrid packing, which is almost

negligible. SNOW 3G and ZUC are accelerated by a factor of 52 and 32, respectively, compared to gate bootstrapping mode. Note that the computational cost of the S-box evaluation is shifted to circuit bootstrapping. The reason that SNOW 3G has higher latency than ZUC using gate bootstrapping is that it still uses more gates to evaluate SB_1/SB_2, MUL_α and DIV_α. Thunderbird, on the other hand, replaces a large number of MULx functions with lookup tables, thus increasing the speed of gate bootstrapping mode by a factor of 52, and outperforming ZUC in latency instead. It is clear that Thunderbird also significantly reduces the latency of the initialization process both for SNOW 3G and ZUC, thus reducing the computational burden on cloud services.

7.2.4 Comparing to FHE-friendly Stream Ciphers

Given that our transciphering goal is to use standardized stream ciphers, we would present a comparison with standardized stream ciphers that have been evaluated, such as Trivium [BOS23]. Further for completeness, we will extend the comparison to include newly designed FHE-friendly ciphers, considering aspects such as security, input type, and efficiency, as shown in Table 6. Note that since these symmetric schemes were tested with different FHE libraries such as Concrete[1], TFHE-rs[2], HElib[3], SEAL[4], Lattigo[5], and FINAL[6], care must be taken when comparing their performance.

Table 6: Comparison of running time [ms] of different FHE-friendly ciphers with 128-bit security. Notice that for Kreyvium and Trivium algorithms, the authors do not provide running time under a single thread, we present a more reasonable estimate, which is labeled using asterisks.

Cipher	Input	Library	Latency	Time per bit	Standardized
FiLIP-144 [CHMS22]	$\mathbb{Z}_2$	Concrete	134	134	×
FiLIP-1216 [CHMS22]	$\mathbb{Z}_2$	Concrete	586	586	×
FiLIP-1280 [CHMS22]	$\mathbb{Z}_2$	Concrete	627	627	×
FiLIP-144 [CDPP22]	$\mathbb{Z}_2$	FINAL	2.62	2.62	×
Elisabeth-4(2 KS) [CHMS22]	$\mathbb{Z}_{2^q}$	Concrete	1485	371.25	×
Elisabeth-4(1 KS) [CHMS22]	$\mathbb{Z}_{2^q}$	Concrete	1648	412.15	×
Pasta-3 [DGH+23]	$\mathbb{F}_p$	SEAL	9280	4.53	×
HERA [HKL+22]	$\mathbb{F}_p$	Lattigo	141580	0.024	×
Rubato [HKL+22]	$\mathbb{F}_p$	Lattigo	106400	0.018	×
Kreyvium [BOS23]	$\mathbb{Z}_2$	TFHE-rs	19200*	300*	×
Trivium [BOS23]	$\mathbb{Z}_2$	TFHE-rs	15488*	242*	√ (80-bit)
SNOW3G	$\mathbb{Z}_2$	TFHEpp	7000	218.75	√
ZUC	$\mathbb{Z}_2$	TFHEpp	9500	296.88	√

To the best of our knowledge, the current $\mathbb{Z}_{2^q}$ stream cipher designed for TFHE's

[1] https://github.com/zama-ai/concrete
[2] https://github.com/zama-ai/tfhe-rs
[3] https://github.com/homenc/HElib
[4] https://github.com/microsoft/SEAL
[5] https://github.com/tuneinsight/lattigo
[6] https://github.com/KULeuven-COSIC/FINAL

functional bootstrapping is Elisabeth, of which the authors provide two versions, Elisabeth with two KS and single KS. In terms of amortization efficiency, both SNOW 3G and ZUC outperform the Elisabeth-4 cipher. For the $\mathbb{Z}_2$ cipher FiLIP, SNOW 3G and ZUC have better amortization than FiLIP-1216 and FiLIP-1280, but slower than FiLIP-144, which were given in [CHMS22]. Note that in [CDPP22], they greatly improved the latency of FiLIP-144 to only 2.62ms per bit, based on the FINAL homomorphic encryption scheme and using freeXOR gate to remove all the cost of the XOR gates. However, as we have previously noted, FiLIP ciphers currently require a longer period of security analysis than standard ciphers.

For $\mathbb{F}_p$ ciphers, Pasta is the state-of-the-art cipher that demonstrates the advantages of application-specific matrix-vector multiplication. In general, these ciphers are better suited to be evaluated in homomorphic libraries supporting SIMD packing techniques, such as HElib and SEAL, which achieve higher throughput but lead to long latency. Two symmetric ciphers designed for CKKS, HERA and Rubato, show the best amortization compared to all current FHE-friendly ciphers, taking only microseconds per bit. Similarly, these ciphers have longer evaluation delays.

The biggest advantage of standardized ciphers in the HHE framework is security and reliability. This is our starting point for this work. The standardized algorithm Trivium in [BOS23] gives an amortized implementation of 1.89ms per bit. However, we note that they use 128 CPUs to accelerate the evaluation. Roughly speaking, Trivium's amortization per bit is around 242ms if using only a single thread. It can be seen that SNOW 3G's amortization time is better than Trivium's. Also, Trivium's design security is *only* 80-bit, which seems to be difficult to meet the needs of real world applications. Therefore, if someone wants to consider using a standard stream cipher with 128-bit security instead of the FHE-friendly cipher in HHE, we recommend the SNOW 3G cipher.

7.3 Key Size

In the transciphering scenario, the evaluation key must be generated by the client and sent to the server. Here, we briefly analyze the key size used in the Thunderbird framework, as shown in Table 7.

Table 7: Key size used in the Thunderbrid framework.

