Further Improvements in AES Execution over TFHE: Towards Breaking the 1 sec Barrier

Sonia Belaïd[1], Nicolas Bon[1,2], Aymen Boudguiga[3], Renaud Sirdey[3], Daphné Trama[3] and Nicolas Ye[3]

[1] CryptoExperts, Paris, France
name.surname@cryptoexperts.com
[2] DIENS, Ecole normale supérieure, PSL University, CNRS, Inria, Paris, France
nicolas.bon@ens.fr
[3] Université Paris-Saclay, CEA LIST, Palaiseau, France
name.surname@cea.fr

Abstract. Making the most of TFHE advanced capabilities such as programmable or circuit bootstrapping and their generalizations for manipulating data larger than the native plaintext domain of the scheme is a very active line of research. In this context, AES is a particularly interesting benchmark, as an example of a nontrivial algorithm which has eluded "practical" FHE execution performances for years, as well as the fact that it will most likely be selected by NIST as a flagship reference in its upcoming call on threshold (homomorphic) cryptography. Since 2023, the algorithm has thus been the subject of a renewed attention from the FHE community and has served as a playground to test advanced operators following the LUT-based, p-encodings or several variants of circuit bootstrapping, each time leading to further timing improvements. Still, AES is also interesting as a benchmark because of the tension between boolean- and byte-oriented operations within the algorithm. In this paper, we resolve this tension by proposing a new approach, coined "Hippogryph", which consistently combines the (byte-oriented) LUT-based approach with a generalization of the (boolean-oriented) p-encodings one to get the best of both worlds. In doing so, we obtain the best timings so far, getting a *single-core* execution of the algorithm over TFHE from 46 down to 32 seconds and approaching the 1 second barrier with only a mild amount of parallelism. We should also stress that all the timings reported in this paper are consistently obtained on the same machine which is often not the case in previous studies. Lastly, we emphasize that the techniques we develop are applicable beyond just AES since the boolean-byte tension is a recurrent issue when running algorithms over TFHE.

1 Introduction

Fully Homomorphic Encryption (FHE) is a corpus cryptographic techniques that allows data to be processed while remaining encrypted, without any need for decryption. Various FHE schemes, such as BGV [BGV12], designed for general computation, CKKS [CKKS17], optimized for approximate arithmetic, and TFHE [CGGI16, CGGI20], specialized for binary operations and low-latency bootstrapping, offer different trade-offs in terms of functionality and performance. Although FHE provides strong end-to-end encryption, it still faces significant efficiency challenges. One of the main limitations is the substantial ciphertext expansion, which hampers fast data transmission to the server.

To mitigate this issue of large uplink data transmission with FHE, it is now standard to rely on a method called *transciphering*. In this approach, the client first encrypts its data using a symmetric encryption scheme and sends both the encrypted data and (once and for all) the FHE-encrypted symmetric key to the server. Leveraging its encrypted-domain computing capabilities, the server can then decrypt the encrypted data *within the homomorphic domain*, ultimately producing homomorphic ciphertexts on which it can perform the requested calculations.

The first attempt to transcipher AES ciphertexts into FHE data was made in 2012 by Gentry, Halevi, and Smart [GHS12]. They used the BGV scheme [BGV12], a fully homomorphic encryption method based on the Ring-LWE problem, as implemented in HElib [HS20], an open-source library for FHE. However, their implementation resulted in an execution latency of 17.5 minutes, with now obsolete parameters (despite an amortized cost of 5.8 seconds per block), highlighting the impracticality of this approach for fast data transmission with AES. That is why many researchers have since developed new "FHE-friendly" symmetric cryptosystems to improve efficiency. Today, several proposals exist, including block ciphers such as LowMC [ARS+16], PRINCE [BCG+12], and CHAGHRI [AMT22], as well as stream ciphers like Elisabeth [CHMS22], PASTA [DGH+21], and Kreyvium [CCF+16]. These new schemes, referred to as hybrid encryption schemes, offer faster and more efficient homomorphic execution compared to the work of Gentry et al. [GHS12], though none have yet been standardized.

In 2022, the National Institute of Standards and Technology (NIST) announced a future call for threshold encryption with a specific focus on FHE, indicating that AES would serve as the benchmark for evaluating proposals. Since then, numerous teams have revisited AES transciphering to improve efficiency. In 2023, the work of Trama et al. [TCBS23] brought AES execution times to under 5 minutes in sequential mode and 30 seconds in parallel mode, leveraging TFHE programmable bootstrapping (PBS) in integer mode and using the Tree-Based Method (TBM) [GBA21] to perform bootstrapping on multiple encrypted inputs. Later in 2023, Bon et al. [BPR24] proposed the *p-encoding* method for binary ciphertexts in TFHE, achieving an AES evaluation in 211 seconds. Other teams then achieved further optimizations using TFHE in leveled homomorphic encryption (LHE) mode and circuit bootstrapping, such as Fregata [WWL+23] and Thunderbird [WLW+24], which reduced sequential execution times to 86 seconds and 46 seconds, respectively, on a single core. The timing results in the above works are summarized in Table 1 where we provide both the original timings given in the papers and the timings obtained on our single machine test bench.

Still, even if AES is often considered a reference benchmark, it is unlikely to be used for transciphering in practical FHE deployments as the stream-cipher based approach intrinsically leads much better performances [CCF+16, TB23]. However, as an example of a nontrivial algorithm that has eluded "practical" FHE execution for years, the algorithm is also interesting since it exemplifies the tension between boolean- and byte-oriented operations that is a recurrent issue when running algorithms over TFHE.

Our Contributions. This paper provides a first set of tools to resolve this kind of tension by consistently combining the (byte-oriented) LUT-based approach with a generalization of the (boolean-oriented) p-encodings one to get the best of both worlds. We then show that this strategy pays off, at least for AES, as we improve the state of the art for a TFHE execution of the algorithm between 30 and 45% and almost break the 1 second latency barrier with a mild amount of parallelism.

Specifically, all the aforementioned approaches rely on TFHE but offer different trade-offs. Binary ciphertext-based techniques, such as those in [BPR24], are faster in sequential

mode but require costly evaluations of the 8-bit Sbox. In contrast, the programmable bootstrapping-based approach of [TCBS23] simplifies Sbox evaluation but is less efficient for the remainder of the AES circuit. Building on these works, we propose a hybrid framework, which we refer to as *Hippogryph*[1], combining the strengths of these two approaches. The Sbox is evaluated using PBS in integer mode as in [TCBS23], while the rest of the AES circuit leverages the p-encoding method from [BPR24]. The integration of these two techniques requires non-trivial transitions between the methods, which constitute a key contribution of our work. This seamless combination sets a new record for AES homomorphic evaluation, achieving execution in approximately 30 seconds on a standard laptop using a single core.

We emphasize that all timings reported in this paper have been consistently obtained on the same machine, which is generally *not* the case in previous studies. To do so, we had to consistently gather and run the codes used in previous studies or reimplement their algorithms in cases where the code was not or only partially made public. As a bonus contribution we thus plan to openly release this software in the close future in order to provide the community with a consistent test bench for further works on AES execution over TFHE.

Table 1: State-of-the-art *single-core* homomorphic evaluation of AES. The table indicates both the original timings, in seconds, provided in the papers and, in brackets, the timings obtained on our single machine test bench (a 12th Gen Intel(R) Core(TM) i7-12700H CPU laptop).

Year	Reference	Method	Timings
2023	[TCBS23]	Tree-Based Method (TBM)	270 (270) s
	[BPR24]	p-encoding method	135 (90) s
	[WWL+23]	TFHE in "LHE" mode	86 (87) s
2024	[WLW+24]	TFHE in "LHE" mode	46 (60) s
2025	This work	Combined TBM/p-encodings	32 s

Organisation. In the following, Section 2 provides the necessary background on TFHE and a concise overview of the AES scheme and its subroutines. Section 3 focuses on the two building blocks of Hippogryph, as introduced in [TCBS23] and [BPR24]. Section 4 introduces our new design. Finally, Section 5 presents a detailed comparison with existing approaches, supported by relevant benchmarks.

2 Preliminaries

2.1 Notations

Let $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ be the real torus, that is to say the additive group of real numbers modulo 1. In practice, torus elements are not represented with an infinite number of digits, but are discretized. We can define the discretized torus $\mathbb{T}_q = \{\frac{a}{q} \mid a \in \mathbb{Z}_q\}$, and identify it with the ring $\mathbb{Z}_q$. Thus, any element $\frac{a}{q}$ of $\mathbb{T}_q$ will be represented in machine by a without losing any properties of the group $\mathbb{T}_q$. The operations of sum + and external product $\cdot$ have to be understood modulo q. Moreover, for a power of two N and a given q, we will denote by $\mathbb{T}_{N,q}[X]$ the polynomial ring $\mathbb{T}_q[X]/(X^N+1)$. The elements of this ring are polynomials of maximum degree $N-1$ and with coefficients in $\mathbb{T}_q$. Like for the scalar version, this ring will be identified with the ring $\mathbb{Z}_{N,q}[X] = \mathbb{Z}_q[x]/(X^N+1)$. Finally, we will denote by $\mathbb{B}$

[1] Following the seemingly emerging tradition of using (possibly mythical) bird names, like Fregata or Thunderbird, for frameworks running AES over TFHE.

