

Coral: General and Efficient Framework for Homomorphic Evaluation of Symmetric Ciphers

Jikang Bai Ruida Wang✉ Xianhui Lu✉ Kunpeng Wang

baijikang@iie.ac.cn wangruida@iie.ac.cn luxianhui@iie.ac.cn wangkunpeng@iie.ac.cn
State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
School of Cyber Security, University of Chinese Academy of Sciences

Abstract

Fully homomorphic encryption (FHE) enables computation directly on encrypted data, but encrypting data under FHE imposes client-side computational costs and produces large ciphertexts. Transciphering addresses these costs by allowing the client to upload symmetric ciphertexts while the server homomorphically evaluates symmetric decryption to obtain FHE ciphertexts. However, transciphering itself can become a performance bottleneck, and a general framework for efficiently supporting structurally diverse symmetric ciphers is still lacking.

To fill this gap, we present CORAL, a general and efficient transciphering framework built on TFHE. Rather than mapping an entire decryption circuit to a uniform evaluation strategy, CORAL decomposes it at operation boundaries and provides optimized procedures for four recurring computational modules: linear Boolean operations, sparse low-degree Boolean functions, modular additions, and small-domain substitutions. These procedures share compatible ciphertext interfaces, allowing them to be reused and composed across different ciphers. Together, these optimized procedures enable CORAL to achieve high efficiency without sacrificing generality.

We instantiate CORAL for Trivium, AES, ChaCha20, and ZUC, covering both individual and combined nonlinear structures. Compared with the corresponding baselines, our implementations achieve speedups of up to $4.055\times$ for ZUC under single-threaded execution and up to $5.517\times$ for ChaCha20 under fully parallel execution.

1 Introduction

Fully homomorphic encryption (FHE) enables computation directly on encrypted data and is therefore a fundamental tool for privacy-preserving cloud computation [7, 8, 15, 16]. However, encrypting data directly under FHE imposes substantial costs on the client and produces ciphertexts much larger than the underlying plaintext, resulting in considerable

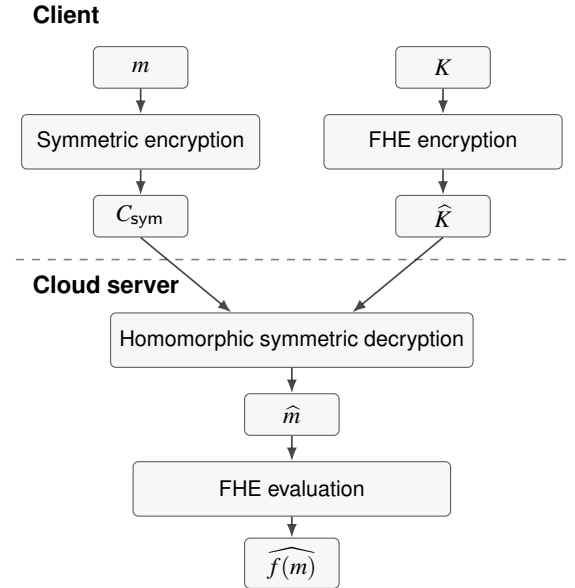


Figure 1: Transciphering workflow.

communication and storage overhead. Symmetric encryption, by contrast, is lightweight and introduces little ciphertext expansion. Transciphering bridges these two forms of encryption as shown in Fig. 1. The client encrypts the symmetric key under FHE once and encrypts its data using the symmetric cipher. The server then homomorphically evaluates the symmetric decryption algorithm on the compact symmetric ciphertext using the FHE-encrypted key, thereby obtaining an FHE ciphertext of the plaintext without learning either the key or the data. Subsequent computation can proceed under FHE. This approach retains the low client-side cost and compact ciphertexts of symmetric encryption while enabling homomorphic computation at the server. However, the cost of transciphering can itself become the primary performance bottleneck.

Although FHE can evaluate arbitrary circuits in principle, different homomorphic evaluation techniques exhibit substan-

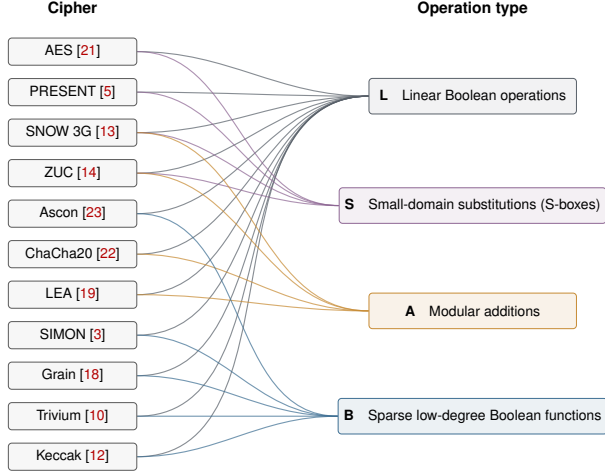


Figure 2: Classification of symmetric ciphers by their fundamental operations. L, S, A, and B denote linear Boolean operations, small-domain substitutions, modular additions, and sparse low-degree Boolean functions, respectively.

tially different performance across computational operations. We observe that the diversity of symmetric ciphers arises largely from different compositions of a small set of recurring operations. As illustrated in Figure 2, their decryption procedures can often be decomposed into four representative computational modules: linear Boolean operations, small-domain substitutions, sparse low-degree Boolean functions, and modular additions.

Most previous work summarized in [1] is generally optimized for particular computational operations. For example, the p -encoding approach [6] performs well for sparse, shallow Boolean circuits and can in principle be extended to circuits of arbitrary size, but its efficiency degrades as circuit complexity increases. Conversely, tree-based programmable-bootstrapping methods [4] efficiently evaluate complex small-domain functions such as the S-box, but incur substantial auxiliary processing when applied to bit-level operations. These complementary strengths indicate that no single evaluation method is efficient for all cipher components. Efficient transciphering requires different homomorphic evaluation procedures tailored to the computational structure of each component. Consequently, a general and efficient transciphering framework remains lacking.

Our goal is to build a general and efficient framework for transciphering using only a small set of optimized, composable modules. Our contributions are summarized as follows:

- **The CORAL FRAMEWORK.** We present CORAL, a general transciphering framework built on TFHE [9] for the homomorphic evaluation of symmetric ciphers. CORAL partitions symmetric cipher evaluation at operation boundaries, assigns each module a suitable homomorphic representation and evaluation procedure mainly

based on circuit bootstrapping (CBS), and connects the resulting modules through compatible GLWE/GGSW interfaces.

- **Optimized evaluation methods for four recurring computational modules.** We optimize each modules by reducing the dominant cost. Since bootstrapping is the most expensive operation in TFHE, our optimizations focus on reducing the number of bootstrapping calls, lowering the latency of each call, and exposing greater parallelism across independent bootstrapping operations.

- *Linear Boolean operations.* We use single-bit encoding, under which XOR is evaluated by homomorphic addition and fixed permutations are implemented by ciphertext reindexing, avoiding unnecessary bootstrapping.
- *Sparse low-degree Boolean functions.* We introduce a maximum independent set based ANF evaluator for sparse low-degree Boolean functions. Unlike other nonlinear modules, it allows a compatible subset of inputs to remain in GLWE form, thereby avoiding their costly conversion to GGSW ciphertexts through circuit bootstrapping. Minimizing the number of required CBS operations substantially improves the efficiency of the overall circuit.
- *Modular additions.* We design a thermometer code based adder for multi-operand additions. Although carry propagation inherently proceeds sequentially from lower to higher bit positions, our construction allows the dominant computational operations to execute fully in parallel, leaving only lightweight state updates on the carry-dependent path. This yields an efficient specialized adder when sufficient parallelism is available.
- *Small-domain substitutions.* We use mixed-packed CMux-tree lookup to pack multiple truth-table entries into GLWE polynomials and select the desired outputs using GGSW controls. Subsequent public linear transformations are fused into the lookup tables, eliminating separate linear operations. This fusion reduces noise growth and enables tighter parameter sets, resulting in faster bootstrapping.

- **Representative instantiations and evaluation.** We instantiate CORAL for Trivium, AES, ChaCha20, and ZUC. Trivium, AES, and ChaCha20 respectively emphasize sparse low-degree Boolean functions, small-domain substitutions, and modular additions, while ZUC demonstrates the composition of substitutions and modular arithmetic. We evaluate both sequential efficiency and parallel scalability under single-threaded and fully parallel configurations. Compared with the corresponding

baselines, our implementations achieve speedups of up to $4.055\times$ for ZUC in the single-threaded configuration and up to $5.517\times$ for ChaCha20 in the fully parallel configuration.

2 Preliminaries

2.1 Notation

We use bold letters for vectors and matrices, and write $\lfloor r \rfloor$ for the nearest integer to r . For $a, b \in \mathbb{R}$ with $a < b$, we define $[a, b[= \{x \in \mathbb{R} : a \leq x < b\}$. When a and b are integers, we write $\llbracket a, b \rrbracket$ for $\mathbb{Z} \cap [a, b]$.

For an integer q , we identify $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ with $\llbracket -q/2, q/2 \rrbracket$ and write $[\cdot]_q$ for reduction modulo q into this interval. We define $\mathbb{B} = \{0, 1\}$ and $[n] = \{1, 2, \dots, n\}$ for a positive integer n , and use $\mathbb{N}$ for the set of positive integers. We denote the multiplicative group of units in $\mathbb{Z}_q$ by $\mathbb{Z}_q^\times$.

For a power-of-two integer N , we define $\mathcal{R}_N[X] = \mathbb{Z}[X]/(X^N + 1)$. We further write $\mathcal{R}_{q,N} = \mathcal{R}_N/q\mathcal{R}_N = \mathbb{Z}_q[X]/(X^N + 1)$ and $\mathbb{B}_N[X] = \mathbb{B}[X]/(X^N + 1)$. For a probability distribution $\mathcal{D}$, we write $a \leftarrow \mathcal{D}$ to indicate that a is sampled from $\mathcal{D}$. Unless specified otherwise, all logarithms are base 2.

2.2 Ciphertext Forms

GLWE. Let $\mathbf{S} \in \mathcal{R}_{q,N}^k$ be a secret key. We encrypt a message $M \in \mathcal{R}_{p,N}$ as the generalized LWE (GLWE) ciphertext

$$\text{GLWE}_{q,\mathbf{S}}(\Delta \cdot M) = \left(A_1, \dots, A_k, B = \sum_{i=1}^k A_i \cdot S_i + [M \cdot \Delta]_q + E \right).$$

Here, we write $\mathbf{S} = (S_1, \dots, S_k)$, sample $A_i \leftarrow \mathcal{R}_{q,N}$ for $i = 1, \dots, k$, and sample $E \leftarrow \chi_\sigma$ from a Gaussian error distribution χ_σ . We refer to kN and k as the GLWE dimension and rank, respectively. As is common in TFHE implementations, we assume binary secret keys. Setting $N = 1$ yields an LWE ciphertext, whereas setting $k = 1$ yields an RLWE ciphertext.

Gadget Decomposition. Let $B \in \mathbb{N}$ be a power of two and $\ell \in \mathbb{N}$. We define the gadget decomposition $\text{GadgetDecomp}^{(B,\ell)}$ with base B and length ℓ as a mapping that decomposes $a \in \mathbb{Z}_q$ into $(a_1, \dots, a_\ell) \in \mathbb{Z}_q^\ell$ such that

$$a = \sum_{j=1}^{\ell} a_j \cdot v_j + e,$$

where $v_j = \lceil q/B^j \rceil$ and $a_j \in \llbracket -B/2, B/2 \rrbracket$ for $j = 1, \dots, \ell$. We bound the decomposition error e by $|e| \leq \lceil q/(2B^\ell) \rceil$ and extend $\text{GadgetDecomp}^{(B,\ell)}$ to polynomials in $\mathcal{R}_{q,N}$ coefficient-wise.

