**Team Details:**

|  |  |
| --- | --- |
| **Name** | **Enrollment Number** |
| Balaga Pavan Sai | 21114025 |
| Bandi Pavan Karthik | 21114026 |
| Male Jithendra | 21114055 |
| Taviti Venkata Manikanta | 21114105 |
| Uppala Vivek Narayan | 21114108 |

**Section 1: Background and Introduction to FHE:**

Homomorphic Encryption (HE) is a set of cryptographic techniques that allow computations on encrypted data without decrypting it. The decrypted result matches what would have been obtained if the computations had been performed on plaintext. There are three types of HE, namely Partially Homomorphic Encryption (PHE), Somewhat Homomorphic Encryption (SHE), and Fully Homomorphic Encryption (FHE). PHE supports only one type of operation, whereas SHE supports a limited number of operations. FHE supports any arbitrary computations. Hence, it provides strong end-to-end encryption, and the property is crucial in several privacy-preserving computations.

Ciphertext expansion is a significant problem for FHE as it increases transmission time to the server. To circumvent this, transciphering is used where a symmetric key encrypted data and FHE-encrypted symmetric key is sent to the server. The server then decrypts the data using this key within the homomorphic domain on which required calculations can be performed. Transciphering AES ciphertexts has been tough and impractical due to the huge amount of time required. Transciphering-friendly encryption schemes were developed to facilitate faster and efficient homomorphic execution. In 2022, NIST announced that AES would be the benchmark for FHE. This has resulted in a new wave of proposals to reduce AES evaluation, and we explore two such proposals in depth in this report.

The homomorphic encryption scheme used in the two approaches is Fast Fully Homomorphic Encryption over the Torus (TFHE). This is used as it provides a Programmable Bootstrapping (PBS) method which is fast and efficiently implemented compared to other FHE schemes.

An important point to note is that Key Expansion is assumed to be done at the client side, and all eleven round keys are sent to the server in FHE encrypted format. This is assumed throughout our report.

**TFHE:**  
TFHE is a homomorphic encryption scheme based on the LWE problem, optimized for low-precision data and known for its programmable bootstrapping feature. It enables evaluation of univariate functions on ciphertexts while refreshing their noise, supporting various homomorphic operations. In this encryption plain text is embedded into discretized torus.

Zp(ring)= Tp(torus)

Mapping =>ρ ∶ Zp→Zq, defined as ρ ∶ m ↦

This corresponds to only p elements in Zq.These elements are evenly distributed across Zq and form as sectors of Zq, represented as:

**LWE Encryption:**

There are two main things: one is the message, and the other is the secret key sk = (s1, s2,...,sn), sampled uniformly at random from Bn.

Encryption:

(Encoding in Zq)  (LWE encryption)

Decryption:

**GLWE Encryption:** It is the generic form of LWE encryption. It is LWE encryption operates within polynomial rings. Here, the secret key is a vector of keys(S1,S2,.....,Sk), Si is randomly sampled from BN,q[X]k .The message is encoded in a polynomial in ZN,q[X].Here the cipher text takes the form C =(A1,...,Ak,B) where

The different linear operations in TFHE are defined below:

**SumTFHE(c1, c2)** where c1 and c2 are ciphertexts of m1, m2 with noise variances a1 and a2 are added, and c3 is obtained, which encrypts m1+m2 with noise variance a1+a2 (This can be intuitively thought of as adding two independent distributions in probability).

**ClearMultTFHE(c1, v)** where v belongs to Z is a constant which is multiplied to c1 to obtain c which encrypts m\*v with noise (v2)\*a (Multiplying a constant k, the variance is multiplied by k2)

**Key Switching** is also an essential feature of TFHE that allows transformation of a ciphertext c1 under key k1 to ciphertext c2 under key k2 easily using a key switching key (KSK), which is just an encryption of k1 under k2. This is helpful when we need shorter ciphertexts (obtained by shorter keys but at the cost of raising its noise) to say, speed up the bootstrapping algorithm.