Key Type	Size
KeySwitching Key used for switching from Level 1 to 0	69.44 MB
PublicKeySwitching Key used for switching from Level 2 to 1	1280 MB
KeySwitching Key for EvalSquareMult	24 MB
Bootstrapping Key used for gate bootstrapping	29.8MB
Bootstrapping Key used for PBSmanyLUT	158.75 MB
Total evaluation key	1538 MB

To summarize, our transciphering framework requires evaluation key size of about 1538 MB. Recalling that since the initial key needs to be encrypted to TLWE homomorphic ciphertext, the client also needs to transmit ciphertexts with the size of 128*2.48 = 317.44 KB. Of course, LWE ciphertext transmission can use the seed-based compression technique proposed in [CDKS21], which can be reduced to only 0.5 KB.

7.4 Comparison of Evaluation of AES with the State-of-the-art

Homomorphic computation of standardized AES is one of the most frequently investigated and attractive work in transciphering, and evaluation strategies based on various fully homomorphic encryption algorithms have been continuously proposed, such as BGV, CKKS

and TFHE. In this subsection, we focus on comparing the homomorphic computation of AES with the state-of-the-art work, as shown in Table 8.

Table 8: Comparison of AES-128 evaluation latency based on different fully homomorphic encryption schemes and different evaluation modes using a single core. It should be noted that the data with asterisks were obtained by utilizing multi-threaded parallel techniques, 88 threads and 16 threads were used in [ADE+23] and [SMK22] respectively.

Scheme	Evaluation mode	Latency	Time per block
BGV	leveled [GHS12]	4 mins	2 s
	bootstrapped[GHS12]	18 mins	6 s
CKKS	bootstrapped [ADE+23]	31 mins*	56.7 ms*
TFHE	FBS [SMK22]	4.2 mins*	4.2 mins*
	FBS [TCBS23]	270 s	211 s
	FBS [BPR23]	211 s	211 s
	LHE [WWL+23]	86 s	86 s
	Our1(LHE+HomoXOR)	**110** s	**110** s
	Our2(LHE+freeXOR)	**46** s	**46** s

Since both the BGV and CKKS scheme support efficient SIMD packing technique, [GHS12, ADE+23] showed the best amortization efficiency when evaluating AES. However, they have higher latency, which is not friendly for some scenarios requiring low latency. The TFHE scheme supports efficient functional bootstrapping, so it outperforms BGV and CKKS in terms of evaluation latency with respect to a single AES block. For homomorphic evaluation of AES based on our Thunderbird evaluation framework, when the XOR gates are used with the HomoXOR gate, i.e., $\{0,1\} \to \{-1/8, 1/8\}$ over torus, evaluating a single AES block takes about 110 seconds, with the computational cost coming from two components: first, the relatively expensive circuit bootstrapping, and second, the large number of gate bootstrappings. This is 2x faster than the current best implementation based on functional bootstrapping [TCBS23, BPR23]. When we use freeXOR, i.e., $\{0,1\} \to \{0,1/2\}$ over torus, the cost mainly comes from circuit bootstrapping, and the latency of a block under a single core is only 46 seconds, which is about a $2\times$ improvement over the current implementation based on circuit bootstrapping. Also compared to [WWL+23], we use less XOR gates due to the 8-to-32-bit lookup table, which leads to a lower decryption error rate. In terms of symmetric key transfer, our evaluation has lower ciphertext size. Our homomorphic AddRoundkey can be performed on the TRLWE ciphertext, so the client encrypts the round keys by simply encrypting them as $\lceil \frac{11}{N/128} \rceil = 2$ TRLWE ciphertexts, not $128 * 11$ TLWE ciphertexts. In other words, we can reduce the homomorphic ciphertext size of the round key from 3492 KB to 16 KB if we don't consider the seed-based compression. Therefore, if someone wants to utilize AES as a symmetric algorithm for transcryption, especially certain companies that make security and latency their first priority, our AES homomorphic evaluation method is the current optimal choice.

Simple Comparison of Evaluation Key Size In [GHS12], the authors give the memory consumption, which are 3.7 GB and 3 GB for bootstrapping and non-bootstrapping versions, respectively. In [TCBS23], the authors do not give the key size explicitly. They gave fast AES evaluation based on functional bootstrapping. However, note that in order to use tree-based lookup table technique, LWEs-to-RLWE packing must be utilized to support

the next functional bootstrapping, which requires larger key-switching keys, typically up to several GBs, e.g., in [GBA21], the key size is up to 4G or more. Thus, our approach still outperforms them in terms of total key size.

7.5 Discussions

The purpose of transciphering is to avoid transmitting large size ciphertexts online. Since a server typically has more computational power than a client, it can effectively use multiple cores to parallelize computation to optimize execution time. Our evaluation framework has natural parallelism:

- **Parallelization** Lookup table parallel computation: circuit bootstrapping is computationally expensive for lookup table evaluation, but their execution in the Thunderbird framework have natural parallelism. For example, AES requires 128 paralleled circuit bootstrappings to be executed per round to perform lookup tables, and if the server has enough cores, we can get at least an order of magnitude more boost. For example, Trama et al. [TCBS23] used the OpenMP library to parallelize and optimize the execution time of homomorphic AES, which was 9× faster using 16 threads than using 1 thread. This implies that with multithreading support, the latency of homomorphic computation of SNOW 3G, ZUC generation of a keyword and an AES block based on our framework is expected to be done at the millisecond level.

  Moreover, instead of using RCA, we can consider utilizing parallel adder such as Carry LookAhead Adder (CLA) or Parallel Prefix Adder (PPA), which further reduces the latency under multi-threading setting.

- **Scalability:** We believe that the Thunderbird framework can be extended to be applied to other SPN-structured ciphers, especially those that use 8-to-8-bit Sboxes, such as the Chinese block cipher encryption standard SM4. Thunderbird-based SM4 evaluation can directly bring 2× speedup compared to [WWL+23]. For other FHE-friendly symmetric algorithms, such as LowMC, Masta, Rasta, Pasta, Chaghri, although they also utilize SPN structures, these ciphers use relatively small-sized Sboxes in order to reduce the computational complexity. Therefore, if Thunderbird is used, the performance may not be as good as the original method.