GGSW. We use GGSW ciphertexts to represent encrypted messages in a form suitable for homomorphic multiplication [9]. Let $B \in \mathbb{N}$ be a power of two and $\ell \in \mathbb{N}$. For a secret

key $\mathbf{S} \in \mathbb{B}_N[X]^k$, a GGSW encryption of $M \in \mathcal{R}_{q,N}$ with base B and gadget length ℓ is

$$\text{GGSW}_{\mathbf{S}}^{(B,\ell)}(M) = (\text{GLWE}_{\mathbf{S}}(v_j \cdot (-S_i \cdot M)))_{(i,j) \in [k+1] \times [\ell]}. \quad (1)$$

We denote the resulting $(k+1)\ell \times (k+1)$ matrix over $\mathcal{R}_{q,N}$ by $\overline{\mathbf{C}} \in \mathcal{R}_{q,N}^{(k+1)\ell \times (k+1)}$.

2.3 Homomorphic Operation Interfaces

This subsection fixes the notation and input–output behavior of the homomorphic operations used later. Appendix A gives their detailed calculations and algorithms.

External Product and CMux. For messages $M_1, M_2 \in \mathcal{R}_{q,N}$, the symbol $\boxtimes$ denotes the external product

$$\text{GGSW}_{\mathbf{S}}(M_1) \boxtimes \text{GLWE}_{\mathbf{S}}(M_2) \longrightarrow \text{GLWE}_{\mathbf{S}}(M_1 M_2).$$

Thus, a GGSW ciphertext acts as the multiplicative operand while the result remains in GLWE form. For a Boolean selector $b \in \mathbb{B}$, we write

$$\begin{aligned} \text{CMux}(\text{GGSW}_{\mathbf{S}}(b), \text{GLWE}_{\mathbf{S}}(M_0), \text{GLWE}_{\mathbf{S}}(M_1)) \\ \longrightarrow \text{GLWE}_{\mathbf{S}}(M_b). \end{aligned}$$

CMux therefore selects between two GLWE messages without revealing b .

Key Switching. For GLWE keys $\mathbf{S}$ and $\mathbf{S}'$ with the same polynomial size, $\text{KS}_{\mathbf{S} \rightarrow \mathbf{S}'}$ denotes the key-switching key, and

$$\text{GLWEKeySwitch}_{\mathbf{S} \rightarrow \mathbf{S}'} : \text{GLWE}_{\mathbf{S}}(M) \longrightarrow \text{GLWE}_{\mathbf{S}'}(M)$$

changes the encryption key while preserving the message. The symbol $\text{AutoKey}_d = \text{KS}_{\mathbf{S}(X^d) \rightarrow \mathbf{S}(X)}$ denotes the corresponding key for restoring $\mathbf{S}(X)$ after the automorphism $X \mapsto X^d$.

LWE Modulus Switching. Modulus switching maps an LWE ciphertext $\mathbf{c} \in \mathbb{Z}_Q^{n+1}$ to $\hat{\mathbf{c}} \in \mathbb{Z}_{2N}^{n+1}$ while approximately scaling its phase by $2N/Q$. The output modulus $2N$ is the rotation modulus used by blind rotation. We use mean composition introduced in [11] to get lower modulus switching noise.

Sample Extraction. For a GLWE plaintext $M(X) = \sum_{j=0}^{N-1} m_j X^j$, sample extraction is denoted by

$$\text{SampleExtract}_i : \text{GLWE}_{\mathbf{S}}(M(X)) \longrightarrow \text{LWE}_{\mathbf{s}^{(i)}}(m_i),$$

where $\mathbf{s}^{(i)}$ is the LWE key induced by the i -th GLWE coefficient. When the index is omitted, SampleExtract extracts the constant coefficient m_0 .

Programmable Bootstrapping. Programmable bootstrapping (PBS) [9] refreshes an LWE ciphertext while evaluating a public negacyclic function f , where $f(x+N) = -f(x)$ for $0 \leq x < N$:

$$\text{PBS}_f : \text{LWE}_{\mathbf{S}}(\Delta_{\text{in}} m) \longrightarrow \text{LWE}_{\mathbf{S}'}(\Delta_{\text{out}} f(m)).$$

Identity PBS refers to the case $f(m) = m$. We denote the bootstrapping key by $\text{BSK} = (\text{GGSW}_{S'}(s_i))_{i \in [n]}$ and the encrypted accumulator rotation inside PBS by BlindRotate .

PBSmanyLUT evaluates up to 2^ϑ negacyclic functions $f_1, \dots, f_{2^\vartheta}$ of the same encrypted input with one blind rotation [9]. We use PBSmanyLUT for its LWE-output interface and $\text{PBSmanyLUT}_{\text{GLWE}}$ when the refreshed accumulator is returned directly in GLWE form without sample extraction.

Automorphisms and Reverse Homomorphic Trace. For $d \in \mathbb{Z}_{2N}^\times$, let $\tau_d(M(X)) = M(X^d)$. We write $\text{EvalAuto}(\mathbf{C}, d)$ for applying τ_d to every component of $\mathbf{C}$ and then using AutoKey_d to restore the original GLWE key. Automorphisms are used to calculate trace. For $M(X) = \sum_{i=0}^{N-1} m_i X^i$, reverse homomorphic trace [20] has the interface

$$\text{RevHomTrace} : \text{GLWE}_{S(X)}(M(X)) \longrightarrow \text{GLWE}_{S(X)}(m_0).$$

It removes the nonconstant coefficients without the phase amplification of the standard homomorphic trace.

Circuit Bootstrapping. For a Boolean message $m \in \mathbb{B}$, circuit bootstrapping (CBS) [24, 25] refreshes an LWE ciphertext and converts it to GGSW form:

$$\text{CBS} : \text{LWE}_S(\Delta m) \longrightarrow \text{GGSW}_{S'}(m).$$

The resulting GGSW ciphertext can be reused as the control input of external products and CMux gates.

3 CORAL Transciphering Framework

CORAL represents the homomorphic decryption procedure of a symmetric cipher as a composition of four computation modules: linear Boolean operations, sparse low-degree Boolean functions, modular additions and small-domain substitutions. Linear Boolean operations are achieved through single-bit encoding. The three nonlinear classes use maximum independent set based ANF evaluation, thermometer code based modular addition and packed CMux-tree lookup with linear-tail fusion respectively. A cipher constructed from these operation classes can reuse the corresponding implementations, however any additional nonlinear structure requires its own realization and is not covered in our framework.

These modules compose through a common data representation. Every module returns its result in GLWE form, and GGSW is introduced only as an additional control representation for inputs used by subsequent nonlinear computation. Linear Boolean operations consume GLWE ciphertexts directly. Maximum independent set based ANF evaluation receives inputs in GLWE form and additionally requires GGSW representations for the promoted set $\mathcal{P}^*$. Thermometer code based modular addition uses GGSW operand bits and an internal GLWE carry state. While packed CMux-tree lookup

uses GGSW selectors. The interfaces are summarized as

$$\begin{aligned} \text{Linear} : & \text{GLWE}^n \longrightarrow \text{GLWE}^m, \\ \text{ANF} : & \text{GLWE}^n \times \text{GGSW}^{|\mathcal{P}^*|} \longrightarrow \text{GLWE}^t, \\ \text{ModAdd} : & \text{GGSW}^{q^w} \times \text{GLWE}_{\text{state}} \longrightarrow \text{GLWE}^w. \\ \text{Lookup} : & \text{GGSW}^n \longrightarrow \text{GLWE}^m, \end{aligned}$$

Module composition must satisfy both ciphertext-format and noise constraints. A GLWE output is passed directly when its propagated noise bound is admissible; otherwise, identity PBS refreshes it. If the next module requires a GGSW control, CBS refreshes the value and produces its GGSW representation. PBS and CBS placement is determined during cipher instantiation from the public module graph and conservative noise bounds.

Each nonlinear module reduces a different source of homomorphic cost. Maximum independent set based ANF evaluation keeps as many GLWE form as possible to minimize the number of circuit bootstrapping. Thermometer code based modular addition replaces a monolithic lookup over all operand bits with local carry transitions whose cost grows linearly with the word length and number of operands rather than exponentially with the total input width. Linear-tail fusion incorporates a subsequent public linear map into the LUT, eliminating the corresponding GLWE operations and reducing noise growth, which result in compacting bootstrapping parameters and faster bootstrapping.

The following subsections describe the four computation modules and their homomorphic realizations.

3.1 Linear Operation Evaluation

We use single-bit encoding, in which a bit $m \in \mathbb{B}$ is encoded as Δm with $\Delta = q/2$. This encoding satisfies

$$[\Delta x + \Delta y]_q = \Delta(x \oplus y), \quad x, y \in \mathbb{B}.$$

Consequently, XOR is evaluated directly through GLWE addition:

$$\text{GLWE}_S(\Delta x) + \text{GLWE}_S(\Delta y) = \text{GLWE}_S(\Delta(x \oplus y)).$$

Fixed bit permutations require no homomorphic operation. Given a public permutation π and a ciphertext vector $(\mathbf{C}_0, \dots, \mathbf{C}_{n-1})$ encrypting $(\Delta x_0, \dots, \Delta x_{n-1})$, the permuted output is obtained by reordering the ciphertexts as

$$\mathbf{C}'_i = \mathbf{C}_{\pi(i)}, \quad 0 \leq i < n.$$

Thus, XOR and fixed permutations preserve the GLWE representation and require no bootstrapping.

3.2 Maximum independent set based ANF Evaluation

3.2.1 ANF representation.

Every Boolean function admits a unique representation in algebraic normal form (ANF). For a multi-output function $F = (f_1, \dots, f_t) : \mathbb{B}^n \rightarrow \mathbb{B}^t$, we write

$$f_j(x_1, \dots, x_n) = c_j \oplus \bigoplus_{S \in \mathcal{M}_j} \prod_{i \in S} x_i, \quad 1 \leq j \leq t,$$

where $c_j \in \mathbb{B}$ and $\mathcal{M}_j$ is a collection of nonempty subsets of $\{1, \dots, n\}$. Each $S \in \mathcal{M}_j$ specifies the variables appearing in the monomial $\prod_{i \in S} x_i$. A set with $|S| = 1$ represents a linear term, whereas $|S| \geq 2$ represents a nonlinear monomial. The ANF uses only AND and XOR operations and makes explicit the number and degrees of the nonlinear monomials, as well as any variable factors shared among them.

3.2.2 Mapping ANF to TFHE.

The sum-of-products structure of an ANF maps naturally to TFHE operations. Under the selected bit encoding, GLWE addition evaluates XOR, while the external product provides the typed multiplication

$$\text{GGSW}_S(x) \boxtimes \text{GLWE}_S(P) \longrightarrow \text{GLWE}_S(xP).$$

A monomial can therefore be evaluated from one GLWE factor and a sequence of GGSW factors. Every external product returns a GLWE ciphertext, so the product remains in GLWE form and can be added directly to the other ANF terms. A nonlinear monomial therefore does not require all of its inputs in GGSW form.