**Programmable Bootstrapping (PBS)** reduces the noise homomorphically to a nominal level. Although it’s theoretically implementable everywhere but practically feasible only in TFHE as it’s very efficient, especially for low-precision values. Not only that, but it’s programmable as well, i.e we can perform an operation on the ciphertext while noise is being reset.

To reduce the noise to a nominal level, server needs to evaluate b-ai\*si(decryption equation) and then round the result to nearest integer in Zp. For this, a special key called bootstrapping key is used which is the encryption of si where si is the ith bit of secret key. Since on the server side only Enc(si) are present and si isn’t present so although it doesn’t have the exact values, it performs the subtraction between the plaintext b and a. Enc(si) to obtain ciphertext corresponding to the actual computation which can be trivially done using linear TFHE operations defined earlier. Next operation is rounding off. This is a complex operation involving various stages:

1. **KeySwitching:** A shorter key is used to keyswitch the ciphertexts to a shorter ciphertext to accelerate computation. This step is an important optimization to decrease latency later.
2. **Modulus Switching:** Modulus is switched from q to 2N. This encoding switch is important as BlindRotate only makes sense when the modulus is for 2N and not q.
3. **BlindRotate:** An accumulator polynomial is constructed from the function f alongside PBS (LUT of f is used to obtain coefficients which encode the outputs of f). Then, this polynomial is multiplied with the X-input to this layer. The first coefficient is the required output. This works as the cycle of rotation group(period) is 2N and after 2N again the same thing repeats.
4. **SampleExtract:** The first coefficient is extracted and converted into an LWE ciphertext which has less noise than before. However, if m>p/2 then the text will acquire an additional negative sign. This is the negacyclicity problem.

Also, we observe that the PBS operation’s timing exponentially grows with the precision bits required. Also, another point to note is that the most expensive step in PBS is BlindRotate which accounts for about 90% of the time of PBS due it’s polynomial rotation operations.

**Section 2: Methodology**

**Homomorphic AES Evaluation Using TFHE: Achieving Sub-30-Second Execution:**

The traditional AES algorithm, although very efficient and highly secure for traditional encryption, it causes challenges when it tries for Fully Homomorphic Encryption (FHE). One of the major issue lies in its reliance on non-linear operations, such as the S-box substitution and finite field arithmetic in MixColumns. These operations are computationally expensive in FHE, because they require complex lookup tables and modular arithmetic, which amplify noise within encrypted data. Moreover, AES combines linear operations (e.g., XOR in AddRoundKey) with non-linear operations, which leads to fragmented homomorphic evaluations that are difficult to optimize. Another major concern is the exponential growth of noise during repeated operations across AES's multiple rounds, so frequent bootstrapping will be necessary which is a costly process that further decreases efficiency.

The motivation behind using TFHE to implement AES comes from its ability to take care these challenges. TFHE introduces programmable bootstrapping, which can reduce noise and can also allow arbitrary functions like S-box LUTs to be computed during noise reduction process. With MVB and tree-based approaches, TFHE is able to perform linear and non-linear computations in an efficient manner, reducing AES to a series of optimized LUT queries. TFHE also has the ability to parallelize on multi-core platforms, facilitating realizable execution times for homomorphic AES. This renders TFHE a suitable instrument that can be employed for use in transciphering applications, in which encrypted information can be processed securely without the need for decryption, maintaining confidentiality and efficiency in sensitive computations.

