The AES-128 decryption circuit for transciphering in TFHE: A Performance Evaluation

Max Tanis[1]
Supervisors: Dr. Zeki Erkin[1]**, Roderick Ras**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Max Tanis
Final project course: CSE3000 Research Project
Thesis committee: Dr. Zeki Erkin, Dr. Merve Gürel

Abstract

Financial institutions need to outsource fraud detection without exposing highly sensitive and private client data. Homomorphic Encryption (HE) enables computations to be performed on encrypted data without revealing the plaintext data. However, one downside is the massive ciphertext expansion. Due to this expansion ratio, it is impractical for clients who do not have the resources to send over the huge amounts of data. Transciphering is a solution to this by combining symmetric encryption with Fully Homomorphic Encryption (FHE). This paper implements and evaluates an AES-128 decryption circuit in TFHE-rs across varying dataset sizes within a transciphering protocol. The implementation for this paper uses a Boolean circuit, provided by Smart [1], to homomorphically evaluate the AES-128 decryption circuit, which results in an initialization phase of approximately 72 seconds regardless of the dataset size. The per-bit latency decreases from 916 ms/bit for a dataset of $S = 10$ (48 data bits) to 371 ms/bit for $S = 500$ (6468 data bits). In addition, AES-128 is compared to Kreyvium. For datasets of $S = 500$, Kreyvium reduces total evaluation time by a factor of 2.22 compared to AES-128, making it the more efficient choice. While current latency is too high for real-time usage, this provides a baseline for future AES transciphering optimization.

1 Introduction

Financial crime is draining economies of countries round the world. The size of this money laundering is estimated somewhere between 2 and 5% of the global GDP [2]. To detect and prevent these crimes, an enormous amount of analysis is required on transactional data. However, this data is highly sensitive, sharing it raises serious privacy concerns, such as exposure of sensitive client information.

To prevent these privacy concerns, Homomorphic Encryption (HE) is a promising solution. HE is a cryptographic scheme that enables computations on encrypted data without having to decrypt it. This allows financial institutions to collaborate on data without exposing sensitive data. However, HE schemes have two fundamental downsides: slow encryption speed and massive ciphertext expansion. These limitations of HE are impractical for large-scale data processing, creating a need for a more efficient approach.

To address these limitations, transciphering, also known as Hybrid Homomorphic Encryption (HHE), has been proposed [3]. The core idea of transciphering is to combine a symmetric cipher with a Fully Homomorphic Encryption (FHE) scheme, in a hybrid model. FHE is the most powerful variant of HE, supporting an unlimited number of operations on encrypted data. In an HHE setting, the client encrypts their data using a lightweight symmetric cipher, such as Advanced Encryption Standard (AES), and sends the ciphertext together with the homomorphically encrypted symmetric key to the server. The server then homomorphically evaluates the symmetric decryption circuit without learning the underlying plaintext. This approach reduces both the encryption overhead on the client side and the ciphertext expansion during transmission, making HE practical for large-scale data processing.

Several FHE schemes can be used in transciphering protocols, each with its own different characteristics. In this paper, Fast Fully Homomorphic Encryption over the Torus (TFHE) is used. Unlike other schemes which encode integers or real numbers into a single ciphertext, TFHE operates at the bit-level, encrypting each bit individually. This allows logic gates such as AND, XOR, and NOT to be applied directly to encrypted bits, making TFHE particularly suitable for evaluating Boolean circuits homomorphically, such as the AES decryption circuit.

AES is one of the most widely deployed symmetric ciphers in the world [4], standardized by NIST in 2001 [5] and used in most modern virtual security protocols, including the banking infrastructure. However, AES was not designed with homomorphic evaluation in mind, making it computationally expensive compared to HE-friendly ciphers that were specifically designed for transciphering, such as Kreyvium. Despite this, its adoption makes AES a natural baseline for evaluating transciphering performance. In a real-world fraud detection setting, the volume of transactional data processed can vary significantly. How the overhead of AES scales with this volume therefore affects its practical viability. This paper investigates the following question:

How does the computational and communication overhead of homomorphically evaluating the AES-128 decryption circuit in TFHE scale with varying dataset sizes?

To answer this question, an experimental research method is used. The AES-128 decryption circuit is implemented in TFHE-rs [6], a Rust library for TFHE, serving as a baseline to compare with other HE-friendly ciphers. AES-128 is evaluated on both computational and communication overhead. To provide a comparison with an HE-friendly cipher, AES-128 is also compared to Kreyvium [7], a stream cipher specifically designed for transciphering in TFHE.

This paper is structured as follows: Chapter 2 introduces the theoretical background. Chapter 3 outlines the work that is related to this paper. Chapter 4 presents the experimental outline. Chapter 5 describes the results from this experiment. Chapter 6 describes how this experiment kept responsible research in mind. Chapter 7 and 8 discuss and conclude the results from this experiments and what research can be done as an extension of this paper.