Let

$$\mathcal{M}_{\text{nl}}(F) = \bigcup_{j=1}^t \{S \in \mathcal{M}_j : |S| \geq 2\}$$

denote the distinct nonlinear supports of the complete multi-output function. We partition the input indices into a GLWE-only set I and a promoted set $\mathcal{P} = [n] \setminus I$. Every input remains available in GLWE form, while the inputs in $\mathcal{P}$ are additionally converted into GGSW ciphertexts.

Because each external product requires a GGSW operand, every nonlinear monomial may contain at most one GLWE-only input. A valid assignment thus satisfies

$$|S \cap I| \leq 1, \quad \forall S \in \mathcal{M}_{\text{nl}}(F).$$

We express this constraint through a conflict graph $\mathcal{G}_{\text{conf}} = (V, E)$ with $V = [n]$. Two inputs are adjacent if they occur together in a nonlinear monomial, so a monomial of degree greater than two induces a clique. A GLWE-only set is valid exactly when it is an independent set of this graph. We select a maximum independent set I^* and promote its complement

Algorithm 1: GGSW Input Assignment

Input: Input indices $V = [n]$ and nonlinear supports $\mathcal{M}_{\text{nl}}(F)$
Output: GLWE-only set I^* and GGSW-promoted set $\mathcal{P}^*$

```

1  $E \leftarrow \emptyset$ 
2 foreach  $S \in \mathcal{M}_{\text{nl}}(F)$  do
3   foreach unordered pair  $\{i, i'\} \subseteq S$  with  $i \neq i'$  do
4      $E \leftarrow E \cup \{\{i, i'\}\}$ 
5  $\mathcal{G}_{\text{conf}} \leftarrow (V, E)$ 
6 foreach  $v \in V$  do
7    $\Gamma_{\text{conf}}(v) \leftarrow \{u \in V : \{u, v\} \in E\}$ 
8  $I^* \leftarrow \emptyset$ 
9 Function  $\text{Search}(I, C)$  :
10  if  $|I| + |C| \leq |I^*|$  then
11    return
12  if  $C = \emptyset$  then
13    if  $|I| > |I^*|$  then
14       $I^* \leftarrow I$ 
15    return
16  Select  $v \in C$  maximizing  $|\Gamma_{\text{conf}}(v) \cap C|$ 
17  Search( $I \cup \{v\}, C \setminus (\{v\} \cup \Gamma_{\text{conf}}(v))$ )
18  Search( $I, C \setminus \{v\}$ )
19 Search( $\emptyset, V$ )
20  $\mathcal{P}^* \leftarrow V \setminus I^*$ 
21 return ( $I^*, \mathcal{P}^*$ )
```

$\mathcal{P}^* = V \setminus I^*$. When every input has the same conversion cost, this choice minimizes the number of GGSW conversions.

For multi-output or repeated computations sharing the same input representations, the conflict graph includes all nonlinear monomials over the complete reuse interval. Algorithm 1 constructs this graph and computes its maximum independent set using an exact branch-and-bound search.

The maximum-independent-set problem has exponential worst-case complexity. However, the search operates only on the public description of a nonlinear module and is performed once before homomorphic evaluation.

3.2.3 Refined homomorphic evaluation.

After determining the type assignment, all inputs remain available in GLWE form, while those in $\mathcal{P}^*$ are additionally provided in GGSW form. Algorithm 2 selects one GLWE factor for each monomial, evaluates its remaining factors through external products, and adds the result to the output. The type constraint guarantees that every remaining factor has a GGSW representation.

For a multi-output function, common monomials can be evaluated once and reused across the corresponding outputs.

Algorithm 2: Maximum independent set based ANF Evaluation

Input: $f = c \oplus \bigoplus_{S \in \mathcal{M}} \prod_{i \in S} x_i$
Input: $\mathbf{X}_i^L = \text{GLWE}_S(x_i)$ for $i \in V$, and
 $\bar{\mathbf{X}}_i^G = \text{GGSW}_S(x_i)$ for $i \in \mathcal{P}^*$
Input: GLWE-only set I^*
Output: $\mathbf{Y} = \text{GLWE}_S(f)$

```

1  $\mathbf{Y} \leftarrow \text{GLWE}_S^0(c)$ 
2 foreach  $S \in \mathcal{M}$  do
3   Select  $i_0 \in S \cap I^*$  if nonempty; otherwise select any  $i_0 \in S$ 
4    $\mathbf{C} \leftarrow \mathbf{X}_{i_0}^L$ 
5   foreach  $i \in S \setminus \{i_0\}$  do
6      $\mathbf{C} \leftarrow \bar{\mathbf{X}}_i^G \boxplus \mathbf{C}$ 
7    $\mathbf{Y} \leftarrow \mathbf{Y} + \mathbf{C}$ 
8 return  $\mathbf{Y}$ 

```

Further savings are possible when several monomials share a promoted factor. For example, if $g \in \mathcal{P}^*$, then

$$x_g x_a \oplus x_g x_b = x_g (x_a \oplus x_b)$$

requires one external product after the GLWE ciphertexts of x_a and x_b are combined. Such factorization factors out only inputs in $\mathcal{P}^*$ and therefore preserves the GGSW–GLWE multiplication interface.

3.2.4 Applicability.

Maximum independent set based ANF evaluation is most effective for functions with few low-degree nonlinear monomials. A degree- d monomial induces a d -clique in the conflict graph, so at most one of its variables can belong to the GLWE-only set. It therefore forces at least $d - 1$ variables to be available in GGSW form, limiting the savings obtained from the maximum independent set.

Before factorization, evaluating all distinct nonlinear monomials requires

$$\sum_{S \in \mathcal{M}_{\text{nl}}(F)} (|S| - 1)$$

external products. For each output f_j , the external-product contribution to its final GLWE noise grows with

$$\sum_{\substack{S \in \mathcal{M}_j \\ |S| \geq 2}} (|S| - 1).$$

Thus, even when the individual monomials have low degree, a large number of them can lead to substantial noise, resulting in a large parameter set that allow high noise capacity. This method therefore targets functions with few low-degree nonlinear monomials, especially when monomials or outputs share reusable factors.

3.3 Thermometer Code Based Modular Addition

Let $A_0, \dots, A_{\kappa-1}$ be κ encrypted w -bit operands, where

$$A_r = \sum_{j=0}^{w-1} a_{r,j} 2^j, \quad a_{r,j} \in \mathbb{B}.$$

The input bits are provided as $\bar{\mathbf{A}}_{r,j} = \text{GGSW}_S(a_{r,j})$, and the result bits are returned as GLWE ciphertexts. We evaluate multi-operand addition through carry-state transitions.

3.3.1 Carry-state formulation.

At bit position j , the addition follows

$$t_j = c_j + \sum_{r=0}^{\kappa-1} a_{r,j}, \quad s_j = t_j \bmod 2, \quad c_{j+1} = \left\lfloor \frac{t_j}{2} \right\rfloor.$$

Since $0 \leq c_j \leq \kappa - 1$, we have $0 \leq t_j \leq 2\kappa - 1$. The carry computation can therefore use a thermometer vector of length $L = 2\kappa - 1$, independent of the word width.

3.3.2 Thermometer Code Based transition.

We represent a state v by a thermometer code with a list of L GLWE ciphertexts:

$$\mathbf{U}_v = (\text{GLWE}_S(\Delta[v \geq 1]), \dots, \text{GLWE}_S(\Delta[v \geq L])).$$

Thus, $\mathbf{U}_v$ contains v encryptions of one followed by encryptions of zero. The entries are independent GLWE ciphertexts rather than coefficients packed into one polynomial.

Let

$$\mathbf{C}_{\text{one}} = \text{GLWE}_S^0(\Delta), \quad \mathbf{C}_{\text{zero}} = \text{GLWE}_S^0(0).$$

Incrementing the state shifts the list by one position and inserts $\mathbf{C}_{\text{one}}$ in the lowest position:

$$\text{Inc}(\mathbf{U}) = (\mathbf{C}_{\text{one}}, \mathbf{U}[0], \dots, \mathbf{U}[L-2]).$$

For each input bit, a componentwise CMux selects either the current state or its incremented form:

$$\begin{aligned} \mathbf{V}[0] &= \text{CMux}(\bar{\mathbf{A}}_{r,j}, \mathbf{U}[0], \mathbf{C}_{\text{one}}), \\ \mathbf{V}[h] &= \text{CMux}(\bar{\mathbf{A}}_{r,j}, \mathbf{U}[h], \mathbf{U}[h-1]), \quad 1 \leq h < L. \end{aligned}$$

Processing all κ bits in column j , starting from $\mathbf{U}_{c_j}$, produces $\mathbf{U}_{t_j}$.

The XOR of all state components gives the result bit and is evaluated by GLWE addition under single-bit encoding:

$$\mathbf{S}_j = \sum_{h=0}^{L-1} \mathbf{U}_{t_j}[h] = \text{GLWE}_S(\Delta(t_j \bmod 2)).$$

Algorithm 3: Thermometer Code Based Carry Scan (CarryScan)

Input: $\bar{\bar{A}}_{r,j} = \text{GGSW}_S(a_{r,j})$ for $0 \leq r < \kappa$ and $0 \leq j < w$
Input: Initial thermometer carry $C_0 = U_{c_0}$
Output: GLWE word $S = (S_0, \dots, S_{w-1})$
Output: Terminal thermometer carry C_w

```

1  $L \leftarrow 2\kappa - 1$ 
2  $C \leftarrow C_0$ 
3 for  $j = 0$  to  $w - 1$  do
4    $U \leftarrow C$ 
5   for  $r = 0$  to  $\kappa - 1$  do
6      $U^+ \leftarrow \text{Inc}(U)$ 
7     for  $h = 0$  to  $L - 1$  do
8        $V[h] \leftarrow \text{CMux}(\bar{\bar{A}}_{r,j}, U[h], U^+[h])$ 
9      $U \leftarrow V$ 
10   $S_j \leftarrow \sum_{h=0}^{L-1} U[h]$ 
11   $C \leftarrow (U[1], U[3], \dots, U[2\kappa - 3], C_{\text{zero}}, \dots, C_{\text{zero}})$ 
12 return  $(S, C)$ 

```

The next carry state is obtained by placing the second, fourth, and subsequent even-numbered components at the low positions:

$$C_{j+1}[u] = U_{t_j}[2u + 1], \quad 0 \leq u < \kappa - 1.$$

The remaining positions are filled with C_{zero} , so the resulting list represents $U_{\lfloor t_j/2 \rfloor} = U_{c_{j+1}}$.

3.3.3 Carry scan and modulus processing.

Algorithm 3 applies the thermometer code based transition across all w bit positions. It starts from an initial carry state $C_0 = U_{c_0}$ and returns both the low w -bit word and the terminal carry state.

For addition modulo 2^w , the scan starts from U_0 and runs once. The returned low word is the modular sum, and the terminal carry is discarded.

For addition modulo $2^w - 1$, let

$$B = 2^w, \quad A = \sum_{r=0}^{\kappa-1} A_r.$$

Since $B \equiv 1 \pmod{B-1}$, the terminal carry must be folded back into the low word. A scan initialized with a carry c_d computes

$$Z_d = (A + c_d) \bmod B, \quad c_{d+1} = \left\lfloor \frac{A + c_d}{B} \right\rfloor.$$

We perform three scans over the same GGSW operands, using the terminal thermometer state of each scan as the initial state of the next.