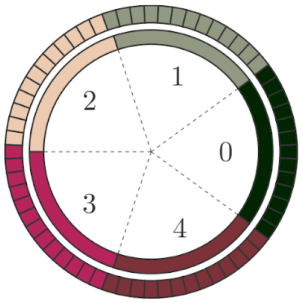

Figure 1: Embedding of $\mathbb{Z}_p$ in $\mathbb{Z}_q$. The inner circle represents $\mathbb{Z}_p$ with $p = 5$, and the outer circle is $\mathbb{Z}_q$, with $q = 64$.

the set of binary digits $\{0, 1\}$. For x and $q \in \mathbb{Z}$, $[x]_q$ denotes the reduction of x modulo q and $\lfloor x \rceil$ is the rounding of x. We denote by $x \xleftarrow{\$} \chi$ a random sampling according to a distribution χ.

2.2 Preliminaries on TFHE

TFHE [CGGI16, CGGI17, CGGI20] is a homomorphic encryption scheme, designed as a successor to FHEW [DM14]. Its security is based on the Learning With Errors (LWE) problem. Optimized for operations on low-precision data (typically less than 6 bits), TFHE offers a distinctive feature: *programmable bootstrapping*. This enables the evaluation of any univariate function on a ciphertext while simultaneously resetting its noise to a nominal level. In what follows, we introduce TFHE, describe the encoding and encryption procedures, and provide an overview of the homomorphic operations it supports.

2.2.1 Plaintext Space and Encryption

Before exploring the TFHE scheme in detail, it is important to define the *plaintext space* and its embedding into the discretized torus.

The plaintext space is the ring $\mathbb{Z}_p$, with $p \in \mathbb{N}^*$. We trivially identify $\mathbb{Z}_p$ with $\mathbb{T}_p$. Let us consider a mapping $\rho : \mathbb{Z}_p \to \mathbb{Z}_q$, defined as $\rho : m \mapsto \left\lfloor \frac{mq}{p} \right\rceil$. The image of this mapping only reaches p elements in $\mathbb{Z}_q$, which take the form $\left\{ \frac{kq}{p} \mid k \in \mathbb{Z}_p \right\}$. These elements are evenly distributed across $\mathbb{Z}_q$ and form what we refer to as *sectors of* $\mathbb{Z}_q$, represented as: $\left\{ \left(\frac{(2k-1)q}{2p}, \frac{(2k+1)q}{2p} \right) \mid k \in \mathbb{Z}_p \right\}$. Such a mapping is represented on Figure 1.

TFHE features two types of encryption that share similar structural patterns but operate within different mathematical spaces.

LWE Encryption. Let $m \in \mathbb{Z}_p$ be a message and let $sk = (s_1, \ldots, s_n)$ represent the secret key, sampled uniformly at random from $\mathbb{B}^n$. First, the message m is encoded in the space $\mathbb{Z}_q$ by $\tilde{m} = \rho(m)$. A small random Gaussian noise $e \xleftarrow{\$} \chi_\sigma$ of variance σ^2 is then added. Since e is small, the noisy message $\tilde{m} + e$ remains within the same sector as $\tilde{m}$. Next, we construct the LWE ciphertext as a vector $c = (a_1, \ldots, a_n, b)$, where the a_i's are sampled uniformly at random from $\mathbb{Z}_q$, and b is defined by $b = \sum_{i=1}^{n} a_i \cdot s_i + \tilde{m} + e$. We denote by $c \in \mathrm{LWE}_{sk}(m)$ an LWE encryption of the message m with secret key sk.

Decryption is performed in two steps: first, we compute $\phi(c) = b - \sum_{i=1}^{n} a_i \cdot s_i$, referred to as the *phase* of the ciphertext. Then we round it to the nearest plaintext value: $\tilde{m} = \left\lfloor \frac{p}{q} \phi(c) \right\rceil$. As long as $e < \frac{q}{2p}$, this rounding produces the right center of sector.

GLWE Encryption. This encryption mode mirrors the structure of LWE encryption but operates within polynomial rings. The secret key SK is here represented as a vector $(S_1, \ldots, S_k)$, sampled uniformly at random from $\mathbb{B}_{N,q}[X]^k$. The message is encoded in a polynomial in $\mathbb{Z}_{N,q}[X]$. The noise is also a polynomial from the same ring, with coefficients drawn from χ_σ. Similar to LWE encryption, the ciphertext takes the form $C = (A_1, \ldots, A_k, B)$ where $B = \sum_{i=1}^{k} A_i \cdot S_i + \tilde{M} + E$.

It is worth noting that LWE encryption can be viewed as a special case of GLWE encryption, where $N = 1$ and $k = n$.

2.2.2 Homomorphic Operations.

TFHE is trivially linearly homomorphic, so we define the following linear operations.

Sum of Ciphertexts. Let c_1 and c_2 be two ciphertexts encrypting m_1 and m_2 with noise variances σ_1^2 and σ_2^2, respectively. Performing a coordinate-wise sum of the two vectors results in a valid ciphertext c', which encrypts $m_1 + m_2$ with noise $\sigma_1^2 + \sigma_2^2$. We denote this operation by $\texttt{SumTFHE}(c_1, c_2)$.

Product with a Cleartext. Let c be a ciphertext encrypting m with noise σ^2. Multiplying each coordinate of c by a constant $\nu \in \mathbb{Z}$ produces a valid ciphertext c', which encrypts $m' = \nu \cdot m$ with noise $\nu^2 \cdot \sigma^2$. We denote this operation as $\texttt{ClearMultTFHE}(c', \nu)$.

These linear operations also have an equivalent with the polynomials when using GLWE encryption. They are extremely fast, particularly in comparison to bootstrapping. However, they increase the noise level, which means that only a limited number of such operations can be performed before the correctness of the results is compromised.

Key Switching (KS). TFHE also features a keyswitching algorithm, that allows the server to homomorphically transform a ciphertext c_1 encrypted under a key s_1 into a ciphertext c_2 encrypted under a key s_2. To do so, it requires a *keyswitching key* KSK, which is simply an encryption of s_1 under the key s_2. To know more about this algorithm, the reader is referred to [CGGI20]. Concretely, the size of a ciphertext can be temporarily reduced by keyswitching it to a shorter key (but raising its noise), to enable some speed-ups in the bootstrapping algorithm.

Programmable Bootstrapping (PBS). Bootstrapping was introduced by Gentry in [Gen09]. It allows to homomorphically reset the noise of a ciphertext to a nominal level. While this operation can theoretically be applied to any homomorphic encryption scheme, it is often deemed too slow for practical use, except for TFHE. Indeed, TFHE bootstrapping is efficient when compared to bootstrapping techniques of other fully homomorphic encryption schemes [BP23], especially for low-precision messages.

In addition, TFHE bootstrapping is implemented in a *programmable* manner: while the noise is being reset, any arbitrary function f can be evaluated on the ciphertext. Indeed, programmable bootstrapping (PBS) allows the *homomorphic evaluation of the Look-Up Table (LUT)* of the function f. We denote the evaluation of a function f on a ciphertext c with PBS as $\texttt{PBS\_TFHE}(c, f)$. We provide hereafter a high-level overview of how PBS works.

Let $(a_1, \ldots, a_n, b)$ be the LWE encryption of a message m with the secret key $(s_1, \ldots, s_n)$ and the noise variance σ^2. To reset the noise to a nominal level following Gentry's framework, the server must homomorphically evaluate $b - \sum_{i=1}^{n} a_i \cdot s_i$, and then round the result to the nearest integer in $\mathbb{Z}_p$. To do so, the server is equipped with a *bootstrapping key*, which consists of encryptions $\mathrm{Enc}(s_i)$ of each bit s_i of the secret key. Using this key, the computation of $\mu = b - \sum_{i=1}^{n} a_i \cdot \mathrm{Enc}(s_i)$ is straightforward, leveraging the linear

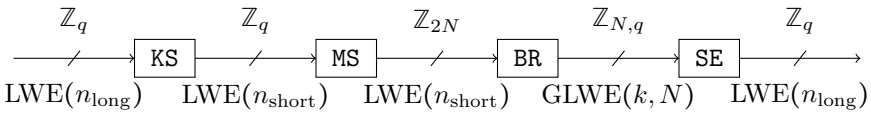

Figure 2: Types and shapes of ciphertexts inside a PBS.

homomorphisms inherent to TFHE. However, since the a_i's are sampled uniformly at random from $\mathbb{Z}_q$, they may have very large norms, leading to substantial noise growth. TFHE circumvents this issue with a technique known as *gadget decomposition*, which helps mitigate noise growth during multiplications with constants (refer to [CGGI20] for further details on gadget decomposition).

Once the linear part is computed, the server homomorphically performs the challenging rounding operation as follows:

1. Keyswitching (KS): The ciphertexts are keyswitched to a smaller key to accelerate the next steps.

2. ModSwitching (MS): The server switches the modulo of μ from q to $2N$, producing $\hat{\mu}$.

3. It constructs an *accumulator polynomial* $acc(X)$, whose coefficients encode the outputs of the function f evaluated alongside the PBS (i.e., the LUT of f). We will give more details on how to construct the accumulator polynomial in Section 2.2.3.