2 Preliminaries

2.1 Homomorphic Encryption

Homomorphic Encryption (HE) is a way of encryption that ensures that computations can still be performed on encrypted data, without having to decrypt the data [8]. The result of these computations, when decrypted, is identical to the result that would have been obtained by performing these computations on the plaintext data. This property is particularly powerful in settings where an untrusted third party performs computations on private data, such as outsourcing fraud detection to an external server.
HE schemes are divided into three categories [9]. Partially Homomorphic Encryption (PHE) supports an unlimited number of computations of a single operation, e.g. addition or multiplication. Somewhat Homomorphic Encryption (SHE) supports both addition and multiplication, but only for a limited number of times. Each operation adds a bit of noise to the ciphertext, if this noise exceeds a certain threshold, it is not possible to decrypt correctly anymore. Fully Homomorphic Encryption (FHE) is an extension of SHE introducing bootstrapping, which reduces the noise of the ciphertext. Because of this noise reduction, operations can be computed an unlimited number of times.
Most FHE schemes, including the ones discussed below, base their security on the Learning With Errors (LWE) problem, or on its ring-based variant Ring Learning With Errors (RLWE) [8]. Informally, given a secret vector $\mathbf{s}$, it is computationally infeasible to recover $\mathbf{s}$ from samples of the form $b = (\mathbf{a} \cdot \mathbf{s}) + e \pmod{q}$, where $\mathbf{a}$ is a public vector, q a large prime and e is a small error term. While LWE operates over vectors in $\mathbb{Z}_q^n$, its ring-based variant,

RLWE, replaces this with a polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, where s, a, and e are polynomials rather than vectors. The error term is what makes the scheme secure, but it is also the source of the noise growth that limits SHE and requires bootstrapping in FHE.
FHE comes in different schemes, with the most relevant ones for transciphering listed below.

BFV/BGV The Brakerski/Fan-Vercauteren (BFV) [10] and Brakerski-Gentry-Vaikuntanathan (BGV) schemes both operate over the ring $R_q = \mathbb{Z}_q[X]/(X^n+1)$ and support Single Instruction Multiple Data (SIMD) operations, allowing multiple values to be packed into a single ciphertext. The schemes operate over a plaintext field modulo p, making them suitable for exact integer arithmetic.

CKKS The Cheon-Kim-Kim-Song (CKKS) [11] scheme operates similarly to BFV but uses a rescaling step instead of keeping the plaintext modulus to control noise growth, at the cost of introducing a small rounding error. This error grows with the number of successive multiplications, resulting in approximate rather than exact results. CKKS is therefore suitable for applications where small errors are acceptable.

TFHE Fully Homomorphic Encryption over the Torus (TFHE) is an FHE scheme introduced by Chillotti et al. [12]. TFHE operates at the bit-level, meaning that each individual bit is encrypted as a separate LWE ciphertext over the torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$. Because each bit is encrypted individually, logical gates such as AND, XOR, and NOT can be applied directly to encrypted bits, making TFHE particularly suitable for evaluating Boolean circuits homomorphically, such as the AES decryption circuit used in this paper. Among these gates, linear operations such as XOR and NOT come essentially for free, since LWE ciphertexts are linear in their noise term, whereas non-linear operations such as AND cannot be expressed this way and instead require a bootstrapping step.

2.2 Transciphering

Transciphering, also known as Hybrid Homomorphic Encryption (HHE), is a way of dealing with the practical limitations of FHE. Direct FHE is impractical for resource-limited clients, such as banks, as it produces a massive ciphertext expansion ratio, which results in high communication overhead between clients and servers [13]. A single integer that is encrypted homomorphically can grow from 4 B to over 20 KiB [14].
The core idea of HHE is to shift the heavy encryption from the client-side to the server-side. Instead of encrypting the data directly under an FHE scheme, it is now being encrypted using a symmetric cipher, like AES. The client then only encrypts the symmetric key under an FHE scheme and sends both ciphertexts to the server. The server homomorphically evaluates the symmetric decryption circuit, obtaining an FHE ciphertext of the plaintext data, as displayed in Figure 1. The server can then perform arbitrary computations on the encrypted data and return the result to the client. At no point does the server have access to the client's plaintext data.

There are three main transciphering frameworks, each targeting a different FHE scheme [13], which are listed below and summarized in Table 1.

The CGGI framework. The CGGI-based transciphering framework, based on the TFHE scheme, operates on Boolean plaintext domains by encoding bits as points on the torus. Each non-linear binary gate can be evaluated using a single bootstrapping step, named gate

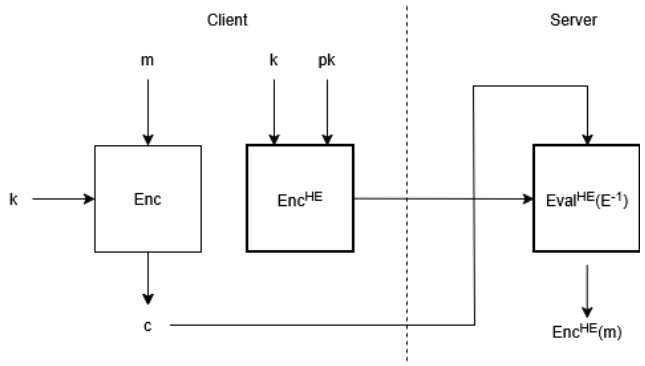

Figure 1: The transciphering protocol. The client encrypts their data (m) symmetrically and the symmetric key (k) homomorphically using public key (pk). The server evaluates the decryption circuit using homomorphic operations (thick-bordered boxes) without learning the plaintext.

bootstrapping (GBS). Besides GBS, the scheme also supports programmable bootstrapping (PBS) and circuit bootstrapping (CBS). CGGI-based methods tend to have lower throughput compared to other FHE schemes due to the absence of SIMD support.