Algorithm 4: Thermometer Code Based Modular Addition (ModAdd)

Input: κ encrypted w -bit GGSW operands $\bar{\bar{A}}_0, \dots, \bar{\bar{A}}_{\kappa-1}$
Input: $M \in \{2^w, 2^w - 1\}$
Output: LSB-first GLWE word Z

```

1  $C_0 \leftarrow U_0$ 
2 if  $M = 2^w$  then
3    $(Z, C_1) \leftarrow \text{CarryScan}(\bar{\bar{A}}_0, \dots, \bar{\bar{A}}_{\kappa-1}; C_0)$ 
4 else
5    $(Z_0, C_1) \leftarrow \text{CarryScan}(\bar{\bar{A}}_0, \dots, \bar{\bar{A}}_{\kappa-1}; C_0)$ 
6    $(Z_1, C_2) \leftarrow \text{CarryScan}(\bar{\bar{A}}_0, \dots, \bar{\bar{A}}_{\kappa-1}; C_1)$ 
7    $(Z_2, C_3) \leftarrow \text{CarryScan}(\bar{\bar{A}}_0, \dots, \bar{\bar{A}}_{\kappa-1}; C_2)$ 
8    $Z \leftarrow Z_2$ 
9 return  $Z$ 

```

The first scan starts from $c_0 = 0$ and obtains the ordinary terminal carry $c_1 = \lfloor A/B \rfloor$. Writing

$$A = Bc_1 + r, \quad 0 \leq r < B,$$

the second scan folds c_1 back and produces

$$c_2 = \left\lfloor \frac{A + c_1}{B} \right\rfloor = c_1 + \left\lfloor \frac{r + c_1}{B} \right\rfloor \in \{c_1, c_1 + 1\}.$$

Here, $c_1 < B$ follows from $2 \leq \kappa \leq B$. If $c_2 = c_1$, the carry is already stable. Otherwise, $r + c_1 \geq B$, and $B \leq r + c_1 + 1 < 2B$ implies that the third scan gives $c_3 = c_1 + 1 = c_2$. Thus, the carry is stable after the third scan.

The third scan also computes the low word with this stable carry:

$$A + c_2 = Bc_2 + Z_2.$$

Consequently,

$$A = (B - 1)c_2 + Z_2, \quad Z_2 \equiv A \pmod{B - 1}.$$

The Mersenne path preserves literal zero and represents a positive multiple of $2^w - 1$ by the all-one word, as required by ZUC. The first two low words Z_0 and Z_1 are discarded and need not be retained in an implementation.

3.3.4 Applicability.

The current thermometer-code construction is specialized to multi-operand modular addition and directly supports only moduli of the form 2^w and $2^w - 1$. Addition modulo 2^w requires a single carry scan and implements ordinary word-level wraparound. Addition modulo $2^w - 1$ uses three scans to stabilize and fold back the terminal carry, assuming $2 \leq \kappa \leq 2^w$. Here, $2^w - 1$ is a Mersenne modulus and is not required to be prime. Arbitrary moduli would require an additional modular-reduction procedure and are outside the scope of the current construction.

3.4 General CMux-Tree Lookup Evaluation

3.4.1 CMux tree and mixed packing.

For CMux-tree evaluation, the input bits are represented as $\overline{\mathbf{C}}_i = \text{GGSW}_{S'}(m_i)$ for $i \in [n]$ and serve as the controls of a lookup table (LUT). We combine this construction with the mixed-packing technique described for TFHE [9], using the N coefficients of a GLWE plaintext polynomial to encode multiple LUT entries.

Consider a function $F : \mathbb{B}^n \rightarrow \mathbb{B}^t$. Its truth table has 2^n rows, each containing the t output bits of F . We arrange these bits in consecutive polynomial coefficients and pack as many complete rows as possible into each GLWE ciphertext. Assuming $t \mid N$, one ciphertext contains N/t rows, and $M = \lceil \frac{t2^n}{N} \rceil$ GLWE ciphertexts represent the complete table. Parameters may be padded to powers of two when the divisions are not exact.

The evaluation proceeds in two stages. The first $\lceil \log M \rceil$ GGSW ciphertexts select the packed GLWE ciphertext that contains the desired truth-table row. This selection is implemented as a binary CMux tree and requires $M - 1$ CMux operations. The remaining $n - \lceil \log M \rceil$ GGSW ciphertexts determine the position of the target row within the selected plaintext polynomial. They are used to homomorphically rotate the selected ciphertext until the corresponding t output coefficients occupy the first t slots. Under exact packing, this stage requires $n - \lceil \log M \rceil$ additional CMux operations. Finally, sample extraction is applied to the first t coefficients, producing the LWE encryptions of $F(m_1, \dots, m_n)$.

3.4.2 Linear-tail fusion.

Substitutions in symmetric ciphers are frequently followed by a public linear transformation. We incorporate such a transformation directly into the LUT instead of evaluating it after the lookup. Given

$$Y = \mathcal{L}(F(U))$$

for a public linear map $\mathcal{L}$, we construct the table from

$$\tilde{F}(h) = \mathcal{L}(F(h))$$

rather than from $F(h)$. The CMux tree then returns the transformed output directly.

If a linear layer combines the outputs of several independent substitutions, we decompose it as

$$\mathcal{L}(S_1(U_1), \dots, S_m(U_m)) = \bigoplus_{j=1}^m \mathcal{L}_j(S_j(U_j)).$$

The table for S_j stores its contribution $\mathcal{L}_j(S_j(\cdot))$, and the resulting GLWE ciphertexts are combined through homomorphic additions. Linear-tail fusion changes only the public table contents and preserves the same GGSW-to-GLWE lookup interface.

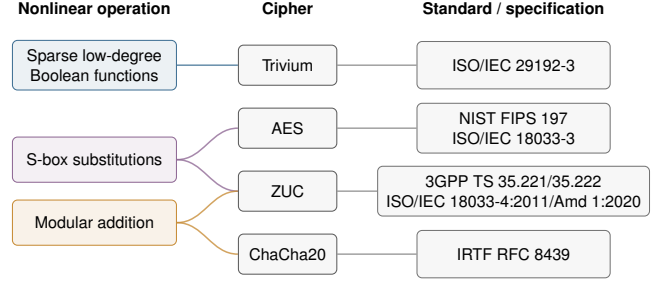


Figure 3: Representative ciphers, their nonlinear operations, and the corresponding standards or specifications.

3.4.3 Applicability.

CMux-tree lookup serves as a general fallback for nonlinear components that are not efficiently handled by the preceding two specialized methods. In particular, it applies when the target function neither has a compact low-degree ANF nor corresponds to one of the supported forms of modular addition. Because the evaluation depends only on the truth table, it makes no assumption about the algebraic or arithmetic structure of the function and can evaluate any function over a sufficiently small input domain. Mixed packing reduces the lookup cost, while linear-tail fusion allows a subsequent public linear transformation to be absorbed into the same lookup.

4 Representative Transciphering Instantiations

We select four well-established ciphers from major standards and protocol specifications: Trivium, AES, ChaCha20, and ZUC. As summarized in Figure 3, they represent the nonlinear component classes supported by our framework. Trivium uses sparse low-degree Boolean functions, AES is dominated by small-domain S-box substitutions, and ChaCha20 relies on word-level additions modulo 2^{32} . These ciphers therefore isolate type-aware ANF evaluation, packed CMux-tree lookup with linear-tail fusion, and thermometer code Based modular addition, respectively. ZUC combines S-box substitutions with additions modulo 2^{32} and $2^{31} - 1$, providing a mixed-structure instantiation.

For each cipher, we summarize the plaintext computation, identify the boundaries between linear and nonlinear modules, and map each nonlinear module to the corresponding homomorphic realization. The resulting GLWE and GGSW interfaces are connected as described in Section 3. The first three instantiations demonstrate the nonlinear modules separately, whereas ZUC shows how they compose within a complete transciphering procedure.

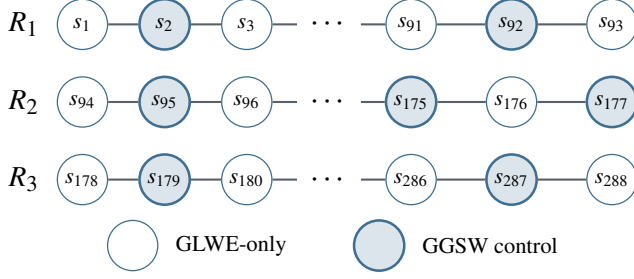


Figure 4: Alternating GLWE/GGSW assignment over the Trivium registers. Adjacent values eventually meet at a quadratic tap.

4.1 Trivium Transciphering

4.1.1 Plaintext computation.

Trivium is a bit-oriented stream cipher standardized in ISO/IEC 29192-3 [10]. Its 288-bit state consists of three shift registers of lengths 93, 84, and 111. At each clock, Trivium computes

$$\begin{aligned} u_1 &= s_{66} \oplus s_{93}, & f_1 &= u_1 \oplus (s_{91}s_{92}) \oplus s_{171}, \\ u_2 &= s_{162} \oplus s_{177}, & f_2 &= u_2 \oplus (s_{175}s_{176}) \oplus s_{264}, \\ u_3 &= s_{243} \oplus s_{288}, & f_3 &= u_3 \oplus (s_{286}s_{287}) \oplus s_{69}, \end{aligned} \quad (2)$$

and outputs $z = u_1 \oplus u_2 \oplus u_3$. The values f_3 , f_1 , and f_2 are inserted into the first, second, and third registers, respectively. The cipher first executes 1,152 initialization clocks without producing a keystream and then outputs one keystream bit per clock. The only nonlinear operations are the three quadratic terms in Equation (2).

4.1.2 ANF-based homomorphic evaluation.

The three nonlinear terms in Equation (2) are quadratic monomials; all remaining operations are XORs and state permutations. We therefore evaluate Trivium using the type-aware ANF method from Section 3.2.

The GLWE/GGSW assignment must account for the complete lifetime of each state value. Two values inserted consecutively into the same register eventually become the two inputs of a quadratic tap, so adjacent values cannot both remain GLWE-only. Alternating the GLWE-only and GGSW-control roles along each register maximizes the number of values that avoid GGSW conversion.

Figure 4 illustrates this assignment. For the initial 288-bit state, one alternating assignment retains 145 values as GLWE-only and requires GGSW representations for the remaining 143 values. As the registers advance, each newly generated feedback value receives the role opposite to the previously inserted value. The role then moves together with the state value throughout its lifetime.

Under this alternating assignment, the two inputs of every quadratic tap have different ciphertext roles. Each of the three

AND terms therefore requires one external product. The resulting GLWE ciphertexts are combined with the linear taps through GLWE additions according to Equation (2). Register shifts only move ciphertext references and require no homomorphic operation. One Trivium clock thus consists of three external products and inexpensive GLWE additions.

After the initialization phase, the three linear output taps are added to obtain the encrypted keystream bit. Adding this GLWE ciphertext to the corresponding public Trivium ciphertext bit produces a GLWE encryption of the plaintext bit.

4.1.3 Feedback noise and refresh.

Because feedback values return to later feedback computations, their noise accumulates across clocks. We estimate this growth under the selected parameters and insert a refresh before the correctness bound is exceeded. The longest register contains 111 bits, so every relevant use of a feedback value occurs within at most 111 clocks.

The refresh method follows the assigned ciphertext role. A feedback value that will serve as a GGSW control is refreshed and converted through CBS, whereas identity PBS refreshes a GLWE-only value. No additional PBS is needed when CBS already provides the required refresh. The alternating type assignment remains fixed, while the selected parameters determine the precise refresh positions.

