Submission to Zama's AES Implementation Bounty: Documentation

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

February 7, 2025

1 Introduction

This is the documentation of our submission to Zama's AES Implementation Bounty. It complements the `README` file of the submission.
Our submission is an optimized version of our recent Hippogryph framework [BBB+25]. This documentation purpose is then to briefly expose the design of our submission (Section 2) and give a hint on the structure of our code (Section 3). For more details and background on the primitives we use, please refer to our article at `https://eprint.iacr.org/2025/075`. The main differences between this submission and the original works are the homomorphic implementation of the Key Schedule, as well as more parallization to take advantage of the benchmarking hardware proposed by Zama.

2 Our Framework

Our submission is an optimized version of the Hippogryph framework. For an exhaustive presentation, please refer to [BBB+25]. We here briefly summarize the methods and tools on which we rely to implement the AES over `tfhe-rs`.

2.1 Tools

The idea of our framework is to rely on TFHE's Programmable Bootstrapping to evaluate the SubBytes step of the AES (as done in [TCBS23]) and to use the p-encodings method [BPR24] to perform all other steps (AddRoundKey, MixColumns and ShiftRows).

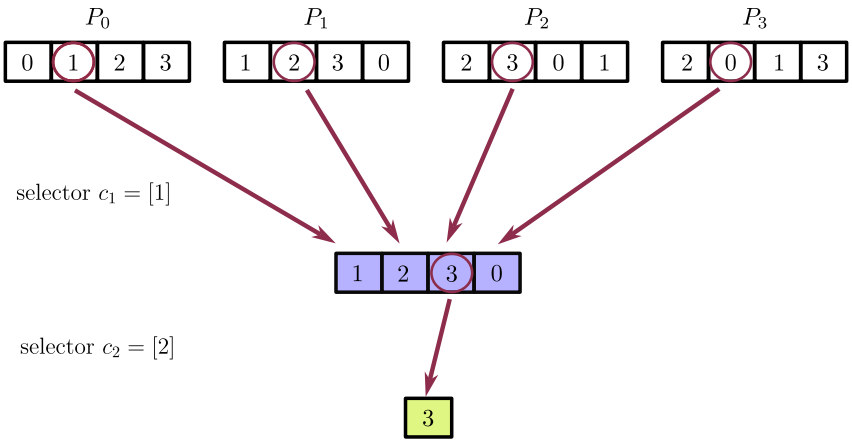

Figure 1: Illustration of the tree-based method on messages $m_1 = 1, m_2 = 2$ in the space $\mathbb{Z}_4$. The corresponding ciphertexts are $c_1 \in (m_1)$ and $c_2 \in \text{LWE}(m_2)$. We apply the addition in $\mathbb{Z}_4$ via programmable bootstrapping. Red arrows indicate bootstrappings.

SubBytes The SubBytes step (which is an evaluation of an 8-bit Sbox) is performed with pairs of 4-bit payload ciphertexts. That is to say, for each byte M of the state matrix, there exist $h, l \in \mathbb{Z}_{16}$ such that $M = 16h + l$. The TFHE ciphertext corresponding to the byte M is $C = (c_0, c_1)$ with $c_0 \in \text{TLWE}(h)$ and $c_1 \in \text{TLWE}(l)$. This decomposition is indeed optimal when working with byte inputs (see [BBB+25] for more details). Then, the idea is to perform the programmable boostrapping on such inputs by taking the AES Sbox as the LookUp Table (LUT) to be evaluated. To do so, we rely on the Tree-Based Method (TBM) introduced in [GBA21] that allows to perform bootstrapping on several encrypted inputs. With this method, evaluating the AES Sbox on input C costs 16 bootstrappings with selector c_1 and only one bootstrapping with selector c_0. This is illustrated in Figure 1.

Moreover, we use a bootstrapping optimization called Multi-Value Bootstrapping (MVB) and introduced in [CIM19]. This optimization allows to perform $k \geq 1$ bootstrappings *on the same input* at the cost of only one BlindRotate (and k SampleExtract). We use this optimization in the first layer of the TBM to factorize the 16 first BlindRotate. This is pictured in Figure 2. Then, the homomorphic evaluation of the AES Sbox on one encrypted bytes only requires 3 BlindRotate (as we need to perform two TBM, one for each nibble encryption and because we can use the MVB to factorize the two first levels of each TBM).