The RtF framework. The Real-to-Finite (RtF) framework is designed around a symmetric cipher whose ciphertext domain is modulo p, matching the plaintext domain of BFV. This allows for an efficient transciphering process into BFV ciphertexts. To enable computation over real numbers, the framework subsequently converts the BFV ciphertext into a CKKS ciphertext, combining the efficiency of CKKS with the practicality of symmetric encryption.

The BtR framework. The Binary-to-Real (BtR) framework addresses the challenge of transciphering into CKKS, which is difficult, due to CKKS's approximate structure and given that a single-bit error leads to a completely different decryption result. The BtR framework overcomes this by representing binary plaintexts as real numbers, where a bit is $x \in \{0, 1\}$ as $x + \varepsilon$ for some small noise $|\varepsilon| \ll 1$. Binary operations are then carried out through arithmetic: AND as $x \cdot y$ and XOR as $(x - y)^2$. The growth of this noise is managed through bootstrapping or step functions. This approach enables the evaluation of any symmetric cipher that can be expressed as a binary circuit.

Table 1: Comparison of the three most important frameworks for transciphering.

Framework	**FHE-scheme**	**Ciphertext-domain**	**Advantage**	**Disadvantage**
CGGI	TFHE	Boolean/torus	Powerful for symmetric ciphers	No SIMD support, lower throughput
RtF	BFV $\rightarrow$ CKKS	mod p	Efficient for real-number computation via CKKS	Loses exactness once converted to CKKS
BtR	CKKS	Real	Enables transciphering of any binary circuit	Single-bit errors could lead to a different result

2.3 Advanced Encryption Standard

Advanced Encryption Standard (AES), also known as the Rijndael Algorithm, is a symmetric block cipher standardized by the National Institute of Standards and Technology (NIST) in 2001 [5]. AES operates on a 4×4 matrix of bytes and encrypts data in a fixed number of rounds depending on the key size. AES-128 uses a 128-bit key and performs 10 rounds, AES-192 uses a 192-bit key and performs 12 rounds, and AES-256 uses a 256-bit key and performs 14 rounds. In this paper, AES-128 is used.
Each round consists of four operations [15]. SubBytes substitutes each byte of the state using a non-linear lookup table called the S-box. ShiftRows shifts the rows of the state by different offsets. MixColumns multiplies each column of the state by a fixed matrix over GF(2^8). AddRoundKey XORs the state with a round key derived from the key schedule. The final round omits MixColumns. A block diagram displaying the encryption and decryption circuit can be seen in figure 2.

2.4 HE-friendly Ciphers

Unlike traditional ciphers such as AES, HE-friendly ciphers are specifically designed to minimize the cost of homomorphic evaluation. They avoid expensive non-linear operations and instead rely on simple arithmetic such as squaring, which is cheap in FHE. There is a large list of HE-friendly ciphers that can be divided into four categories [13]:

FHE-friendly $\mathbb{Z}_2$ Ciphers Ciphers in this category are created to minimize the amount of Boolean gates in their encryption/decryption. This category includes ciphers like Kreyvium.

$\mathbb{F}_p$ Ciphers This group contains ciphers like Masta and Pasta. Ciphers in this category are defined over arithmetic modulo a prime. These ciphers fit perfectly for FHE schemes like BFV or BGV.

Standard Ciphers This is the most standardized group of ciphers, containing ciphers like AES. Most standard ciphers are $\mathbb{Z}_2$ ciphers, but not designed for FHE compatibility.

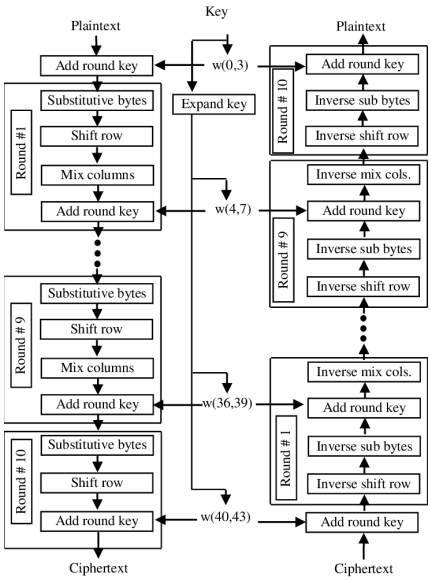

Figure 2: Block diagram of the AES-128 encryption (left) and decryption (right) circuit [15].

$\mathbb{F}_{2^n}$ **Ciphers** Ciphers in this category have operations and plaintexts defined over the finite field $\mathbb{F}_{2^n}$. An important limitation is that transciphering over $\mathbb{F}_{2^n}$ restricts subsequent FHE computations to the same field $\mathbb{F}_{2^n}$.

2.5 Threshold Private Set Intersection (TPSI)

Private Set Intersection (PSI) is a privacy-preserving protocol that allows multiple parties to compute the intersection of their datasets without revealing their individual data to each other [16]. The threshold variant, TPSI, adds the requirement that an element must appear in at least t out of n parties' datasets to be included in the result. This makes TPSI particularly suitable for financial crime detection, where multiple banks can, for example, jointly identify suspicious IP addresses that appear across multiple institutions without exposing their individual transaction data.

3 Related work

3.1 Homomorphic evaluation for AES in BGV

One of the first implementations of homomorphic evaluations of AES was shown by Gentry, Halevi, and Smart [17] using the BGV scheme. They demonstrated that AES-128 can be evaluated homomorphically, achieving a throughput of approximately 2 seconds per block without bootstrapping. This paper also motivates the transciphering settings, where AES encrypted data can be converted to FHE encrypted data via homomorphic AES decryption, enabling further computation without having to reveal the plaintext.

