






E2E near-standard and practical authenticated transciphering

Ehud Aharoni 
IBM Research - Israel

Nir Drucker 
IBM Research - Israel

Gilad Ezov 
IBM Research - Israel

Eyal Kushnir 
IBM Research - Israel

Hayim Shaul 
IBM Research - Israel

Omri Soceanu 
IBM Research - Israel

Abstract—Homomorphic encryption (HE) enables computation delegation to untrusted third-party while maintaining data confidentiality. Hybrid encryption (a.k.a Transciphering) allows a reduction in the number of ciphertexts and storage size, which makes HE solutions practical for a variety of modern applications. Still, modern transciphering has two main drawbacks: 1) lack of standardization or bad performance of symmetric decryption under FHE; 2) lack of input data integrity. In this paper, we discuss the concept of Authenticated Transciphering (AT), which like Authenticated Encryption (AE) provides some integrity guarantees for the transciphered data. For that, we report on the first implementations of AES-GCM decryption and Ascon decryption under CKKS. Moreover, we report and demonstrate the first end-to-end process that uses transciphering for real-world applications i.e., running deep neural network inference (ResNet50 over ImageNet) under encryption.

I. INTRODUCTION

Nowadays, many organizations move their workloads from in-house data centers to public cloud environments. This trend has not skipped conservative industries like finance and healthcare. However, the use of these third-party services can be restricted by the need to comply with government regulations such as GDPR [35] and HIPAA [15], which ensure data privacy.

Modern cryptography provides useful and standardized solutions for ensuring the confidentiality and integrity of organizations’ data in transit, for example, through the use of TLS 1.3 [57], or at rest, using advanced encryption standard (AES)-Galois / counter mode (GCM) [34]. While the combination of these solutions allows users to take advantage of the storage services provided by the cloud, they do not provide a secure way to utilize its computing capabilities. The reason is that the computations are done in the clear.

One cryptographic solution to the above issue, which gains popularity nowadays is homomorphic encryption (HE) because it enables computation to be performed on encrypted data. The potential of HE can be observed in Gartner’s report [38], which states that by 2025, 50% of large enterprises are expected to adopt privacy-enhancing computation for processing data in untrusted environments e.g., by using HE. Another example that highlights its widespread adoption is the extensive list of enterprises and academic institutions actively involved in initiatives like HEBench [60] and the standardization efforts for HE [4].

The literature on HE primarily focuses on demonstrating its practicality for specific use cases, such as performing

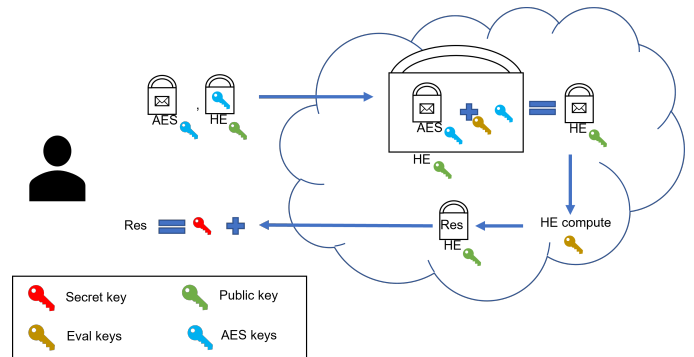


Figure 1: A Hybrid encryption a.k.a. transciphering flow. A user possesses a symmetric AES key (blue) and an HE secret key (red). They upload the HE public key (green) and the evaluation key (brown) to the cloud. The user encrypts the data using AES and sends it to the cloud, along with an encryption of the AES keys under HE. The cloud replaces the AES encryption with HE encryption, processes the data, and returns the results FHE encrypted back to the user. Finally, the user utilizes the secret HE key to decrypt the results.

classification through inferencing over deep neural networks [50], [2], [7]. These studies often assume an ephemeral application, where data is encrypted using HE, uploaded to an untrusted environment for HE computation, and the results are promptly returned to the user for decryption.

However, in reality, the situation becomes more complex when users need to store their data encrypted at one point in time and use it in the cloud for computation at a later point in time. In such cases, the large size of HE ciphertexts can result in extra costs. For instance, HE ciphertexts may have an expansion ratio of more than 2:1 compared to storing the original plaintext, or compared with the 1:1 compression ratio when using symmetric encryption such as AES. These costs impact not only the storage “at rest” but also the bandwidth required for uploading and downloading the data to and from the cloud.

Hybrid encryption, a.k.a., transciphering (see e.g., [39], [58]), enables the encryption of data using symmetric block ciphers, which can then be transmitted and stored in the cloud at moderate costs. Subsequently, the encrypted data can be moved to a computing service that employs HE. Through

transciphering, the service effectively “replaces” the encryption scheme from symmetric encryption to HE encryption. Once the data is encrypted using HE, the service can perform computations on it and return the results to the user or store them for future use. Figure 1 illustrates the hybrid encryption process.

Till this work, there were three main drawbacks to hybrid encryption: a) it is either considered unpractical for many applications; b) it involves non-standardized symmetric ciphers, which again places some restrictions on organizations from using such a solution; c) it does not authenticate the data that is fed to the HE service, when its origin is another untrusted entity such as a storage service.

Our contribution. Our contribution is as follows:

- We provide the first hybrid encryption implementation that can be considered practical by many applications that use standardized block-cipher, specifically AES-CTR. We do that by using the IBM[®] HElayers library [2] compiled with the CryptoLab[®] HEaAN [26] library and running on a commodity GPU. Our implementation runs in 4 minutes for 512KB of data, with amortized latency of 7 msec per AES block.
- Today, many applications attempt to ensure both the confidentiality and the integrity of the user data. To this end, using an authenticated encryption with associated data (AEAD) scheme is advantageous over using solely a symmetric cipher. Consequently, we have enhanced our implementation to encompass the novel integration of hybrid encryption with a standardized AEAD, specifically AES-GCM. Our implementation exhibits an efficient performance, processing 512KB of data in a mere 11.5 minutes, with an amortized latency of 21 milliseconds per AES block.
- In addition to our AES-GCM implementation, we also implemented and evaluated the Ascon cryptosystem under HE. Ascon is the winner of the national institute for standards and technology (NIST) lightweight project [55] and is about to be standardized.
- We study and discuss the security guarantees of the above implementation using a notion that we call authenticated transciphering (AT). Here the integrity of the input data to the HE service is guaranteed. We discuss the advantages and disadvantages of this notion and how it should be used in threat models.
- We demonstrate for the first time an end-to-end application that uses near-standard hybrid encryption. We say that it is near-standard because an HE standard is only expected in 2024. Our demonstration involved a large task of running neural network inference over a large network, ResNet-50, and a large dataset – ImageNet with images of size 224x224x3.

Roadmap. The rest of the paper is organized as follows. Section II reports the state-of-the-art hybrid encryption constructions. Section III lists the notation used in the paper, and briefly explains the concepts of HE and AEAD. The paper proceeds by describing our fast AES-CTR implementation in Section IV. The handling of the error of CKKS is explained in Section V, and the notion that we call AT is explored in

Table I: Standardized AES-ECB implementations under FHE, Reported measurements are of the cited references, which may derived using different hardware.

Ref.	Scheme	Security bits	Latency (hours)	Amortized latency (min/AES block)
[39]	BGV	128	65	5
[39]	BGV	128	36	40
[61]	BGV	128	0.023	1.4
[16]	DGHV	72	113	12
[23]	DGHV	72	18.3	33
[23]	DGHV	72	3.58	0.38
[23]	DGHV	80	102	3.25
[59]	TFHE	128	0.07	4.2

Section VI. Using this notion, AT schemes using AES-GCM or Ascon, and CKKS are implemented in Sections VII and VIII, respectively. The latency measurements of running these schemes are reported in Section IX. The above implementation is utilized to construct and demonstrate the first end-to-end flow for performing deep neural network inference over encrypted data in Section X. Finally, Section XI discusses some takeaways from the study, and we conclude the paper in Section XII.

II. RELATED WORK.

Related work for transcribers involve two principle categories: a) implementations of transcribers that use standard symmetric encryption e.g., AES-electronic codebook (ECB) [54]; b) implementations of transcribers that offer a new non-standardized symmetric encryption schemes.

The first type of implementations is compared in Table I. Some implementations offer less than 128-bit security e.g., [16], [23], while others took over an hour to complete e.g., [39], [61]. Among these implementations, the one with the fastest amortized latency was [23] that used the BGV HE scheme, which took 0.38 minutes per AES-128 block while the FHE security level was only 72 bits. In contrast, our implementation is several orders of magnitude faster, taking only 7 amortized milli-seconds to decrypt an AES-256 block when decrypting a batch of 32,768 AES blocks, using HE configuration of 128 bits security. We consider amortized latency instead of latency, because the input to modern tasks is often large, e.g., 602 KB of data are required for just one image sample from the commonly used ImageNet dataset. This means that we need to consider the decryption latency of 37,632 AES blocks and not just the latency of a single AES block.

The second type of implementations is summarized in Figure 2 and it includes Kreyvium [14], FLIP [53] and Elizabeth [25] that target the FHEW/TFHE [33], [20] HE schemes, LowMC [5], Rasta [28], Dasta [47], Pasta [31], Masta [44], Fasta [22], and Chaghri [6] that target the B/FV [36], [11] and BGV [12] schemes, and Hera [21] and Rubato [45] that target CKKS. However, none of these schemes are yet standardized, making them unsuitable for our case. Additionally, comparing their performance to the standardized scheme would be meaningless as it would not be an apples-to-apples comparison. We include them here for the sake of completeness. Another demonstration of why the thorough process of standardization is needed was recently demonstrated in [40] that present a key recovery attack on Rubato with some recommendations for parameter modifications.

FHEW/TFHE (Z_2)	BFV/BGV (Z_p/Z_2)		CKKS (R)
Kreyvium	LowMC	Pasta	Hera
FLIP	Rasta	Masta	Rubato
Elizabeth	Dasta	Fasta	
		Chaghri	

Figure 2: Non-standardized block cipher proposals. Each block cipher is described below the HE scheme that it targets. The order in which the block ciphers is listed is arbitrary.

Another type of implementation was conducted to assess the performance of HE on lightweight stream ciphers that participated in the NIST lightweight cryptography project [55] before the selection of Ascon as the finalist for standardization. For example, [8] reported the implementations of the stream ciphers Trivium [27], Kreyvium [14], and Grain-128a [41] under TFHE [20]. Additionally, it also implemented Grain128-AEAD [41] and thus initiated the process of studying AT constructions. Section VI provides further details on AT. Unfortunately, none of the above constructions was selected for standardization and thus the reported implementations cannot be used by those who require standardized cryptography. Moreover, due to the use of TFHE, no batching is possible, which results in latency in the orders of several dozen seconds per 64-bit block.

III. PRELIMINARIES AND NOTATION

We denote a sequence of x bits, where all bits are 0 by 0^x . Concatenation of two strings a and b is denoted by $a||b$. For a byte b we access its bits by b_i , where $b = \sum_{0 \leq i < 8} b_i \cdot 2^i$. Galois Fields (GF) of characteristic 2^a are denoted by $GF(2^a)$, e.g., $GF(2^8)$ for the AES state elements. Hexadecimal values are prefixed by $0x$, e.g., $0xe = 14$. The symbol \oplus and \odot denote the Boolean XOR and AND operations of two bits, bytes, or 64-bit words dependent on the context. We denote by $a := f()$ a deterministic assignment of $f()$ to a , and by $a \leftarrow f()$ probabilistic assignment. We denote by $x \ggg k$ the left rotation of the bits of x by k .