- **Potential applications:** We hope that Thunderbird can be applied to other scenarios, such as privacy-preserving machine learning, but this may require special models. For example, in [BGPS23] they propose a neural network model based on lookup tables. We hope that our evaluation program will accelerate the evaluation of such models.

8 Conclusion

The hybrid homomorphic encryption framework solves the ciphertext size expansion problem of fully homomorphic encryption by introducing symmetric encryption schemes. In this paper, we investigate the possibility of applying the standardized symmetric encryption algorithms in 3GPP to hybrid homomorphic encryption: SNOW 3G, ZUC, and AES. Specifically, we propose the Thunderbird evaluation framework, which combines the TFHE's leveled evaluation mode and gate bootstrapping mode to address the different function types involved in symmetric encryption algorithms. As a result, our experimental results further promote the application of standardized algorithms in real-world scenarios. We also believe that our evaluation strategies would also provide some new ideas for designing new FHE-friendly symmetric encryption algorithms.

Open Problems: Circuit bootstrapping is a computational bottleneck in our evaluation mode, so how to further improve its efficiency is one of the open problems in FHE research. One possible direction is to optimize PublicKeySwitching using the method of Chen et al. [CDKS21]. However, it requires $N^{-1} \bmod Q$ to exist and can only be accelerated using NTT. We would explore this in our future work. Another possible direction is to construct more efficient circuit bootstrapping based on the NTRU assumption to further optimize our evaluation framework.

Acknowledgments

We are very grateful to the anonymous reviewers for their helpful comments. This work was supported by the Huawei Technologies Co., Ltd and CAS Project for Young Scientists in Basic Research (Grant No. YSBR-035).

References

[ADE+23] Ehud Aharoni, Nir Drucker, Gilad Ezov, Eyal Kushnir, Hayim Shaul, and Omri Soceanu. E2E near-standard and practical authenticated transciphering. *IACR Cryptol. ePrint Arch.*, page 1040, 2023.

[AMT22] Tomer Ashur, Mohammad Mahzoun, and Dilara Toprakhisar. Chaghri - A fhe-friendly block cipher. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022*, pages 139–150. ACM, 2022.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *Advances in Cryptology - EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 430–454. Springer, 2015.

[BBS21] Adda-Akram Bendoukha, Aymen Boudguiga, and Renaud Sirdey. Revisiting stream-cipher-based homomorphic transciphering in the TFHE era. In *Foundations and Practice of Security - 14th International Symposium, FPS 2021*, volume 13291 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2021.

[BCBS23] Adda-Akram Bendoukha, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. Optimized stream-cipher-based transciphering by means of functional-bootstrapping. In *Data and Applications Security and Privacy XXXVII, DBSec 2023*, volume 13942 of *Lecture Notes in Computer Science*, pages 91–109. Springer, 2023.

[BCK+23] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, Jai Hyun Park, and Damien Stehlé. HERMES: efficient ring packing using MLWE ciphertexts and application to transciphering. In *Advances in Cryptology - CRYPTO 2023*, volume 14084 of *Lecture Notes in Computer Science*, pages 37–69. Springer, 2023.

[BGG18] Christina Boura, Nicolas Gama, and Mariya Georgieva. Chimera: a unified framework for B/FV, TFHE and HEAAN fully homomorphic encryption and predictions for deep learning. *IACR Cryptol. ePrint Arch.*, page 758, 2018.

[BGGJ19] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Simulating homomorphic evaluation of deep learning predictions. In *Cyber Security Cryptography and Machine Learning, CSCML 2019*, volume 11527 of *Lecture Notes in Computer Science*, pages 212–230. Springer, 2019.

[BGPS23] Adrien Benamira, Tristan Guérand, Thomas Peyrin, and Sayandeep Saha. TT-TFHE: a torus fully homomorphic encryption-friendly neural network architecture. *CoRR*, abs/2302.01584, 2023.

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

[BOS23] Thibault Balenbois, Jean-Baptiste Orfila, and Nigel P. Smart. Trivial transciphering with trivium and TFHE. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Copenhagen, Denmark, 26 November 2023*, pages 69–78. ACM, 2023.

[BPR23] Nicolas Bon, David Pointcheval, and Matthieu Rivain. Optimized homomorphic evaluation of boolean functions. Cryptology ePrint Archive, Paper 2023/1589, 2023. https://eprint.iacr.org/2023/1589.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in Cryptology - CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.

[CCF+16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In *Fast Software Encryption - 23rd International Conference, FSE 2016*, volume 9783 of *Lecture Notes in Computer Science*, pages 313–333. Springer, 2016.

[CCR19] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: efficient constant bandwidth oblivious RAM from (leveled) TFHE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019*, pages 345–360. ACM, 2019.

[CDKS21] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. In *Applied Cryptography and Network Security - 19th International Conference, ACNS 2021*, volume 12726 of *Lecture Notes in Computer Science*, pages 460–479. Springer, 2021.

[CDPP22] Kelong Cong, Debajyoti Das, Jeongeun Park, and Hilder V. L. Pereira. Sortinghat: Efficient private decision tree evaluation via homomorphic encryption and transciphering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022*, pages 563–577. ACM, 2022.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *Advances in Cryptology - ASIACRYPT 2017*, volume 10624 of *Lecture Notes in Computer Science*, pages 377–408. Springer, 2017.

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

[CHK19] Jung Hee Cheon, Kyoohyung Han, and Duhyeong Kim. Faster bootstrapping of FHE over the integers. In *Information Security and Cryptology - ICISC 2019*, volume 11975 of *Lecture Notes in Computer Science*, pages 242–259. Springer, 2019.