4.1.4 Parallel evaluation.

Although Trivium is defined as a recurrent computation, a newly generated feedback value is not used immediately. The feedbacks inserted into the first and third registers first reach the taps at s_{66} and s_{243} after 66 clocks.

Consequently, 66 consecutive Trivium clocks can be evaluated as one parallel batch: no clock consumes a feedback value generated by another clock in the same batch. Once the batch has produced its feedback ciphertexts, their PBS or CBS operations have no mutual data dependencies and can also run in parallel. The refreshed feedback values are inserted into the registers before the next batch begins.

4.2 AES Transciphering

4.2.1 Plaintext decryption.

AES-128 decrypts a 128-bit state using an initial addition of K_{10} , nine inverse rounds, and a final inverse round without InvMixColumns [21]. In each middle round, AddRoundKey follows InvSubBytes, allowing the round key to be incorporated directly into the inverse-S-box lookup.

4.2.2 Homomorphic decryption.

We evaluate AES decryption with the packed-lookup method from Section 3.4, using encrypted key-dependent inverse-S-box tables. This organization resembles the lookup-oriented AES evaluation of Wang et al. [24] but is adapted to the inverse-round structure required for transciphering. During preprocessing, the client expands the master key and encrypts the resulting lookup tables. Separate round-key ciphertexts and homomorphic AddRoundKey operations are therefore unnecessary.

Let $\pi(j)$ denote the source byte selected by InvShiftRows for destination position j , and let $\mathcal{A} = \{14, 11, 13, 9\}$ contain the coefficients of InvMixColumns. We use three families of encrypted tables:

$$\mathbf{T}_j^{\text{init}}(x) = (\alpha(S^{-1}(x \oplus K_{10, \pi(j)}) \oplus K_{9, j}))_{\alpha \in \mathcal{A}},$$

$$\mathbf{T}_j^{(r)}(x) = (\alpha(S^{-1}(x) \oplus K_{r, j}))_{\alpha \in \mathcal{A}}, \quad r = 8, \dots, 1,$$

$$\mathbf{T}_j^{\text{final}}(x) = S^{-1}(x) \oplus K_{0, j}.$$

The initial and middle tables implement logical 8-to-32 lookups that return the four field-multiplication contributions required by InvMixColumns. Each logical lookup consists of four packed 8-to-8 component lookups. With $N = 1024$, one fused 8-to-32 lookup uses eight accumulators. The final round omits InvMixColumns and therefore uses a single 8-to-8 keyed lookup.

Since the AES ciphertext is public, the initial table entries are selected using public accumulator rotations and require no CBS. Their GLWE outputs are combined to evaluate InvMixColumns and then reindexed by InvShiftRows.

Each of the following eight middle layers converts the 128 state bits through CBS, evaluates sixteen keyed 8-to-32 lookups, and combines their outputs with GLWE additions. A final CBS layer followed by sixteen keyed 8-to-8 lookups returns the plaintext block directly. In total, AES decryption requires 1,152 bitwise CBS operations.

4.2.3 Refresh and parallel evaluation.

Every encrypted-input lookup layer begins with CBS, which refreshes the state while producing its GGSW controls. No additional identity PBS is required between rounds. Our optimized CBS supports a more compact parameter set than the direct instantiation of Wang et al. [24], modestly reducing the complete transciphering latency.

Within one layer, the 128 CBS operations and the sixteen byte lookups are independently parallelizable. AES rounds remain sequential because each layer consumes the state produced by its predecessor.

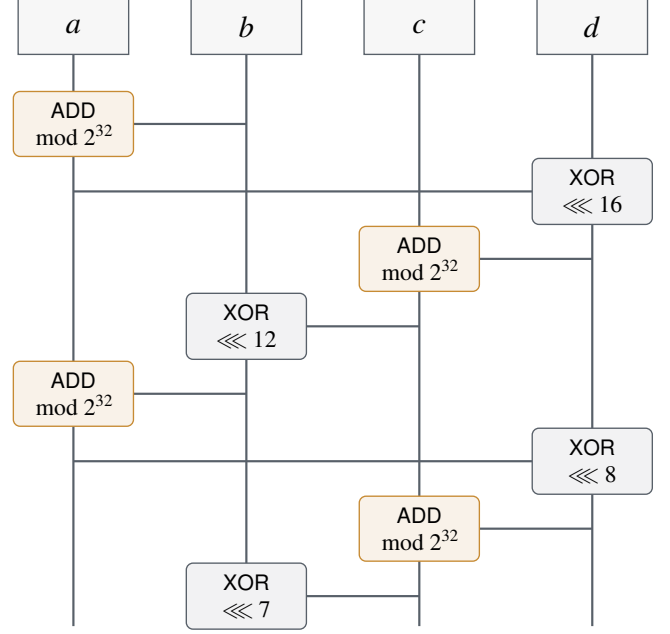


Figure 5: Data flow of the ChaCha20 quarter round.

4.3 ChaCha20 Transciphering

4.3.1 Plaintext computation and quarter-round scheduling.

ChaCha20 is a stream cipher specified in RFC 8439 [22]. Its state contains sixteen 32-bit words arranged as a 4×4 matrix. The initial state consists of four constants, eight key words, one block counter, and three nonce words. ChaCha20 executes ten double rounds, each consisting of one column round followed by one diagonal round. The four quarter rounds in each round operate on disjoint words:

$$\begin{aligned} \text{column: } & \text{QR}(0, 4, 8, 12), & \text{QR}(1, 5, 9, 13), \\ & \text{QR}(2, 6, 10, 14), & \text{QR}(3, 7, 11, 15), \\ \text{diagonal: } & \text{QR}(0, 5, 10, 15), & \text{QR}(1, 6, 11, 12), \\ & \text{QR}(2, 7, 8, 13), & \text{QR}(3, 4, 9, 14). \end{aligned}$$

As shown in Figure 5, a quarter round alternates four additions modulo 2^{32} with XORs and fixed rotations. After twenty rounds, the working state is added word-wise to the initial state to form the keystream block, which is then XORed with the input message.

4.3.2 Homomorphic quarter-round evaluation.

We represent each ChaCha20 word with 32 GLWE ciphertexts. GLWE addition evaluates XOR, while fixed rotations only permute ciphertext references. The four additions in Figure 5 use the thermometer code based modular addition from Section 3.3 with modulus 2^{32} .

The modular adder consumes GGSW input bits, so CBS converts every bit of its two GLWE operands before each ADD. A quarter round contains four ADD operations and therefore requires 256 CBS operations. Each ADD returns a 32-bit GLWE word that is used directly by the subsequent XOR and rotation. If that word later enters another ADD, the next CBS stage converts it to GGSW form.

CBS refreshes the input ciphertexts while producing the representation required by modular addition. Each ADD therefore starts from refreshed inputs, and no additional identity PBS is needed within the quarter round. The final word-wise addition with the initial state uses the same modular-adder construction. Adding the resulting GLWE keystream to the public ChaCha20 ciphertext produces GLWE encryptions of the plaintext bits.

4.3.3 Parallel evaluation.

The four quarter rounds in each column or diagonal round operate on disjoint state words and form four independent evaluation lanes. They can therefore execute concurrently.

Within a lane, the 32 bitwise CBS operations for each input word are independent, and the two words can be converted simultaneously. Across the four quarter-round lanes, a corresponding ADD stage therefore exposes up to 256 parallel CBS operations. The four ADD stages within each quarter round remain sequential because each consumes the result of the preceding modular addition and XOR.

4.4 ZUC Transciphering

4.4.1 Plaintext computation.

ZUC is a stream cipher used in the 3GPP confidentiality and integrity algorithms [14]. Its state contains sixteen 31-bit LFSR cells and two 32-bit memory words R_1 and R_2 . Bit reorganization selects portions of the LFSR cells to form four words X_0, X_1, X_2, X_3 . The nonlinear function computes

$$\begin{aligned} W &= (X_0 \oplus R_1) \boxplus_{32} R_2, \\ W_1 &= R_1 \boxplus_{32} X_1, \\ W_2 &= R_2 \oplus X_2, \end{aligned}$$

where $\boxplus_{32}$ denotes addition modulo 2^{32} . The byte-wise S-box layer then updates

$$\begin{aligned} R_1 &\leftarrow S(L_1(W_{1,L} \parallel W_{2,H})), \\ R_2 &\leftarrow S(L_2(W_{2,L} \parallel W_{1,H})), \end{aligned}$$

where L_1 and L_2 are fixed linear maps.

The LFSR feedback is

$$\begin{aligned} v &= (2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} \\ &\quad + 2^{20}s_4 + (1 + 2^8)s_0) \bmod (2^{31} - 1). \end{aligned}$$

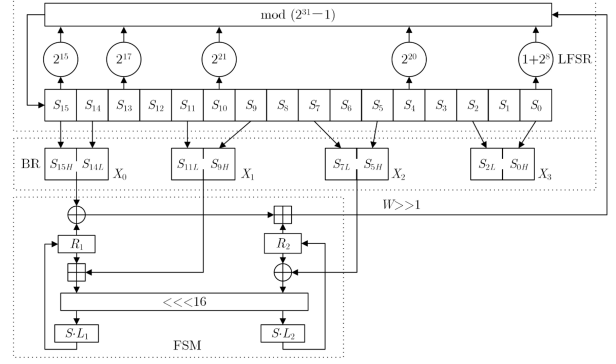


Figure 6: Structure of ZUC

This is a six-operand modular addition. During the 32 initialization clocks, $W \gg 1$ is appended as a seventh operand. After initialization and one discarded working clock, each subsequent clock outputs $Z = W \oplus X_3$.

4.4.2 Homomorphic module composition.

ZUC combines thermometer code based modular addition with packed CMux-tree lookup. Persistent LFSR cells and the memory words R_1, R_2 remain in GGSW form so that they can serve directly as modular-adder inputs and CMux controls. Intermediate arithmetic and S-box results remain in GLWE form.

Bit reorganization, word splitting, concatenation, rotations, and the maps L_1, L_2 require only ciphertext reindexing and GLWE addition. For the LFSR feedback, multiplication by a power of two modulo $2^{31} - 1$ becomes a cyclic bit rotation. The resulting six GGSW operands, together with $W \gg 1$ during initialization, are supplied to the Mersenne modular adder. Strong CBS converts its GLWE output to GGSW form before insertion into the circular LFSR buffer. The all-one output for a positive multiple of $2^{31} - 1$ directly matches the representation required by ZUC.

Within F , we form $X_0 \oplus R_1$ by componentwise addition of their GGSW ciphertexts. When both input bits are one, the resulting message is 2 rather than a normalized Boolean value. Under without-padding single-bit encoding, this message still produces the required XOR action in the modular adder but introduces additional error. The second addition directly receives the GGSW representations of R_1 and X_1 . Both additions return GLWE words.

To form W_2 , each GGSW bit of R_2 and X_2 is externally multiplied by a trivial GLWE encryption of one, and the resulting GLWE ciphertexts are added. The words W_1 and W_2 are then rearranged and transformed by L_1 and L_2 in GLWE form.

Before the S-box layer, weak CBS converts the two 32-bit linear outputs to GGSW controls. ZUC applies four byte substitutions to each word, giving eight packed 8-to-8 lookups per clock. With $N = 1024$, each byte substitution uses two

packed accumulators. The lookups return the updated R_1 and R_2 in GLWE form, and strong CBS then converts them to GGSW form for the next clock.