4. BlindRotate (BR): The server computes $X^{-\hat{\mu}} \cdot acc(X)$ which rotates the polynomial and specifically moves its coefficient $v_{\hat{\mu}} = m$ to the first position. The rotation works because the order of X in $\mathbb{Z}_{N,q}[X]$ is $2N$. If the LUT is properly encoded in the polynomial's coefficients, the first coefficient now contains an encryption of the LUT output.

5. SampleExtract (SE): The server then extracts this first coefficient and converts it into a new LWE ciphertext, which has significantly less noise than the original one. However, if $m > \frac{p}{2}$, the extracted coefficient will acquire an additional negative sign. This phenomenon is known as the *negacyclicity problem*. In Section 2.2.3, we discuss how we address this issue by using an odd value for the plaintext modulus p.

Figure 2 sums up the bootstrapping procedure of TFHE, and clarifies the types of ciphertext used at each step. Note that TFHE bootstrapping is by far the most computationally expensive operation in TFHE, and its cost increases significantly with the modulus p of the plaintext. Figure 3 gives examples of TFHE bootstrapping timings, on a standard laptop, in function of the input message's precision (number of bits).

2.2.3 Negacyclicity Problem and Parity of Plaintext Space

A common choice for TFHE is using a small power of 2 for the modulus p of the plaintext space, aligning with the format of (small-precision) binary numbers. However, selecting such an even modulus introduces an additional constraint: any function $f : \mathbb{Z}_p \to \mathbb{Z}_p$ used within a PBS must be *negacyclic* (i.e., it must satisfy $f(x + p/2) = -f(x)$ for all $x \in \mathbb{Z}_p$) due to the minus signs that appear during the BlindRotate step). To circumvent this issue, a possible approach consists in adding a bit of padding fixed to 0 in the most significant bit, effectively embedding the plaintext space $\mathbb{Z}_p$ into $\mathbb{Z}_{2p}$. However, this padding leads to an important overhead: linear operations are no longer virtually free, as frequent bootstrapping becomes necessary to maintain the padding bit cleared.

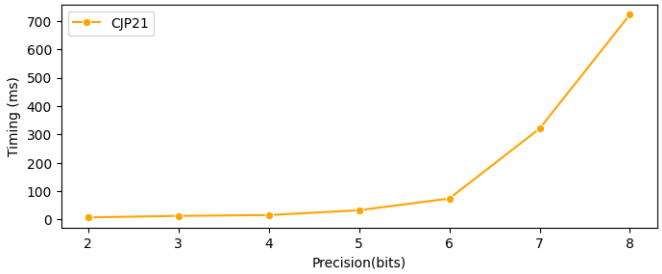

Figure 3: Timing of a PBS (obtained on a laptop) with respect to the precision of the ciphertext.

Another solution is to adopt an odd modulus p, which completely eliminates the negacyclicity problem. This approach, introduced in [BPR24], requires only a minor modification of the accumulator polynomial (acc) of the bootstrapping algorithm, but allows for arbitrary PBS function $f : \mathbb{Z}_p \to \mathbb{Z}_p$ without the need for a padding bit. We give in Eq. (1) a concrete formula for the accumulator acc in the PBS algorithm with an odd plaintext modulus.

Definition 1 (accumulator)**.** If p is an odd modulus, and $f : \mathbb{Z}_p \mapsto \mathbb{Z}_p$ is a function, then the accumulator $acc(X) \in \mathbb{Z}_{N,q}[X]/(X^N + 1)$ has the form:

$$acc(X) = X^{-\frac{N}{2p}} \cdot \sum_{j=0}^{N/p-1} X^j \cdot \left(\sum_{i=0}^{\frac{p-1}{2}} f(i) X^{i\frac{2N}{p}} + \sum_{i=0}^{\frac{p-1}{2}-1} -f\left(i + \frac{p+1}{2}\right) X^{i\frac{2N}{p} + \frac{N}{p}} \right) \tag{1}$$

In this work, we also make use of plaintext modulus $p = 2$. Even though 2 is even, we will use PBS without a bit of padding to evaluate a negacyclic function.

2.3 A Short Reminder on AES

The Advanced Encryption Standard (AES), based on the Rijndael algorithm winner of the NIST competition in 2000 [DR02], is a symmetric block cipher supporting key sizes of 128, 192, and 256 bits. Depending on the key size, AES uses 10, 12, or 14 rounds of processing, each applying a fixed sequence of substitution, permutation, and mixing steps to transform plaintext into ciphertext (or ciphertext into plaintext for decryption). A key schedule generates round keys for each encryption round, plus an initial key.

This work focuses on AES with 128-bit keys, which uses 10 rounds. The 16-byte input (plaintext for encryption or ciphertext for decryption) is treated as a structured *state* matrix, which is progressively updated during the encryption process. The encryption begins with an `AddRoundKey` step, followed by 10 rounds. Each round includes four steps: `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`, except the final round, which omits `MixColumns`. Below, we recall the key expansion and the subroutines:

- `Key Expansion`: The `Key Expansion` operation is performed once for a given secret key. Starting from the 128-bit key (in our context), it generates eleven 128-bit round keys, which are then used in the `AddRoundKey` operation throughout the AES encryption or decryption process, without needing access to the original key. The key expansion involves XORs and `GF`(256) multiplications.
- `SubBytes`: The `SubBytes` operation is the only non-linear transformation in the cipher. It involves a substitution step, where each byte in the state matrix is replaced

according to a fixed S-box. Since it operates independently on each byte of the state, `SubBytes` can be easily parallelized, allowing for more efficient execution.

- `AddRoundKey`: During this transformation, the state is updated by combining it with the current round key using a bitwise XOR operation. Specifically, the 128-bit round key is organized into a matrix format to align with the structure of the state matrix, and the two matrices are XORed element-wise to produce the new state.
- `ShiftRows`: The `ShiftRows` step is a byte transposition that cyclically shifts the rows of the state by different offsets. For AES with 128-bit keys, the first row remains unchanged, the second row is shifted by one byte, the third by two bytes, and the fourth row by three bytes.
- `MixColumns`: The `MixColumns` step processes the state column by column through matrix multiplication. To compute each byte of the state matrix, they combine scalar multiplication in $\texttt{GF}(256)$ with XOR operations. This approach facilitates parallelization of the operation.

3 Building Blocks of Hippogryph

In this section, we present the two approaches from [TCBS23] and [BPR24] that we use as building blocks for our new algorithm. [TCBS23] is simply recalled, while [BPR24] is generalized beyond just the Boolean case. We also formally present some advanced homomorphic primitives used in these works that we reuse as well.

3.1 The "Full-LUT" Approach

In the "Full-LUT" approach of [TCBS23], AES is evaluated entirely with TFHE's programmable bootstrapping, encoding exclusively all operations within LUTs. To meet the performance constraints outlined in Section 2.2, this method operates on elements in $\mathbb{Z}_{16}$, ensuring efficient computation.

3.1.1 AES Subroutines as LUTs

The `SubBytes` step, which involves the evaluation of an Sbox, is inherently a LUT operation and is therefore naturally implemented in FHE using a PBS. However, it must be evaluated over $\mathbb{Z}_{16}$ rather than $GF(256)$ for efficiency reasons. Converting the other AES steps into LUT evaluations also requires additional effort.

In particular, in the original AES design [DR02], the `MixColumns` step is computed using a series of XOR operations and multiplications in $GF(256)$. Unfortunately, TFHE's native cleartext-ciphertext multiplication cannot directly handle these $GF(256)$ multiplications because of the polynomial nature of the elements of this field. As a result, `MixColumns` must be reformulated as a LUT evaluation.

Additionally, the `AddRoundKey` step, which uses XOR as its key operation, presents its own challenges because XOR is a bivariate operation that requires two inputs. Classical bootstrapping, which operates on single inputs, is insufficient for this purpose. To address this, the authors utilize a specialized bootstrapping method that supports operations on multiple encrypted inputs.

3.1.2 LUTs Evaluation

Since the AES evaluation involves computing an 8-bit Sbox, a straightforward solution would be to work with 8-bit messages. With such messages, the homomorphic Sbox

evaluation would require only one bootstrapping per byte. However, as discussed in Section 2.2, processing messages with more bits demands larger TFHE parameters, which significantly slow down the bootstrapping process. For example, with 8-bit inputs, TFHE parameters result in bootstrapping times of approximately 1.5 seconds per byte on a standard laptop, making direct evaluation of the 8-bit Sbox (and other LUTs) infeasible.

To address this issue, the authors of [TCBS23] propose a decomposition approach and demonstrate that the optimal representation of 8-bit inputs for their purpose is in $\mathbb{Z}_{16}$. Specifically, a message $M \in \{0, \cdots, 255\}$ is split into two 4-bit chunks (or *nibbles*) h and l such that $M = 16h + l$. The encryption of M is then represented as a vector containing the encryptions of h and l with the same key sk: $C = (c_0, c_1) \in \mathrm{LWE}_{sk}(h) \times \mathrm{LWE}_{sk}(l)$.