3.2 HE-friendly ciphers

Several works have proposed HE-friendly ciphers as alternatives to AES for transciphering. Cho et al. [18] introduced HERA as part of the Real-to-Finite (RtF) framework, achieving a ciphertext expansion ratio of 1.23 to 1.54 at least 23 times smaller than direct CKKS encryption. Ha et al. [19] proposed Rubato as an improvement over HERA, reducing multiplicative depth while improving throughput by up to 32.2%. A comparison of different HE-friendly ciphers, including HERA and Rubato, is provided by Niu et al. [13].
Despite these advances, HE-friendly ciphers lack the extensive cryptanalysis that AES has undergone since its standardization in 2001 [5]. AES therefore remains a natural baseline for evaluating transciphering performance.

3.3 Transciphering for AES

Trama et al. [20] proposed an AES-128 implementation in TFHE using programmable bootstrapping. They optimized the FHE operators for AES evaluation and benchmarked different decomposition bases to minimize execution time, achieving under 30 seconds in parallel mode.

Bae et al. [21] proposed a high-throughput AES transciphering implementation using CKKS, achieving a per-block evaluation time of under 1 ms via SIMD parallelism. While this approach achieves significantly higher throughput than TFHE-based methods, it relies on CKKS's approximate arithmetic, making it less suitable for exact computations such as fraud detection thresholds.

3.4 Research Gap

While existing work has demonstrated the feasibility of homomorphic AES evaluation in both TFHE and CKKS, these works focus on optimizing latency and throughput. To the best of our knowledge, no existing work evaluates the computational and communication overhead of AES transciphering across varying dataset sizes. This paper addresses this gap by implementing and evaluating AES-128 transciphering, providing information on its practicality.

4 AES-128 transciphering

In this section we describe the AES-128 implementation for transciphering in TFHE-rs and the evaluation methodology that is used for its communication and computation overhead.

4.1 AES-128 implementation

For the implementation that is created for this paper, Tauber's method [22] is used as a reference. This method is a Fully Homomorphic version of AES-128 cryptosystem in Rust using the TFHE-rs library [6]. In our implementation, we focus on the decryption circuit of AES-128 in Counter Mode (CTR). CTR Mode was chosen because, in this mode, each block is encrypted independently, without depending on the output of previous blocks. This makes CTR mode well-suited for homomorphic evaluation, as each block can be processed independently [23].

The implementation uses the Boolean API of TFHE-rs, evaluating the AES circuit gate-by-gate. The complete AES-128 decryption, including all operations, is provided by the Smart [1] as a Boolean circuit of AND, XOR and INV gates. It consists of 6,400 AND gates, 28,176 XOR gates and 2,087 INV gates with a multiplicative depth of 60. The circuit is loaded and evaluated directly in TFHE-rs. In TFHE, each AND gate requires a bootstrapping operation, while XOR and INV gates are evaluated for free, since these are linear operations. The circuit is evaluated layer by layer, where all independent gates within the same layer are executed in parallel. The two-phase evaluation process is described in Section 4.2.
The Boolean API was chosen because AES is inherently a Boolean circuit, operating on individual bits through AND, XOR, and NOT operations. Since the complete AES circuit is already provided as a Boolean circuit by Smart, the Boolean API is the natural choice for evaluation in TFHE-rs.
The AES-128 transciphering implementation is integrated into a larger project aimed at computing TPSI [16]. This paper extends the TPSI framework by adding a transciphering path, in which the client encrypts their dataset using AES-128 and the server homomorphically evaluates the AES-128 decryption circuit on the encrypted data. The data set of the client consists of a list of S IP addresses which are compared between multiple parties to compute their intersection, where S denotes the party set size.

4.2 Transciphering Protocol

The transciphering protocol consists of two parties. In our method, we have the client, who encrypts its dataset using AES-128 in Counter Mode (CTR) and encrypts the AES key under TFHE, and the server, who evaluates the AES decryption circuit homomorphically on the encrypted data. The client sends both the AES ciphertext and the TFHE-encrypted key to the server. At no point does the server have access to the plaintext data. For our implementation, we mainly focus on the server-side, as this is where the homomorphic decryption of AES-128 takes place. The protocol consists of two phases:

Init phase. This phase is executed only once per session. The value of the TFHE-encrypted AES key from the client is inserted into the pre-loaded Boolean circuit. The value of the 128 encrypted bits is assigned to the first 128 wires of the circuit. After which, all gates that depend solely on the key are evaluated. This requires 1,280 AND gate evaluations, all requiring a bootstrapping operation in TFHE. This phase is independent of the size of the dataset, as it depends only on the key.

Per-block phase. This phase is executed for every 128-bit block in the client's dataset. If the data is not a multiple of 128 bits, it is padded with zeros to the nearest multiple of 128 bits before evaluation. For each block, a counter value is constructed based on the block number and trivially encrypted under TFHE. The counter bits are assigned to the remaining 128 inputs wires of the circuit, alongside the TFHE-encrypted key from the init phase. The circuit then homomorphically evaluates the AES encryption on the counter, producing a TFHE-encrypted keystream of 128 bits. This keystream is XORed with the trivially encrypted ciphertext bits of the corresponding block, resulting in the homomorphically encrypted plaintext block. The complete per-block evaluation consists of 5,120 AND gates, each requiring a bootstrapping operation in TFHE.