During initialization, $W \gg 1$ is additionally converted through strong CBS before entering the LFSR feedback. In working mode, the six-operand feedback is independent of the current W . The keystream word $Z = W \oplus X_3$ is returned in GLWE form and added to the public ZUC ciphertext to obtain encrypted plaintext bits.

4.4.3 Refresh and state maintenance.

The strong and weak CBS paths use different parameters but produce compatible ciphertexts under the same final GLWE key. Strong CBS handles persistent state, including the LFSR feedback, $W \gg 1$, and the updated R_1, R_2 . Weak CBS handles only the transient GGSW controls immediately preceding the S-box layer.

The first addition in F requires two additional carry refreshes because the GGSW representation of $X_0 \oplus R_1$ may contain messages equal to 2. We refresh its active thermometer carry after bit columns 10 and 21 using Boolean identity PBS. No carry refresh is required for $R_1 + X_1$ or for the Mersenne LFSR feedback.

A working-mode clock performs 64 weak CBS operations, 95 strong CBS operations, and two identity PBS operations. Initialization additionally converts the 31 bits of $W \gg 1$, increasing the strong-CBS count to 126. Persistent LFSR and memory values remain in GGSW form until they are replaced by the next state update.

4.4.4 Parallel execution.

During initialization, the LFSR feedback depends on $W \gg 1$, so F , the conversion of $W \gg 1$, and the feedback computation execute sequentially. In working mode, F and the six-operand LFSR feedback read the same pre-update state and can execute concurrently.

Within F , the L_1 and L_2 branches are independent. Their bitwise weak-CBS conversions expose up to 64 parallel tasks, as do the strong-CBS conversions of the updated R_1 and R_2 . The current implementation evaluates the eight byte lookups and the two modular additions sequentially because each group shares one reusable workspace.

For the LFSR feedback, active thermometer positions are divided into at most four parallel CMux groups. After modular addition, the 31 feedback bits undergo strong CBS in parallel. In working mode, this conversion can overlap with the 64-bit CBS batches in F , exposing up to 95 independent CBS tasks.

5 Experiments and Results

We evaluate the four transciphering instantiations from the preceding section: Trivium, AES, ChaCha20, and ZUC. For

each cipher, we report single-threaded and multi-threaded performance and compare it with the corresponding state-of-the-art transciphering result.

We conduct all experiments on a server with two 48-core AMD EPYC 9474F processors running at 3.6 GHz and 1.5 TB of memory. The platform provides 192 hardware threads. The multi-threaded experiments impose no additional concurrency limit and may use all available hardware threads. For consistency with prior work, AVX-512 is disabled in every experiment. Each reported result is averaged over more than 100 executions.

5.1 Parameters and security

We use the Lattice Estimator¹ to verify that the underlying LWE and GLWE parameter sets provide at least 128-bit security. Homomorphic operations subsequently increase the ciphertext noise, and decryption remains correct only while the accumulated noise stays within the corresponding decoding interval. Following the noise-estimation methodology of the refined TFHE evaluation [24], we select the parameters in Table 1 so that the estimated decryption-failure probability is at most 2^{-40} .

In Table 1, Q is the ciphertext modulus; n and σ_{LWE} are the LWE dimension and error standard deviation; and N , k , and σ_{GLWE} are the polynomial size, GLWE dimension, and per-coefficient GLWE error standard deviation. A GLWE ciphertext contains k mask polynomials and one body polynomial over $\mathbb{Z}_Q[X]/(X^N + 1)$. The standard deviations are reported in the integer scale modulo Q . For $\star \in \{\text{ks}, \text{pbs}, \text{tr}, \text{ss}\}$, $(\ell_\star, B_\star)$ denotes the gadget-decomposition level and base used for key switching, PBS, reverse homomorphic trace, and scheme switching, respectively. The split-FFT parameter b_{tr} specifies the coefficient-splitting bit position used during trace evaluation, while (ℓ, B) denotes the gadget parameters of the GGSW ciphertext produced by circuit bootstrapping. A dash indicates that the corresponding operation is not used.

5.2 Results

Table 2 compares our implementations with the corresponding state-of-the-art results for AES [24], ChaCha20 [17], Trivium [2], and ZUC [26]. It reports the average processing time per bit and the end-to-end latency under single-threaded (ST) and multi-threaded (MT) settings. Latency is measured for a 128-bit AES block, a 512-bit ChaCha20 block, one Trivium output bit, and a 32-bit ZUC word.

Each parenthesized factor is the baseline result divided by ours under the same threading configuration. A value above one means that our implementation is faster; a value below one means that the baseline is faster. The final column reports our internal speedup from ST to MT execution.

¹<https://github.com/malb/lattice-estimator>

Table 1: Parameters used by the homomorphic cipher implementations.

Name	Q	n	$Std. Dev. LWE$	N	k	$Std. Dev. GLWE$	ℓ_{ks}	B_{ks}	ℓ_{pbs}	B_{pbs}	ℓ_{tr}	B_{tr}	b_{tr}	ℓ_{ss}	B_{ss}	ℓ	B
AES	2^{64}	640	1.2928×10^{15}	1024	2	5.4240×10^3	4	2^3	1	2^{23}	2	2^{17}	45	2	2^{17}	6	2^2
ChaCha20	2^{64}	640	1.2928×10^{15}	256	7	2.0282×10^6	5	2^2	2	2^{14}	3	2^{11}	38	2	2^{15}	2	2^6
ZUC-Strong	2^{64}	640	1.2928×10^{15}	1024	2	5.4240×10^3	5	2^2	2	2^{15}	2	2^{17}	45	2	2^{17}	3	2^5
ZUC-Weak	2^{64}	640	1.2928×10^{15}	1024	2	5.4240×10^3	5	2^2	1	2^{23}	2	2^{17}	45	2	2^{17}	4	2^4
ZUC-PBS	2^{64}	640	1.2928×10^{15}	1024	2	5.4240×10^3	5	2^2	2	2^{15}	—	—	—	—	—	—	—
Trivium-CBS	2^{64}	640	1.2928×10^{15}	1024	2	5.4240×10^3	5	2^2	1	2^{23}	2	2^{17}	45	2	2^{17}	3	2^4
Trivium-PBS	2^{64}	640	1.2928×10^{15}	1024	2	5.4240×10^3	5	2^2	1	2^{23}	—	—	—	—	—	—	—

Table 2: Average per-bit execution time and end-to-end latency compared with prior implementations.

Cipher	Metric	Ours		Baseline		Parallel Speedup	
		ST	MT	ST	MT	Ours	
AES-128	Average (ms)	108.037	4.053	127.562 (1.181 $\times$)		—	26.654 $\times$
	Latency (s)	13.829	0.519	16.328 (1.181 $\times$)		—	
ChaCha20	Average (ms)	662.228	15.247	265.598 (0.401 $\times$)		84.113 (5.517 $\times$)	43.434 $\times$
	Latency (s)	339.061	7.806	135.986 (0.401 $\times$)		43.066 (5.517 $\times$)	
Trivium	Average (ms)	33.206	1.218	122.996 (3.704 $\times$)		5.815 (4.772 $\times$)	27.253 $\times$
	Latency* (s)	2.918	0.107	7.87 (2.69 $\times$)		0.37 (3.46 $\times$)	
ZUC	Average (ms)	73.206	6.236	296.880 (4.055 $\times$)		—	11.739 $\times$
	Latency (s)	2.343	0.200	9.500 (4.055 $\times$)		—	

5.3 Discussion

5.3.1 Comparison with AES transciphering

AES achieves a 1.181 $\times$ single-threaded speedup because its baseline already uses a similar lookup-oriented evaluation; the improvement mainly comes from tighter bootstrapping parameters. The baseline does not implement parallel evaluation, so we leave that data empty.

5.3.2 Comparison with Trivium

For Trivium, Maximum independent set based ANF Evaluation reduces the number of inputs requiring CBS. Under current parameter setting, it will execute 132 normal step and followed by 111 refresh step. resulting in end-to-end average speedups of 3.704 $\times$ and 4.772 $\times$ under the ST and MT setting respectively. Different output bits incur different homomorphic workloads, so the slowest evaluation path determines latency and yields a smaller improvement than the average per-bit time. We estimate the latency of each 64 bits block which achieve 2.69 $\times$ and 3.46 $\times$ under the ST and MT setting respectively.

5.3.3 Comparison with ZUC.

ZUC combines lookup-based evaluation with modular addition. Under the single-threaded configuration, CORAL

achieves a 4.055 $\times$ speedup over the previously reported state-of-the-art result. Because the prior work does not provide a publicly available end-to-end implementation, we use the performance reported in the corresponding article as the baseline. This comparison should therefore be interpreted as a cross-paper comparison rather than a direct re-execution of the baseline. Notably, the baseline employs AVX-512 acceleration, whereas our implementation does not use AVX-512 instructions; nevertheless, CORAL still achieves a substantial improvement. Since the prior work reports no multi-threaded measurements, we mark the corresponding table entries as unavailable rather than extrapolating unreported results.

5.3.4 Comparison with ChaCha20

ChaCha20 single-threaded execution is slower than the baseline, as indicated by the 0.401 $\times$ relative factor, because the thermometer code based adder performs substantial homomorphic work. The computation nevertheless exposes extensive parallelism across the four independent quarter-round lanes and the bitwise CBS conversions. As a result, the MT implementation is 5.517 $\times$ faster than the MT baseline and achieves a 43.434 $\times$ internal ST-to-MT speedup.

Ethical Considerations

Research procedures and stakeholders. This work studies the design, implementation, and performance of homomorphic transciphering. All experiments use randomly generated keys and plaintexts or publicly available standard test vectors. The study involves no human participants, personal or proprietary data, user accounts, live services, third-party systems, or vulnerability exploitation. Its direct stakeholders are cryptographic researchers, implementers, and potential system operators, while its indirect stakeholders include users whose data may eventually be protected by homomorphic processing.

Benefits and potential risks. The intended benefit of this work is to improve the practicality of privacy-preserving computation by allowing compact symmetric-ciphertext inputs to be processed homomorphically. The main potential risk is premature or incorrect deployment. An implementation error, an unsuitable parameter set, or an incorrect interpretation of the reported noise analysis could undermine the confidentiality or correctness expected by users.

Mitigations and publication decision. To reduce the risk of incorrect use or interpretation, we provide explicit parameter sets, plaintext reference implementations, standard-vector and end-to-end correctness tests, and documented benchmark boundaries.

Open Science

We embrace the principles of transparency, reproducibility, and long-term availability in scientific research. All research artifacts associated with this work are publicly available through an anonymized repository at <https://anonymous.4open.science/r/transciphering-CBS-16D8>.

The repository contains:

- **Source Code:** A complete Rust implementation of the proposed bitwise homomorphic transciphering framework, primarily based on TFHE-rs. It includes homomorphic implementations of AES-128, ChaCha20, ZUC-128, and Trivium, together with the shared cryptographic components used by these constructions, including circuit bootstrapping, programmable bootstrapping, thermometer-state modular addition, GGSW addition, and packed CMux-tree lookup tables.
- **Correctness Tests:** End-to-end tests compare the outputs of the homomorphic cipher implementations with their corresponding plaintext reference implementations. Additional unit tests validate the shared homomorphic components, parameter compatibility, state transitions, and standard test vectors where applicable.
- **Benchmark Evaluation:** The repository provides benchmarks for both complete transciphering constructions and their principal homomorphic building blocks. These include:
 - end-to-end performance evaluation of homomorphic AES-128 decryption;
 - evaluation of the homomorphic ChaCha20 block function;
 - evaluation of the initialization and working phases of homomorphic ZUC-128;
 - evaluation of the initialization and keystream-generation phases of homomorphic Trivium;
- **Parallel Evaluation:** The implementations provide dedicated single-threaded and multithreaded execution paths. The number of worker threads can be selected through the `RAYON_NUM_THREADS` environment variable, enabling reproducible evaluation under different levels of parallelism.
- **Documentation:** The top-level `README.md` provides instructions for compiling, testing, and benchmarking the artifact. Layered documentation further describes the homomorphic computation flow, state representation, refresh strategy, and parallelization design of each supported cipher. An offline deployment guide is also included for reproducing the experiments on Linux machines without network access.