A. Homomorphic Encryption

We start by describing the high-level background and basic concepts of HE schemes. HE schemes allow us to perform operations on encrypted data [46]. Modern HE instantiations such as BGV [12], B/FV [36], [11], and CKKS [17] rely on the complexity of the Ring-LWE problem [52] for security and support single instruction multiple data (SIMD) operations. The HE system has an encryption operation $\text{HE.Enc} : \mathcal{R}_1 \rightarrow \mathcal{R}_2$ that encrypts input plaintext from the ring $\mathcal{R}_1(+, *)$ into ciphertexts in the ring $\mathcal{R}_2(\star, \cdot)$ and an associated decryption operation $\text{HE.Dec} : \mathcal{R}_2 \rightarrow \mathcal{R}_1$. An HE scheme is correct if for every valid input $x, y \in \mathcal{R}_1$

$$\text{HE.Dec}(\text{HE.Enc}(x)) = x \quad (1)$$

$$\text{HE.Dec}(\text{HE.Enc}(x) \star \text{HE.Enc}(y)) = x + y \quad (2)$$

$$\text{HE.Dec}(\text{HE.Enc}(x) \cdot \text{HE.Enc}(y)) = x \cdot y \quad (3)$$

and is approximately correct (as in CKKS) if for some small $\epsilon > 0$ that is determined by the key, it follows that $|x - \text{HE.Dec}(\text{HE.Enc}(x))| \leq \epsilon$. Equations 2, and 3 are modified in the same way. In this paper, we used CKKS for the experiments, as state-of-the-art deep neural network inference studies such as [50], [7] are based on CKKS. In CKKS, \mathcal{R}_1 is a vector space over the complex plane \mathbb{C}^n and \mathcal{R}_2 is the polynomial quotient ring over the integers $\mathbb{Z}[X]/(X^n - 1)$. We call every element in the plaintext vector a *slot*.

When designing an HE application, it is important to consider that certain operations incur higher computational costs than others. For instance, additions are significantly faster compared to multiplications of plaintexts by ciphertexts, which, in turn, are faster compared to ciphertext-ciphertext multiplications. The slowest operation in HE is known as the bootstrap operation. The bootstrap operation is required after a series of consecutive multiplications in order to refresh the state of the ciphertext, enabling further computations. In the CKKS scheme, on modern hardware, the bootstrap operation is several orders of magnitude slower compared to regular multiplications. Consequently, minimizing the need for bootstrapping is essential for efficient HE computations.

There are two primary methods to mitigate the need for bootstrapping: 1) reducing the multiplication depth of the evaluated circuit. By minimizing the number of sequential multiplications, the frequency of bootstrapping operations can be reduced; 2) Avoiding the wait until the last moment to perform a bootstrap operation. Instead, strategically identify locations in the computation where the number of ciphertexts in memory requiring bootstrap operations is minimal. This approach involves manual inspection and careful placement of the bootstrap operation to optimize efficiency. In this work, we adopted the latter approach. The decision regarding bootstrap placement is elaborated upon in the relevant sections.

To support binary inputs within the CKKS scheme, it is necessary to effectively handle binary inputs, we adopt the methodology proposed by BLEACH [32] and employ a cleanup utility after a specific number of Boolean gates. See an analysis of the error management in Section V.

B. Authenticated Encryption

Authenticated encryption with associated data (AEAD) is a cryptosystem that offers users both confidentiality and authenticity guarantees. Similar to block ciphers, AEAD schemes consist of three methods: AEAD.KeyGen, AEAD.Enc, and AEAD.Dec, which operate over various spaces. The key space is denoted as \mathcal{K} , the nonce space as \mathcal{N} , the plaintext and additional data space as $\{0, 1\}^*$, and the ciphertext space as \mathcal{C} .

The key generation method $k \leftarrow \text{AEAD.KeyGen}$ generates a new (pseudo)random symmetric key $k \in \mathcal{K}$. The encryption function $(c, t) \leftarrow \text{AEAD.Enc}_k(a, n, m)$ receives a plaintext message $m \in \{0, 1\}^*$ a nonce $n \in \mathcal{N}$, some authentication data $a \in \{0, 1\}^*$ and the key k . It outputs a ciphertext $c \in \mathcal{C}$ in addition to an authentication tag t over the pair (a, m) . The decryption method $\{m, \perp\} = \text{AEAD.Dec}_k(a, n, c)$ receives a ciphertext $c \in \mathcal{C}$ a nonce $n \in \mathcal{N}$ and some authentication data $a \in \{0, 1\}^*$ and an authentication tag t . If the verification of the authentication tag succeed, it returns the decryption of $c(m)$, otherwise, it returns \perp .

IV. ADVANCED ENCRYPTION STANDARD (AES)

This chapter starts by briefly describing the AES block cipher. Subsequently, using this info we describe our implementation of AES-CTR under decryption.

A. The AES block cipher

AES was officially standardized by NIST in 2001 [54] and has since become widely accepted and the most commonly used block cipher in modern cryptographic systems and applications. Its importance is exemplified by the rapid growth of AES-encrypted online data, which is strongly supported by industry players like IBM in its Z systems [48], and Intel, who have introduced AES-NI processor instructions [43], [42] to enhance AES performance.

We briefly describe the AES encryption algorithm, which we illustrate in Fig. 3, the decryption procedure is described in [54]. AES encryption operates on a plaintext block consisting of 128 bits and a key that can be either 128, 192, or 256 bits in size. The encryption process generates a ciphertext block of 128 bits. The key undergoes an expansion process, resulting in the creation of 10, 12, or 14 round keys, depending on the key size. To begin the encryption, the plaintext block is XOR-ed with the first 128 bits of the key, which serves as a whitening step. The resulting value then undergoes a series of 39, 47, or 55 consecutive transformations. These transformations can be organized into 9, 11, or 13 identical AES rounds, respectively, followed by an additional final round. The j th AES round, $j=1, \dots, 9/11/13$, is the sequence of transformations

$$\text{MixColumns}(\text{ShiftRows}(\text{SubBytes}(S))) \oplus \text{RoundKey}[j]$$

operating on the 128-bits state S , where $\text{RoundKey}[j]$ is the j th round key. The last round $j = 10/12/14$ is the sequence

$$\text{ShiftRows}(\text{SubBytes}(S)) \oplus \text{RoundKey}[j]$$

Encryption modes. Various cryptographic modes of operation can be utilized with the AES algorithm, including ECB, cipher block chaining (CBC), counter (CTR), and GCM. In our implementation, we chose to implement the CTR and GCM modes. The selection of these modes is driven by the fact that ECB is deemed insecure due to its vulnerability to certain attacks, thus making it unsuitable for our purposes. In addition, the CBC mode poses challenges in terms of parallelization, limiting its efficiency in certain scenarios, especially when considering HE, which may already introduce some slowdowns. In contrast, AES-GCM, which extends AES-CTR with authentication capabilities, has been designated as the preferred AES mode of operation in security protocols such as TLS 1.3 [57].

In AES-CTR mode, each plaintext block is XORed with the output of an AES encryption operation using a 32-bit nonce (n) concatenated with a 96-bit counter (c), with a specified key (k). It is crucial for the security and confidentiality of AES that the concatenated 128-bit value ($n||c$) remains unique for a given key (k). A notable advantage of AES-CTR is its ability to encrypt multiple plaintext blocks in parallel, making it highly efficient for processing large volumes of data simultaneously. Leveraging this inherent parallelizability, our implementation takes advantage of the AES-CTR mode to enhance the decryption performance and the overall efficiency.

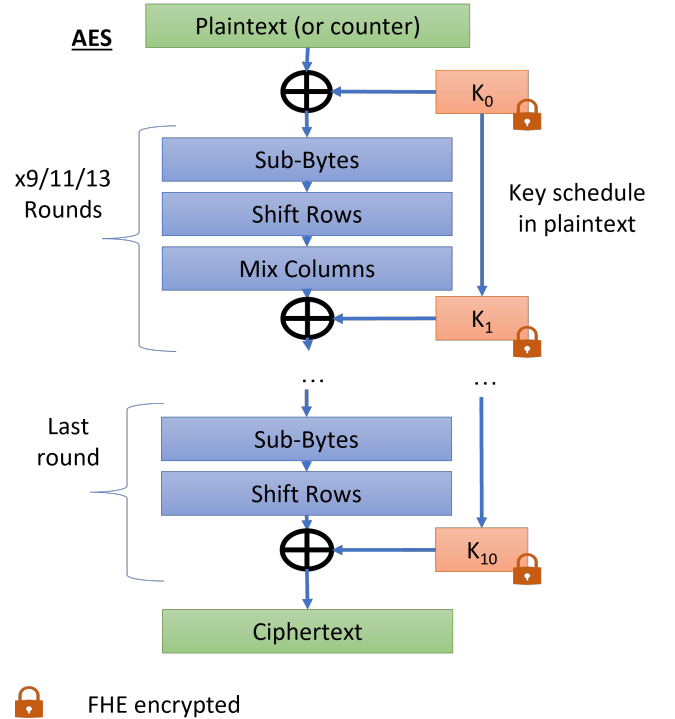


Figure 3: An illustration of an AES-128/192/256 block cipher. When considering hybrid encryption, the key is encrypted using HE.

Remark 1. The generation of AES keys, their subsequent cleartext expansion, and the encryption of all keys are conducted offline by the client only once. The procedure itself is straightforward, assuming consistent data packing as outlined below. For the sake of brevity, we omit the detailed description of this process.

B. Implementing AES-CTR over HE

Our implementation follows a bit-sliced approach, where each HE plaintext slot is treated individually per AES block, leveraging the HE SIMD capabilities. Because every AES block consists of 128 bits, this implementation requires 128 ciphertexts (or in tile tensor shape notation, $[128, \frac{s}{s}]$, where s is the number of slots. See [2]). In the case where each ciphertext occupies approximately 12 MB, calculated as 65,536 coefficients multiplied by a multiplication depth of ~ 12 , multiplied by 2 polynomials, and further multiplied by 8 bytes per coefficient, the overall state size amounts to $\sim 12 \text{ MB} \times 128 = 1.5 \text{ GB}$.

The advantage of employing this method is the elimination of rotational operations entirely. However, a trade-off arises in the form of the requirement for users to decrypt 512 KB at a time, derived from the multiplication of 128-bit blocks by 32K slots per ciphertext. Nonetheless, this limitation becomes negligible when the primary use case involves handling substantial amounts of data, such as in the scale of megabytes (MB), gigabytes (GB), or even petabytes (PT).

During our design process, we considered various alternative approaches. One such approach involves utilizing the CKKS

scheme to operate on bytes instead of individual bits through techniques like the BLEACH cleanup method [32]. However, CKKS currently lacks support for performing Boolean-XOR operations directly on these bytes. Instead, it requires the decomposition of numbers into bits before applying the XOR operation, and the subsequent reconstruction of bits into bytes. While this approach may potentially reduce the number of ciphertexts and thereby improved cache and memory utilization, the associated costs of decomposing and reconstructing bytes, as well as managing the S-Box look-up table, would have been considerably higher for the same number of AES blocks. As a result, we chose not to use this approach, and preferred the above bit-sliced implementation.

Another approach involved consolidating all the bits of the AES blocks into a single ciphertext, placing them adjacent to each other. In this approach, the MixColumns and ShiftRows stages would require numerous rotations, while the AddKey and SubBytes operations would still involve the same amount of computation as in the bit-sliced approach. Similar to the byte-sliced approach, this approach offers the advantage of reducing the number of ciphertexts and imposing a lower limit on the number of blocks that need to be decrypted at a time. However, as mentioned earlier, this limit is generally not a concern when operating with large volumes of data in cloud environments. This approach was taken for example in [18].

Next, we describe our implementation of the four AES methods: AddKey, MixColumns, ShiftRows, and SubBytes.

AddKey Operation. The AddKey operation, within the context of CKKS, is realized through a straightforward XOR operation. In CKKS, this XOR operation is implemented using the equation $x \oplus y = (x - y)^2$, where x and y represent individual bits, which in turn are represented by values within the range $[0 \pm \epsilon, 1 \pm \epsilon]$, where ϵ denotes an extremely small value, see Section V. For efficient parallelization, the XOR operation is performed in parallel for all the 128 ciphertexts of the AES state, ensuring efficient and simultaneous processing. As mentioned in Remark 1, the keys must also be packed using the bit-sliced approach, which means that the same key is duplicated over all slots of the 128-ciphertexts (in tile tensor shape notation, $[128, \frac{*}{s}]$, where s is the number of slots).