**Multi-Value Bootstrapping (MVB):** MVB is a clever technique which allows us to calculate k functions on a single input with a single PBS where 1 function would require a PBS. This helps us reduce the number of expensive operations (PBS) and thus reduce the time required for AES evaluation. It does this by cleverly factoring the accumulator polynomial so that one PBS will be enough to perform all the k calculations. Each accumulator polynomial has a shared factor v0(X) and acci(X) is a multiplication of this common factor v0(X) and vi(X). This vi(X) is a factor specific to each function fi. v0(X) is common because it matches the rotation-friendly structure required by BlindRotate in TFHE and is one of the cores to the design of bootstrapping in TFHE. This is because v0(X) is the inverse of (1-X) modulo (XN+1) which ensures that the structure of LUT is preserved during rotation. Having this knowledge, we can perform k functions with one BlindRotate Operation. It proceeds as follows:

1. First, we do a BlindRotate on an accumulator polynomial initialized with v0(X).
2. Then, multiplying it with ClearMultTFHE and each fi’s corresponding vi(X), we can get respective acci(X).

This works because rotating a polynomial can be expressed as rotating one of its factors and leaving the other factor as it is. It can be intuitively thought of as analogous to dividing with something (although rotating and dividing are two mathematically different things the concept, I want to convey here is that we get isolate a part and apply transformation on it and then multiply the other part). When you divide a polynomial by something you can just express it as multiplication of a factor divided by that thing and the second factor. Here it just so happens that we know one factor and analogous to division BlindRotate is there and it’s an expensive operation and to reduce that they just apply it on the shared factor and with cheap multiplications for k functions, we get them with just a single BlindRotate. This is quite a clever trick and it’s very elegant.

This way, we get evaluation of k different fi’s on a single encrypted input with k cleantext-ciphertext GLWE multiplications.

**Conversion of Multiplication and XOR operations to LUT evaluation using TFHE:**

In AES in mixed column step we will have matrix multiplication, MUL(a,b) where a,b are two 8-bit integers. But it is observed that a can only take certain values that is a ∈ {2,3,9,b,d,e}. So we can change our MUL(a,b) function into 6 mul\_a(b) functions(for each value of a). Now for each function we can use a table which contains precalculated values for each b (a is fixed and for all possible values of b compute MUL(a,b) and store). So like this we can change multiplication operation into LUT evaluation(for each a,b we have to just search in table).

So now our mix column operation becomes

b[0][j] = 2\*(a[0][j]) ^ 3\*(a[1][j]) ^ a[2][j] ^ a[3][j] = mul\_2(a[0][j]) ^ mul\_3(a[1][j]) ^ a[2][j] ^ a[3][j];

b[1][j] = 2\*(a[1][j]) ^ 3\*(a[2][j]) ^ a[3][j] ^ a[0][j] = mul\_2(a[1][j]) ^ mul\_3(a[2][j]) ^ a[3][j] ^ a[0][j];

Now we have to convert XOR operations into LUT evaluation as they are computationally expensive. We can transform the binary XOR operation into LUT evaluations using a tree-based method with basis-16 decomposition. Since a direct implementation would require 256×256 lookup table for 8-bit inputs which is quiet difficult, we can decompose each byte into two 4-bit limbs as m = m₀ + m₁.16, allowing them to work with more manageable 16×16 LUTs. For computing the XOR of two ciphertexts c and c′, we can break them as c⊕c′ = (c₀,c₁)⊕(c₀′,c₁′) = (c₀⊕c₀′,c₁⊕c₁′), handling each limb separately. First we construct 16 test polynomials, with each encoding a unary XOR operation (xor\_by\_0, xor\_by\_1, etc.). First, we should apply bootstrapping with selector c₀ to these polynomials, obtaining 16 ciphertexts representing xor\_by\_m₀ results. These are combined into a polynomial Pfinal, which should undergo a second bootstrapping with selector c₀′, yielding the final c₀⊕c₀′ result. The same process is applied for c₁⊕c₁′, and the results are recombined. Using this approach we have converted a XOR operation into LUT evaluation.