All input preparation, key generation, and correctness verification are performed outside the timed benchmark regions unless explicitly stated otherwise. This organization enables independent verification of both the functional correctness and the reported performance of the proposed constructions. To ensure long-term accessibility, we also intend to deposit the artifact in a permanent, citable archive.

References

- [1] Hossein Abdinassibfar, Camille Nuoskala, and Antonis Michalas. HHE Kombit: Benchmarking hybrid homomorphic encryption schemes. *Cryptology ePrint Archive*, 2026.
- [2] Thibault Balenbois, Jean-Baptiste Orfila, and Nigel Smart. Trivial transcribing with Trivium and TFHE. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 69–78, 2023.
- [3] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference, DAC ’15*, pages 175:1–175:6. ACM, 2015. doi:10.1145/2744769.2747946.
- [4] Sonia Belaïd, Nicolas Bon, Aymen Boudguiga, Renaud Sirdey, Daphné Trama, and Nicolas Ye. Further improvements in aes execution over tfhe. *IACR Communications in Cryptology*, 2(1), 2025.
- [5] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and Charlotte VIKKESOE. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007. URL: <https://www.iacr.org/archive/ches2007/47270450/47270450.pdf>, doi:10.1007/978-3-540-74735-2_31.
- [6] Nicolas Bon, David Pointcheval, and Matthieu Rivain. Optimized homomorphic evaluation of boolean functions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024:302–341, 2024.
- [7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory*, 6(3):1–36, 2014.
- [8] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.
- [9] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.
- [10] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In *International Conference on Information Security*, pages 171–186. Springer, 2006.
- [11] Thomas de Ruijter, Jan-Pieter D’Anvers, and Ingrid Verbauwhede. Don’t be mean: Reducing approximation noise in TFHE through mean compensation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(1):82–104, 2026. URL: <https://doi.org/10.46586/tches.v2026.i1.82-104>, doi:10.46586/TCHES.V2026.I1.82-104.
- [12] Morris J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Information Processing Standards Publication 202, National Institute of Standards and Technology, Gaithersburg, MD, August 2015. URL: <https://csrc.nist.gov/pubs/fips/202/final>, doi:10.6028/NIST.FIPS.202.
- [13] ETSI/SAGE. Specification of the 3GPP confidentiality and integrity algorithms UEA2 & UIA2: Document 2, SNOW 3G specification. Algorithm Specification Version 1.1, European Telecommunications Standards Institute, September 2006. URL: <https://www.gsma.com/security/wp-content/uploads/2019/05/snow3gspec.pdf>.
- [14] ETSI/SAGE. Specification of the 3GPP confidentiality and integrity algorithms 128-EEA3 & 128-EIA3; document 2: ZUC specification. Algorithm Specification Version 1.6, European Telecommunications Standards Institute, June 2011. URL: <https://www.gsma.com/security/wp-content/uploads/2019/05/eea3eia3zucv16.pdf>.
- [15] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.
- [16] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, pages 169–178, 2009.
- [17] Antonio Guimarães, Gabriela M. Jacob, and Hilder V. L. Pereira. PETCHA: Post-quantum efficient transcribing with ChaCha. In *International Conference on Post-Quantum Cryptography*, pages 178–203. Springer, 2026.
- [18] Martin Hell, Thomas Johansson, Alexander Maximov, and Willi Meier. The Grain family of stream ciphers. In Matthew Robshaw and Olivier Billet, editors, *New Stream Cipher Designs: The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 179–190. Springer, 2008. doi:10.1007/978-3-540-68351-3_14.

- [19] Deukjo Hong, Jung-Keun Lee, Dong-Chan Kim, Dae-sung Kwon, Kwon Ho Ryu, and Dong-Geon Lee. LEA: A 128-bit block cipher for fast encryption on common processors. In *Information Security Applications – WISA 2013*, volume 8267 of *Lecture Notes in Computer Science*, pages 3–27. Springer, 2014. doi:[10.1007/978-3-319-05149-9_1](https://doi.org/10.1007/978-3-319-05149-9_1).
- [20] Kang Hoon Lee and Ji Won Yoon. Homomorphic field trace revisited: Breaking the cubic noise barrier. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(1):105–134, 2026.
- [21] National Institute of Standards and Technology. Advanced Encryption Standard (AES). Federal Information Processing Standards Publication FIPS 197-upd1, National Institute of Standards and Technology, 2023. URL: <https://csrc.nist.gov/pubs/fips/197/final>, doi:[10.6028/NIST.FIPS.197-upd1](https://doi.org/10.6028/NIST.FIPS.197-upd1).
- [22] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF protocols. Request for Comments RFC 8439, RFC Editor, June 2018. doi:[10.17487/RFC8439](https://doi.org/10.17487/RFC8439).
- [23] Meltem Sönmez Turan, Kerry McKay, Jinkeon Kang, John Kelsey, and Donghoon Chang. Ascon-based lightweight cryptography standards for constrained devices: Authenticated encryption, hash, and extendable output functions. NIST Special Publication NIST SP 800-232, National Institute of Standards and Technology, 2025. URL: <https://csrc.nist.gov/pubs/sp/800/232/final>, doi:[10.6028/NIST.SP.800-232](https://doi.org/10.6028/NIST.SP.800-232).
- [24] Ruida Wang, Jincheol Ha, Xuan Shen, Xianhui Lu, Chunling Chen, Kunpeng Wang, and Jooyoung Lee. Refined TFHE leveled homomorphic evaluation and its application. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 2309–2323, 2025. doi:[10.1145/3719027.3744873](https://doi.org/10.1145/3719027.3744873).
- [25] Ruida Wang, Yundi Wen, Zhihao Li, Xianhui Lu, Benqiang Wei, Kun Liu, and Kunpeng Wang. Circuit bootstrapping: Faster and smaller. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 342–372. Springer, 2024.
- [26] Benqiang Wei, Xianhui Lu, Ruida Wang, Kun Liu, Zhihao Li, and Kunpeng Wang. Thunderbird: Efficient homomorphic evaluation of symmetric ciphers in 3GPP by combining two modes of TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):530–573, 2024.

A Detailed Homomorphic Operation Procedures

This appendix gives the detailed calculations for the homomorphic interfaces summarized in Section 2.3. The notation and ciphertext formats are defined in Section 2; here we retain the full computational procedures, noise-related transformations, and algorithms needed to reproduce the implementation.

A.1 Basic Homomorphic Operations

External Product. The external product implements the mapping $(\text{GGSW}(M_1), \text{GLWE}(M_2)) \rightarrow \text{GLWE}(M_1 \cdot M_2)$. Let $\bar{\bar{C}}_1$ be a GGSW encryption of M_1 , and let $C_2 = (A_1, \dots, A_{k+1})$ be a GLWE encryption of M_2 . For each $i \in [k+1]$, write $(A_{i,1}, \dots, A_{i,\ell})$ for the gadget decomposition of A_i . Their external product $\boxtimes$ is

$$\bar{\bar{C}}_1 \boxtimes C_2 = \sum_{i=1}^{k+1} \sum_{j=1}^{\ell} A_{i,j} \cdot \text{GLWE}(v_j \cdot (-S_i \cdot M_1)).$$

Here, $\text{GLWE}(v_j \cdot (-S_i \cdot M_1))$ denotes the (i, j) -th GLWE component of $\bar{\bar{C}}_1$. Polynomial multiplication is applied to every component of a GLWE ciphertext. The result is a GLWE encryption of $M_1 M_2$.

CMux. We use the controlled multiplexer CMux to select between two GLWE ciphertexts according to an encrypted Boolean value. Given $\bar{\bar{C}} = \text{GGSW}_S(b)$ and two GLWE ciphertexts C_0 and C_1 under S , we define

$$\text{CMux}(\bar{\bar{C}}, C_0, C_1) = C_0 + \bar{\bar{C}} \boxtimes (C_1 - C_0).$$

The result encrypts the message in C_b .

Key Switching. Let $S = (S_1, \dots, S_k)$ and $S' = (S'_1, \dots, S'_{k'})$ be two GLWE secret keys with the same polynomial size N . GLWE key switching converts a ciphertext under S into a ciphertext encrypting the same message under S' . For a decomposition base B and length ℓ , we define the key-switching key as $\text{KS}_{S \rightarrow S'} = \{\text{KS}_{S \rightarrow S'}[i][j]\}_{i \in [k], j \in [\ell]}$, where $\text{KS}_{S \rightarrow S'}[i][j] = \text{GLWE}_{S'}(v_j \cdot S_i)$ and $v_j = \lceil q/B^j \rceil$.

Given $C = (A_1, \dots, A_k, A_{k+1}) = \text{GLWE}_S(M)$, we initialize $C' = (0, \dots, 0, A_{k+1}) \in \mathcal{R}_{q,N}^{k'+1}$. For each $i \in [k]$, we decompose A_i as $A_i = \sum_{j=1}^{\ell} A'_{i,j} \cdot v_j + E'_i$ and update

$$C' \leftarrow C' - \sum_{i=1}^k \sum_{j=1}^{\ell} A'_{i,j} \cdot \text{KS}_{S \rightarrow S'}[i][j].$$

The resulting ciphertext $C' = \text{GLWE}_{S'}(M)$ encrypts the same message under S' . Its additional error comes from the gadget decomposition and the key-switching ciphertexts. We denote this operation by $\text{GLWEKeySwitch}_{S \rightarrow S'}$.

LWE Modulus Switching. We use modulus switching to map an LWE ciphertext from modulus Q to the rotation modulus $2N$ used by blind rotation. Let $\mathbf{c} = (a_1, \dots, a_n, b) \in \mathbb{Z}_Q^{n+1}$ be an LWE ciphertext under the secret key $\mathbf{s} = (s_1, \dots, s_n)$. We first compute $\hat{a}_i = \lfloor (2N/Q)a_i \rfloor$ and the corresponding rounding error $u_i = (2N/Q)a_i - \hat{a}_i$ for each $i \in [n]$. Following mean-compensated modulus switching [11], we use the public mean μ_s of the secret-key coefficients to correct the ciphertext body before rounding:

$$\hat{b} = \left\lfloor \frac{2N}{Q}b - \mu_s \sum_{i=1}^n u_i \right\rfloor, \quad \hat{\mathbf{c}} = (\hat{a}_1, \dots, \hat{a}_n, \hat{b}) \in \mathbb{Z}_{2N}^{n+1}.$$

The phase of $\hat{\mathbf{c}}$ approximates the input phase scaled by $2N/Q$ under the new modulus. The compensation term centers the rounding contribution associated with the secret-key coefficients and thereby reduces the modulus-switching noise.