Remark 2. *It is feasible to “share” HE ciphertexts for multiple AES decryptions, employing distinct keys. This can be achieved by either the clients broadcasting the respective keys to the corresponding HE ciphertext slots beforehand or by requesting the server to select the pertinent keys per slot using application masks. These masks consist of binary values, with a value of 1 in the relevant positions and 0 elsewhere. They are multiplied by the associated HE ciphertexts that encrypt the corresponding AES keys and summed together.*

ShiftRows Operation. Using the bit-sliced representation, the ShiftRows operation is achieved without any additional computational cost. In this representation, the operation simply involves replacing the ciphertext location. More precisely, it is implemented by permuting the pointers to the corresponding ciphertexts.

MixColumns Operation. One reason that we preferred implementing AES-CTR over other alternatives, e.g., AES-CBC,

is that its decryption process involves only AES encryption operations. This is especially critical when considering the MixColumns Step. If we consider the AES state as a 4×4 matrix of elements in $GF(2^8)$ multiplied modulo the polynomial $x^4 + 1$ then the output of the MixColumns operation (in encryption) on every column input $[b_0, b_1, b_2, b_3]^T$ is

$$\begin{aligned} D0 &= x \cdot b_0 + (x + 1) \cdot b_1 + b_2 + b_3 \\ D1 &= b_0 + x \cdot b_1 + (x + 1) \cdot b_2 + b_3 \\ D2 &= b_0 + b_1 + x \cdot b_2 + (x + 1) \cdot b_3 \\ D3 &= (x + 1) \cdot b_0 + b_1 + b_2 + x \cdot b_3 \end{aligned}$$

As described in [37], these equations can be simplified to the following:

$$\begin{aligned} D0 &= x \cdot (b_0 + b_1) + b_1 + b_2 + b_3 \\ D1 &= x \cdot (b_1 + b_2) + b_2 + b_3 + b_0 \\ D2 &= x \cdot (b_2 + b_3) + b_3 + b_0 + b_1 \\ D3 &= x \cdot (b_3 + b_0) + b_0 + b_1 + b_2 \end{aligned}$$

Here, $+$ translates in $GF(2^8)$ to the XOR operation and multiplication by x of a value $a \in GF(2^8)$ is done using the equation

$$\begin{aligned} &(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) = \\ &(a_6, a_5, a_4, a_3 \oplus a_7, a_2 \oplus a_7, a_1, a_0 \oplus a_7, a_7) \end{aligned}$$

These simplified equations primarily involve repeated XOR operations. In contrast, during the AES decryption, which is used by AES-CBC decryption, the InvMixColumns operation is performed using the following equations:

$$\begin{aligned} D0 &= (x^3 + x^2 + x) \cdot b_0 + (x^3 + x + 1) \cdot b_1 + \\ &(x^3 + x^2 + 1) \cdot b_2 + (x^3 + 1) \cdot b_3 \\ D1 &= (x^3 + 1) \cdot b_0 + (x^3 + x^2 + x) \cdot b_1 + \\ &(x^3 + x + 1) \cdot b_2 + (x^3 + x^2 + 1) \cdot b_3 \\ D2 &= (x^3 + x^2 + 1) \cdot b_0 + (x^3 + 1) \cdot b_1 + \\ &(x^3 + x^2 + x) \cdot b_2 + (x^3 + x + 1) \cdot b_3 \\ D3 &= (x^3 + x + 1) \cdot b_0 + (x^3 + x^2 + 1) \cdot b_1 + \\ &(x^3 + 1) \cdot b_2 + (x^3 + x^2 + x) \cdot b_3 \end{aligned}$$

These equations involve multiple serial multiplications, which leads to an increase in the circuit’s multiplication depth when executed under HE. Hence, evaluating MixColumns is significantly faster compared to InvMixColumns.

SubBytes Operation. The AES S-box involves an affine transformation on the inverse of the input in $GF(2^8)$. However, computing the inverse efficiently is not an easy task. Extensive research has been dedicated to achieve this task in various contexts, such as hardware implementation and secure multi party computation (MPC) protocols. Notable studies include [9], [10], [56], [13]. One prominent approach involves transforming the AES Galois field data to a tower (composite) field with a minimized number of gates. For instance, in [10], a circuit was achieved using only 34 AND gates and a multiplication depth of 4, while [9] presented a circuit with 32 AND gates and a multiplication depth of 6.

However, a drawback of prior-art designs is their assumption that XOR gates are computationally free. Consequently, they

propose minimization functions that primarily aim to reduce the number of AND gates. While this assumption holds true in hardware implementations, MPC protocols, and some HE schemes such as BGV or BF/V, it does not hold for the CKKS scheme. In CKKS, both XOR and AND gates require one multiplication operation, thereby increasing the overall multiplication depth of the circuit.

Our implementation utilizes the lookup table approach, commonly employed in hardware systems. Usually these hardware implementations are vulnerable to memory access attacks. However, in our case the nature of HE imposes oblivious computations, thereby eliminating this drawback. For AES, we employ a lookup table consisting of 256 entries, where each entry represents a unique 8-bit value expressed in plaintext bits.

To compute the inverse function, we begin by calculating the indicator mask for each table cell by comparing the cell index with the input value. We leverage the following observations: 1) when an output bit is 0, we can disregard the indicator ciphertext entirely, and 2) when a bit is 1, we can utilize the indicator ciphertext, particularly during the summation process involved in collecting all the indicators of all cells to get the final output. Computing the indicators requires 272 multiplications with a multiplication depth of 3. However, the second part of selecting the value from the table is computationally “free” in the context of HE. Appendix A describes another approach for computing lookup tables with finite range under encryption. This approach achieves less multiplication but it is less generic so that it introduces additional code overhead. Thus, we decided to avoid it in our implementation and to report it only for the completeness of the paper.

Overall, the multiplication depth associated with each round in our implementation is 9 as follows: AddKey: 1, MixColumns: 3, ShiftRows: 0, SubBytes: 3, and the cleanup function h_1 (Section V): 2. For AES128/192/256, which require 9, 11, and 13 rounds respectively, as well as an additional final round, which does not include the MixColumns operation, the total multiplication depth is calculated as 87, 105, and 123 respectively.

Bootstrap Policy. As part of our implementation, at every round, we incorporated a bootstrap operation after SubBytes following by a cleanup utility. The bootstrap operation is executed independently on each of the 128 ciphertexts, and hence can be parallelized, dependent on the capabilities of the hardware being utilized. The above bootstrap policy fits nicely when the maximal multiplication depth is 12 and a bootstrap is needed when a ciphertext reaches chain index 3.

V. BINARY CIRCUITS OVER CKKS

To achieve high throughput we decided to leverage the approximated HE scheme CKKS [17]. Moreover, CKKS is the leading HE scheme when considering state-of-the-art inference applications, for example, [50], [7]. For that, we leveraged a recent technique called BLEACH [32], which has demonstrated the practicality of executing binary circuits over CKKS. Specifically, it showed (Lemma 1) that performing XOR (\oplus), AND (\wedge), or OR (\vee) operations on two encrypted bits, followed by the cleanup function $h_1(x) = -2x^3 + 3x^2$ [19], does not introduce any significant increase in the ciphertext

error. This allows us to efficiently execute these operations while maintaining the desired level of accuracy in the ciphertext.

Lemma 1 ([32] Lemma 3). *Let $x = b_x + e_x$ and $y = b_y + e_y$ be input to a binary operation, $b_x, b_y \in \{0, 1\}$ and $|e_x|, |e_y| < e \leq 0.001$, and the error added when multiplying and rescaling two ciphertexts is e_{ckks} such that $2.1e_{ckks} < 0.5e$. Then $z = b_z + e_z$, where $b_z \in \{0, 1\}$ and $|e_z| < e$ for $z = h_1(x \wedge y)$ or $h_1(x \vee y)$ or $h_1(x \oplus y)$*

To avoid bleaching after every boolean gate, we extend this lemma and show that it is enough to perform a cleanup operation after every several steps that depend on the scheme parameters e.g., the fractional part accuracy. We start by reminding:

Lemma 2 ([32][Lemma 2]). *For $x, y \in [0 \pm \epsilon, 1 \pm \epsilon]$, i.e., $x = b_x + e_x$ and $y = b_y + e_y$, where $b_x, b_y \in \{0, 1\}$ and $|e_x|, |e_y| < e < 0.25$. Then*

$$\begin{aligned} |(x \wedge y) - (b_x \wedge b_y)| &< 5e, \\ |(x \vee y) - (b_x \vee b_y)| &< 5e, \\ |(x \oplus y) - (b_x \oplus b_y)| &< 2.25e. \end{aligned}$$

Using this gate error bounds (5,5,2.25) we state the following lemma.

Lemma 3. *Let $0 < e < 1$ be the bound on the initial error in the scheme, let B be the gate error bound, and let f be a Boolean circuit with multiplication depth d and input values $x_i = b_{x_i} + e_{x_i}$, $1 \leq i \leq n$, $b_{x_i} \in \{0, 1\}$ and $|e_{x_i}| < e$. If the error added when multiplying and rescaling two ciphertexts is e_{ckks} such that $e_{ckks} < 0.25e$. Then $z = h_1(f(x_1, \dots, x_n)) = b_z + e_z$, where $b_z \in \{0, 1\}$ and*

$$|e_z| < 3 \cdot (B + 0.25)^{2d} \cdot e^2 + 2 \cdot (B + 0.25)^{3d} \cdot e^3$$

Proof: Consider the expression $w = b_w + e_w = f(x_1, \dots, x_n)$, where $b_w \in \{0, 1\}$ represents the result obtained by applying the function f to binary inputs. Assuming that the error incurred when applying a gate is $B + e_{ckks} < (B + 1)e$, we can establish a bound on the final error e_w as $e_w < (B + 0.25)^d \cdot e$. When applying the cleanup utility h_1 , the resulting value z is:

$$\begin{aligned} z &= h_1(w) = h_1(b_w + e_w) \\ &= -2(b_w + e_w)^3 + 3(b_w + e_w)^2 \\ &= -2b_w^3 - 6b_w^2e_w + 3b_w^2 - 6b_we_w^2 + 6b_we_w - 2e_w^3 + 3e_w^2 \\ &= \begin{cases} 3e_w^2 - 2e_w^3 & b_w = 0 \\ 1 - 3e_w^2 - 2e_w^3 & b_w = 1 \end{cases} \end{aligned}$$

and

$$\begin{aligned} |e_z| &= |z - b_w| < |3(B + 0.25)^{2d}e^2 \pm 2(B + 0.25)^{3d}e^3| \\ &< 3(B + 0.25)^{2d} \cdot e^2 + 2(B + 0.25)^{3d} \cdot e^3 \end{aligned}$$

We can now use the lemma to find the largest d for which $e_z < e$. This will allow stability of the evaluation process. While this can be solved analytically, the results are not displayed nicely, and instead we chose to use a SageMath script to plot

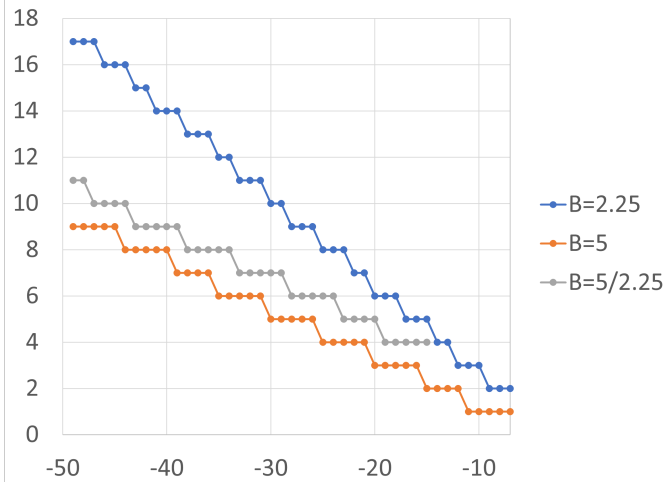


Figure 4: Maximum d (y-axis) as a function of $\log_2(e)$ (x-axis). Three functions (f) are considered: f with AND gates only ($B = 5$), f with XOR gates only ($B = 2.25$), and f with XOR gates except for the last three levels ($B = 5/2.5$).