A last optimization is to use the plaintext modulus $p = 17$, and to embed the nibbles from $\mathbb{Z}_{16}$ to $\mathbb{Z}_{17}$. We explain further in the section why we made this choice.

AddRoundKey, MixColumns and ShiftRows The other steps of the AES round are performed using a Boolean representation, so with a plaintext modulus $p = 2$. With such an encoding, XOR and shifts operations are free. As AddRoundKey is only composed of XOR, that ShiftRows is only composed of shifts and that there exist a MixColumns circuit that only relies on XOR and shifts, this part of the AES is extremely fast.
So the only costly operations that remain are the decomposition of 4-bit encryptions into four binary ciphertexts and the recomposition: from four binary ciphertexts, we want to obtain one 4-bit encryption.

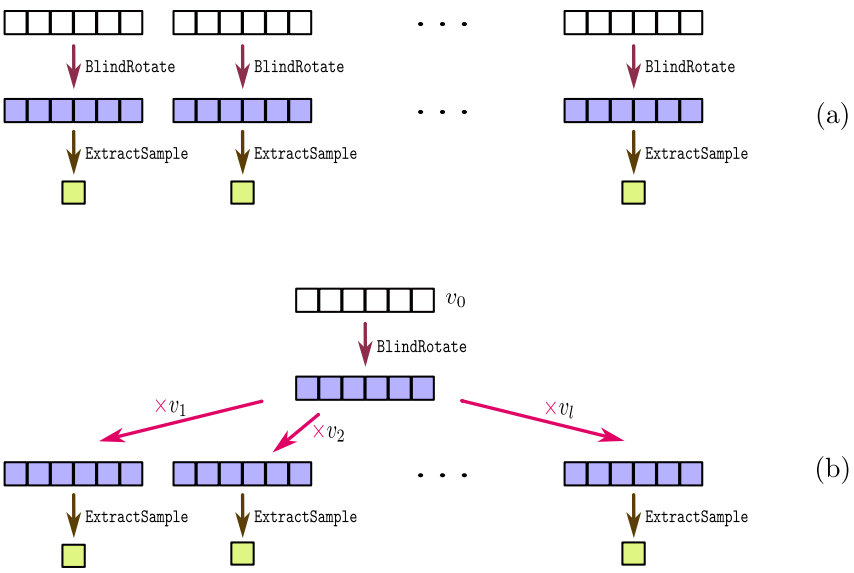

Figure 2: Difference between classic bootstrapping of several LUTs on a single input (a) and the use of MVB (b). Pink arrows represent cleartext-ciphertext RLWE multiplications.

Decomposer To decompose a 4-bit ciphertext into four binary ciphertext, it is sufficient to perform four programmable bootstrappings (one per bit) to recover the encrypted inputs. This is exactly what we do, but we use the MVB optimization. So we obtain the four ciphertexts at the cost of only one BlindRotate.

Recomposer Recomposing four binary ciphertexts into a 4-bit ciphertext is slightly trickier than the decomposition. As shown in our paper, this is only doable efficiently if the output plaintext modulus is odd. First we have to bootstrap each binary ciphertext to send it in $\mathbb{Z}_{17}$. We then use the ciphertext-plaintext multiplication to align each bits (for instance the ciphertext corresponding to the most significant bit of the message has to be multiplied by 8, and so on), and then we sum the four ciphertexts.

These four steps are represented on Figure 3. More informations about each block of the design can be found in our paper.

KeyExpansion: Our approach is quite similar than with the main encryption algorithm: the Sbox is evaluated with our optimized tree bootstrapping algorithm, and the linear operations are performed in $\mathbb{Z}_2$. Recomposer and Decomposer algorithm are used to move from one representation to another.

Parameters Our parameters set is defined to ensure 128 bits of security. We ensure this level with the `lattice-estimator`. The output of this tool on our parameter set is reproduced on Figure 4.