Sample Extraction. We use sample extraction to convert a selected coefficient of a GLWE ciphertext into an LWE ciphertext. Let $C = \text{GLWE}_S(M(X))$, where $M(X) = \sum_{i=0}^{N-1} m_i X^i$ and $S = (S_1, \dots, S_k)$. For $S_j(X) = \sum_{t=0}^{N-1} s_{j,t} X^t$, we define the LWE secret-key component associated with the coefficient of degree i as $\pi_i(S_j) = (s_{j,i}, s_{j,i-1}, \dots, s_{j,0}, -s_{j,N-1}, \dots, -s_{j,i+1})$, and write $\mathbf{s}^{(i)} = (\pi_i(S_1), \dots, \pi_i(S_k))$. We then define

$$\text{SampleExtract}_i(C) = \text{LWE}_{\mathbf{s}^{(i)}}(m_i).$$

The negacyclic arrangement of $\mathbf{s}^{(i)}$ depends on the extracted coefficient. In particular, extracting the constant coefficient yields $\pi_0(S_j) = (s_{j,0}, -s_{j,N-1}, \dots, -s_{j,1})$, whereas extracting a higher-degree coefficient produces the corresponding rotated and sign-adjusted key. When we omit the index, SampleExtract extracts m_0 .

A.2 Bootstrapping

Programmable Bootstrapping. Programmable bootstrapping (PBS) refreshes an LWE ciphertext while evaluating a public negacyclic function [9]. We call $f: \mathbb{Z}_{2N} \rightarrow \mathbb{Z}_{2N}$ negacyclic if $f(x+N) = -f(x)$ for $0 \leq x < N$. This property follows from $X^{x+N} = -X^x$ in $\mathcal{R}_{2N,N}$ and allows us to encode f as a test polynomial.

Given $\mathbf{c} = \text{LWE}_S(\Delta m) \in \mathbb{Z}_{2N}^{n+1}$, PBS encodes f in a trivial accumulator $C_f = \text{GLWE}_{S'}^0(P_f)$ and uses $\text{BSK} = (\text{GGSW}_{S'}(s_i))_{i \in [n]}$ to rotate the accumulator by the encrypted phase of $\mathbf{c}$. We denote this operation by BlindRotate . After blind rotation, the designated accumulator coefficient contains $f(m)$, which can be converted into an LWE ciphertext by sample extraction.

PBSmanyLUT evaluates up to 2^ϑ negacyclic functions of the same encrypted input with one blind rotation [9]. It rounds the LWE coefficients to multiples of 2^ϑ and encodes $f_1, \dots, f_{2^\vartheta}$ in consecutive coefficients of a redundant lookup polynomial.

Algorithm 5: PBSmanyLUT

Input: $\mathbf{c}_{\text{in}} = \text{LWE}_{\mathbf{s}}(m \cdot \Delta_{\text{in}}) = (a_1, \dots, a_n, a_{n+1} = b) \in \mathbb{Z}_{2N}^{n+1}$ under $\mathbf{s} = (s_1, \dots, s_n)$

Input: $\text{BSK} = (\text{GGSW}_{\mathbf{S}'}(s_i))_{1 \leq i \leq n}$ under $\mathbf{S}' = (S'_1, \dots, S'_k)$ with decomposition base B_{pbs} and level ℓ_{pbs}

Input: $P_{(f_1, \dots, f_{2^\vartheta})}$: a redundant LUT for the negacyclic functions $f_1, \dots, f_{2^\vartheta}$

Output: $\mathbf{c}_1, \dots, \mathbf{c}_{2^\vartheta}$, where $\mathbf{c}_j = \text{LWE}_{\mathbf{S}'}((-1)^\beta \cdot f_j(m') \cdot \Delta_{\text{out}})$

- 1 **for** $i = 1$ **to** n **do**
- 2 $a'_i \leftarrow \lfloor [a_i \cdot 2^{-\vartheta}] \cdot 2^\vartheta \rfloor_{2N}$
- 3 $u_i \leftarrow a_i \cdot 2^{-\vartheta} - \lfloor a_i \cdot 2^{-\vartheta} \rfloor$
- 4 $a'_{n+1} \leftarrow \lfloor [b \cdot 2^{-\vartheta} - \mu_s \sum_{i=1}^n u_i] \cdot 2^\vartheta \rfloor_{2N}$
- 5 $\mathbf{C}_{(f_1, \dots, f_{2^\vartheta})} \leftarrow \text{GLWE}_{\mathbf{S}'}^0(P_{(f_1, \dots, f_{2^\vartheta})})$
- 6 $\mathbf{C} \leftarrow \text{BlindRotate}(\mathbf{C}_{(f_1, \dots, f_{2^\vartheta})}, \{a'_i\}_{i=1}^{n+1}, \text{BSK})$
- 7 **for** $j = 1$ **to** 2^ϑ **do**
- 8 $\mathbf{c}_j \leftarrow \text{SampleExtract}_{j-1}(\mathbf{C})$
- 9 **return** $\mathbf{c}_1, \dots, \mathbf{c}_{2^\vartheta}$

We also use PBSmanyLUT without the final sample-extraction loop and denote this GLWE-output interface by $\text{PBSmanyLUT}_{\text{GLWE}}$.

Homomorphic Trace. For $d \in \mathbb{Z}_{2N}^\times$, we define the ring automorphism τ_d by $\tau_d(M(X)) = M(X^d)$. Applying τ_d to a GLWE ciphertext changes its secret key from $\mathbf{S}(X)$ to $\mathbf{S}(X^d)$. We therefore apply the automorphism to every ciphertext component and then use $\text{GLWEKeySwitch}_{\mathbf{S}(X^d) \rightarrow \mathbf{S}(X)}$ to restore the original key. We denote the resulting homomorphic operation by $\text{EvalAuto}(\mathbf{C}, d)$.

For $M(X) = m_0 + m_1X + \dots + m_{N-1}X^{N-1}$, the field trace satisfies

$$\text{Tr}(M(X)) = \sum_{d \in \mathbb{Z}_{2N}^\times} \tau_d(M(X)) = N \cdot m_0.$$

Thus, homomorphic trace eliminates all nonconstant coefficients, but its standard evaluation also multiplies the message and ciphertext error by N .

We use the amplification-removing reverse homomorphic trace RevHomTrace of Lee and Yoon [20]. The algorithm evaluates the automorphisms in reverse order and divides the entire GLWE ciphertext by two before each automorphism step. It therefore returns an encryption of m_0 rather than Nm_0 . This removes the phase-amplification factor and reduces trace-noise accumulation from $O(N^3)$ to $O(N \log N)$.

Circuit Bootstrapping. Circuit bootstrapping converts an LWE encryption of a Boolean message into a GGSW encryption of the same message:

$$\text{CBS}(\text{LWE}_{\mathbf{s}}(\Delta m)) = \text{GGSW}_{\mathbf{S}'}(m), \quad m \in \mathbb{B}.$$

Algorithm 6: Reverse Homomorphic Trace Evaluation (RevHomTrace [20])

Input: $\mathbf{C} = \text{GLWE}_{\mathbf{S}(X)}(M(X))$, where $M(X) = m_0 + m_1X + \dots + m_{N-1}X^{N-1}$

Input: $\text{AutoKey}_d = \text{KS}_{\mathbf{S}(X^d) \rightarrow \mathbf{S}(X)}$ for $d \in \mathbb{Z}_{2N}^\times$

Output: $\mathbf{C}' = \text{GLWE}_{\mathbf{S}(X)}(m_0)$

- 1 $\mathbf{C}' \leftarrow \mathbf{C}$
- 2 **for** $d = 1$ **to** $\log N$ **do**
- 3 $(A'_1, \dots, A'_{k+1}) \leftarrow \mathbf{C}'$
- 4 $\mathbf{C}' \leftarrow \left(\lfloor \frac{A'_1}{2} \rfloor, \dots, \lfloor \frac{A'_{k+1}}{2} \rfloor \right)$
- 5 $\mathbf{C}' \leftarrow \mathbf{C}' + \text{EvalAuto}(\mathbf{C}', 2^d + 1)$
- 6 **return** $\mathbf{C}'$

$$\boxed{\text{LWE}_{\mathbf{s}}(\Delta m)} \xrightarrow{\text{PBSmanyLUT}_{\text{GLWE}}} \boxed{\{\text{GLWE}_{\mathbf{S}'}(v_j m + R_j(X))\}_{j \in [\ell]}} \xrightarrow{\text{RevHomTrace}} \boxed{\{\text{GLWE}_{\mathbf{S}'}(v_j m)\}_{j \in [\ell]}} \xrightarrow{\text{SchemeSwitch}} \boxed{\text{GGSW}_{\mathbf{S}'}(m)}$$

Figure 7: The single-bit circuit-bootstrapping workflow. The refresh step produces gadget-scaled GLWE encryptions with redundant polynomial coefficients, while the conversion step removes these coefficients and assembles the resulting ciphertexts into GGSW form.

We follow the refined CBS construction of Wang et al. [24], replacing its trace evaluation with RevHomTrace [20]. The procedure has two stages: refresh and conversion.

The refresh step uses $\text{PBSmanyLUT}_{\text{GLWE}}$ to evaluate the gadget-scaled functions $f_j(m) = v_j m$ for $j \in [\ell]$, where $\ell \leq 2^\vartheta$. It returns one GLWE accumulator whose coefficient of degree $j-1$ contains $v_j m$. We multiply the accumulator by $X^{-(j-1)}$ to move this coefficient to the constant position, obtaining $\mathbf{C}_j = \text{GLWE}_{\mathbf{S}'}(v_j m + R_j(X))$, where $R_j(X)$ contains the remaining redundant coefficients.

The conversion step independently applies RevHomTrace to each $\mathbf{C}_j$, producing $\mathbf{C}'_j = \text{GLWE}_{\mathbf{S}'}(v_j m)$. We then apply SchemeSwitch to the collection $(\mathbf{C}'_1, \dots, \mathbf{C}'_\ell)$. Scheme switching constructs the GLWE components encrypting $v_j(-S'_i m)$ for $(i, j) \in [k+1] \times [\ell]$ and arranges them as the GGSW ciphertext defined in Equation (1).

The resulting GGSW ciphertext can be reused as the control of external products and CMux gates.

Algorithm 7: Single-Bit Circuit Bootstrapping with RevHomTrace

Input: $\mathbf{c} = \text{LWE}_s(\Delta m)$, where $m \in \mathbb{B}$

Input: BSK and a redundant LUT P_{CBS} encoding

$$f_j(m) = v_j m \text{ for } j \in [\ell]$$

Input: Evaluation keys for RevHomTrace and SchemeSwitch

Output: $\overline{\overline{\mathbf{C}}} = \text{GGSW}_{S'}(m)$

```

1  $\mathbf{C}_{\text{acc}} \leftarrow \text{PBSmanyLUT}_{\text{GLWE}}(\mathbf{c}, P_{\text{CBS}}, \text{BSK})$ 
2 for  $j = 1$  to  $\ell$  in parallel do
3    $\mathbf{C}_j \leftarrow X^{-(j-1)} \cdot \mathbf{C}_{\text{acc}}$  //  $\mathbf{C}_j = \text{GLWE}_{S'}(v_j m + R_j(X))$ 
4    $\mathbf{C}'_j \leftarrow \text{RevHomTrace}(\mathbf{C}_j)$  //  $\mathbf{C}'_j = \text{GLWE}_{S'}(v_j m)$ 
5  $\overline{\overline{\mathbf{C}}} \leftarrow \text{SchemeSwitch}(\mathbf{C}'_1, \dots, \mathbf{C}'_\ell)$ 
6 return  $\overline{\overline{\mathbf{C}}}$ 

```