Figure 4. The graph illustrates the relationship between the logarithm of the error bound ($\log_2(e)$) on the x-axis and the corresponding maximum value of d allowed before invoking the $h_1()$ function. Three different functions, are considered: a function that solely performs AND gates, a function that only performs XOR gates, and a function that primarily performs XOR gates, except for the last three multiplication levels, where it incorporates AND gates. The last function is the one we have in our AES-CTR implementation. As can be seen, calling h_1 only at the end of every AES round (which has a multiplication depth of 7) is possible when the initial error satisfies $e < 2^{-29}$. In our experiments, we chose a scale of 2^{42} and since the initial HE noise is only a few bits it guarantees that our initial error meets the requirement.

Remark 3. *The analysis provided in this section relies on the worst case scenario. In practice, it is possible to derive bounds that depend on the error expectation and are achieved with some probability. We leave this research to future work.*

VI. AUTHENTICATED TRANSCIPHERING (AT)

This section informally define the notion of AT. This definition will be presented in a step-by-step manner, where each step describes an expanded threat model encompassing additional capabilities.

In the basic HE scenario, there exists a user and an untrusted *semi-honest* environment, such as the cloud. The user generates the HE secret, public, and evaluation keys. The secret key is securely stored in a private location, while the user publishes the public and evaluation keys to the cloud. To utilize the cloud HE service, the user encrypts data using either the HE secret key or the public key and uploads the encrypted data to the cloud. Subsequently, the cloud performs operations on the data, such as running a neural network inference, and returns the encrypted results to the user. The user can then decrypt and view the results using their private key.

Modern HE schemes are designed to be either IND-CPA secure (e.g., BGV [12] or B/FV [36], [11]) or IND-CPA^D secure [51] like CKKS. In either case, these schemes offer semantic security to the uploaded ciphertexts, meaning that the cloud gains no knowledge about the user’s data solely by observing the ciphertexts.

In the context of hybrid encryption, we expand upon the aforementioned scenario by introducing a partition in the cloud infrastructure, dividing it into two distinct entities with varying capabilities. Specifically, we consider a semi-honest HE service that adheres to the established protocol, while characterizing the remaining components of the cloud as malicious. In particular, we identify the database that stores AES ciphertexts of the client as a malicious entity.

In this setup, a user initiates a request to the HE service, specifying a list of keys to be utilized for accessing data from the database. The HE service, in turn, communicates with the database to acquire the ciphertexts associated with the provided keys. To ensure the integrity of the ciphertexts, certain assumptions are made. Specifically, it is assumed that the user has encrypted the data using an AEAD scheme, and the keys form a part of the additional authentication data (AAD) associated with the ciphertext. This enables the HE service to authenticate the data on behalf of the user. We refer to the combination of AEAD and HE construction as AT. The concept of AT was also explored in [8] with the Grain128-AEAD implementation. Below we provide further discussion on AT that leads to our near-standardized implementations in Section VII and Section VIII.

The fundamental concept underlying AT is to ensure that the authentication tag propagates seamlessly from the AEAD ciphertexts to the HE decryption process, where decryption failure occurs if the original AEAD tag check would have failed. This objective can be accomplished through two distinct approaches. The first approach involves transmitting both the consumed tags by the HE service and the tags generated during the AEAD decryption under HE process to the user. Alternatively, the second approach utilizes a single bit sent (encrypted) from the server to the client to indicate the validity of the returned results. In the first option, the client is responsible for comparing the two lists of tags and releasing the HE decrypted results only if the lists are identical. This places some computational burden on the client. Conversely, in the second option, only one bit is sent, which saves bandwidth and computation to the client but increases the overhead on the server side.

In the context of AT, there are two crucial aspects that deserve attention. First, it is imperative to ensure the confidentiality of the AES key encrypted under HE from any potential adversary. Even though the key is encrypted under HE, if an adversary gains access to this key, they can encrypt their own ciphertexts, thereby compromising the authenticity guarantees of the scheme. This concern does not apply to the HE service itself since we assume it to be semi-honest. Moreover, regardless of the situation, the HE service can always provide a bit of choice to the client, thereby indicating whether the returned ciphertext is valid or not. Note also that revealing the encrypted key to an adversary does not harm privacy of the AT scheme because the adversary still does not hold the HE secret key and thus cannot decrypt HE ciphertexts.

Alternatives to AT. There exist alternatives to the aforementioned construction, such as employing asymmetric encryption instead of symmetric encryption in conjunction with the HE scheme. However, this alternative solution is less practical compared to using AEAD, primarily due to the prevalence of AEAD usage among users in current systems. Adopting asymmetric encryption would require significant modifications in software or, in some cases, even hardware, to encrypt or reencrypt all existing data under the asymmetric scheme. Additionally, the expansion rate of data would no longer remain at a 1:1 ratio, as with AEAD, which deviates from the goal of compression that was initially pursued.

Another option, which faces similar challenges involves requesting the user to sign each symmetric or AEAD ciphertext. While this approach enables the HE service to efficiently validate the authenticity of the data (in plaintext), it suffers from the same practical issues as the previous alternative. Furthermore, the existing standardized signature schemes are either not post-quantum secure or require significant space, rendering them unsuitable for integration into IoT devices. Considering these factors, it becomes evident that the use of AEAD within the AT scheme presents a more practical and efficient solution.

Verifiable authenticated transciphering (VAT). Once we established what an AT is, we need also to say what guarantees it does not provide. HE schemes are susceptible to malleability issues, which allows malicious entities to manipulate the ciphertext data. For example, operations like subtracting a ciphertext from itself or multiplying it by a plaintext value are possible without informing the original data owner. While there are methods available to protect the integrity of HE ciphertexts, such as using verifiable computation (VC) or trusted execution environments (TEEs) like Intel[®] SGX [49] or ARM[®] Trustzone [24], these approaches are still considered impractical, and the latter requires involving third-party entities in the user trusted computing base (TCB), e.g, Intel. As a result, most prior works have assumed a *semi-honest* cloud that faithfully executes computations on the encrypted ciphertexts without deviation.

In the context of AT, we also make the semi-honest assumption **on the HE service**. As a result, the authenticity guarantees provided by AT pertain solely to the inputs obtained from external storage or other services, rather than ensuring the integrity of the computations performed by the HE service itself. A compelling area for further research lies in the combination of VC techniques with AT, which can yield intriguing possibilities. We propose the term *verifiable authenticated transciphering (VAT)* as a potential name for this novel approach.

Remark 4. *AT is not limited to one client or one client key per HE computation. The client can ask the HE service to collect data that was encrypted using multiple AES keys that may belong to different users. As long as the server holds the required keys encrypted under HE, it can combine them in the evaluation process.*

Remark 5. *As a desirable practice, it is preferable for the HE service to promptly delete the content of any unauthenticated decrypted data as soon as it becomes aware of its authenticity status, even when under HE. By doing so, the server minimizes the potential risk posed by attackers who may capture ciphertexts containing potentially maliciously crafted data.*

We start with some background on AES-GCM and then continue by describing our implementation.

A. Background

The Galois / counter mode (GCM) [34] is a mode of operation specifically developed for symmetric block ciphers, such as AES. Unlike other modes like CTR, ECB, and CBC, which primarily aim for confidentiality, GCM is classified as an AEAD scheme. As such, it provides guarantees for both confidentiality and integrity. This is accomplished through the combination of the AES-CTR mode with a GHASH function, which ensures the authenticity of the data being processed.

Presently, AES-GCM has gained widespread adoption due to its high throughput rates on modern processors. It is among the few allowed ciphers when using TLS 1.3 [57] and is highly recommended by prominent companies libraries like AWS encryption SDK [1]. Additionally, in terms of ciphertext expansion rate, AES-GCM incurs a minimal overhead of only an additional 128-bit tag compared to AES-CTR ciphertexts. Figure 5 illustrates the AES-GCM scheme, where it highlights the parts to be eventually encrypted under HE.

The GHASH function is defined over the Galois field $\mathbb{F}_{GCM} = GF(2^{128})$ with a polynomial reduction $x^{128} + x^7 + x^2 + x + 1$. To generate the authentication tag the ciphertext blocks are XORed and multiplied by an encrypted value $H = AES_k(0^{128})$ in \mathbb{F}_{GCM} .

An illustration of the AES-GCM AEAD scheme within the context of HE is presented in Figure 5. The figure provides a visual representation of the components that are encrypted with AEAD, encrypted with HE, or remain in plaintext. It is important to note that at the end of the process, both the ciphertexts and the authentication tag are preserved in an encrypted form under HE. Moreover, because $H = AES_k(0^{128})$ is encrypted, the entire tag computation must be done under HE.

B. An implementation of AES-GCM

Our implementation of the AES-CTR mode is discussed in detail in Section IV-B. This implementation serves as the foundation for our AES-GCM implementation, as well as an additional implementation of the GHASH function under HE.

The code presented in Figure 6 provides an overview of our GHASH implementation. It is implemented using SageMath with Numpy and incorporates a basic GF-mul algorithm. This code is later adapted to operate on real HE ciphertexts. The multiplication function takes two elements from $GF(2^{128})$ as input, where ct represents a ciphertext and pt represents a plaintext. The function computes the product $ct \cdot pt$ within $GF(2^{128})$, while also accommodating the CKKS scheme by replacing XOR operations with $(x - y)^2$ operations. Furthermore, it assumes that each bit in the first axis of the array corresponds to a distinct ciphertext, enabling the HE rotate operation (*np.roll*) to be executed without incurring additional computational cost.

HTBL. Consider the AAD data $A = a_1, \dots, a_m$ and ciphertext data $C = c_1, \dots, c_n$ as elements from $GF(2^{128})$ on which we

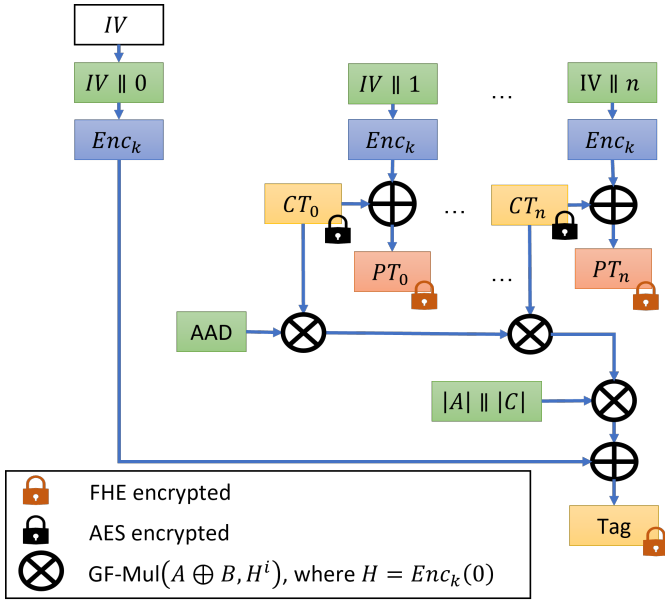


Figure 5: An illustration of the AES-GCM AEAD scheme within the context of HE. Green blocks represent plaintext blocks, blue blocks represent AES-GCM encryption methods, yellow blocks represent AES-GCM encrypted blocks, and red blocks represent the AES-GCM decrypted plaintext that remains encrypted under HE.

```

def gf_mul(ct, pt):
    z = np.zeros(pt.shape, dtype=np.int32)
    v = ct.copy()
    for i in range(128):
        z = np.power((z - (pt[i]*v)), 2)
        c = v[127,:]
        v = np.roll(v, 1, axis=0)
        v[0] = c
        v[1] = (c - v[1])**2
        v[2] = (c - v[2])**2
        v[7] = (c - v[7])**2
    return z

```

Figure 6: An illustration of our GHASH implementation using SageMath and Numpy.

apply the function $\text{GHASH}(A, C, H)$, defined as:

$$\text{GHASH}(A, C, H) = \sum_{i=1}^m a_i \cdot H^i + \sum_{i=1}^n c_i \cdot H^{m+i}$$

Figure 5 presents an alternative approach to compute the GHASH tag, utilizing Horner’s rule, which states that

$$\sum_{i=1}^n x_i \cdot H^i = (x_1 \cdot H) \oplus x_2 \cdot H \dots \oplus x_n \cdot H$$

This technique is commonly employed to avoid the expensive computation of powers of H . However, due to the SIMD nature of HE, we adopt a different commonly used strategy by precomputing a table called $HTBL$ that stores the s powers of

H , where s represents the number of slots in the HE ciphertext. Note that even though a new nonce or IV are required per ciphertext, the $HTBL$ is the same for all ciphertexts under the same AES-GCM key. This means that a user can precompute the $HTBL$ once, maybe at an offline stage, and use in many different occasions in the online phase.