[CHK+21] Jihoon Cho, Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. Transciphering framework for approximate homomorphic encryption. In *Advances in Cryptology - ASIACRYPT 2021*, volume 13092 of *Lecture Notes in Computer Science*, pages 640–669. Springer, 2021.

[CHMS22] Orel Cosseron, Clément Hoffmann, Pierrick Méaux, and François-Xavier Standaert. Towards case-optimized hybrid homomorphic encryption - featuring the elisabeth stream cipher. In *Advances in Cryptology - ASIACRYPT 2022*, volume 13793 of *Lecture Notes in Computer Science*, pages 32–67. Springer, 2022.

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In *Topics in Cryptology - CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 106–126. Springer, 2019.

[CIR22] Carlos Cid, John Petter Indrøy, and Håvard Raddum. FASTA - A stream cipher for fast FHE evaluation. In *Topics in Cryptology - CT-RSA 2022*, volume 13161 of *Lecture Notes in Computer Science*, pages 451–483. Springer, 2022.

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *Cyber Security Cryptography and Machine Learning, CSCML 2021*, volume 12716 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2021.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *Advances in Cryptology - ASIACRYPT 2017*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[CLOT21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In *Advances in Cryptology - ASIACRYPT 2021*, volume 13092 of *Lecture Notes in Computer Science*, pages 670–699. Springer, 2021.

[CLT14] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In *Public-Key Cryptography - PKC 2014*, volume 8383 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 2014.

[DEG+18] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. Rasta: A cipher with low anddepth and few ands per bit. In *Advances in Cryptology - CRYPTO 2018*, volume 10991 of *Lecture Notes in Computer Science*, pages 662–692. Springer, 2018.

[DGH+23] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Pasta: A case for hybrid homomorphic encryption. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):30–73, 2023.

[DHS16] Yarkin Doröz, Yin Hu, and Berk Sunar. Homomorphic AES evaluation using the modified LTV scheme. *Des. Codes Cryptogr.*, 80(2):333–358, 2016.

[DLR16] Sébastien Duval, Virginie Lallemand, and Yann Rotella. Cryptanalysis of the FLIP family of stream ciphers. In *Advances in Cryptology - CRYPTO 2016*, volume 9814 of *Lecture Notes in Computer Science*, pages 457–475. Springer, 2016.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology - EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640. Springer, 2015.

[DR13] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer Berlin Heidelberg, 2013.

[DSES14] Yarkin Doröz, Aria Shahverdi, Thomas Eisenbarth, and Berk Sunar. Toward practical homomorphic evaluation of block ciphers using prince. In *Financial Cryptography and Data Security - FC 2014 Workshops, BITCOIN and WAHC 2014*, volume 8438 of *Lecture Notes in Computer Science*, pages 208–220. Springer, 2014.

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

[GAH+23] Lorenzo Grassi, Irati Manterola Ayala, Martha Norberg Hovd, Morten Øygarden, Håvard Raddum, and Qingju Wang. Cryptanalysis of symmetric primitives over rings and a key recovery attack on rubato. In *Advances in Cryptology - CRYPTO 2023*, volume 14083 of *Lecture Notes in Computer Science*, pages 305–339. Springer, 2023.

[GBA21] Antonio Guimarães, Edson Borin, and Diego F. Aranha. Revisiting the functional bootstrap in TFHE. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):229–253, 2021.

[GBA22] Antonio Guimarães, Edson Borin, and Diego F. Aranha. MOSFHET: optimized software for FHE over the torus. *IACR Cryptol. ePrint Arch.*, page 515, 2022.

[GBJR23] Henri Gilbert, Rachelle Heim Boissier, Jérémy Jean, and Jean-René Reinhard. Cryptanalysis of elisabeth-4. In *Advances in Cryptology - ASIACRYPT 2023*, volume 14440 of *Lecture Notes in Computer Science*, pages 256–284. Springer, 2023.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford University, 2009.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology - CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012.

[GHW23] Xinxin Gong, Yonglin Hao, and Qingju Wang. Combining MILP modeling with algebraic bias evaluation for linear mask search: Improved fast correlation attacks on SNOW. *IACR Cryptol. ePrint Arch.*, page 145, 2023.

[HKC+20] Jincheol Ha, Seongkwang Kim, Wonseok Choi, Jooyoung Lee, Dukjae Moon, Hyojin Yoon, and Jihoon Cho. Masta: An he-friendly cipher using modular arithmetic. *IEEE Access*, 8:194741–194751, 2020.

[HKL+22] Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Jooyoung Lee, and Mincheol Son. Rubato: Noisy ciphers for approximate homomorphic encryption. In *Advances in Cryptology - EUROCRYPT 2022*, volume 13275 of *Lecture Notes in Computer Science*, pages 581–610. Springer, 2022.

[HL20] Phil Hebborn and Gregor Leander. Dasta - alternative linear layer for rasta. *IACR Trans. Symmetric Cryptol.*, 2020(3):46–86, 2020.

[HMR20] Clément Hoffmann, Pierrick Méaux, and Thomas Ricosset. Transciphering, using filip and TFHE for an efficient delegation of computation. In *Progress in Cryptology - INDOCRYPT 2020*, volume 12578 of *Lecture Notes in Computer Science*, pages 39–61. Springer, 2020.

[KLD+23] Andrey Kim, Yongwoo Lee, Maxim Deryabin, Jieun Eom, and Rakyong Choi. LFHE: fully homomorphic encryption with bootstrapping key size less than a megabyte. *IACR Cryptol. ePrint Arch.*, page 767, 2023.

[LAW+23] Fukang Liu, Ravi Anand, Libo Wang, Willi Meier, and Takanori Isobe. Coefficient grouping: Breaking chaghri and more. In *Advances in Cryptology - EUROCRYPT 2023*, volume 14007 of *Lecture Notes in Computer Science*, pages 287–317. Springer, 2023.

[LHH+21] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. PEGASUS: bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *42nd IEEE Symposium on Security and Privacy, SP 2021*, pages 1057–1073. IEEE, 2021.