Let’s go through how different layers of AES are implemented in detail:

1. **SubBytes:** The SubBytes phase uses TFHE's configurable bootstrapping to apply the AES S-box to each byte. Each 8-bit byte is divided into two 4-bit limbs (base 16). The S-box is converted into two parallel 4-bit Look-Up Tables (LUTs) and evaluated using the Tree-Based Method (TBM). Splitting the S-box into smaller 16-entry LUTs and processing them with Multi-Value Bootstrapping (MVB) reduces bootstrapping steps by 50%.
2. **Shiftrows:** This step is implemented by simply permuting the input ciphertexts as per requirement.
3. **MixColumns:** MixColumns, which mixes columns via GF(256) multiplications and XORs is transformed into LUT evaluations for FHE compatibility. GF(256) multiplications by constants (2, 3, 9, 0x0b, 0x0d, 0x0e) are replaced with precomputed 8-bit→8-bit LUTs, evaluated during programmable bootstrapping. XOR operations are optimized using the tree-based method: 8-bit inputs are split into 4-bit limbs (basis 16) and processed via parallel 16×16 LUTs. Basis 16 is used as it is optimal for performance.
4. **AddRound key:** This is just a XOR bit by bit operation between ciphertext and key. Optimization can be done by merging the AddRoundKey operation with the SubBytes transformation. As AddRoundKey always precedes SubBytes in the AES round structure, this can be done and it also eliminates sequential dependencies between operations.
5. **Noise:** It is an inherent feature introduced during encryption to ensure security. It acts as a safeguard against attacks, making it difficult for attackers to extract meaningful information from ciphertexts. Noise is tightly managed by operating with error probability <2-23 ensuring precise parameterization for decryption correctness.
6. **Parallelization:** OpenMP can be used to parallelize the homomorphic AES evaluation across multiple CPU cores for significantly improving execution times. Another optimization can be the implementation by parallelizing all round functions except ShiftRows, because they involve much simpler ciphertext reorganization. Additionally, AddRoundKey can also be merged with SubBytes step to enhance parallelism.

**AES Evaluation over TFHE in almost 1 second: Hippogryph:**

**Negacyclicity Problem:** Due to the additional minus signs introduced in BlindRotate step we need to have an additional constraint on the function f to be evaluated. This is the need to be negacyclic i.e for all x.

To circumvent the negacyclicity problem, one possible approach is to add a bit of padding fixed to 0 in the most significant bit, effectively embedding the plaintext space Zp into Z2p. However, this padding introduces a significant overhead: linear operations are no longer virtually free, as frequent bootstrapping becomes necessary to maintain the padding bit cleared. Also, this introduces more complexities into the logic and more PBS operations are required later.

Another approach involves utilising an odd modulus and modifying the accumulator polynomial and this eliminates the negacyclicity problem. This way we also don’t need to sacrifice the linear operations efficiency unlike the padding approach.

The earlier approach utilizes TBM and converts all the AES layers into LUT operations. The LUT operations are done using PBS but this is a waste of time and computing for trivial operations like XOR, Byte shifts while it’s required for S-box (inspite of optimizing by dividing a byte into 2 4-bit chunks and performing PBS in Z16 instead of Z256 the point still remains that using PBS is overkill). Still, this way AES evaluation time was reduced to 270s (1 core). After this, another approach was implemented which made use of p-encoding method which further reduced the evaluation time. Here the binary operations are easily completed but for s-box it requires a complex Boolean circuit which makes it costly (Here too they haven’t used the naïve method of gate bootstrapping which uses PBS after every gate and instead made use of gadget bootstrapping where the complex circuit is decomposed into small circuits with need of single bootstrapping but still it is slow). But still, it reduced the AES evaluation time further to 90s (1 core).

So, this motivated a hybrid approach combining the best of both approaches utilizing the LUT based approach to evaluate the S-box while the rest of the layers are evaluated using p-encodings. To accomplish this, these had to be combined through a non-trivial transition between the two methods.