There are two options for computing $HTBL$, either the client precomputes it and sends it encrypted under HE to the server, or the client encrypts only H , and the server computes all the relevant powers of H . This computation requires $\log_2 s$ GF multiplications. Precomputing the data on the client side offers the advantage of faster computations in plaintext, and in any case, the bandwidth remains the same as at least one HE ciphertext needs to be transmitted from the client to the server. However, this approach places an additional burden on the client, which sometimes needs to be avoided. Another option is to combine the two approaches sending only partial $HTBL$ and complete it if needed on the server.

The size of the $HTBL$ is similar to the size of an AES ciphertext under HE encryption, i.e., $\sim 12 \text{ MB} \times 128 = 1.536 \text{ GB}$. If the number of AES blocks to be processed under the same key is more than 32,768, i.e., it fits in more than one ciphertext, one can either use the Horner rule, or precompute the power of H also for the extra slots.

VIII. ASCON

A. Background

Ascon [30] stands as an alternative to AES-GCM in the presence of lightweight and low-end devices. Recently, it was selected by NIST for standardization [55]. Additionally, Ascon emerged as the top choice for AE in the CAESAR competition [29]. What makes Ascon particularly appealing is its ease of implementation in both software and hardware. With a compact state size of 320 bits (comprised of five 64-bit words), Ascon can benefit from parallelization through SIMD operations. Consequently, it exhibits compatibility not only with large-end CPUs but also with HE. Another advantageous aspect of Ascon is its avoidance of look-up tables, an original motivation stemming from the need to ensure constant-time implementations that avoid timing-based information leaks. This property also aligns with our implementation, which uses CKKS that does not provide native support for look-up tables.

The encryption process of Ascon involves iteratively applying a round transformation based on the substitute permutation network (SPN) to the Ascon state. This state is composed of five 64-bit words, denoted as x_0, \dots, x_5 , resulting in a total of 320 bits. The Ascon encryption process involves four distinct phases: an initial phase comprising 12 permutation rounds to establish the ciphertext state, a final phase consisting of an additional 12 rounds to complete the encryption process.

In between, the encryption of plaintext blocks, Ascon128 and Ascon128a utilize 6 rounds to process blocks of size 64-bit and 128-bit, respectively, for the AAD and ciphertext data. Each round encompasses three essential steps: the addition of round constants, a non-linear substitution layer (depicted in Figure 7), and a linear diffusion layer described by equations 4 to 8. This systematic approach ensures the secure transformation of data during encryption.

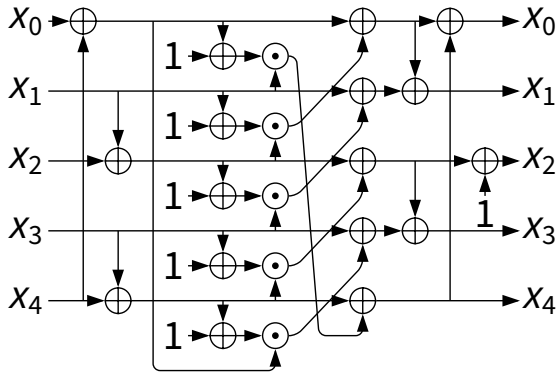


Figure 7: Schematic representation of the Ascon s-box, image was taken from [30]. x_0, \dots, x_4 are 64-bit word elements.

$$x_0 := x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \quad (4)$$

$$x_1 := x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \quad (5)$$

$$x_2 := x_2 \oplus (x_2 \ggg 01) \oplus (x_2 \ggg 06) \quad (6)$$

$$x_3 := x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \quad (7)$$

$$x_4 := x_4 \oplus (x_4 \ggg 07) \oplus (x_4 \ggg 41) \quad (8)$$

B. Implementing Ascon

For the purpose of Ascon decryption within the context of HE, we made a decision to employ a 64-bit word sliced implementation instead of a bit-sliced implementation as we did for AES-CTR. This choice was motivated by the fact that the 320-bit state of Ascon would require the utilization of 320 ciphertexts, resulting in a total size of approximately $12 \text{ MB} \times 320 \approx 3.84 \text{ GB}$, which was less practical. Instead, a strategy was adopted wherein only five ciphertexts were employed, with a total size of approximately 60 MB. This configuration allowed for the parallel decryption of a batch consisting of $32,768/64 = 512$ Ascon blocks in parallel.

However, unlike AES-CTR/GCM, where parallel operations can be performed on different blocks of the same ciphertext, the adapting state of Ascon necessitated the decryption of blocks from different ciphertexts. These blocks either employed different keys or different nonces. Similar to the AES-CTR implementation, the placement of Ascon keys within the relevant slots in the HE ciphertexts can either be done directly by the clients or using masks on the server side. In summary, the advantage of Ascon lies in its relatively small number of ciphertexts, while the limitation lies in the requirement to operate on orthogonal Ascon blocks during HE-based decryption.

Overall, the multiplication depth associated with each Ascon round in our implementation is 9 as follows: The addition of round constants: 1 XOR; the non-linear substitution layer: 4 (3 XORs and 1 AND); the linear diffusion layer: 2 XORs; and the cleanup function $h_1(\cdot)$: 2. The total multiplication depth is therefore $(12 + 12 + 6 * m) * 9 = 216 + 54m$, where m is the number of AAD and ciphertext blocks. The number of bootstraps is $(24 + 6m) * 5 = 120 + 30m$ due to the 5 ciphertexts that hold the Ascon state.

Table II: A comparison of decryption methods under HE

Cipher	Process unit	Size (KB)	Latency (min)	Amortized latency mSec/block	Peak Memory (GB)
CTR	CPU	512	31	56.7	127.62
CTR	GPU	512	4	7.3	60.10
GCM	GPU	512	11.4	21	65.10
ASCON	CPU	4	21	2,460	50.10
ASCON	GPU	4	0.55	64.5	45.10
ASCON	GPU	512	14	12.8	49.10

IX. EXPERIMENTS

A. Experimental setup

For the experiments, we considered two platforms

- 1) CPU: An Intel® Xeon® CPU E5-2699 v4 @ 2.20GHz machine with 44 cores (88 threads) and 750GB memory.
- 2) GPU: A100 SXM4 80 GB GPU, on a server with an AMD® EPYC 7763 64-Core Processor 2.45GHz machine with 64 cores (128 threads). Used single CPU thread by setting `OMP_NUM_THREADS=1`.

The experiments were conducted using HElayers [2], a software development kit (SDK) for privacy-preserving computations that offers various programming capabilities for developers working with HE. Each experiment was repeated 10 times, and the reported result represents the minimum measured running time. In our experiments, we configured HElayers to utilize a bootstrappable HEaAN context (using the CryptoLab HEaAN library) with a security level of 128 bits. The ciphertexts employed had a capacity of 32,768 slots, allowing for parallel processing of multiple data elements. We set the multiplication depth to 12, furthermore, the fractional part precision (scale) was set to 42 bits, while the integer part precision (number of additional bits in the first prime) was set to 18 bits. In addition, the chain index after bootstrap is 12 and the minimal chain index for bootstrap is 3.

B. Experiments results

Table II presents¹ the benchmark results for the decryption process of various block ciphers in different modes of operation. The chosen data sizes, specifically 512KB for AES-CTR/GCM and 4KB for Ascon, correspond to the number of blocks required to fill the ciphertexts representing the block cipher states: 128 ciphertexts for AES-CTR/GCM and 5 ciphertexts for Ascon. These sizes were selected to maximize the utilization of our implementation, as lower values would leave unused slots in the ciphertexts and result in under-utilization. It is important to note that the decryption time will double if the data size is doubled, as our implementation currently utilizes only one thread on the GPU device. For a fair comparison between our AES-GCM and Ascon implementations we also include the runtime of decrypting 512 KB using Ascon, where we increased the number of blocks in the original 512 ciphertext. Figure 8 illustrates how the amortized latency is reduced when increasing the ciphertexts size. The reason is that cost of the constant overhead of the initialization and finalization steps, which include 12 permutation rounds each becomes negligible with the ciphertext size. Particularly, one permutation round

¹The AES-CTR/GCM results were also presented in a poster [3].

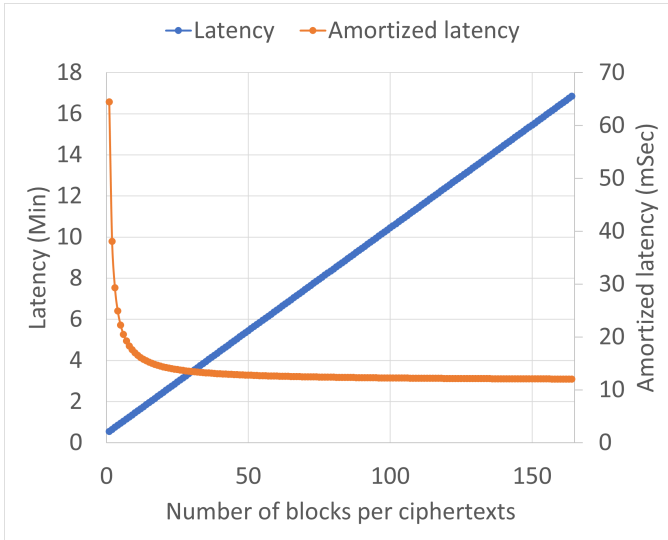


Figure 8: Latency and amortized latency of Ascon for a batch of 512 ciphertexts and different number of blocks (either AAD or ciphertext data).

takes around 1.125 seconds. The latency on the other hand increases linearly.

We make the assumption that users who intend to harness the capabilities of HE will utilize specialized devices such as GPUs, and potentially in the future, FPGAs or ASICs. Accordingly, we present the reported results for all our constructions on a GPU device. To provide a point of reference regarding the performance disparity between GPUs and CPUs, we also include the runtime of the CPU implementation for AES-CTR. As depicted in the table, even with 88 threads, the CPU implementation is nearly $7.75\times$ slower compared to the GPU implementation using a single thread.

The reported latency values are given in minutes, while the amortized latency values are reported in milliseconds, which represents a significant improvement compared to the previous methods outlined in Table I. It is evident that the fastest implementation among the tested implementations is AES-CTR, as it solely provides confidentiality guarantees. Conversely, AES-GCM and Ascon offer both confidentiality and authenticity capabilities, resulting in slower performance. Among the two, our AES-GCM implementation demonstrated faster speeds. It should be noted that AES-GCM operates on a block size of 16 bytes, whereas Ascon operates on a block size of 8 bytes. When comparing amortized latency per 16 bytes, the reported value for Ascon (25.6 mSec) is higher than that for AES-GCM (21 mSec). Another observation is that in our experiment we used a GPU with a single thread, in practice the computation of the AES-CTR and the GHASH functions can be parallelized, which will result in latency of 7.4 minutes and amortized latency of 13.5 milli-seconds.