4.3 Evaluation Metrics

To evaluate the performance of AES-128 transciphering in TFHE-rs, we focus on the computational and communication overhead. The results are then compared to an already implemented HE-friendly cipher, Kreyvium.

4.3.1 Computational Overhead

The computational overhead is measured in terms of latency. Two phases are distinguished, which are described in section 4.2. The initialization phase is measured in seconds. The per-block latency is measured in milliseconds per bit (ms/bit).

4.3.2 Communication Overhead

The communication overhead is measured as the size of the data transferred from the client to the server. Three values are reported and compared: the plaintext size, the direct FHE ciphertext size, and the transcipher ciphertext size. Additionally, the bandwidth saving is given, which expresses how much smaller the ciphertext is for transciphering compared to direct FHE-encryption.

4.4 Kreyvium

In order to have an idea of how AES-128 performs against an HE-friendly cipher, we compare AES-128 with Kreyvium. Kreyvium is a stream cipher based on Trivium, specifically designed for transciphering in TFHE [7]. Unlike AES-128, Kreyvium was designed with homomorphic evaluation in mind, making it a natural candidate for comparison within the same TFHE framework.
The metrics described in Section 4.3.1 are used to evaluate Kreyvium, allowing a direct comparison with AES-128.

5 Results

5.1 Experimental Setup

During our experiment, we have conducted all results on the same laptop. The laptop is running on Windows 11 with a 13th Gen Intel Core i7-13620H processor (2.40 GHz, 10 cores) and 16 GB of RAM, using Windows Subsystem for Linux (WSL). The implementation is written in Rust using the TFHE-rs library (version 0.8.7). While executing the experiment, no other programs were running, except for the AES-128 decryption circuit.
The implementation was benchmarked on party set sizes $S \in \{10, 25, 50, 75, 100, 200, 500\}$ where S represents the number of IP addresses in each party's dataset. This corresponds to the following amount of bits used for each run $\{48, 168, 408, 674, 959, 2206, 6468\}$ respectively. Each configuration was executed 30 times. The initialization phase and per-block evaluation phase were measured separately. Results are reported as mean and standard deviation.

5.2 Results

5.2.1 Computational Overhead

Table 2: Computational overhead of AES-128 transciphering per dataset size. With initialization time and per-bit latency as mean (μ) and standard deviation (σ) over 30 runs.

S	**# Bits**	**Init μ (s)**	**Init σ (s)**	**ms/bit μ**	**ms/bit σ**
10	48	74.44	0.85	916.17	10.63
25	168	74.28	0.97	541.41	7.41
50	408	75.16	1.29	456.18	7.57
75	674	74.38	1.10	413.27	4.35
100	959	71.15	2.00	383.56	8.55
200	2206	68.88	1.58	381.52	6.5
500	6468	69.04	0.89	371.11	6.76

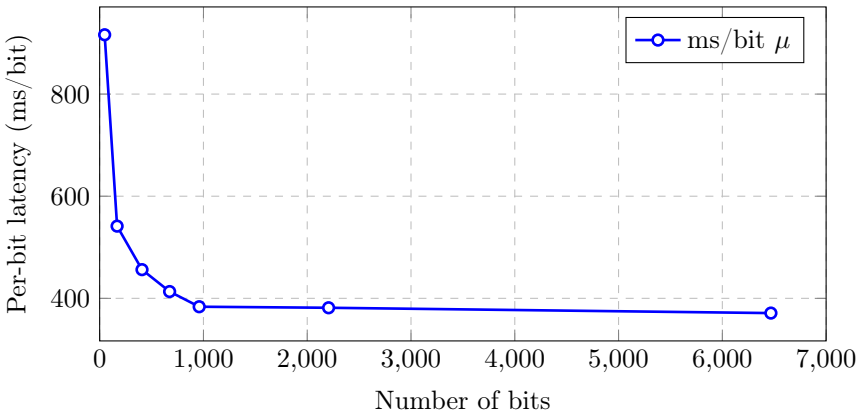

Figure 3: Per-bit latency of AES-128 transciphering vs number of bits. Data used from Table 2.

Table 2 shows the computational overhead of AES-128 transciphering for different dataset sizes. The initialization phase remains approximately constant around 72 seconds independent from the dataset size. The per-bit latency drops from 916.17 ms/bit for $S = 10$ to 371.11 ms/bit for $S = 500$. As Figure 3 shows, the per-bit latency drops really fast for small dataset size, and stabilizes at approximately 370 ms/bit for larger dataset sizes. The relatively low standard deviation (σ), for both phases, shows the consistency for each different run.

5.2.2 Communication Overhead

Table 3: Communication overhead of AES-128 transciphering per dataset size, showing plaintext size, direct FHE ciphertext size, transciphering ciphertext size, and bandwidth savings for using transciphering compared to direct FHE.