[LKSM23] Fukang Liu, Abul Kalam, Santanu Sarkar, and Willi Meier. Algebraic attack on fhe-friendly cipher HERA using multiple collisions. *IACR Cryptol. ePrint Arch.*, page 1800, 2023.

[LN14] Tancrède Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes FV and YASHE. In *Progress in Cryptology - AFRICACRYPT 2014*, volume 8469 of *Lecture Notes in Computer Science*, pages 318–335. Springer, 2014.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology - EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.

[LSMI21] Fukang Liu, Santanu Sarkar, Willi Meier, and Takanori Isobe. Algebraic attacks on rasta and dasta using low-degree equations. In *Advances in Cryptology - ASIACRYPT 2021*, volume 13090 of *Lecture Notes in Computer Science*, pages 214–240. Springer, 2021.

[MBM+21] Kotaro Matsuoka, Ryotaro Banno, Naoki Matsumoto, Takashi Sato, and Song Bian. Virtual secure platform: A five-stage pipeline processor over TFHE. In *30th USENIX Security Symposium, USENIX Security 2021*, pages 4007–4024. USENIX Association, 2021.

[MCJS19] Pierrick Méaux, Claude Carlet, Anthony Journault, and François-Xavier Standaert. Improved filter permutators for efficient FHE: better instances and implementations. In *Progress in Cryptology - INDOCRYPT 2019*, volume 11898 of *Lecture Notes in Computer Science*, pages 68–91. Springer, 2019.

[MHSB21] Kotaro Matsuoka, Yusuke Hoshizuki, Takashi Sato, and Song Bian. Towards better standard cell library: Optimizing compound logic gates for TFHE. In *WAHC '21: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 63–68. WAHC@ACM, 2021.

[MJSC16] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. Towards stream ciphers for efficient FHE with low-noise ciphertexts. In *Advances in Cryptology - EUROCRYPT 2016*, volume 9665 of *Lecture Notes in Computer Science*, pages 311–343. Springer, 2016.

[MP21] Daniele Micciancio and Yuriy Polyakov. Bootstrapping in fhew-like cryptosystems. In *WAHC '21: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 17–28. WAHC@ACM, 2021.

[NLV11] Michael Naehrig, Kristin E. Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011*, pages 113–124. ACM, 2011.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):34:1–34:40, 2009.

[RKMR23] R. Radheshwar, Meenakshi Kansal, Pierrick Méaux, and Dibyendu Roy. Differential fault attack on rasta and $\text{FiLIP}_{\text{dsm}}$. *IEEE Trans. Computers*, 72(8):2418–2425, 2023.

[SEA23] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[SMK22] Roy Stracovsky, Rasoul Akhavan Mahdavi, and Florian Kerschbaum. Faster evaluation of aes using tfhe. Poster Session, FHE.Org - 2022, 2022. https://rasoulam.github.io/data/poster-aes-tfhe.pdf.

[TCBS23] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. At last! A homomorphic AES evaluation in less than 30 seconds by means of TFHE. *IACR Cryptol. ePrint Arch.*, page 1020, 2023.

[Tea21] ZUC Design Team. An addendum to the ZUC-256 stream cipher. *IACR Cryptol. ePrint Arch.*, page 1439, 2021.

[WWL+23] Benqiang Wei, Ruida Wang, Zhihao Li, Qinju Liu, and Xianhui Lu. Fregata: Faster homomorphic evaluation of AES via TFHE. In *Information Security - 26th International Conference, ISC 2023*, volume 14411 of *Lecture Notes in Computer Science*, pages 392–412. Springer, 2023.

[YJM19] Jing Yang, Thomas Johansson, and Alexander Maximov. Vectorized linear approximations for attacks on SNOW 3g. *IACR Trans. Symmetric Cryptol.*, 2019(4):249–271, 2019.

A Gate bootstrapping

Algorithm 6 TFHE's Gate Boostrapping [CGGI20]

Input: A TLWE ciphertext $c = (a, b) \in \mathbb{T}^{n+1}$
Input: A constant $\mu \in \mathbb{T}$
Input: A bootstrapping key $\mathrm{BK}_i \in \mathrm{TRGSW}_S(s_i)$, for $i \in [1, n]$
Output: $c'' \in \mathrm{TLWE}_s(\bar{m} \cdot \mu)$, where $\bar{m} = \begin{cases} 1, & \text{if } m < \frac{1}{2} \\ -1, & \text{otherwise.} \end{cases}$

1: $\bar{b} \leftarrow \lfloor 2N \cdot b \rceil \in \mathbb{Z}_{2N}$ and $\bar{a_i} \leftarrow \lfloor 2N \cdot a_i \rceil \in \mathbb{Z}_{2N}$ for each $i \in [1, n]$
2: $v \leftarrow \left(1, X, X^2, \ldots, X^{N-1}\right) \cdot \mu \in \mathbb{T}_N[X]$
3: ACC $\leftarrow$ BlindRotate $\left((0, v), \left(\bar{a_1}, \ldots, \bar{a_n}, \bar{b}\right), (\mathrm{BK}_1, \ldots, \mathrm{BK}_n)\right)$
4: $c' = \mathrm{SampleExtract}_0(\mathrm{ACC})$
5: **return** $\mathrm{KeySwitch}_{S \to s}(c')$

B KeySwitch

B.1 TLWE-to-T(R)LWE Key Switching

Algorithm 7 Public Functional KeySwitching [CGGI20]

Input: p TLWE samples $c^{(z)} = (a^{(z)}, b^{(z)}) \in \mathrm{TLWE}_s(\mu_z), z \in [1, p]$
Input: a public R-Lipschitz linear function $f : \mathbb{T}^p \to \mathbb{T}_N[X]$
Input: a precision parameter $t \in \mathbb{Z}$
Input: a Key Switching key $\mathrm{KS}_{i,j} \in \mathrm{T(R)LWE}_{s'}(\frac{s_i}{2^j})$, for $i \in [1, n]$ and $j \in [1, t]$
Output: a T(R)LWE sample $c' \in \mathrm{T(R)LWE}_{s'}(f(\mu_z))$, for $z \in [1, p]$