Our parameters ensure also an error probability of 2^{-64}. To check this, we measured the noise variance of the ciphertexts right before each bootstrapping. Using this value, it is

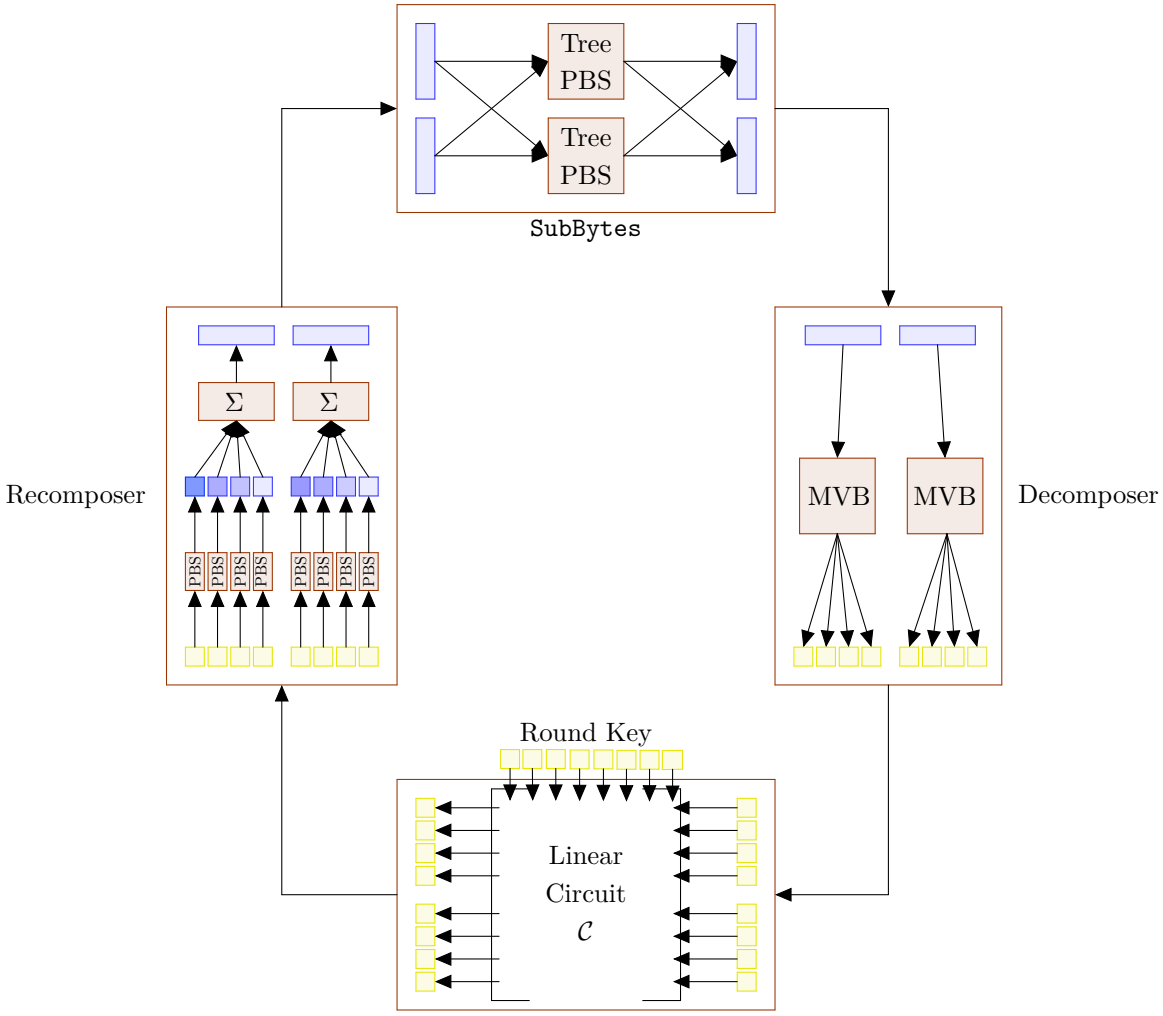

Figure 3: Structure of one round of AES with our method. Ciphertexts in blue live in $\mathbb{Z}_{17}$ while the ones in yellow are in $\mathbb{Z}_2$. Squares represent encryptions of one single bit while rectangles represent nibbles.

```
bkw                 :: rop: ≈2^213.2, m: ≈2^199.5, mem: ≈2^196.1, b: 3, t1: 0, t2: 17, ℓ: 2, #cod: 735, #top: 0, #test: 109, tag: coded-bkw
usvp                :: rop: ≈2^134.3, red: ≈2^134.3, δ: 1.004218, β: 368, d: 1555, tag: usvp
bdd                 :: rop: ≈2^154.9, red: ≈2^123.8, svp: ≈2^154.9, β: 331, η: 479, d: 1347, tag: bdd
dual                :: rop: ≈2^138.0, mem: ≈2^88.5, m: 753, β: 378, d: 1595, ν: 1, tag: dual
dual_hybrid         :: rop: ≈2^128.1, red: ≈2^128.1, guess: ≈2^123.2, β: 342, p: 2, ζ: 10, t: 90, β': 352, N: ≈2^72.0, m: 842
```