Let’s go through how different layers of AES are implemented in detail:

1. **SubBytes:** First each byte is divided into 2 4-bit limbs. The canonical (16,17) encoding is utilized to transform each element in Z16 to a subset of the discretized Torus Z17. This is done for compatibility with Recomposer. Then for computing the output limbs, 2 instances of TBM are used. We use MVB to reduce the number of bootstrapping required. Also, further for one byte, 1 BlindRotate is enough as we can use the same common factor for both evaluations thus requiring only one BlindRotate.
2. **ShiftRows:** This step is implemented by simply permuting the input ciphertexts as per requirement.
3. **MixColumns:** The XOR-only circuit representation is used and for XORring, simple addition in Z2 using native TFHE Addition is used.
4. **AddRoundKey:** This is also a XOR bit by bit between ciphertext and key. So, can be done using native TFHE addition.

An important point to note is that native additions increase noise in the ciphertexts. For Z2, we need not worry much as gap between valid message values is large on the encrypted torus representation. But for Z17 we need to be cautious and apply PBS as required because otherwise noise may grow and give us incorrect answers. In Z17 the noise tolerance is low and sectors on torus are smaller. This results in the SubBytes step being a bottleneck.

As SubBytes and linear operations operate in different encodings, we need to have additional modules to convert between these representations. Thus, we have two additional modules, Decomposer and Recomposer.

1. **Decomposer:** Here, first we decompose the ciphertexts in 4-bit nibbles from SubBytes to 1-bit chunks for use in further layers. Then, we switch the encoding from E17 to E2. Both can be done by using a PBS with MVB at once for each nibble. MVB helps us evaluate all 4 functions at once with a single PBS. This is accomplished by PBS as encoding switching can be done in a straightforward manner using a PBS. This is illustrated, proved and used in the paper.
2. **Recomposer:** Converse to the Decomposer, we also need a module to convert from Boolean domain to Arithmetic domain. This too has two steps, casting the ciphertext from modulus 2 to modulus 17 and recombining the 4 bits to form a ciphertext encrypting the whole nibble. Over here, we need to use PBS for each bit as here we have different ciphertexts. MVB only helps us apply different functions to the same input and hence here it isn’t helpful as the inputs themselves are different necessitating a PBS per bit. The existence of a boolean recomposition algorithm relying solely on one PBS and linear operations per bit depends on the parity of the output plaintext modulus. The plaintext modulus needs to be odd to mitigate the negacyclicity problem. The least odd modulus which can represent all numbers in Z16 is 17 and hence we use Z17. Padding is another possible solution, but we don’t use it due to the problems stated earlier.

**Section 3: Experimental Results and Code Link**

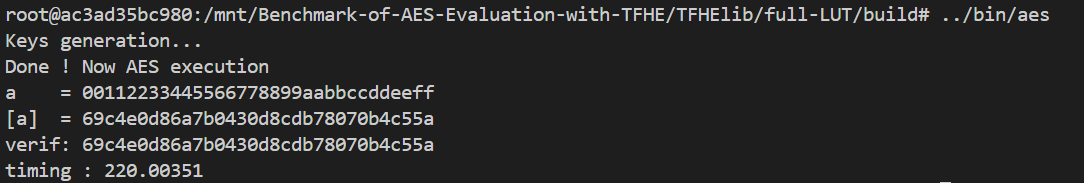
We have performed all the parallelized experiments using AMD Ryzen 9 9950X 16-Core Processor.