1: **for** $i = 1$ to n **do**
2: $\quad a_i \leftarrow f(a_i^{(1)}, a_i^{(2)}, \cdots, a_i^{(p)})$
3: $\quad$ Let $\tilde{a}_i = \lceil a_i \rfloor_{\frac{1}{2^t}}$ be the closest multiple of $\frac{1}{2^t}$ to a_i
4: $\quad$ Decompose each $\tilde{a}_i = \sum_{j=1}^{t} \tilde{a}_{i,j} \cdot 2^{-j}$, where $\tilde{a}_{i,j} \in \mathbb{B}_N[X]$
5: **end for**
6: **return** $(0, f(b_i^{(1)}, b_i^{(1)}, \cdots, b_i^{(p)})) - \sum_{i=1}^{n} \sum_{j=1}^{t} \tilde{a}_{i,j} \cdot \mathrm{KS}_{i,j}$

B.2 TRLWE-to-TRLWE Key Switching

Key switching for TRLWE ciphertext include the following two algorithms:

- $\mathrm{KeySwitchGen}(s, z)$: Compute $ksk = \mathrm{TRLWE}_z(s \cdot B_g^j)$ for $j = 1, \cdots, \ell$.
- $\mathrm{KeySwitch}_{s \to z}(\mathrm{TRLWE}_s(m), ksk)$: Given an $\mathrm{TRLWE}_s(m) = (a, b)$, it evaluates
  - Decompose the first part $a(x)$ into some small polynomials $\sum_{i=1}^{\ell} a_i(x) B_g^i$
  - Compute the inner product

$$\mathrm{TRLWE}_z(m) = (0, b) - (\sum a_i(x) \cdot ksk[0]_i, \sum a_i(x) \cdot ksk[1]_i).$$

Algorithm 8 Private Functional KeySwitching [CGGI20]

Input: p TLWE samples $c^{(z)} = (a^{(z)}, b^{(z)}) \in \mathrm{TLWE}_s(\mu_z), z \in [1, p]$
Input: a precision parameter $t \in \mathbb{Z}$
Input: a Key Switching key $\mathrm{KS}_{i,j,z} \in \mathrm{T(R)LWE}_{s'}(\frac{f_z(s_i)}{2^j})$ for $i \in [1, n]$ and $\mathrm{KS}_{n+1,j,z} \in \mathrm{T(R)LWE}_{s'}(\frac{f_z(-1)}{2^j})$, for $j \in [1, t]$ and $k \in [1, p]$, where f_z are linear morphisms
Output: a T(R)LWE sample $c' \in \mathrm{T(R)LWE}_{s'}(f(\mu_z))$, for $z \in [1, p]$
1: **for** $z = 1$ to p **do**
2: **for** $i = 1$ to n **do**
3: Let $\tilde{a}_{k,i} = \lceil a_{z,i} \rfloor_{\frac{1}{2^t}}$ be the closest multiple of $\frac{1}{2^t}$ to $a_{z,i}$
4: Decompose each $\tilde{a}_{z,i} = \sum_{j=1}^{t} \tilde{a}_{z,i,j} \cdot 2^{-j}$, where $\tilde{a}_{z,i,j} \in \mathbb{B}_N[X]$
5: **end for**
6: **end for**
7: **return** $-\sum_{z=1}^{p} \sum_{i=1}^{n} \sum_{j=1}^{t} \tilde{a}_{z,i,j} \cdot \mathrm{KS}_{z,i,j}$

C Recall on SNOW 3G

C.1 Some functions used in SNOW 3G

(1)The MULx function:

$$\mathrm{MULx}(V, c) = \begin{cases} (V <<_8 1) \oplus c & \text{, if the most significant bit of V equals 1,} \\ V <<_8 1 & \text{, else.} \end{cases}$$

(2)The MULxPOW function:

$$\mathrm{MULxPOW}(V, i, c) = \begin{cases} V & \text{, if } i = 0, \\ \mathrm{MULx}(\mathrm{MULxPOW}(V, i-1, c), c) & \text{, else.} \end{cases}$$

(3)The MUL_α function:

$$\begin{aligned} \mathrm{MUL}_\alpha(c) =& (\mathrm{MULxPOW}(c, 23, 0xa9)||\mathrm{MULxPOW}(c, 245, 0xa9) \\ & ||\mathrm{MULxPOW}(c, 48, 0xa9)||\mathrm{MULxPOW}(c, 239, 0xa9)). \end{aligned}$$

(4)The DIV_α function:

$$\begin{aligned} \mathrm{DIV}_\alpha(c) =& (\mathrm{MULxPOW}(c, 16, 0xa9)||\mathrm{MULxPOW}(c, 39, 0xa9) \\ & ||\mathrm{MULxPOW}(c, 6, 0xa9)||\mathrm{MULxPOW}(c, 64, 0xa9)). \end{aligned}$$

C.2 The 32x32-bit S-Box.