S	# Bits	Plaintext	FHE direct	Transcipher	Savings
10	48	6 B	152.81 KiB	407.52 KiB	0.38
25	168	21 B	534.84 KiB	407.53 KiB	1.31
50	408	51 B	1.27 MiB	407.56 KiB	3.20
75	674	85 B	2.10 MiB	407.59 KiB	5.28
100	959	120 B	2.98 MiB	407.62 KiB	7.49
200	2206	276 B	6.86 MiB	407.77 KiB	17.2
500	6468	809 B	20.11 MiB	408.30 KiB	50.4

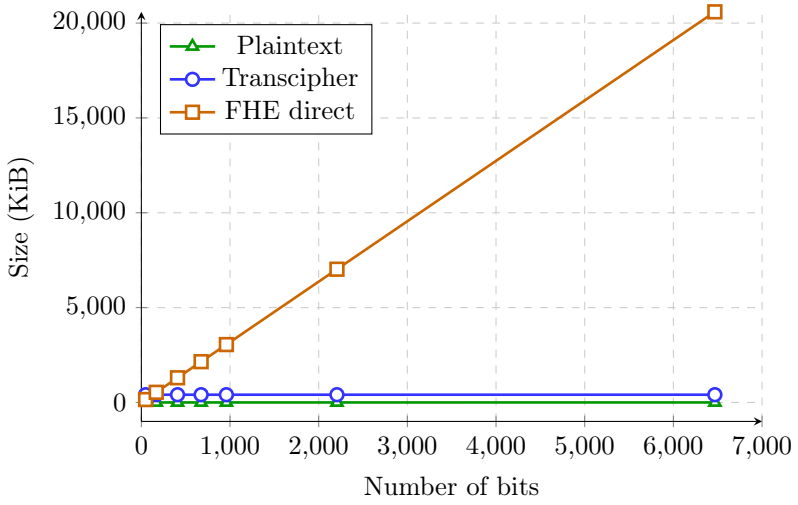

Figure 4: Comparison of ciphertext sizes between direct FHE and AES-128 transciphering for different dataset sizes. Data used from Table 3.

Table 3 shows the ciphertext sizes for using direct FHE and AES-128 transciphering for different dataset sizes. Based on these results Figure 4 shows these results as a graph. The transciphering ciphertext remains approximately constant at 407 KiB, due to the dominance of the TFHE-encrypted AES-128 key, which consists of 128 FHE encrypted bits. The symmetric encrypted plaintext is only a small fraction of the total ciphertext size. The FHE direct ciphertexts grows linearly with the dataset size, from 152.81 KiB for $S = 10$ to 20.11 MiB for $S = 500$. As a result, the bandwidth savings of transciphering over direct FHE encryption increases with the dataset size.

5.2.3 AES-128 vs Kreyvium

Table 4: AES-128 vs Kreyvium: computational overhead comparison per dataset size. The total time is calculated as Total = Init + (ms/bit × #bits)/1000. The total speedup is the factor saved using Kreyvium instead of AES-128.

		AES-128			**Kreyvium**			**Speedup (×)**	
S	**# Bits**	**Init (s)**	**ms/bit**	**Total (s)**	**Init (s)**	**ms/bit**	**Total (s)**	**Per-bit**	**Total**
10	48	74.44	916.17	118.42	161.72	149.22	168.88	6.14	0.70
25	168	74.28	541.41	165.24	167.12	155.55	193.25	3.48	0.86
50	408	75.16	456.18	261.28	171.24	159.92	236.49	2.85	1.10
75	674	74.38	413.27	352.92	169.91	158.57	276.79	2.61	1.28
100	959	71.15	383.56	438.98	170.45	159.09	323.02	2.41	1.36
200	2206	68.88	381.52	910.51	158.09	147.57	483.63	2.58	1.88
500	6468	69.04	371.11	2468.37	157.82	147.69	1113.08	2.51	2.22

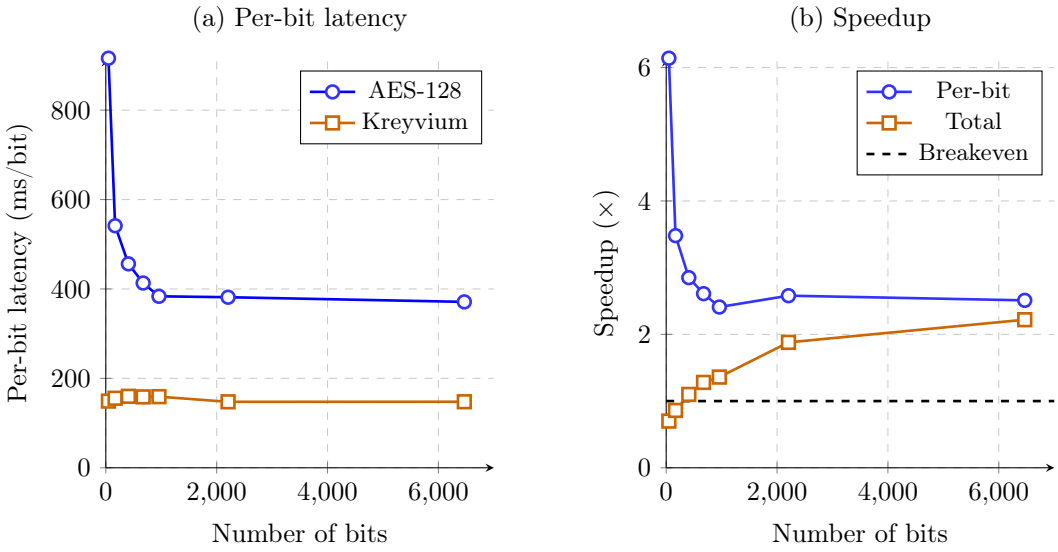

Figure 5: AES-128 vs Kreyvium: (a) per-bit latency and (b) speedup comparison. Data used from Table 4.