However, bootstrapping these decomposed inputs requires a method capable of handling multiple encrypted inputs. The authors explore several approaches for this, namely the chain-based method and the tree-based method [GBA21]. Their analysis concludes that the Tree-Based Method (TBM) is the most suitable for their needs. They also relies on the Multi-Value Bootstrapping (MVB) to produce several outputs for the cost of one PBS. We provide details about TBM and MVB in the following:

Multi-Value Bootstrapping from [CIM18]. Multi-Value Bootstrapping (MVB) is a technique that enables the evaluation of k distinct Look-up tables $(f_i)_{1 \le i \le k}$ on a single encrypted input, using only one BlindRotate. This method is based on the factorization of the accumulator polynomials $acc_i(X)$ associated with each function f_i. Specifically, each accumulator polynomial is expressed as:

$$acc_i(X) = \sum_{j=0}^{N-1} \alpha_{i,j} X^j, \quad \alpha_{i,j} \in \mathbb{Z}_q.$$

The factorization then splits it into two parts:

$$acc_i(X) = v_0(X) \cdot v_i(X) \mod (X^N + 1),$$

where $v_0(X)$ is a common factor shared across all accumulators:

$$v_0(X) = \frac{1}{2} \cdot (1 + X + \cdots + X^{N-1}),$$

and $v_i(X)$ is a distinct factor specific to each function f_i:

$$v_i(X) = \alpha_{i,0} + \alpha_{i,N-1} + (\alpha_{i,1} - \alpha_{i,0}) \cdot X + \cdots + (\alpha_{i,N-1} - \alpha_{i,N-2}) \cdot X^{N-1}.$$

This factorization is made possible thanks to the identity:

$$(1 + X + \cdots + X^{N-1}) \cdot (1 - X) \equiv 2 \mod (X^N + 1).$$

By leveraging this factorization and as illustrated on Figure 4, multiple LUTs can be evaluated on a single encrypted input by performing the following steps:

1. Computing a BlindRotate operation on an accumulator polynomial initialized with the value of v_0.
2. Then multiplying with `ClearMultTFHE` the obtained rotated polynomial by each $v_i(X)$ corresponding to the LUT of f_i to obtain the respective $acc_i(X)$.

Finally, at the cost of a single BlindRotate and k cleartext-ciphertext GLWE multiplications, one can obtain the evaluation of k different LUTs on one single encrypted input. Moreover,

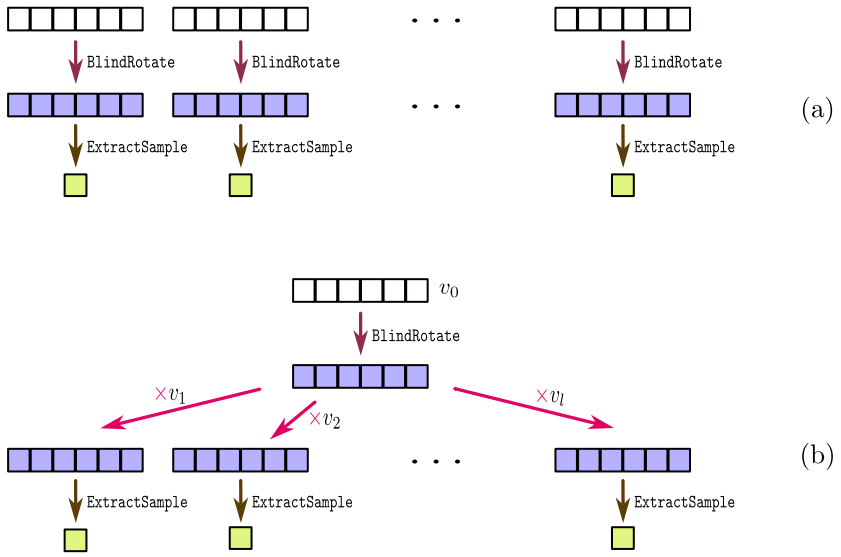

Figure 4: Difference between classic bootstrapping of several LUTs on a single input (a) and the use of MVB (b). Pink arrows represent cleartext-ciphertext RLWE multiplications. Figure extracted from [TCBS23].)

this specific choice of factorization allows for a very-low norm for the vectors v_i's (which in practice are very sparse), and so a very-low noise expansion.

This MVB primitive thus allows significant speed-ups in the implementation of [TCBS23], in particular in the evaluation of the Sbox or in the multiplications in $GF(256)$ that occur during the `MixColumns` step. Indeed, since each byte is decomposed into two nibbles h and l, the LUT corresponding, for instance, to the Sbox must also be decomposed into two tables: one providing the most significant nibble and one providing the least significant nibble. That is to say:

$$\mathtt{tab}_{\mathrm{MSN}}[i] = \left\lfloor \frac{\mathtt{Sbox}[i]}{16} \right\rfloor \quad \text{and} \quad \mathtt{tab}_{\mathrm{LSN}}[i] = \mathtt{Sbox}[i] \mod 16.$$

Each of these tables must be evaluated on an 8-bit payload ciphertext.

Tree-Based Method from [TCBS23]. Let $B, B', d \in \mathbb{N}^*$. The Tree-Based Method (TBM) allows to evaluate a LUT $f : \mathbb{Z}_{B^d} \mapsto \mathbb{Z}_{B'}$ with a large input size B^d, by processing d limbs of data in $\mathbb{Z}_B$. We consider input messages that are written as:

$$m = \sum_{i=0}^{d-1} m_i B^i, \quad \text{with } m_i \in \mathbb{Z}_B,$$

and that are represented by d ciphertexts $(c_0, c_1, \ldots, c_{d-1})$ corresponding to the d message components $(m_0, m_1, \ldots, m_{d-1})$. To evaluate f, we encode a LUT for f using B^{d-1} accumulators, each represented by a polynomial $acc_i(X)$. These accumulators encode the functions:

$$\begin{aligned} f_i : \mathbb{Z}_B &\to \mathbb{Z}_{B'} \\ x &\mapsto f(i + x \cdot B^{d-1}) \end{aligned}$$

Next, we apply a BlindRotate and a SampleExtract to each accumulator $acc_i(X)$, using c_{d-1} as the selector. This operation produces B^{d-1} LWE ciphertexts, each encrypting

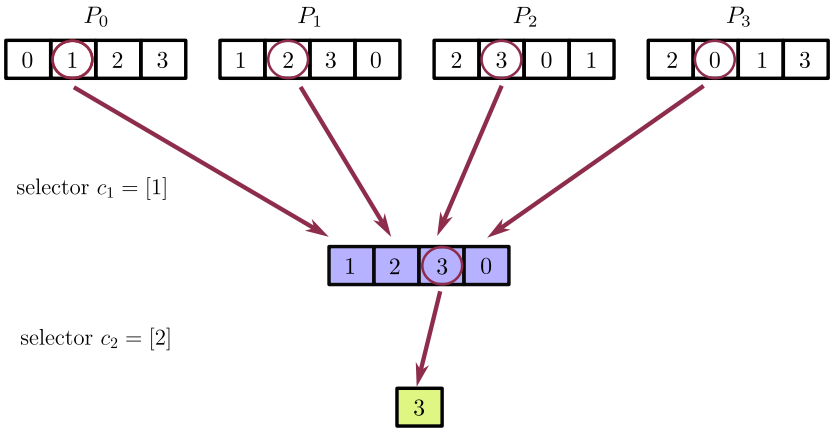

Figure 5: Illustration of the tree-based method on messages $m_1 = 1, m_2 = 2$ in the space $\mathbb{Z}_4$. The corresponding ciphertexts are $c_1 \in (m_1)$ and $c_2 \in \mathrm{LWE}(m_2)$. We apply the addition in $\mathbb{Z}_4$ via programmable bootstrapping. Red arrows indicate bootstrappings. (Figure inspired by [TCBS23].)

$f(i + m_{d-1} \cdot B^{d-1})$ for $i \in \mathbb{Z}_{B^{d-1}}$. Finally, a Keyswitch operation from LWE to GLWE aggregates these ciphertexts into B^{d-2} GLWE encryptions, representing the LUT of h, defined as:

$$\begin{aligned} h : (\mathbb{Z}_B)^{d-1} &\mapsto \mathbb{Z}'_B \\ (a_0, \ldots, a_{d-1}) &\mapsto f \circ g(a_0, \ldots, a_{d-2}, m_{d-1}) \end{aligned}$$

using the bijection g, which reverses the decomposition:

$$\begin{aligned} g : (\mathbb{Z}_B)^d &\to \mathbb{Z}_{B^d} \\ (a_0, \ldots, a_{d-1}) &\mapsto \sum_{i=0}^{d-1} a_i \cdot B^i \end{aligned}$$

This process is repeated iteratively, using the next ciphertext at each step, until a single LWE ciphertext encrypting $f(m_0, \ldots, m_{d-1})$ is obtained.

In the implementation described in [TCBS23], this primitive is employed to evaluate an 8-bit LUT by dividing it into two limbs of 4 bits each, which they determined to be optimal for their specific setting. To further enhance the performance of the TBM, the blind rotations for the accumulators $acc_i(X)$ of *the first layer of the tree* can be performed simultaneously using the MVB technique (as discussed in [GBA21]).

