# 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 parties while maintaining data confidentiality. Hybrid encryption (a.k.a transciphering) allows a reduction in the number of ciphertexts and storage size, which makes FHE solutions practical for a variety of modern applications. Still, modern transciphering has three main drawbacks: 1) lack of standardization or bad performance of symmetric decryption under FHE; 2) post-HE-evaluation is limited to small-size applications; 3) lack of input data integrity. Interestingly, modern-size secure inference applications were demonstrated using approximated FHE schemes such as CKKS. However, implementing transciphering using standard Authenticated Encryption (AE) over CKKS is challenging due to its approximated nature.

In this paper, we aim to close these gaps. First, we report and demonstrate the first end-to-end process that uses transciphering for real-world applications i.e., running deep neural network (DNN) inference under encryption. For that, we discuss the concept of Authenticated Transciphering (AT), which like AE, provides some integrity guarantees for the transciphered data. Finally, to demonstrate the AT concept, we report on the first implementation of Ascon decryption under CKKS, and complete the picture with a detailed technical description of our AES-GCM implementation under CKKS.

## KEYWORDS

homomorphic encryption, hybrid encryption, transciphering, standard implementations, AES256-GCM

## 1 INTRODUCTION

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 [61], or at rest, using advanced encryption standard (AES)-Galois / counter mode (GCM) [36]. The combination of these solutions allows users to enjoy, for example, the storage services provided by public cloud environments. Still, to better utilize the cloud, organizations and industries like finance and healthcare aim to move their workloads from in-house data centers to the cloud in order to use its scalable and reliable compute capabilities. A major issue with the above is that the use of these third-party services can be restricted by the need to comply with government regulations such as GDPR [37] and CCPA [1], which ensure data privacy.

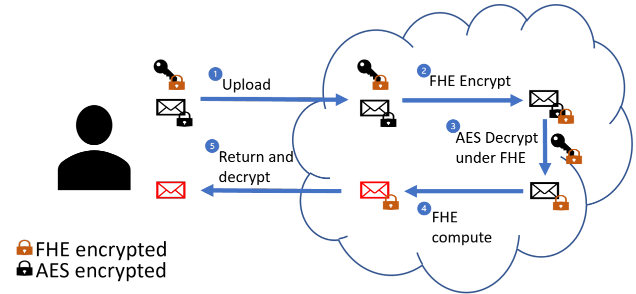


**Figure 1: A hybrid encryption flow. 1. A user encrypts some data using AES (black key and lock), locks the key using FHE encryption (brown lock) and uploads both to the cloud. 2. The cloud encrypts the AES encrypted data using FHE (double-encryption). 3. The cloud removes the AES decryption under FHE using the encapsulated AES key. 4. The cloud performs some computation on the FHE encrypted data. 5. The cloud returns the computation results encrypted to the user who decrypt and consume the desired results.**

Fully homomorphic encryption (FHE), which gains popularity nowadays, can address the above issue because it enables computation to be performed on encrypted data. The potential of FHE can be observed in Gartner's report [40], 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 FHE. Another example is the extensive list of enterprises and academic institutions actively involved in initiatives like HEBench [65] and the standardization efforts for FHE [5].

Only in the last few years FHE solutions become practical enough for complex tasks such as secure DNN inference over relatively large datasets and architectures as in [2, 9, 54]. These often rely on the CKKS FHE scheme due to its approximated nature, which leads to efficient solutions but also raises unique challenges for handling the accumulated error without destroying the original data.

Moreover, these studies often assume an ephemeral application, where data is encrypted using FHE, uploaded to an untrusted environment for FHE computation, and the results are 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 FHE ciphertexts can result in extra costs. For instance, FHE 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

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

*Proceedings on Privacy Enhancing Technologies YYYY(X), 1–17*
© YYYY Copyright held by the owner/author(s).
https://doi.org/XXXXXXX.XXXXXXX

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., [41, 62]), 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 FHE. Through transciphering, the service effectively "replaces" the encryption scheme from symmetric encryption to FHE encryption. Once the data is encrypted using FHE, the service can perform computations on it and return the results to the user or store them for future use. Fig. 1 illustrates this process. We note that the term hybrid encryption was introduced in [27] where key and data encapsulation mechanisms (KEM/DEM) were combined to generate a public key encryption (PKE). In the context of FHE it is common to use the same term, where the data undergoes encryption through a symmetric encryption method, while the key of the symmetric encryption undergoes encryption using a PKE scheme such as FHE.