Table 4 compares the computational overhead of AES-128 and Kreyvium for different dataset sizes. The initialization phase of Kreyvium is significantly higher than that of AES-128, averaging 165 seconds compared to 72 seconds for AES-128. However, Kreyvium's per-bit latency is consistently lower, resulting in a per-bit speedup of at least 2.41× in its favor as shown in Figure 5 (a).
Besides this lower per-bit latency, the higher initialization time for Kreyvium results in a higher total evaluation time for small datasets. As shown in Figure 5 (b), AES-128 has a lower total evaluation time at $S = 10$ and $S = 25$ (total speedup below 1×), while Kreyvium becomes more efficient for larger dataset sizes, reaching a speedup of 2.22× at $S = 500$.

6 Responsible Research

Ethical Considerations This research is motivated by helping financial institutions detect fraud without exposing sensitive client data. We believe this work is unlikely to aid adversaries, as it does not introduce new attack techniques but instead evaluates the performance of an already existing cipher (AES). All experiments used synthetic IP addresses rather than real client data, eliminating any risk of leaking sensitive information during this research.

Reproducibility. To ensure reproducibility for this research, the experimental setup is described in detail in Section 5.1, including hardware specifications and software versions. Furthermore, this paper builds upon open source tools including TFHE-rs and the Boolean circuit provided by Smart. The source code is publicly available in a GitHub repository[1] to reproduce this experiment on other devices. However, the exact reproduction of the results may vary due to hardware differences.

7 Discussion

Computational overhead The results in Table 2 show that the initialization phase of AES-128 takes an average of approximately 72 seconds. This is explained by the 1,280 AND gates that must be evaluated during this phase (Section 4.2), each requiring a bootstrapping operation. Since this phase depends only on the key, it is independent of the dataset size. The standard deviation of this phase is at most 2 seconds, which indicates that it is relatively stable across all runs.
The per-bit latency decreases as the dataset size increases. This is because a full evaluation of the per-block phase is required for every 128-bit block, regardless of how many of its bits represent actual data. For small datasets, this fixed per-block cost is spread over only a few data bits, resulting in a high per-bit latency. For larger datasets, the same cost is spread over many more bits, reducing the per-bit latency accordingly. This suggests that the relative overhead of AES-128 transciphering improves as it is applied to larger, more realistic workloads.

Communication overhead Figure 4 is a visualization based on Table 3. The transciphering ciphertext consists of two parts: the symmetric encrypted plaintext and the TFHE-encrypted AES key. Since symmetric encryption produces a ciphertext of almost the same size, the first component almost grows linearly with the plaintext size. The second component, the TFHE-encrypted AES key, is always 128 bits and therefore remains constant at approximately 407 KiB, independent of the dataset size. For this paper, the plaintext is relatively small compared to the key component, which results in the total transciphering size being almost constant.
Using direct FHE, on the other hand, encrypts each individual bit as a TFHE ciphertext. As a result, the ciphertext size grows linearly with the number of bits in the plaintext.

Comparison AES-128 vs Kreyvium The results in Table 4 show that the choice between AES-128 and Kreyvium depends on the dataset size. For really small dataset sizes, AES-128 is preferable due to its lower initialization time. However, for larger dataset sizes

[1] `https://github.com/maxtanis03/AES-128-decryption-circuit-in-TFHE`

Kreyvium becomes more suitable because of its lower per-bit latency. In a practical situation where banks outsource their data, Kreyvium would be a more efficient choice. However, AES-128 remains relevant as it is one of the most widely deployed symmetric ciphers [4].

Limitations When we look at the implementation used for this research, we can already determine one major limitation. The AES-128 decryption circuit is evaluated using a Boolean circuit which consists of 6,400 AND gates, all requiring a bootstrap operation. This requires more bootstrapping than a PBS-based approach such as the implementation of Trama et al. [20]. Using a Boolean circuit results in a higher per-bit latency compared to the optimized situation.
Another limitation for this paper is the number of HE-friendly ciphers that are being compared. AES-128 is only compared to Kreyvium on computational overhead, while other HE-friendly ciphers, such as HERA and Pasta, are not included.

8 Conclusions and Future Work

8.1 Conclusion

This paper aimed to measure the performance of AES-128, one of the most widely used symmetric ciphers, in a transciphering protocol by answering the following question: *How does the computational and communication overhead of homomorphically evaluating the AES-128 decryption circuit in TFHE scale with varying dataset sizes?*
We implemented the AES-128 decryption circuit in TFHE-rs as a Boolean circuit. Computationally, each run requires a fixed initialization phase of approximately 72 seconds, after which the per-bit cost decreases from 916 ms/bit (48 data bits) to 371 ms/bit (6468 data bits). This is because the per-block phase must be evaluated for every 128-bit block, regardless of how many of its bits are actual data. For smaller datasets, this cost is spread over only a few data bits, whereas for larger datasets, it is spread over many more. This indicates that the computational overhead of AES-128 transciphering scales sub-linearly with the dataset size.
Communication overhead grows linearly with the number of bits for direct FHE ciphertext size, whereas the ciphertext size for transciphering remains almost constant around 407 KiB, dominated by the fixed-size FHE encrypted key. This results in a bandwidth saving from 1.31× (168 data bits) up to 50.4× (6468 data bits), confirming that transciphering's advantage over direct FHE grows with the dataset size. However, for the smallest dataset size we have tested (48 data bits), transciphering is less efficient than direct FHE since the fixed cost of the FHE encrypted key outweighs the small amount of data being transmitted. When comparing AES-128 to Kreyvium, AES-128 is only cheaper for smaller dataset sizes ($\leq$ 168 data bits). Beyond that, Kreyvium is up to 2.22× faster overall. This shows that the choice between a standardized and HE-friendly cipher depends on the data scale rather than one cipher being universally preferable.