Finally, the "full-LUT" approach facilitates efficient computation of the Sbox through the Tree-Based Method, as opposed to directly evaluating the corresponding Boolean circuit. However, this approach also requires LUT-based computation of XOR operations and other intermediary steps, which is notably slower when operating in $\mathbb{Z}_{16}$ compared to binary messages. Consequently, our new method Hippogryph proposed in this paper strategically applies LUT evaluation exclusively where it is most effective and yields the best performance, namely for the evaluation of the Sbox.

3.2 Generalization of the "p-encodings" Approach

The work of [BPR24] takes an orthogonal approach compared to the previous one. In this method, data is encrypted bit per bit and only Boolean operations are performed. It

leverages the fact that, in the plaintext space $\mathbb{Z}_2$, the `SumTFHE` operation actually performs a XOR. Thus, the linear operations `MixColumns` and `AddRoundKey` can be efficiently performed with minimal cost, using only the homomorphic sum of TFHE. Specifically, they leverage on the circuit representation of `MixColumns` proposed in [Max19]. Furthermore, because operations are performed on individual bits, the `ShiftRows` transformation can be evaluated for free, as it merely involves rearranging the ciphertexts.

Evaluating `SubBytes` is trickier. Using $p = 2$, it is impossible to perform a bivariate Boolean gate other than XOR. Thus, evaluating the boolean circuit of the Sbox cannot be done. To deal with this problem, the authors introduced the notion of p-encoding, that embeds the bits into a larger space $\mathbb{Z}_p$ with $p > 2$.

In this work, we now generalize this notion beyond the Boolean case by defining the (o, p)- encoding construction. Informally, instead of embedding the Boolean space in $\mathbb{Z}_p$, we embed any space $\mathbb{Z}_o$ in $\mathbb{Z}_p$ (with $o < p$). So, what was called p-encoding in [BPR24] corresponds to a $(2, p)$-encoding in this work. Definition 2 formalizes this generalization:

Definition 2 ((o,p)-encoding)**.** Let $\mathbb{Z}_o$ be the message space. A (o, p)-*encoding* is a function $\mathcal{E} : \mathbb{Z}_o \mapsto 2^{\mathbb{Z}_p}$ that maps each element of $\mathbb{Z}_o$ to a subset of the discretized torus $\mathbb{Z}_p$. A (o, p)-encoding is *valid* if and only if:

$$\begin{cases} \forall (i,j) \in \mathbb{Z}_o^2, i \neq j, \mathcal{E}(i) \cap \mathcal{E}(j) = \varnothing \quad \text{and} \\ \text{if } p \text{ is even: } \forall\, x \in \mathbb{Z}_p, \forall i \in \mathbb{Z}_o : x \in \mathcal{E}(i) \iff \left\lfloor x + \frac{p}{2} \right\rceil_p \in \mathcal{E}([-i]_o) \end{cases} \tag{2}$$

The latter property is a direct consequence of the negacyclicity problem, which we presented in Section 2.2.3.

In this work, we focus exclusively on cases where $p = 2$ or p is an odd prime. As a result, a lot of the subleties of negacyclicity can be overlooked. Furthermore, among the various types of (o, p)-encodings, one particular class proves especially useful for our purposes: the *canonical* (o, p)-encoding.

Definition 3 (canonical (o,p)-encoding)**.** A (o, p)-encoding $\mathcal{E}$ is said *canonical* if and only if it verifies:

$$\begin{aligned} \mathcal{E} : \mathbb{Z}_o &\to \mathbb{Z}_p \\ x &\mapsto x \end{aligned}$$

(with $o < p$). Informally, we simply embed a smaller space into a larger one, without altering the order of the elements.

In [BPR24], the Boolean space is used (so $o = 2$). The `SubBytes` circuit is evaluated using $(2, 11)$-encoding, while the rest is evaluated with a $(2, 2)$-encoding (i.e. the trivial encoding of TFHE). Consequently, an *Encoding Switching* operation is required. This operation can be straightforwardly performed using a PBS.

Definition 4 (Encoding Switching)**.** Let c be a ciphertext encrypting a message $m \in \mathbb{Z}_o$ under the (o, p)-encoding $\mathcal{E}$. Its encoding can be switched to the (o, p')-encoding $\mathcal{E}'$ by applying a PBS on c evaluating the function:

$$\begin{aligned} \mathtt{Cast}_{\mathcal{E} \mapsto \mathcal{E}'} : \mathbb{Z}_p &\to \mathbb{Z}_{p'} \\ x &\mapsto x' \end{aligned}$$

where x' is defined as $\forall i \in \mathbb{Z}_o, x \in \mathcal{E}(i) \implies x' \in \mathcal{E}'(i)$

It remains to explain how the `SubBytes` circuit is evaluated in [BPR24]. The authors use a circuit representation for the Sbox (the one of [BP10]), and decompose it into so-called *gadgets*, which are smaller subcircuits evaluable in one single bootstrapping if the inputs are provided under the right p-encodings. This makes the evaluation of the circuit of the Sbox much faster than with the naive approach of "gate bootstrapping" where every logic gate is evaluated with a bootstrapping. As an order of magnitude, the authors of [WWL+23] have also implemented the AES with the "gate bootstrapping" approach and report a sequential evaluation timing of more than an hour on a standard laptop.

Manipulating the data bit-per-bit makes the evaluation of the linear part blazingly fast, however the circuit decomposition into gadgets is less efficient (even if better than gate bootstrapping).

4 Design of Hippogryph

Building on the foundation of the two previous works, we develop a hybrid approach, Hippogryph, that not only combines their respective strengths but also introduces new contributions to enable their effective integration. The guiding principles of this design are outlined below:

- The `SubBytes` step, which was the weak point of [BPR24], is evaluated using the strategy of [TCBS23].
- Conversely, the linear steps (namely `ShiftRows` , `MixColumns` and `AddRoundKey`) are computed using a trivial $(2, 2)$-encoding, which makes them extremely fast.
- Since the two aforementioned points rely on different data representations (arithmetic for `SubBytes` and Boolean for the other steps), a decomposition layer and a recomposition layer are necessary to transition from one to another. The decomposition and recomposition steps are denoted by `Decomposer` and `Recomposer`, respectively.

Our design for one round of AES is summed up on Figure 6. In the following we explain each of its components.

`SubBytes`. The `SubBytes` step is implemented following the design of [TCBS23]. Each 8-bit input is represented by two ciphertexts, each encrypting a 4-bit limb. Two instances of the TBM are then used to compute the limbs of the output. The only modification from the design of [TCBS23] is the adoption of the canonical $(16, 17)$-encoding, as specified in Definition 3:

$$\begin{aligned} \mathcal{E}_{17} : \mathbb{Z}_{16} &\to \mathbb{Z}_{17} \\ i &\mapsto i. \end{aligned}$$

This modification is introduced to ensure compatibility with the `Recomposer` operation, a point which will be explained in the dedicated paragraph. In Figure 6, ciphertexts encrypted under this $(16, 17)$-encoding are represented by blue rectangles. This process is repeated 16 times, once for each byte of the AES state. An additional improvement comes from the fact that the two TBM are using a MVB to evaluate the first step. So, the same common factor can be used for both evaluations, requiring only one `BlindRotate` per byte for this first step.

Linear Circuit. For this part, we follow the design of [BPR24]. The ciphertexts manipulated in this block are encoded under the trivial $(2, 2)$-encoding $\mathcal{E}_2$, and encrypt a single bit each. They are represented by yellow squares on Figure 6. Consequently, this