X. AN END2END IMPLEMENTATION

Our end-to-end process is illustrated in Fig. 9, which shows the steps involved in our approach. In this demonstration, we utilize our implementations of AES-GCM and CKKS. The

objective is to perform an inference operation on a deep neural network, specifically ResNet-50, using a large image with dimensions $224 \times 224 \times 3$, which is currently the state-of-the-art when considering inference over HE. We have also experimented with AES-CTR, but the flow for AES-GCM is more complex due to the additional requirement of integrity checks. Therefore, we focus on describing the AES-GCM based flow in detail. It is worth noting that using Ascon instead of AES-GCM would result in a similar flow.

The demonstration begins with a client who employs AES-GCM to encrypt the sample data, which in this case is an image consisting of $224 \times 224 \times 3 = 150,528$ pixels represented as 32-bit floating-point elements. The total size of the data is approximately 588 KB, and the encryption size closely matches that of the plaintext (taking into account the overhead of adding a 128-bit tag). Finally, the user saves the data in some database location, in our case, it was our local file system.

Subsequently, a server process was executed on the same machine, which received an AES ciphertext to be decrypted. The total size of the ciphertext, 588 KB, can be accommodated within 2 units of 512 KB (in tile tensor shape notation, $[128, \frac{150,528 \times 4}{32,768}]$, see [2] for more info on tile tensors). In other words, 2 blocks of 128 HE ciphertexts are required to store the AES-encrypted data. In our implementation, we load all the 256 ciphertexts (approximately 3.072 GB) into memory. Alternatively, a lazy evaluation mode could be employed, where blocks of 1.5 GB are loaded at a time.

Upon completing the decryption process, we obtained the original data encrypted under HE in a bit-sliced representation, along with the authentication tag. We first compared the original AES-GCM plaintext tag with the resulting tag using an $IsEq()$ HE utility, which generates an authentication indicator. This indicator is then transmitted to the client, who can utilize it during the decryption process to determine whether to release the inference results or not. At this point we also decrypted the results and measured the generated noise after the AES-GCM decryption process. The average noise ($avg(|pt - HE.Dec(ct)|)$) observed was 1.16×10^{-10} , with an even smaller standard deviation of 7.30562×10^{-17} . These measurements confirm the expected behavior discussed in Section V.

Prior to executing the inference step, two additional steps were incorporated into the process. First, we needed to convert the bit-sliced data into numerical representation, which is discussed in greater detail in Section X-A. Once the data was prepared, it was necessary to ensure that it was packed using the same packing methodology selected for the inference operation, where the specific packing methodology employed depends on the particular model to be executed.

To accomplish this, the server initially loaded the final application (model inference) and queried it to determine the expected input format of the data. Utilizing this information, the server performed a permutation of the elements of the input ciphertext to their respective destinations. Further insights about permuting the data are provided in Section X-B.

Before running the inference step we needed to equip the process with two extra steps, the first convert the bit sliced data to number and is described in more details in Section X-A. Once we have the data ready we still need to make sure

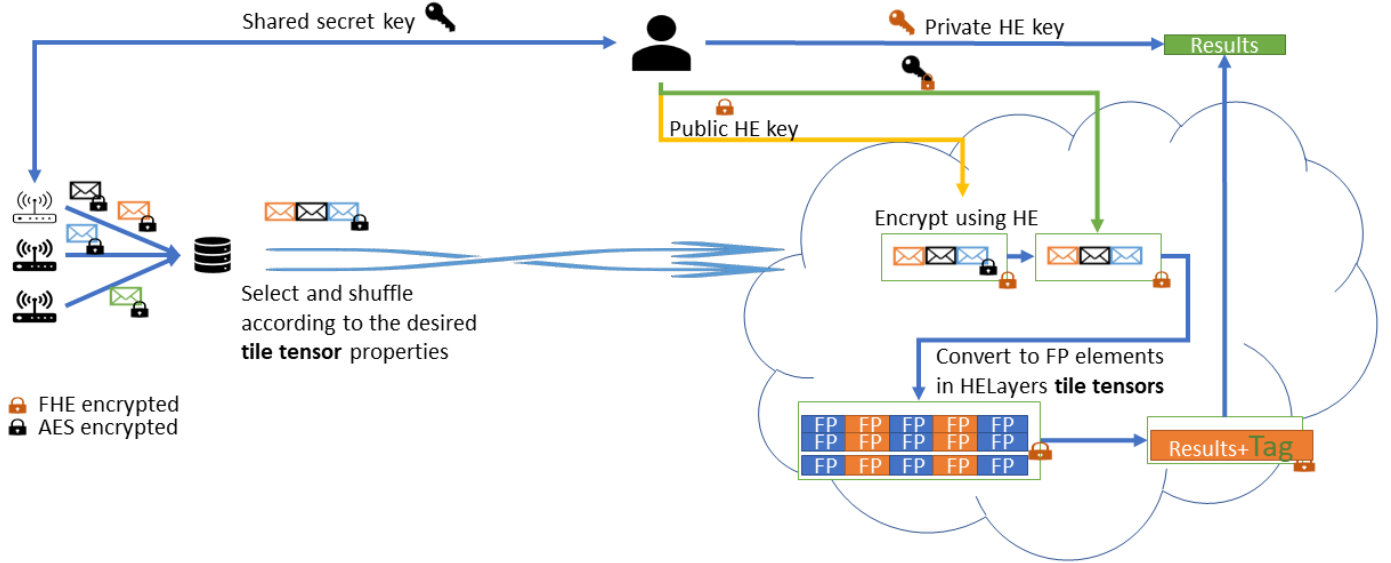


Figure 9: An illustration of an end-to-end flow using AES-GCM and CKKS.

that it is packed using the same packing methodology that was chosen to run the inference operation. This, of course depends on the model to be run. To this end, the server first needs to load the final application, and query it to learn the expected output format of the data. Using this data the server permutes the elements of the ciphertext to their destination. The overhead of the permutation depends on the number of rotations that need to be done on the input data and vary between different applications. Luckily it may only consume one multiplication depth due to the use of masking, which means that often no extra bootstraps are required on the data. We provide some insights on the topic in Section X-B.

To execute the inference operation, we leveraged the existing AI over HE capability provided by HLayers [2], as documented in [7]. Specifically, we utilized their pre-trained ResNet-50 model that is compatible with HE computations and trained on the ImageNet dataset. Notably, the latency and accuracy achieved in our implementation closely aligned with the results reported in [7]. This outcome was anticipated since our approach introduced no additional overhead to the inference process, and the negligible error introduced during the decryption process had minimal impact on the overall accuracy.

A. From bits to numbers

Upon completing the decryption process, we store the serialized data in a bit-sliced representation. However, subsequent applications require the data to be casted back to its original data type, which can include signed or unsigned integers, floating-point numbers, or fixed-point elements with sizes of 8, 16, 32, or 64 bits. In our implementation, we assume that knowledge of the original data type is common, similar to many other applications that utilize AES encryption. Nevertheless, if necessary, it is possible to include this information as an AAD of the AES ciphertexts.

It is important to note that not all conversions are feasible due to the inherent error involved in the restoration process.

For instance, if the HE ciphertext’s integer part consists of 16 bits, it does not make sense to restore a 32-bit integer within it, unless we have some guarantees on the input upper bound. Similarly, if both the integer part and the fractional part are 32 bits each, attempting to restore 32-bit integers would not be meaningful, as we would need to multiply the most significant bit (MSB) by 2^{32} . This operation would result in the error also growing by 2^{32} , potentially corrupting the lower bits of the integer. Therefore, it is crucial to consider the scheme parameters before attempting such conversions. Fortunately, many applications require the integer part to have a relatively small number of bits, allowing most of the data to be allocated to the fractional part. We stress that the restoration process may introduce some level of error, and careful consideration of the scheme’s limitations is necessary to ensure accurate and meaningful conversions.

Algorithm 1 presents a methodology for reconstructing numbers in scenarios where the desired type is a fixed-point or integer representation. It takes as input an array of bits in that encodes the number and precisely positions each bit according to its designated location. In order to mitigate potential errors arising from zero-valued bits, the algorithm employs a quadratic operation that effectively restores the original error magnitude. The choice of whether to perform one or two square operations dependence on the HE configuration and specifically, the error bound e .

B. Packing the data

The overhead associated with organizing the AES-decrypted data for consumption by subsequent applications, such as model inference, primarily involves rotating and masking operations. The extent of this overhead depends on the number of rotations needed for the input data and can vary across different applications. Fortunately, in many scenarios, this overhead is limited to a single multiplication depth, thanks to the utilization of masking techniques. Consequently, additional

Algorithm 1 Constructing numbers from bits

Input: in – an array of n encrypted bits, e the bound on the error
Output: out an integer with n bits and error below e .

```
1: procedure CONSTRUCTINT
2:    $out = in_0 + 2in_1$ 
3:   for  $i = 2$  to  $n - 1$  do
4:     if  $i < \frac{-\log_2(e)}{2}$  then
5:        $b = (2^{\lfloor i/2 \rfloor} \cdot in_i)^2$ 
6:       if  $i \pmod{2} = 1$  then
7:          $b = 2b$ 
8:       end if
9:     else
10:       $b = (2^{\lfloor i/4 \rfloor} \cdot in_i)^4$ 
11:       $b = 2^{i - (4 \lfloor i/4 \rfloor)} \cdot b$ 
12:       $out = out + b$   $\triangleright$  Here,  $out = out + 2^i in_i$ 
13:    end if
14:  end for
15:  return  $out$ 
16: end procedure
```

bootstraps are typically unnecessary for the data.

There are methods to mitigate this extra permutation cost. For instance, if the client possesses knowledge of the expected packing requirements, they can encrypt the data with AES in the desired format. However, in most cases, it is not anticipated that this approach will be feasible since data is often stored well in advance of its usage by the target model. Consequently, the specific model type and, hence, the required input packing style are typically unknown in advanced.

There are additional approaches that can expedite the process. First, compilers such as HElayers [2] could optimize the end-to-end process by considering the permutation costs when selecting the packing style to be used. By incorporating knowledge of the permutation overhead, compilers can make more informed decisions that minimize the overall computational requirements. Second, data preparation for packing can be performed earlier, specifically when the data is retrieved from the database. At this stage, the server can apply permutations to the AES ciphertext blocks, aligning them in a manner that reduces the subsequent number of required permutations. This approach is applicable to our AES-GCM implementation, as the encrypted HTBL powers and the IV+CTR inputs for the AES encryption calls can be permuted in the same way. However, this cannot be achieved with Ascon due to its serialization characteristic.

XI. DISCUSSION

Our demonstration establishes the feasibility and practicality of an end-to-end AT approach that enables the inference process. However, it is essential to acknowledge the additional components required by products that will utilize our implementation. These components include a key management system (KMS) for securely storing the AES, HE, and AES encrypted under HE keys. Additionally, a public key infrastructure (PKI) is necessary to manage the transfer of keys and validate their authenticity.

These additional components play a crucial role in ensuring the security and integrity of the system. Without proper safeguards, a malicious adversary could potentially provide the server with a manipulated ciphertext and a malicious encryption of the AES key under HE. While the ciphertexts may pass authentication, the resulting inference results would be compromised and incorrect.

Using scheme switching. An intriguing research avenue involves investigating the use of e.g., the B/FV scheme [36], [11] instead of CKKS [17] for AES decryption, followed by scheme switching from B/FV to CKKS for performing the inference operation. Exploring the optimal point at which to perform the scheme switching, such as before the bits-to-numbers conversion or after, or even after the permutation step, presents an interesting direction for further investigation. In the scope of this paper, we did not delve into this option, as the implementation scheme switching capabilities is anticipated to be realized in HE libraries after the submission of this paper.

Using AES-CTR only. The primary focus of our paper is on AT, and we propose the utilization of AES-GCM or ASCON for this purpose. However, there are certain scenarios where the use of AES-CTR alone is sufficient. One such example is when the sample data is transmitted directly to the server through a secure channel, such as TLS 1.3 [57]. In such cases, the client and server can rely on TLS 1.3 for data authentication, and no further guarantees are necessary. In this context, AES-CTR serves mainly to enable efficient and compressed data transmission, as opposed to encrypting the data directly under HE. Note that in this case the data is encrypted twice once with AES-CTR and another time with the AES-GCM or Poly-Chacha AEAD of TLS 1.3.