**Full LUT based approach Code Execution:**

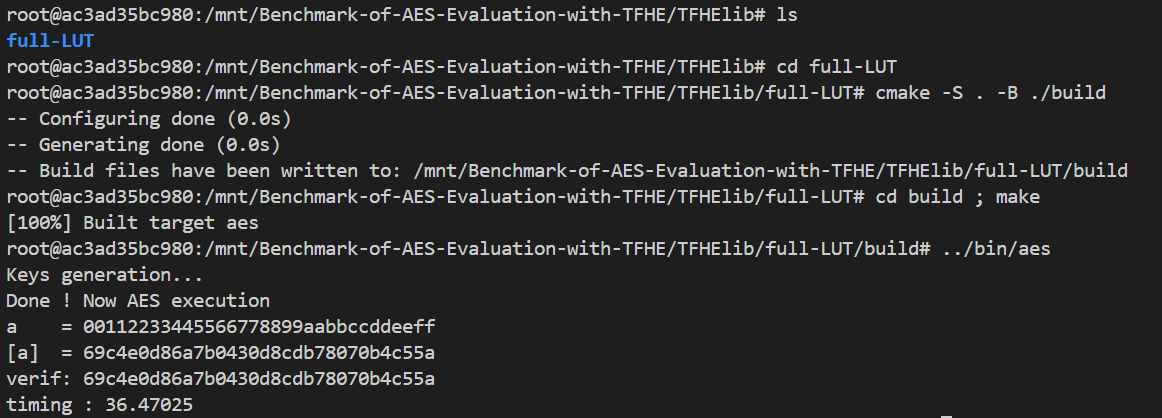
The code used to run Full LUT is mentioned in the [paper](https://github.com/daphnetrm/Benchmark-of-AES-Evaluation-with-TFHE/tree/main/TFHElib/full-LUT) .

When the program runs, the plaintext **a** is encoded and encrypted using TFHE, Encrypted data passes through homomorphic versions of all AES operations. After all rounds are completed, the result is FHE decrypted to verify correctness and this is displayed as **[a]** in the output. The **verif** line in the output confirms that the homomorphic implementation produces the same result as standard AES would when applied to the same input.

The following snapshot shows the single core execution of full-LUT based approach with 32 threads. It took 220 sec for execution in our machine.



The following snapshot shows the parallelized execution of full-LUT based approach with 32 threads. It took 36.47 sec for execution in our machine.



**Hippogryph Execution and Optimization**

The code executed can be found here:<https://github.com/CryptoExperts/Hippogryph>

We executed the Hippogryph codebase, written primarily in Rust, using the commands for optimized execution, cargo build --release and cargo run –release.   
**A screenshot of a computer

AI-generated content may be incorrect.**

The program logs timestamps (in seconds) after each major cryptographic transformation, including SUB\_BYTES, BOOLEAN\_DECOMPOSITION, SHIFT\_ROWS, MIX\_COLUMNS, ADD\_ROUND\_KEYS, and BOOLEAN\_RECOMPOSITION. These stepwise logs enable fine-grained performance analysis of the homomorphic encryption pipeline.

Key Observations:

* Non-Linear Dominance: SubBytes accounted for 23.7% of total runtime (940ms/4013ms), despite using tree-based MVB optimizations.
* Representation Transition: Boolean decomposition/recomposition consumed 19.4% of total time (776ms), emphasizing the cost of switching representations.
* Linear Efficiency: ShiftRows, MixColumns, and AddRoundKeys averaged only 0.8ms per round (~0.02%), validating the effectiveness of boolean circuit optimizations.

For correctness validation, each transformation logs the decrypted actual output (from the FHE domain using a private key) and compares it with the pre-computed expected AES output.

Output:   69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a

Expected: 69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a

The full encryption-decryption cycle took approximately 4.01 seconds, driven largely by bootstrapping overhead.

**Code Optimizations:**

The link to our updated code is [here](https://github.com/pavansai444/CSL-513-Project) <https://github.com/pavansai444/CSL-513-Project>  
We improved performance by avoiding dynamic memory allocation. In particular, clear\_sub\_bytes in clear.rs no longer creates a Vec<u64> on each call. Instead, we use a const static array. Furthermore, initially parsed from text files at runtime in the LinearCircuit module, circuit definitions are now precomputed and embedded as static structures, eliminating repeated file I/O and string parsing. These optimizations reduced the runtime to 3.70 seconds, marking a 0.31s improvement. A snapshot of the modified code’s log is shown below:

**A computer screen shot of numbers and letters

AI-generated content may be incorrect.**