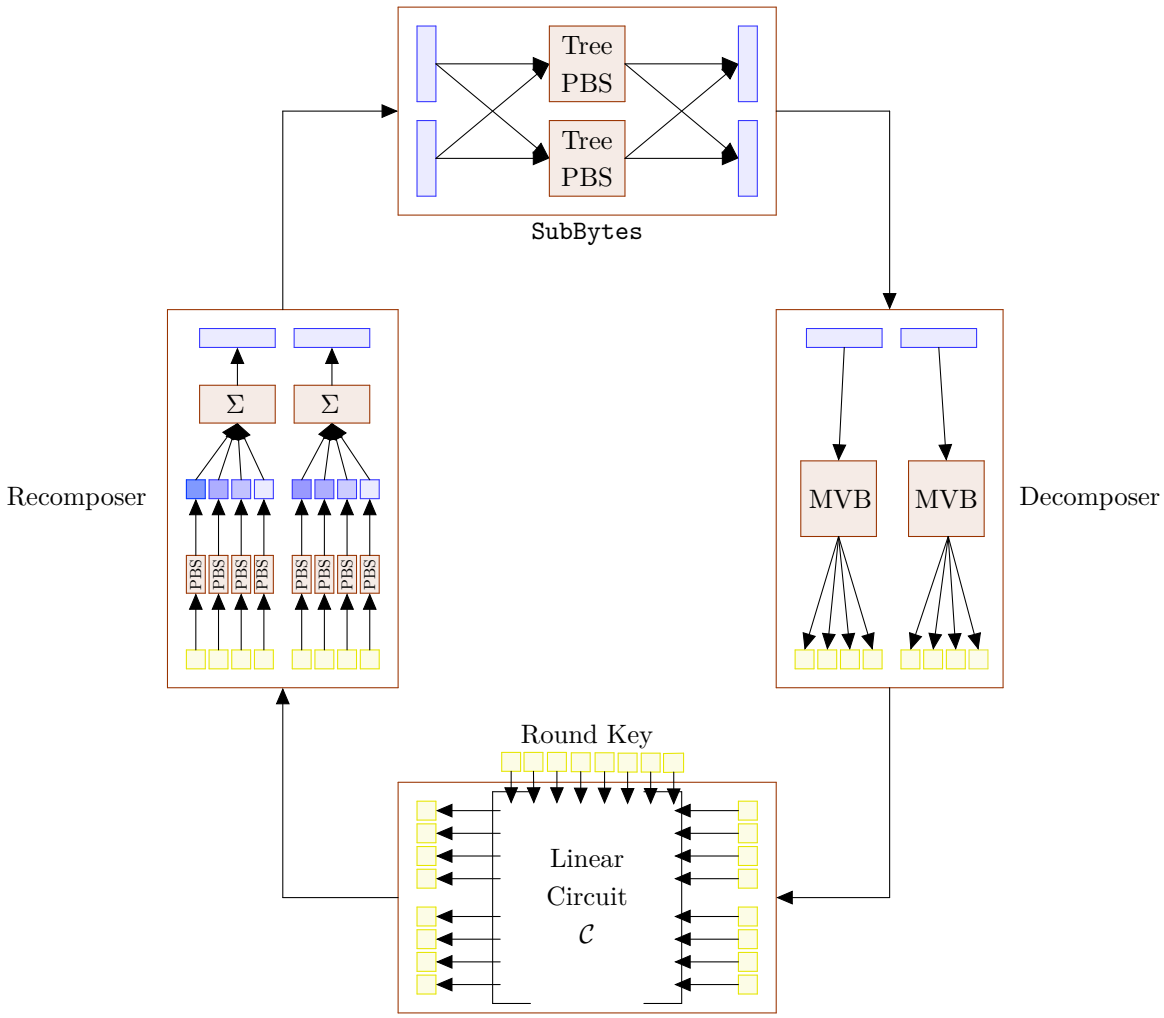

Figure 6: Structure of one round of AES with our method. Ciphertexts in blue live in $\mathbb{Z}_{17}$ while the ones in yellow are in $\mathbb{Z}_2$. Squares represent encryptions of one single bit while rectangles represent nibbles.

circuit takes 256 inputs (one for each of the 128 bits in an AES block, and one for each of the 128 bits in the current round key), and outputs a new state of 128 bits, by combining the three following steps:

- `ShiftRows` : This step is trivially implemented in FHE by permuting the input ciphertexts according to the AES spec.
- `MixColumns` : Here, we use the XOR-only circuit representation of [Max19]. Evaluating a XOR on ciphertexts under $\mathcal{E}_2$ is simply done using the native addition of TFHE `SumTFHE`.
- `AddRoundKey` : This step is a simple XOR between the state and the round key.

Evaluating the sums within this circuit increases the noise in the ciphertexts. However, this problem can actually be overlooked: using $p = 2$ there is plenty of room for the noise to grow, so the bottleneck of the construction in terms of noise is actually the TBM in $\mathbb{Z}_{17}$. In our experimentations, we made sure to select parameters ensuring correctness up to the target probability of success.

Decomposer. From the `SubBytes` step to the linear circuit steps, a switch of representation is needed at two levels. First, we need to decompose each ciphertext of a 4-bit limb into 4 ciphertexts each encrypting a single bit. Secondly, we need to switch the encoding from $\mathcal{E}_{17}$ to $\mathcal{E}_2$. Fortunately, by combining the MVB primitive and the encoding switching primitive (from Definition 4), it is possible to do both changes at once for each nibble with a single PBS. Formally, the MVB will evaluate the four functions:

$$\begin{aligned}\forall i \in \{0,\dots,3\}, f_i : \mathbb{Z}_{17} &\to \mathbb{Z}_2 \\ x &\mapsto \mathcal{E}_2((\mathcal{E}_{17}^{-1}(x))_i)\end{aligned}$$

where $(y)_i$ refers to the extraction of the i-th bit of y.

Recomposer. Conversely, a transformation from the Boolean domain to the arithmetic domain is required. As in the `Decomposer` operation, this involves two key steps:

- Casting the ciphertexts from a plaintext modulus of 2 to 17.
- Recombining each group of 4 bits into a single ciphertext encrypting the whole nibble.

To achieve this efficiently, we introduce four intermediary $(2, 17)$-encodings, namely:

$$\begin{aligned}\forall i \in \{0,\dots,3\}, \mathcal{E}_{17}^{(i)} : \mathbb{Z}_2 &\to \mathbb{Z}_{17} \\ x &\mapsto \begin{cases} 0 \text{ if } x = 0 \\ 2^{i+1} \text{ if } x = 1 \end{cases}.\end{aligned}$$

Using little-endian representation, we perform an encoding switching (Definition 4) on the i-th bit of each nibble, transitioning from $\mathcal{E}_2$ to $\mathcal{E}_{17}^{(i)}$. In Figure 6, the resulting ciphertexts are representing by squares filled with different shades of blue. Once the bits are expressed in this intermediary representation, we simply sum them to reconstruct the result in $\mathcal{E}_{17}$.

The inputs to the `Recomposer` are encrypted modulo 2. Since no padding bits are used, the negacyclicity problem necessitates that the PBS in the `Recomposer` evaluates a negacyclic function. As stated in Property 1, the existence of a Boolean recomposition algorithm relying solely on PBS and linear operations depends on the parity of the output plaintext modulus.

Property 1. A `Recomposer` using only linear operations and one PBS per bit exists only if the output modulo is **odd**.

Proof. Let p be an integer. Let $(b_0, \dots, b_{d-1})$ be the bits to encrypt, and let $(c_0, \dots, c_{d-1})$ denote their corresponding ciphertexts, encoded with the trivial $(2,2)$-encoding $\mathcal{E}_2$. We aim to construct a `Recomposer` that uses only one programmable bootstrapping (PBS) per bit and linear operations to homomorphically compute an encryption of the message $m = \sum_{i=0}^{d-1} b_i 2^i$ under the canonical $(2^d, p)$-encoding $\mathcal{E}_p$. The purpose of this proof is to demonstrate how the parity of p influences the existence of such an algorithm.

To do so, following the blueprint introduced earlier in the section, we want to bootstrap the ciphertext c_i into $\mathbb{Z}_p$ with the p-encoding $\mathcal{E}_p^{(i)} = \begin{cases} \mathbb{Z}_2 \mapsto \mathbb{Z}_p \\ 0 \mapsto \{0\} \\ 1 \mapsto \{2^{i+1}\} \end{cases}$. Once we have those, a simple sum will reconstruct the message under the canonical $(2^d, p)$-encoding. Let us analyze if this bootstrapping is possible.

As the ciphertexts are encrypted modulo 2, there is no bit of padding. So, if we send them modulo p with a PBS, the result will necessarily be encoded under a negacyclic $(2,p)$-encoding, that is to say of the form: $\mathcal{E}^{(\text{neg})} = \begin{cases} \mathbb{Z}_2 \mapsto \mathbb{Z}_p \\ 0 \mapsto \{\gamma\} \\ 1 \mapsto \{[-\gamma]_p\} \end{cases}$ with $\gamma \in \mathbb{Z}_p$.

Now, we need a linear transformation that casts a ciphertext from $\mathcal{E}^{(\text{neg})}$ to $\mathcal{E}_p^{(i)}$. Let us denote this hypothetical linear transformation by $\mathcal{L}$, and define it as:

$$\begin{aligned} \mathcal{L} : \mathbb{Z}_p &\mapsto \mathbb{Z}_p \\ x &\mapsto a \cdot x + b \end{aligned}$$

By simply considering the encoding switching from $\mathcal{E}^{(\text{neg})}$ to $\mathcal{E}_p^{(0)}$, it is clear that the constants a and b need to verify the property:

$$\begin{cases} a \cdot \gamma + b = 0 \mod p \\ a \cdot (-\gamma) + b = 1 \mod p \end{cases}$$

which can be rewritten as:

$$\begin{cases} b = 2^{-1} \mod p \\ \gamma = (b-1) \cdot a^{-1} \mod p \end{cases}$$

It is clear that such a b only exists if and only if 2 has an inverse modulo p. This latter argument forces p to be odd. In that case, fixing a to 1, the $(2,p)$-encoding

$$\mathcal{E}^{(neg)} = \begin{cases} \mathbb{Z}_2 \mapsto \mathbb{Z}_p \\ 0 \mapsto \{[2^{-1} - 1]_p\} \\ 1 \mapsto \{[1 - 2^{-1}]_p\} \end{cases}$$

is supposed to be what we are looking for.