Other AEADs. Our choice to implement AES-GCM and Ascon was influenced by the fact that these schemes have either already been standardized or are on the verge of being standardized by NIST. However, there is an intriguing alternative known as Poly1305-ChaCha, which is an AEAD scheme that is also recommended for use with TLS 1.3. Upon examining its design, we observed that Poly1305-ChaCha involves numerous transitions between integers and bits. Specifically, it performs integer addition and immediately follows it with an XOR operation on the results. As mentioned earlier, the process of composing integers from bits and subsequently decomposing them for the XOR operation can be computationally expensive under the CKKS scheme. It remains an interesting alternative worthy of further investigation and evaluation in scenarios where the cost of transitioning between integers and bits is less of a concern.

XII. CONCLUSION

We explored the properties of a recent security notion that we term AT, which enhances the use case of hybrid encryption by incorporating an integrity layer to the inputs of the symmetric cipher. We have discussed the advantages and disadvantages of this approach, highlighting its potential benefits and limitations. Additionally, we have proposed a stronger notion called VAT, which represents an intriguing avenue for future research and development.

To demonstrate the practical feasibility of near-standardized hybrid encryption and AT, we have presented a novel implementation of an end-to-end neural network inference application that employs transciphering using standardized AEAD algorithms, specifically AES-GCM and Ascon. Our experimental results showcase that, when leveraging GPUs, the application achieves satisfactory execution times for various applications. We anticipate that upcoming HE accelerators will further enhance the speed and efficiency of our solution. This implies that within a relatively short time frame, approximately one to two years from now, when the HE standardization process is finalized, users will be able to adopt standardized hybrid encryption, eliminating certain barriers associated with the adoption of HE in general use cases.

REFERENCES

- [1] “AWS Encryption SDK,” jun 2023, last accessed Jun 2023. [Online]. Available: <https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/faq.html> 8
- [2] E. Aharoni, A. Adir, M. Baruch, N. Drucker, G. Ezov, A. Farkash, L. Greenberg, R. Masalha, G. Moshkovich, D. Murik *et al.*, “HElayers: A tile tensors framework for large neural networks on encrypted data,” *PoPETs*, 2023. [Online]. Available: <https://doi.org/10.56553/popets-2023-0020> 1, 2, 4, 10, 11, 12, 13
- [3] E. Aharoni, N. Drucker, G. Ezov, E. Kushnir, H. Shaul, and O. Soceanu, “E2E near-standard hybrid encryption,” March 2023, poster session at 6th HomomorphicEncryption.org Standards Meeting, Seoul, South Korea. [Online]. Available: <https://homomorphicencryption.org/6th-homomorphicencryption-org-standards-meeting/> 10
- [4] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, “Homomorphic encryption security standard,” HomomorphicEncryption.org, Toronto, Canada, Tech. Rep., November 2018. [Online]. Available: <https://HomomorphicEncryption.org> 1
- [5] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner, “Ciphers for MPC and FHE,” in *Advances in Cryptology – EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 430–454. [Online]. Available: https://doi.org/10.1007/978-3-662-46800-5_17 2
- [6] 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*, ser. CCS ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 139–150. [Online]. Available: <https://doi.org/10.1145/3548606.3559364> 2
- [7] M. Baruch, N. Drucker, G. Ezov, E. Kushnir, J. Lerner, O. Soceanu, and I. Zimmerman, “Sensitive Tuning of Large Scale CNNs for E2E Secure Prediction using Homomorphic Encryption,” 2023. 1, 3, 6, 12
- [8] A.-A. Bendoukha, A. Boudguiga, and R. Sirdey, “Revisiting Stream-Cipher-Based Homomorphic Transciphering in the TFHE Era,” in *Foundations and Practice of Security*, E. Aïmeur, M. Laurent, R. Yaich, B. Dupont, and J. Garcia-Alfaro, Eds. Cham: Springer International Publishing, 2022, pp. 19–33. [Online]. Available: https://doi.org/10.1007/978-3-031-08147-7_2 3, 7
- [9] J. Boyar, P. Matthews, and R. Peralta, “Logic Minimization Techniques with Applications to Cryptology,” *Journal of Cryptology*, vol. 26, no. 2, pp. 280–312, 2013. [Online]. Available: <https://doi.org/10.1007/s00145-012-9124-7> 5
- [10] J. Boyar and R. Peralta, “A Small Depth-16 Circuit for the AES S-Box,” in *Information Security and Privacy Research*, D. Gritzalis, S. Furnell, and M. Theoharidou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 287–298. [Online]. Available: https://doi.org/10.1007/978-3-642-30436-1_24 5
- [11] Z. Brakerski, “Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP,” in *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds., vol. 7417 LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 868–886. [Online]. Available: https://doi.org/10.1007/978-3-642-32009-5_50 2, 3, 7, 13
- [12] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, “(Leveled) Fully Homomorphic Encryption without Bootstrapping,” *ACM Transactions on Computation Theory*, vol. 6, no. 3, jul 2014. [Online]. Available: <https://doi.org/10.1145/2633600> 2, 3, 7
- [13] D. Canright, “A Very Compact S-Box for AES,” in *Cryptographic Hardware and Embedded Systems – CHES 2005*, J. R. Rao and B. Sunar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 441–455. [Online]. Available: https://doi.org/10.1007/11545262_32 5
- [14] 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,” *Journal of Cryptology*, vol. 31, no. 3, pp. 885–916, 2018. [Online]. Available: <https://doi.org/10.1007/s00145-017-9273-9> 2, 3
- [15] Centers for Medicare & Medicaid Services, “The Health Insurance Portability and Accountability Act of 1996 (HIPAA),” 1996. [Online]. Available: <https://www.hhs.gov/hipaa/> 1
- [16] J. H. Cheon, J.-S. Coron, J. Kim, M. S. Lee, T. Lepoint, M. Tibouchi, and A. Yun, “Batch Fully Homomorphic Encryption over the Integers,” in *Advances in Cryptology – EUROCRYPT 2013*, T. Johansson and P. Q. Nguyen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 315–335. [Online]. Available: https://doi.org/10.1007/978-3-642-38348-9_20 2
- [17] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437. [Online]. Available: https://doi.org/10.1007/978-3-319-70694-8_15 3, 6, 13
- [18] J. H. Cheon, D. Kim, and D. Kim, “Efficient Homomorphic Comparison Methods with Optimal Complexity,” in *Advances in Cryptology – ASIACRYPT 2020*, S. Moriai and H. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 221–256. [Online]. Available: https://doi.org/10.1007/978-3-030-64834-3_8 5
- [19] —, “Efficient homomorphic comparison methods with optimal complexity,” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2020, pp. 221–256. [Online]. Available: https://doi.org/10.1007/978-3-030-64834-3_8 6
- [20] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, “TFHE: Fast Fully Homomorphic Encryption Over the Torus,” *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020. [Online]. Available: <https://doi.org/10.1007/s00145-019-09319-x> 2, 3
- [21] 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*, M. Tibouchi and H. Wang, Eds. Cham: Springer International Publishing, 2021, pp. 640–669. [Online]. Available: https://doi.org/10.1007/978-3-030-92078-4_22 2
- [22] C. Cid, J. P. Indrøy, and H. Raddum, “FASTA – A Stream Cipher for Fast FHE Evaluation,” in *Topics in Cryptology – CT-RSA 2022*, S. D. Galbraith, Ed. Cham: Springer International Publishing, 2022, pp. 451–483. [Online]. Available: https://doi.org/10.1007/978-3-030-95312-6_19 2
- [23] J.-S. Coron, T. Lepoint, and M. Tibouchi, “Scale-Invariant Fully Homomorphic Encryption over the Integers,” in *Public-Key Cryptography – PKC 2014*, H. Krawczyk, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 311–328. [Online]. Available: https://doi.org/10.1007/978-3-642-54631-0_18 2
- [24] -. A. Corporation, “Arm security technology - building a secure system using trustzone technology whitepaper,” http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, April 2009. 8
- [25] O. Cosserson, C. Hoffmann, P. Méaux, and F.-X. Standaert, “Towards Case-Optimized Hybrid Homomorphic Encryption,” in *Advances in Cryptology – ASIACRYPT 2022*, S. Agrawal and D. Lin, Eds. Cham: Springer Nature Switzerland, 2022, pp. 32–67. 2
- [26] CryptoLab, “HEaaN: Homomorphic Encryption for Arithmetic of Approximate Numbers, version 3.1.4,” 2022. [Online]. Available: <https://www.cryptolab.co.kr/eng/product/haan.php> 2
- [27] C. De Cannière and B. Preneel, *Trivium*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 244–266. [Online]. Available: https://doi.org/10.1007/978-3-540-68351-3_18 3