8.2 Future Work

Future work based on this paper should first focus on the improvement of the AES-128 decryption circuit. The per-bit latency could be reduced by implementing the S-box evaluation using PBS via the ShortInt API, instead of the Boolean circuit approach used in this paper. This would reduce the number of bootstrapping operations required per block.

Furthermore, the performance comparison could be extended by including other HE-friendly ciphers such as HERA and Pasta. This provides a better overview of the performance of other HE-friendly ciphers compared to AES-128.
Finally, the dataset sizes used in this paper remain small compared to real-world banking datasets. It is not yet known whether the observed scaling continues for larger dataset sizes.

References

[1] N. P. Smart, "MPC circuits." [Online]. Available: `https://nigelsmart.github.io/MPC-Circuits/`. visited on 01/06/2026.

[2] Europol, "Criminal finances and money laundering." [Online]. Available: `https://www.europol.europa.eu/crime-areas/criminal-finances-and-money-laundering`. visited on 01/06/2026.

[3] I. Thakur, A. Karmakar, C. Li, and B. Preneel, "A survey on transciphering and symmetric ciphers for homomorphic encryption." Cryptology ePrint Archive, Paper 2025/093, 2025.

[4] C. Paar and J. Pelzl, *The Advanced Encryption Standard (AES)*, pp. 87–121. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[5] M. J. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. D. Jr., "Advanced encryption standard (aes)," Nov. 2001.

[6] Zama, "TFHE-rs: A pure Rust implementation of TFHE." `https://docs.zama.org/tfhe-rs`, 2022. visited on 01/06/2026.

[7] 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 *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers* (T. Peyrin, ed.), Lecture Notes in Computer Science, pp. 313–333, Springer, 2016.

[8] C. Marcolla, V. Sucasas, M. Manzano, R. Bassoli, F. H. P. Fitzek, and N. Aaraj, "Survey on fully homomorphic encryption, theory, and applications," *Proc. IEEE*, vol. 110, no. 10, pp. 1572–1609, 2022.

[9] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A survey on homomorphic encryption schemes: Theory and implementation," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 79:1–79:35, 2018.

[10] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[11] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I* (T. Takagi and T. Peyrin, eds.), Lecture Notes in Computer Science, pp. 409–437, Springer, 2017.

[12] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, pp. 34–91, 2020.

[13] C. Niu, B. Wei, Z. Huang, Z. Yang, C. Hong, M. Wang, and T. Wei, "Sok: Fhe-friendly symmetric ciphers and transciphering," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2025, no. 3, pp. 583–613, 2025.

[14] S. Gupta, R. Cammarota, and T. Simunic, "Memfhe: End-to-end computing with fully homomorphic encryption in memory," *ACM Trans. Embed. Comput. Syst.*, vol. 23, no. 2, pp. 28:1–28:23, 2024.

[15] B. Sarkar, A. Saha, D. Dutta, G. Sarkar, and K. Karmakar, "A survey on the advanced encryption standard (aes): A pillar of modern cryptography," *International Journal of Computer Science and Mobile Computing*, vol. 13, pp. 68–87, 04 2024.

[16] F. Liu, E. Zhang, and L. Qin, "Efficient multiparty probabilistic threshold private set intersection," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023* (W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, eds.), pp. 2188–2201, ACM, 2023.

[17] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 99, 2012.

[18] J. Cho, J. Ha, S. Kim, B. Lee, J. Lee, J. Lee, D. Moon, and H. Yoon, "Transciphering framework for approximate homomorphic encryption," 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* (M. Tibouchi and H. Wang, eds.), Lecture Notes in Computer Science, pp. 640–669, Springer, 2021.

[19] J. Ha, S. Kim, B. Lee, J. Lee, and M. Son, "Rubato: Noisy ciphers for approximate homomorphic encryption (full version)," *IACR Cryptol. ePrint Arch.*, vol. 2022, p. 537, 2022.

[20] D. Trama, P. Clet, A. Boudguiga, and R. Sirdey, "At last! A homomorphic AES evaluation in less than 30 seconds by means of TFHE," *IACR Cryptol. ePrint Arch.*, vol. 2023, p. 1020, 2023.

[21] Y. Bae, J. H. Cheon, M. Kang, and T. Kim, "High-throughput AES transciphering using CKKS: less than 1ms," *IACR Cryptol. ePrint Arch.*, vol. 2025, p. 1865, 2025.

[22] T. Tauber, "FHE-AES: Fully homomorphic AES in Rust." [Online]. Available: `https://github.com/tomtau/fhe-aes/tree/main`. visited on 01/06/2026.

[23] Zama, "Implement a fully homomorphic version of the AES-128 cryptosystem using TFHE-rs." [Online]. Available: `https://www.zama.org/post/implement-fhe-aes-128-tfhe-rs`, 2023. version 0.8.7, visited on 01/06/2026.

A LLM

During the writing process, an LLM was used to check grammar, sentence structure, and clarity of the written text. The tool was not used to generate original research content, results, or analysis.