Let us check if that is the case. As it is negacyclic, the PBS is evaluable. Then, the linear transformation $x \mapsto x + 2^{-1} \mod p$ produces a ciphertext under the right p-encoding. Trivially, adding a constant to a TFHE ciphertext do not increase its noise. The same reasoning can be followed for the others bits.

Finally, summing the produced ciphertexts gives an encryption of m under $\mathcal{E}_p$. The whole procedure is only possible if p is odd. □

Thus, an odd modulo is required and the best choice to fit 4-bit nibbles is $p = 17$.

Key Expansion. Note that, as done, to the best of our knowledge, in all previous works on AES transciphering, we do not perform the key expansion in the homomorphic domain. Instead, we work under the assumption that FHE encryptions of the eleven AES round keys are directly available. Since the round keys need to be computed only once for a given secret key, this makes sense in a client-server setting as the client is then expected to compute the key expansion and to send encryptions of the resulting round keys (rather than sending an encryption of the secret key under the homomorphic scheme).

5 Experimental Results

In this section, we compare our new framework to several state-of-the-art homomorphic AES executions, including the ones performed with the two building blocks of our new design. We particularly emphasize that *all implementations have been tested on the same machine*, a 12th Gen Intel(R) Core(TM) i7-12700H CPU laptop with 64 Gib total system memory with an Ubuntu 22.04.2 LTS operating system. All execution timings can be found in Table 3. Parameter sets used to obtain these results are presented in Table 2. Depending on the framework, we had to use different implementations of TFHE as available in the `TFHElib`[2], `tfhe-rs`[3] or `TFHEpp`[4] libraries.

Table 2: Parameters sets used for our homomorphic AES evaluation, with $\lambda \approx 128$ bits as the security parameter. P_{err} denotes the probability of bootstrapping failure. B_{PBS} and l denote the basis and levels associated with the gadget decomposition in KeySwitch, B_{KS} and t denote the decomposition basis and the precision of the decomposition of BlindRotate. σ_{LWE} and σ_{GLWE} are the standard deviations of noises used in LWE and GLWE ciphertexts, respectively.

P_{err}	n	N	k	l	B_{PBS}	B_{KS}	t	σ_{LWE}	σ_{GLWE}
2^{-40}	754	1024	1	2	2^{23}	2^{4}	3	$2^{46.4}$	$2^{16.7}$
2^{-128}	900	4096	1	2	2^{15}	2^{3}	6	$2^{44.5}$	2^{2}

Table 3: Comparison of our method with different state-of-the art approaches *on a single core*. The only execution timing that was not obtained on our machine is marked with a *, i.e. for Thunderbird, making the comparison more in favor of that method. See the discussion at the end of Section 5.1.2.

Year	**Reference**	**Framework**	**Library**	**Timings (s)**
2023	[TCBS23]	Tree-Based Method (TBM)	`TFHElib`	270
	[BPR24]	p-encoding method	`tfhe-rs`	90
	[WWL+23]	Fregata	`TFHEpp`	87
2024	[WLW+24]	Thunderbird	`TFHEpp`	46*
2025	this work	Hippogryph	`tfhe-rs`	32

5.1 State-Of-The-Art Homomorphic AES Executions

The approaches introduced in [TCBS23] and [BPR24], which form the foundation of our proposal, are discussed in Section 3. Additionally, we briefly describe the two other main state-of-the-art methods for homomorphic AES executions: Fregata [WWL+23] and Thunderbird [WLW+24].

[2] https://tfhe.github.io/tfhe/
[3] https://github.com/zama-ai/tfhe-rs
[4] https://github.com/virtualsecureplatform/TFHEpp

5.1.1 Fregata [WWL+23]

In this work, the authors present a novel evaluation framework especially designed for faster AES homomorphic evaluation. Instead of relying on functional bootstrapping, they decided to use CMUX gate as the building block of their framework. They also propose a new technique for an efficient S-box evaluation relying on mixed packing (which combines different ways of organizing encrypted data within polynomials to balance parallelism and flexibility). But one of the major contributions of this work is the optimization of TFHE's circuit bootstrapping. Indeed, they propose to use PBSManyLUT [CLOT21] in the first step of circuit bootstrapping. As their framework relies on the use of TFHE in LHE mode, this optimization of circuit bootstrapping is the key to an efficient homomorphic AES evaluation. Fregata being designed to perform one round of AES without any bootstrapping and to use circuit bootstrapping on each bit of the state matrix after a full round evaluation, running these circuit bootstrappings then becomes the most time consuming part. Finally, they also leverage on encoding messages in $\{0,1\}$ as $\{0,\frac{1}{2}\}$ over the torus to transform XOR operations into simple LWE sums (which is the same thing as using our $(2,2)$-encoding in the linear parts).

Their results, obtained with the TFHEpp library [Mat20], reached an AES homomorphic evaluation latency of 86 seconds on a 12th Gen Intel(R) Core(TM) i5-12500× 12 with 15.3 GB RAM machine. When running the Fregata implementation[5] on our machine, we also obtained a latency of about 87 seconds.

5.1.2 Thunderbird [WLW+24]

The work presented in the Thunderbird paper leverages on the Fregata framework to produce an even faster AES homomorphic evaluation, still using TFHEpp. Specifically, Thunderbird combines the gate bootstrapping and leveled evaluation modes of TFHE to cater to various function types within symmetric encryption algorithms. More specifically, their approach builds upon the Fregata framework with additional optimizations:

- The circuit bootstrapping proposed in Fregata is optimized by replacing the second step (namely a private keyswitch) by a public keyswitch followed by a new operation called `EvalSquareMult`.
- Instead of following a standard AES implementation, the authors introduce a LUT-based AES implementation that merges SubBytes, ShiftRows and MixColumns operations into 8-to-32-bit tables (which results in a smaller number of XOR operations when running the overall AES).

Moreover, as in [WWL+23], they rely on encoding the messages in $\{0,1\}$ as $\{0,\frac{1}{2}\}$ over the Torus. With such encoding, XOR operation can be performed for free. They call this optimization `FreeXOR`. They also propose another technique to evaluate XOR, namely `HomoXOR` relying on gate bootstrapping with messages encoded in $\{\frac{-1}{8},\frac{1}{8}\}$ over the Torus. The evaluation of AES with this technique is less efficient than with `FreeXOR`. For this work, the tests were run on an Intel(R) Core(TM) i5-11500 CPU @ 2.70GHz machine with 32 GB of RAM and they obtained an average execution latency of 46 seconds.

It is important to note that the implementation of the Thunderbird framework is not publicly available. To obtain a fair comparison with our work, we tried to reproduce their results by implementing the framework ourselves, starting from Fregata on which Thunderbird is based. Although our implementation of Thunderbird (using the most efficient `FreeXOR` variant) still induces unexpected decryption errors, it executes the AES in 60 secs, compared to the 46 seconds reported by the authors. This hints that our

[5] https://github.com/WeiBenqiang/Fregata

machine is approximately 1.3 times slower than the one used in their paper (a ratio which is further confirmed by lower-grain unitary measurements on the circuit bootstrapping alone). *As a result, comparing the execution times of our new framework to those reported in the Thunderbird paper (i.e. 46 secs) may slightly disadvantage Hippogryph.*

5.2 Results

To measure the performances of our method, we implemented it using a fork of `tfhe-rs` [Zam22] that supports odd moduli. The results were then compared against the current state-of-the-art frameworks.

For a fair comparison, all implementations were tested on the same machine, *using a single core*. As shown in Table 3, our novel framework achieves the lowest latency when evaluating the AES as the evaluation of the algorithm only takes about 30 seconds. Hence, Hippogryph is between 1.44 and 1.87 times faster than the best-in-class Thunderbird approach (depending, as discussed above, whether we respectively consider the 46 secs timing given in the Thunderbird paper or a timing of 60 secs as measured with our implementation). Moreover, when enabling several cores on our 12th Gen Intel(R) Core(TM) i7-12700H CPU laptop, we can reach an execution time that is smaller than 5 seconds, using only 6 cores, and further reduce this timing to 1.6 seconds by using 16 cores on a more powerful machine as discussed below.

A Few Words About Parallelisation. The purpose of transciphering is to minimize the bandwidth overhead when transferring large amount of data. Given that servers typically have more computational resources than clients, they can effectively leverage multiple cores to parallelize computations and enhance execution times. In this context, AES offers inherent parallelizability, as operations within each encryption round can be executed concurrently on each byte of the state matrix, with the exception of the ShiftRows step.

To implement this parallelization, we used Rust's `rayon` crate. Our tests were conducted on two distinct machines to assess performance across different setups. First, we used the same 12th Gen Intel(R) Core(TM) i7-12700H CPU laptop that was previously used for testing with a single core. This time, we ran the code on the laptop using its six available cores. Specifically, we parallelized every round function except for the ShiftRows function, which mainly involves reordering ciphertexts within the state matrix. Second, we ran the code on a server with an AMD Ryzen Threadripper PRO 7995WX, equipped with 96 cores, allowing for extended parallelization. *This setup brought us remarkably close to breaking the 1-second barrier, with an execution time of just 1.6 seconds.* Detailed execution timings illustrating these improvements can be found in Table 4.