The S-Box SB_1 maps a 32-bit input to a 32-bit output. Let $w = w_0||w_1||w_2||w_3$ the 32-bit input and then $SB_1(w) = r_0||r_1||r_2||r_3$. Specifically, r_0, r_1, r_2, r_3 are defined as

$$\begin{aligned} r_0 &= \mathrm{MULx}(S_R(w_0), 0x1B) \oplus S_R(w_1) \oplus S_R(w_2) \oplus \mathrm{MULx}(S_R(w_3), 0x1B) \oplus S_R(w_3), \\ r_1 &= \mathrm{MULx}(S_R(w_0), 0x1B) \oplus S_R(w_0) \oplus \mathrm{MULx}(S_R(w_1), 0x1B) \oplus S_R(w_2) \oplus S_R(w_3), \\ r_2 &= S_R(w_0) \oplus \mathrm{MULx}(S_R(w_1), 0x1B) \oplus S_R(w_1) \oplus \mathrm{MULx}(S_R(w_2), 0x1B) \oplus S_R(w_3), \\ r_3 &= S_R(w_0) \oplus S_R(w_1) \oplus \mathrm{MULx}(S_R(w_2), 0x1B) \oplus S_R(w_2) \oplus \mathrm{MULx}(S_R(w_3), 0x1B). \end{aligned}$$

where S_R is 8-to-8-bit Rijndael S-Box. The S-Box SB_2 also maps a 32-bit input to a 32-bit output as SB_1, except that it uses another Rijndael S-Box S_R.

C.3 Initialization Mode and Keystream Mode

In the Initialization Mode the LFSR receives a 32-bit input word F, which is the output of the FSM. Let $s_0 = s_{0,0}||s_{0,1}||s_{0,2}||s_{0,3}$ and $s_{11} = s_{11,0}||s_{11,1}||s_{11,2}||s_{11,3}$. Compute the intermediate value v as

$$v = (s_{0,1}||s_{0,2}||s_{0,3}||0x00) \oplus \mathrm{MUL}_\alpha(s_{0,0}) \oplus s_2 \oplus (0x00||s_{11,0}||s_{11,1}||s_{11,2}) \oplus \mathrm{DIV}_\alpha(s_{11,3}) \oplus F,$$

then update LFSR: $s_i = s_{i+1}, 0 \leq i \leq 14$ and $s_{15} = v$.

In the Keystream Mode, the LFSR performs the same operation as the initialization Mode, except that it does not receive any input for calculating v.

C.4 Clocking the FSM

The FSM has two input words s_{15} and s_5 from the LFSR. It produces a 32-bit output word $F = (s_{15} \boxplus R1) \oplus R2$. Compute the intermediate value $r = R2 \boxplus (R3 \oplus s_5)$, then the registers are updated:

$$R3 = SB_2(R2), R2 = SB_1(R1), R1 = r.$$

C.5 Initialization and Keystream Generation

C.5.1 Initialization

SNOW 3G is initialized with a 128-bit key consisting of four 32-bit words k_0, k_1, k_2, k_3 and an 128-bit initialization variable consisting of four 32-bit words IV_0, IV_1, IV_2, IV_3 as follows. Let $\mathbf{1}$ be the all-ones word $(0xffffffff)$.

$s_{15} = k_3 \oplus IV_0$	$s_{14} = k_2$	$s_{13} = k_1$	$s_{12} = k_0 \oplus IV_1$
$s_{11} = k_3 \oplus \mathbf{1}$	$s_{10} = k_2 \oplus \mathbf{1} \oplus IV_2$	$s_9 = k_1 \oplus \mathbf{1} \oplus IV_3$	$s_8 = k_0 \oplus \mathbf{1}$
$s_7 = k_3$	$s_6 = k_2$	$s_5 = k_1$	$s_4 = k_0$
$s_3 = k_3 \oplus \mathbf{1}$	$s_2 = k_2 \oplus \mathbf{1}$	$s_1 = k_1 \oplus \mathbf{1}$	$s_0 = k_0 \oplus \mathbf{1}$

The FSM is initialised with $R1 = R2 = R3 = 0$. Then the cipher runs the following two steps 32 times without producing output:

- The FSM is clocked producing the 32-bit word F.
- The LFSR is clocked in Initialization Mode consuming F.

C.5.2 Keystream Generation

First the FSM is clocked once. The output word of the FSM is discarded. Then the LFSR is clocked once in Keystream Mode. After that n 32-bit words of keystream are produced: for $t = 1$ to n

1. The FSM is clocked and produces a 32-bit output word F.
2. The next keystream word is computed as $z_t = F \oplus s_0$.
3. Then the LFSR is clocked in Keystream Mode.

D Recall on ZUC

ZUC consists of three layers: the top layer is a linear feedback shift register (LFSR) of 16 stages; the bottom layer is a nonlinear block which is called F function; while the middle layer, called BitReorganization layer, is a connection layer between the LFSR and F.

Now we would give some details of the three layers. The linear feedback shift register (LFSR) has 16 31-bit words $s_0, s_1, \cdots, s_{15}$ and consists of two modes of operations: the initialization mode in Figure 11 and working mode in Figure 4.

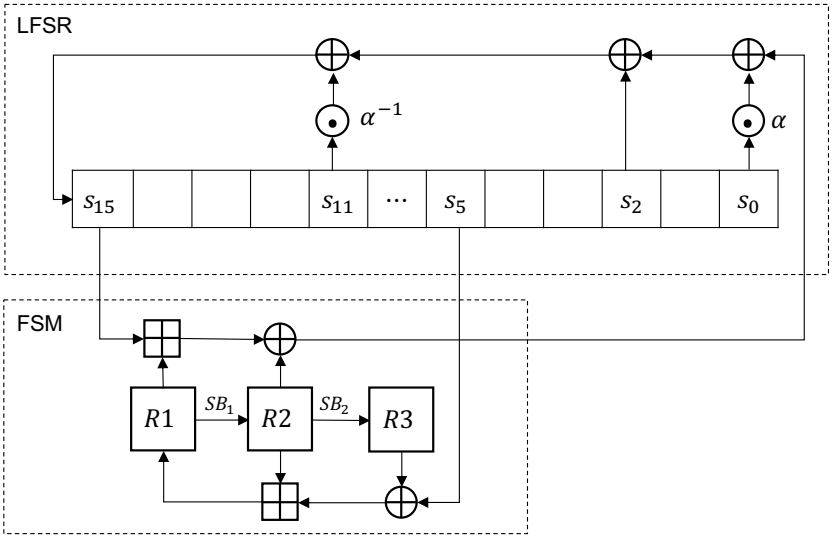

Figure 10: Initialization of SNOW 3G.