- [28] C. Dobraunig, M. Eichlseder, L. Grassi, V. Lallemand, G. Leander, E. List, F. Mendel, and C. Rechberger, “Rasta: A Cipher with Low ANDdepth and Few ANDs per Bit,” in *Advances in Cryptology – CRYPTO 2018*, H. Shacham and A. Boldyreva, Eds. Cham: Springer International Publishing, 2018, pp. 662–692. [Online]. Available: https://doi.org/10.1007/978-3-319-96884-1_22 2
- [29] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schl  ffer, “Ascon v1.2,” Submission to Round 3 of the CAESAR competition, 2016. [Online]. Available: <https://competitions.cr.yt.to/round3/asconv12.pdf> 9
- [30] —, “Ascon v1.2,” Submission to Round 1 of the NIST Lightweight Cryptography project, 2019. [Online]. Available: <https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/ascon-spec.pdf> 9, 10
- [31] C. Dobraunig, L. Grassi, L. Helming, C. Rechberger, M. Schofnegger, and R. Walch, “Pasta: A case for hybrid homomorphic encryption,” Cryptology ePrint Archive, Paper 2021/731, 2021. [Online]. Available: <https://eprint.iacr.org/2021/731> 2
- [32] N. Drucker, G. Moshkovich, T. Pelleg, and H. Shaul, “Bleach: Cleaning errors in discrete computations over cks,” Cryptology ePrint Archive, Paper 2022/1298, 2022. [Online]. Available: <https://eprint.iacr.org/2022/1298> 3, 5, 6
- [33] L. Ducas and D. Micciancio, “FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second,” in *Advances in Cryptology – EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 617–640. [Online]. Available: https://doi.org/10.1007/978-3-662-46800-5_24 2
- [34] M. Dworkin, “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC,” 2007. [Online]. Available: <https://doi.org/10.6028/NIST.SP.800-38d> 1, 8
- [35] EU General Data Protection Regulation, “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation),” *Official Journal of the European Union*, vol. 119, 2016. [Online]. Available: <http://data.europa.eu/eli/reg/2016/679/oj> 1
- [36] J. Fan and F. Vercauteren, “Somewhat Practical Fully Homomorphic Encryption,” *Proceedings of the 15th international conference on Practice and Theory in Public Key Cryptography*, pp. 1–16, 2012. [Online]. Available: <https://eprint.iacr.org/2012/144> 2, 3, 7, 13
- [37] H. Fujii, F. C. Rodrigues, and J. L  pez, “Fast AES Implementation Using ARMv8 ASIMD Without Cryptography Extension,” in *Information Security and Cryptology – ICISC 2019*, J. H. Seo, Ed. Cham: Springer International Publishing, 2020, pp. 84–101. [Online]. Available: https://doi.org/10.1007/978-3-030-40921-0_5 5
- [38] Gartner, “Gartner identifies top security and risk management trends for 2021,” Tech. Rep., March 2021. [Online]. Available: <https://www.gartner.com/en/newsroom/press-releases/2021-03-23-gartner-identifies-top-security-and-risk-management-t> 1
- [39] C. Gentry, S. Halevi, and N. P. Smart, “Homomorphic Evaluation of the AES Circuit,” in *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds., vol. 7417 LNCS. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 850–867. [Online]. Available: https://doi.org/10.1007/978-3-642-32009-5_49 1, 2
- [40] L. Grassi, I. M. Ayala, M. N. Hovd, M.   ygarden, H. Raddum, and Q. Wang, “Cryptanalysis of symmetric primitives over rings and a key recovery attack on rubato,” Cryptology ePrint Archive, Paper 2023/822, 2023. [Online]. Available: <https://eprint.iacr.org/2023/822> 2
- [41] M.   gren, M. Hell, T. Johansson, and W. Meier, “Grain-128a: a new version of grain-128 with optional authentication,” *International Journal of Wireless and Mobile Computing*, vol. 5, no. 1, pp. 48–59, 2011. [Online]. Available: <https://doi.org/10.1504/IJWMC.2011.044106> 3
- [42] S. Gueron, “Intel’s New AES Instructions for Enhanced Performance and Security,” in *Fast Software Encryption*, O. Dunkelman, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 51–66. [Online]. Available: https://doi.org/10.1007/978-3-642-03317-9_4 4
- [43] —, “Intel® Advanced Encryption Standard (AES) New Instructions Set Rev. 3.01,” *Intel Software Network*, 2010. 4
- [44] J. Ha, S. Kim, W. Choi, J. Lee, D. Moon, H. Yoon, and J. Cho, “Masta: An he-friendly cipher using modular arithmetic,” *IEEE Access*, vol. 8, pp. 194741–194751, 2020. [Online]. Available: <https://doi.org/10.1109/ACCESS.2020.3033564> 2
- [45] J. Ha, S. Kim, B. Lee, J. Lee, and M. Son, “Rubato: Noisy Ciphers for Approximate Homomorphic Encryption (Full Version),” Cryptology ePrint Archive, Paper 2022/537, Tech. Rep. Report 2022/537, 2022. [Online]. Available: <https://eprint.iacr.org/2022/537> 2
- [46] S. Halevi, “Homomorphic Encryption,” in *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, Y. Lindell, Ed. Cham: Springer International Publishing, 2017, pp. 219–276. [Online]. Available: https://doi.org/10.1007/978-3-319-57048-8_5 3
- [47] P. Hebborn and G. Leander, “Dasta – alternative linear layer for rasta,” *IACR Transactions on Symmetric Cryptology*, vol. 2020, no. 3, p. 46–86, Sep. 2020. [Online]. Available: <https://doi.org/10.13154/tosc.v2020.i3.46-86> 2
- [48] IBM, “Ibm z15 performance of cryptographic operations,” 2020. [Online]. Available: <https://www.ibm.com/downloads/cas/6K2653EJ> 4
- [49] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, “Intel® Software Guard Extensions: EPID provisioning and attestation services,” *White Paper*, April 2016. [Online]. Available: <https://software.intel.com/sites/default/files/managed/ac/40/2016%20WW10%20sgx%20provisioning%20and%20attestation%20final.pdf> 8
- [50] E. Lee, J.-W. Lee, J. Lee, Y.-S. Kim, Y. Kim, J.-S. No, and W. Choi, “Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions,” in *Proceedings of the 39th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 17–23 Jul 2022, pp. 12403–12422. [Online]. Available: <https://proceedings.mlr.press/v162/lee22e.html> 1, 3, 6
- [51] B. Li and D. Micciancio, “On the Security of Homomorphic Encryption on Approximate Numbers,” in *Advances in Cryptology – EUROCRYPT 2021*, A. Canteaut and F.-X. Standaert, Eds. Cham: Springer International Publishing, 2021, pp. 648–677. [Online]. Available: https://doi.org/10.1007/978-3-030-77870-5_23 7
- [52] V. Lyubashevsky, C. Peikert, and O. Regev, “On Ideal Lattices and Learning with Errors over Rings,” in *Advances in Cryptology – EUROCRYPT 2010*, H. Gilbert, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23. [Online]. Available: https://doi.org/10.1007/978-3-642-13190-5_1 3
- [53] P. M  aux, A. Journault, F.-X. Standaert, and C. Carlet, “Towards Stream Ciphers for Efficient FHE with Low-Noise Ciphertexts,” in *Advances in Cryptology – EUROCRYPT 2016*, M. Fischlin and J.-S. Coron, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 311–343. [Online]. Available: https://doi.org/10.1007/978-3-662-49890-3_13 2
- [54] NIST, “FIPS pub 197: Advanced encryption standard (AES),” p. 311, 2001. [Online]. Available: <https://doi.org/10.6028/NIST.FIPS.197> 2, 4
- [55] NIST, “Lightweight cryptography,” 2023, last accessed 28 June 2023. [Online]. Available: <https://csrc.nist.gov/Projects/lightweight-cryptography> 2, 3, 9
- [56] C. Rebeiro, D. Selvakumar, and A. S. L. Devi, “Bitslice Implementation of AES,” in *Cryptography and Network Security*, D. Pointcheval, Y. Mu, and K. Chen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 203–212. [Online]. Available: https://doi.org/10.1007/11935070_14 5
- [57] E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.3,” RFC 8446, Aug. 2018. [Online]. Available: <https://doi.org/10.17487/RFC8446> 1, 4, 8, 13
- [58] N. P. Smart and F. Vercauteren, “Fully homomorphic SIMD operations,” *Designs, Codes and Cryptography*, vol. 71, no. 1, pp. 57–81, 2014. [Online]. Available: <https://doi.org/10.1007/s10623-012-9720-4> 1
- [59] R. Stracovsky, R. Akhavan, and F. Mahdavi Kerschbaum, “Faster Evaluation of AES using TFHE,” 2022, fHE.org 2022, Last accessed July 2023. [Online]. Available: <https://drive.google.com/file/d/1WMBjjM416BXGoiLf16gPn6q5aLt4zZqi/view> 2
- [60] The HEBench Organization, “HEBench,” 2022. [Online]. Available: <https://hebench.github.io/> 1
- [61] D. Toprakhisar, “Behaviour of algebraic ciphers in fully homomorphic encryption,” Aug 2021. [Online]. Available: <https://research.tue.nl/nl/studentTheses/behaviour-of-algebraic-ciphers-in-fully-homomorphic-encryption> 2

APPENDIX A

EFFICIENT LOOKUP TABLE WITH LIMITED VALUE RANGE

Consider a vector v consisting of n elements, where each element satisfies the condition $a \leq v_i \leq a + b$. We are given an address x , represented in binary form as x_i , such that $x = \sum_{i=0}^{\lceil \log_2 n \rceil} x_i 2^i$. The objective is to preprocess v in a way that facilitates efficient computation of v_x under HE, even when the binary values $x_0, \dots, x_{\lceil \log_2 n \rceil}$ are encrypted.

A straightforward approach to calculate v_x is as follows:

$$v_x = \sum_{i=0}^n \text{IsEq}(x, i) \cdot v_i \quad (9)$$

Here, IsEq represents a polynomial that yields an approximation of 1 when $c_1 = c_2$ and approximately 0 otherwise. It is important to note that evaluating IsEq typically incurs a significant computational cost and this naive method requires performing n evaluations of IsEq , resulting in potential inefficiency.

We propose a method that offers a significant improvement over the aforementioned approach. To achieve this, we make the assumption that $a = 0$ and observe that we can treat the elements of vector v as being within the range $0 \leq v_i \leq b$. The rationale behind this assumption is that we can introduce a new vector v' , where $v'_i = v_i - a$, for all i . By doing so, we can compute v_x as $v'_x + a$. Denote by $\bar{x}_i = 1 - x_i$ then

$$\begin{aligned} v_x &= (\bar{x}_{\log n} \cdots \bar{x}_1 x_0) v_0 + (\bar{x}_{\log n} \cdots \bar{x}_1 x_0) v_1 + \\ &\quad (\bar{x}_{\log n} \cdots \bar{x}_2 x_1 x_0) v_2 + (\bar{x}_{\log n} \cdots \bar{x}_2 x_1 x_0) v_3 + \\ &\quad \dots + (x_{\log n} \cdots x_1 x_0) v_n \\ &= \sum c_{m_i} m_i, \end{aligned}$$

where m_1, \dots, m_n represent n monomials, namely $1, x_0, x_1, \dots, x_{\log n}, x_0 x_1, \dots, x_0 x_1 \dots x_{\log n}$ and the coefficients c_i depend on the values of v and can be computed using the formula:

$$c_m = \sum_j v_j f_m(j). \quad (10)$$

For example, $c_1 = v_0$ and $c_{x_i} = v_{2^i} - v_0$, for any i .

The coefficients c_m and the functions f_m possess certain properties: a) for a monomial m with k variables, the summation in Equation 10 contains 2^k terms for which $f_m(i) \neq 0$; b) Except for the monomial $m = 1$, the number of occurrences where $f_m(j) = 1$ is equal to the number of occurrences where $f_m(j) = -1$. It follows that c_m have a binomial distribution with $\mathbb{E}[c_m] = 0$.

A. Computing all monomials efficiently

To compute all the monomials $1, x_1, \dots, x_{\log n}, x_1 x_2, x_1 x_3, \dots, x_1 x_2 \dots x_{\log n}$, a recursive approach can be employed.

We start with the given monomials $1, x_1, \dots, x_{\log n}$ as input. Then, we recursively compute a monomial m by multiplying two existing monomials, denoted as m_1 and m_2 , where the number of variables in m_1 and m_2 is approximately half of those in m . This recursive process continues until all desired

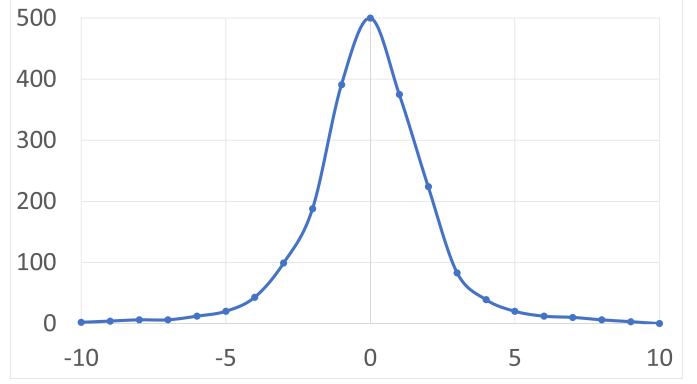


Figure 10: The number of monomials (y-axis) with a coefficient c_m (x-axis) derived from the values of the AES S-box.

monomials are computed. The number of multiplications required in this process is $n - \log n - 1$, and the depth of the computation is $\log n$.

Our proposed method offers several advantages over the approach described in Equation 9:

- 1) It requires fewer ciphertext-ciphertext multiplications. Specifically, our method requires $n - \log n - 1$ multiplications, which is significantly fewer compared to the $n \log n$ multiplications required by the other method.
- 2) Due to the Binomial distribution nature of the coefficients c_m , many monomials have coefficients of zero value. This means that the computation of these monomials can be skipped entirely, leading to further reduction in the number of required multiplications.
- 3) In addition to zero-valued coefficients, many monomials have coefficients of 1 or -1. Since multiplication by these scalar values does not require actual multiplications, the computation becomes even more efficient.

B. AES S-box as an example

We tested our method on computing an AES S-box, which showed slightly improvement over the method described in Section IV-B. Computing S-box requires a lookup table of 256 entries, where each value is from the range $[0, 255]$. A boolean function is then performed on the bits of the value read from the table. To perform this efficiently, implementations split the table into 8 tables where each table holds a single bit of the output.

We tested our method on an AES S-box that involves a lookup table with 256 entries, where each entry corresponds to a unique value in the range of $[0, 255]$. A boolean function is applied to the bits of the value retrieved from the table. As in Section IV-B, to optimize this computation, we divided the table into 8 sub-tables, with each sub-table responsible for a single output bit of the original values. Fig. 10 shows the distribution of the coefficients.

Based on our evaluation, our proposed methodology exhibited a moderate advancement over the implementation of Section IV-B when implemented for the computation of the AES S-box. However, it also incurred supplementary overhead in the form of coding the boolean functions. Consequently,

we made a decision to forgo this optimization during the experimentation phase, as elaborated in Section IX, and include it solely as an appendix for the sake of completeness.