Furthermore, finer-grained parallelism could help reduce this timing even further. For instance, we could exploit the independence between TBM computations for each byte output of the S-box layer.

What About Recent CPA$^{\mathbf{D}}$ Attacks? To obtain a fair comparison, we use parameters equivalent to those used in the state-of-the-art, that typically achieve an error probability of about 2^{-40}. But to take into account recent attacks in the CPA^D model [LM21] on several FHEs (including TFHE) [CSBB24, CCP+24], we also give execution times of our approach with a example parameters set achieving an error probability of 2^{-128} (Table 2). When running with such parameters, an AES evaluation takes about 463 seconds on our machine, still using a single core (see also Table 4). Although more optimal parameters may be found, this timing also illustrates that achieving CPA^D security may have a significant cost on FHE performances. At this point, we leave that cost mitigation as a future work.

Table 4: Different evaluation timings of Hippogryph for different setups.

Machine	# cores	P_{err}	Timings (s)
laptop	1	2^{-40}	32
laptop	1	2^{-128}	463
laptop	6	2^{-40}	4.6
server	16	2^{-40}	1.6

6 Conclusion

Even if this paper focuses primarily on AES, it should be seen as a first step towards solving the boolean-vs-byte tension which often occurs when attempting to run algorithms over TFHE. Beyond the quest for "the fastest AES-over-TFHE in the west", this paper's approach will clearly benefit to other block-ciphers such as PRINCE [BCG+12], SKINNY [BJK+16] or PRESENT [BKL+07], which also alternate boolean- and byte-friendly operations. For instance the 4-bit SBox of PRINCE is byte-friendly (and even more efficient with the full-LUT approach than the 8-bit Sbox of the AES). But the PRINCE matrix multiplication is a real efficiency bottleneck for this approach, as it only consists of XOR and AND operations on bits of the state matrix.

Furthermore, although AES may seem an arbitrary benchmark, it can however be expected that works on this algorithm prefigure more widely applicable advances. For instance, the work in [TCBS23] later leads to the full-blown instruction set in [TCB+25] as a systematization of the LUT-based approach. An interesting perspective would then be to revisit that latter instruction set by taking advantage of the toolbox proposed in the present paper.

Lastly, we plan to open source the unified software test bench for AES execution over TFHE we created for obtaining the consistent same-machine experimental results given in this paper in the close future. We hope that it will be valuable resource for enabling fair comparisons in further works on AES in the community.

Acknowledgements

This work was supported by the France 2030 ANR Projects ANR-22-PECY-003 SecureCompute and ANR-23-PECL-0009 TRUSTINCloudS. The authors would like to thank Pierre-Emmanuel Clet for providing the parameters with $P_{\text{err}} = 2^{-128}$ in Table 2. They also would like to thank Matthieu Rivain for his valuable insights and thoughtful discussions.

References

[AMT22] T. Ashur, M. Mahzoun, and D. Toprakhisar. Chaghri - a fhe-friendly block cipher. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 139–150, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3559364.

[ARS+16] M. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. Ciphers for mpc and fhe. Cryptology ePrint Archive, Paper 2016/687, 2016. https://eprint.iacr.org/2016/687. URL: https://eprint.iacr.org/2016/687.

[BCG+12] J. Borghoff, A. Canteaut, T. Güneysu, E. Bilge Kavun, M. Knezevic, L. Ramkilde Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçin. Prince - a low-latency block cipher for pervasive

computing applications - extended abstract. In *International Conference on the Theory and Application of Cryptology and Information Security*, 2012.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. ACM. doi:10.1145/2090236.2090262.

[BJK+16] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, Thomas Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer, 2016. URL: https://doi.org/10.1007/978-3-662-53008-5_5.

[BKL+07] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 450–466, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[BP10] J. Boyar and R. Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2010. doi:10.1007/978-3-642-13193-6_16.

[BP23] A. Al Badawi and Y. Polyakov. Demystifying bootstrapping in fully homomorphic encryption. Cryptology ePrint Archive, Paper 2023/149, 2023. URL: https://eprint.iacr.org/2023/149.

[BPR24] N. Bon, D. Pointcheval, and M. Rivain. Optimized homomorphic evaluation of boolean functions. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):302–341, 2024. URL: https://doi.org/10.46586/tches.v2024.i3.302-341, doi:10.46586/TCHES.V2024.I3.302-341.

[CCF+16] A. Canteaut, S. Carpov, C. Fontaine, T. Lepoint, M. Naya-Plasencia, P. Paillier, and R. Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In Thomas Peyrin, editor, *Fast Software Encryption*. Springer Berlin Heidelberg, 2016.

[CCP+24] J. H. Cheon, H. Choe, A. Passelègue, D. Stehlé, and E. Suvanto. Attacks against the IND-CPAD security of exact FHE schemes. Technical Report 127, IACR ePrint, 2024.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 3–33, Hanoi, Vietnam, December 4–8, 2016. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-53887-6_1.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 377–408, Hong Kong, China, December 3–7, 2017. Springer, Cham, Switzerland. doi:10.1007/978-3-319-70694-8_14.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020. doi:10.1007/s00145-019-09319-x.

[CHMS22] O. Cosseron, C. Hoffmann, P. Méaux, and F.-X. Standaert. Towards globally optimized hybrid homomorphic encryption - featuring the elisabeth stream cipher. Cryptology ePrint Archive, Paper 2022/180, 2022. https://eprint.iacr.org/2022/180.

[CIM18] S. Carpov, M. Izabachène, and V. Mollimard. New techniques for multi-value input homomorphic evaluation and applications. Cryptology ePrint Archive, Paper 2018/622, 2018. https://eprint.iacr.org/2018/622.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Cham, Switzerland. doi:10.1007/978-3-319-70694-8_15.

[CLOT21] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap. Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe. In *Advances in Cryptology – ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III*, 2021. URL: https://doi.org/10.1007/978-3-030-92078-4_23.

[CSBB24] M. Checri, R. Sirdey, A. Boudguiga, and J.-P. Bultel. On the practical cpad security of "exact" and threshold FHE schemes. In *CRYPTO*, 2024.

[DGH+21] C. Dobraunig, L. Grassi, L. Helminger, C. Rechberger, M. Schofnegger, and R. Walch. Pasta: A case for hybrid homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2021:731, 2021.

[DM14] L. Ducas and D. Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. Cryptology ePrint Archive, Paper 2014/816, 2014. URL: https://eprint.iacr.org/2014/816.

[DR02] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard (Information Security and Cryptography)*. Springer, 1 edition, 2002.

[GBA21] A. Guimarães, E. Borin, and D. F. Aranha. Revisiting the functional bootstrap in tfhe. 2021, 2021. doi:10.46586/tches.v2021.i2.229-253.

[Gen09] C. Gentry. Fully homomorphic encryption using ideal lattices. STOC '09, 2009. doi:10.1145/1536414.1536440.

[GHS12] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. In R. Safavi-Naini and R. Canetti, editors, *Advances in Cryptology – CRYPTO 2012*. Springer Berlin Heidelberg, 2012.

[HS20] Shai Halevi and Victor Shoup. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481, 2020. URL: https://eprint.iacr.org/2020/1481.

[LM21] B. Li and D. Micciancio. On the security of homomorphic encryption on approximate numbers. In *EUROCRYPT*, pages 648–677, 2021.

[Mat20] K. Matsuoka. TFHEpp: pure C++ implementation of TFHE cryptosystem. https://github.com/virtualsecureplatform/TFHEpp, 2020.

[Max19] A. Maximov. AES mixcolumn with 92 XOR gates. *IACR Cryptol. ePrint Arch.*, page 833, 2019. URL: https://eprint.iacr.org/2019/833.

[TB23] N. Smart T. Balenbois, J.-B. Orfila. Trivial transciphering with trivium and tfhe. In *WAHC*, pages 69–78, 2023.

[TCB+25] D. Trama, P.-E. Clet, A. Boudguiga, R. Sirdey, and N. Ye. Designing a general-purpose 8-bit (T)FHE processor abstraction. To appear in TCHES 2025. URL: https://eprint.iacr.org/2024/1201.

[TCBS23] D. Trama, P.-E. Clet, A. Boudguiga, and R. Sirdey. A homomorphic AES evaluation in less than 30 seconds by means of TFHE. In Michael Brenner, Anamaria Costache, and Kurt Rohloff, editors, *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Copenhagen, Denmark, 26 November 2023*, pages 79–90. ACM, 2023. doi:10.1145/3605759.3625260.

[WLW+24] B. Wei, X. Lu, R. Wang, K. Liu, Z. Li, and K. Wang. Thunderbird: Efficient homomorphic evaluation of symmetric ciphers in 3gpp by combining two modes of TFHE. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):530–573, 2024. URL: https://doi.org/10.46586/tches.v2024.i3.530-573, doi:10.46586/TCHES.V2024.I3.530-573.

[WWL+23] B. Wei, R. Wang, Z. Li, Q. Liu, and X. Lu. Fregata: Faster homomorphic evaluation of aes via tfhe. In *Information Security: 26th International Conference, ISC 2023, Groningen, The Netherlands, November 15–17, 2023, Proceedings*, page 392–412, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-3-031-49187-0_20.

[Zam22] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.