Figure 4: Output of `lattice estimator` on our parameter set

possible to compute the error probability of this PBS with the following formula:

$$p_{\text{err}} = \mathsf{erfc}\left(\frac{\tau}{2\sqrt{2}\cdot\sigma_{\text{critical}}}\right) \tag{1}$$

where τ denotes the size of a torus sector, and erfc the complementary error function of Gauss, defined as

$$\mathsf{erfc}(z) = 1 - \frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$$

.

The chosen parameters are reproduced in Table 1.

Table 1: 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}	q	n	N	k	l	B_{PBS}	B_{KS}	t	σ_{LWE}	σ_{GLWE}
2^{-64}	2^{64}	841	2048	1	2	2^{13}	2^4	4	2^{45}	$2^{13.8}$

2.2 A Word on Parallelization

Finally, we would like to emphasize that AES being easily parallelizable (each round can be run separately on each byte), we rely on the `rayon` crate to parallelize our AES evaluation. The 16 Sbox evaluation of a round can be trivially parallelized. Moreover, the evaluation itself of the Sbox can also benefit from some parallelization: the tree bootstrapping for both the most significant nibble and the least significant nibble are independent and are performed in parallel. As the machine on which the code will be run for the bounty is an AWS hpc7a.96xlarge instance, we decided to fully exploit the parallelization offered by our framework.

3 Code Organization

We list here all the modules in the `hippogryph/src` folder and briefly summarize their architecture:

The `clear_aes` module: This module contains a classical AES implementation, built on the `aes` crate. It is used to check the validity of our homomorphic results.

The `encrypted_aes` module: This module contains our homomorphic implementation of the AES cipher.

- `casts.rs`: this file implements our Recomposer and Decomposer features, that are discussed in Section 2.
- `clear.rs`: this file simply holds the plaintext representation of the AES Sbox.

- `linear_circuits.rs`: this file defines a structure implementing the parsing of a "circuit file" and executing it homomorphically. It is typically used to perform the MixColumns step of the AES.
- `key_expansion.rs`: this file implements the KeyExpansion of AES.
- `mod.rs`: this file is at the center of our submission as it implements all the AES steps. In particular, we can find functions such as `sub_bytes`, `add_round_key`, `shift_rows`, and `mix_columns`, that all appear in the `encrypted_aes` function which implement a full homomorphic AES execution.
- `data`: this folder contains the circuit file for the homomorphic execution of our MixColumns operation, taken from [Max19].

The `main.rs` File This file contains our main function that calls both `demo_clear_aes` and `demo_encrypted_aes` on the same inputs. This allows us to run our homomorphic design and to compare the results with the one of the clear implementation with `assert` clauses.

`tfhe-rs` : Our framework requires the manipulation of odd plaintext moduli. This feature being not available in `tfhe-rs`, we implemented it in `tfhe/src/odd`. The encryption and decryption functions have been modified accordingly, as well as the bootstrapping algorithm that required some minor tweaks (namely, on the construciton of the accumulator polynomial). We also implemented the tree bootstrapping and the multi-value bootstrapping, which required the development of a packing keyswitch operation as well.

References

[BBB+25] S. Belaïd, N; Bon, A. Boudguiga, R; Sirdey, D. Trama, and N. Ye. Further improvements in AES execution over TFHE: Towards breaking the 1 sec barrier. Cryptology ePrint Archive, Paper 2025/075, 2025.

[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.

[CIM19] S. Carpov, M. Izabachène, and V. Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In *Topics in Cryptology – CT-RSA 2019: The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4–8, 2019, Proceedings*, page 106–126, 2019.

[GBA21] A. Guimarães, E. Borin, and D. F. Aranha. Revisiting the functional bootstrap in tfhe. 2021, 2021.

[Max19] A. Maximov. AES mixcolumn with 92 XOR gates. *IACR Cryptol. ePrint Arch.*, page 833, 2019.

[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.