D.1 LFSRWithInitializationMode(u)

The LFSR receives the input u and then performs:

1. $v = 2^{15} \cdot s_{15} + 2^{17} \cdot s_{13} + 2^{21} \cdot s_{10} + 2^{20} \cdot s_4 + (1 + 2^8) \cdot s_0 \bmod (2^{31} - 1)$;
2. $s_{16} = (v + u) \mod (2^{31} - 1)$;
3. if $s_{16} = 0$ then set $s_{16} = 2^{31} - 1$;
4. $(s_{16}, s_{15}, \cdots, s_2, s_1) \rightarrow (s_{15}, s_{14}, \cdots, s_1, s_0)$.

D.2 LFSRWithworkMode()

The LFSR performs similar operations to the initialization process, except that it does not receive any input and does not require a second step of computation.

1. $s_{16} = 2^{15} \cdot s_{15} + 2^{17} \cdot s_{13} + 2^{21} \cdot s_{10} + 2^{20} \cdot s_4 + (1 + 2^8) \cdot s_0 \bmod (2^{31} - 1)$;
2. if $s_{16} = 0$ then set $s_{16} = 2^{31} - 1$;
3. $(s_{16}, s_{15}, \cdots, s_2, s_1) \rightarrow (s_{15}, s_{14}, \cdots, s_1, s_0)$.

D.3 BitReorganization

BitReorganization extracts 128 bits from the words of LFSR and forms four 32-bit words, in which the first three words will be passed to the next layer, that is, the nonlinear function F, and the last word will participate in generating the key stream.

$$X_0 = s_{15H} \mid\mid s_{14L}, X_1 = s_{11L} \mid\mid s_{9H}, X_2 = s_{7L} \mid\mid s_{5H}, X_3 = s_{2L} \mid\mid s_{0H},$$

where s_{iH} is the high 16 bits of the word s_i and s_{jL} is the low 16 bits of the word s_j.

D.4 The Nonlinear function F

There are two 32-bit memory words R_1 and R_2 in the nonlinear function F. The input of F is X_0, X_1, X_2, which are the first three words of output of the BitReorganization. It outputs a 32-bit word W.

1. $W = (X_0 \oplus R_1) \boxplus R_2$; $W_1 = R_1 \boxplus X_1$; $W_2 = R_2 \oplus X_2$;
2. $R_1 = S(L_1(W_{1L} \mid\mid W_{2H}))$; $R_2 = S(L_2(W_{2L} \mid\mid W_{1H}))$.

where $S = (S_0, S_1, S_0, S_1)$ is the 4 parallel S-boxes and L_1, L_2 are the two 32-bit linear transformation defined by

$$L_1(X) = X \oplus (X \lll 2) \oplus (X \lll 10) \oplus (X \lll 18) \oplus (X \lll 24);$$
$$L_2(X) = X \oplus (X \lll 8) \oplus (X \lll 14) \oplus (X \lll 22) \oplus (X \lll 30).$$

D.5 The key loading

The key loading procedure will expand the 128-bit initial key $k(k_0 \mid\mid k_1 \mid\mid \cdots k_{15})$ and the 128-bit initial vector $iv(iv_0 \mid\mid iv_1 \mid\mid \cdots iv_{15})$ into 16 31-bit integers $s_i = k_i \mid\mid d_i \mid\mid iv_i$ as the initial state of the LFSR, where d_i is a known constant.

D.6 The Execution of ZUC

The execution of ZUC has two stages: the initialization stage and the working stage.

D.6.1 The initialization stage

1. Load the key, IV and constants into the LFSR;
2. Let $R_0 = R_1 = 0$;
3. for $i = 0$ to 31 do
   - BitReconstruction();
   - $W = F(X_0, X_1, X_2)$;
   - LFSRWithInitializationMode($W \gg 1$);

D.6.2 The working stage

After the initialization stage, the algorithm enters the working stage. During the working stage, the algorithm performs the following operations once:

- BitReconstruction();
- $W = F(X_0, X_1, X_2)$, diszard W;
- LFSRWithworkMode().

Lastly, the algorithm enters the keystream generation phase, each iteration performs the following operations to generate a 32-bit word Z as output:

1. BitReorganization();
2. $Z = F(X_0, X_1, X_2) \oplus X_3$;
3. LFSRWithworkMode().

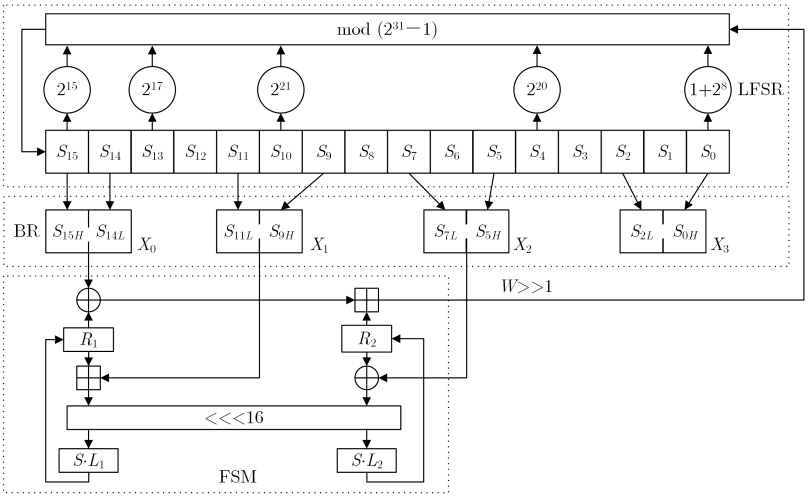

Figure 11: Initialization of ZUC. The picture is taken from [Tea21]