**Our contribution.** We demonstrate, for the first time, an end-to-end application that uses practical near-standard hybrid encryption and subsequently practically evaluates a large task such as running DNN inference over a modern size network, ResNet-50, and a large dataset – ImageNet with images of size $224 \times 224 \times 3$. Our demonstration relies on CKKS because: 1) it was demonstrated to show good results for secure inference under FHE; 2) among the FHE schemes, it is the only one that was already standardized (in South Korea [63]). **Note that we say that our solution is near-standard because a (CKKS) FHE standard is only available in South Korea or otherwise expected in late 2024 [5].** In addition, we consider and address the following research challenges.

1. **Can we implement transciphering from a standardized cipher to CKKS?** CKKS is an approximated FHE scheme, and thus considered unsuitable for implementing standard symmetric ciphers, which operate over finite fields. This is due to the extra accumulated error that does not disappear when bootstrapping. To address this issue, we leverage a new technique called BLEACH [34] that allows maintaining the error size when running binary circuits over CKKS. Specifically, we use it to implement algorithms such as AES-GCM and Ascon in bit-wise representation.
2. **How to make BLEACH more efficient for large binary circuits?** We found that performing a cleanup operation after every gate, as in [34], can lead to a slow implementation. To mitigate this, we expand BLEACH in Section 5 to support cleaning after multiple gates, which enables a more efficient implementation of binary circuits.
3. **Can the above mentioned transciphering be efficient?** We answer affirmatively to this question and provide details on how to construct a hybrid encryption implementation that can be considered practical by many applications and which uses NIST standardized block-cipher, specifically AES-counter (CTR)-128. We do that by using IBM® HElayers [2] compiled with CryptoLab® HEaaN [28] and running on a commodity GPU. Our implementation runs in 1.65 minutes for 512KB of data, with an amortized latency of 3 msec per AES block. In addition, we report a 1.71× speedup over our initial results for AES-CTR-256 that were presented (without technical details) in a poster [4].
4. **What is authenticated transciphering (AT) and why is it required?** We study and discuss the security guarantees of the above implementation using a notion that we call AT. Here the integrity of the input data to the FHE service is guaranteed. An AT implementation was presented e.g., in [10] but without labeling it as such and without any discussion about the motivation and consequences of using it. We complete this gap and discuss the pros and cons of this notion and how it should be used in threat models.
5. **Does AES-CTR is enough to provide integrity to the encrypted data?** No, 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 such as AES-CTR. Thus, we also consider the integration of an end-2-end application with a standardized AEAD, specifically AES-GCM. Our AES-GCM-128 implementation exhibits efficient performance, processing 512KB of data in a mere 6.95 minutes, with an amortized latency of 13 msec per AES block, whereas our AES-GCM-256 implementation runs in 7.66 minutes, with an amortized latency of 14 msec per AES block, 1.5× faster compared to the initial poster results [4]. The paper provides the implementation details and describes the way we handle the nullification of data for invalid tags.
6. **How to support IoT devices that cannot afford AES-GCM?** in addition to our AES-GCM implementation, we also implemented and evaluated the Ascon cryptosystem under CKKS. Ascon is the winner of the national institute for standards and technology (NIST) lightweight project [59] and is about to be standardized.
7. **What challenges exist when implementing an AT solution under CKKS?** AEAD schemes must erase the plaintext data upon receiving invalid tags. This is harder to achieve in CKKS because the error is blended with the plaintext. In Section 6 we discuss challenges and solutions for verifying the AEAD tags and nullify faulty ciphertexts under FHE.

**Roadmap.** The rest of the paper is organized as follows. Sec. 2 reports the state-of-the-art hybrid encryption constructions. Sec. 3 lists the notation used in the paper, and briefly explains the concepts of FHE and AEAD. The paper proceeds by describing our fast AES-CTR implementation in Sec. 4. The handling of the error of CKKS is explained in Sec. 5, and our AT notion is explored in Sec. 6. Using this notion, AT schemes using AES-GCM or Ascon, and CKKS are implemented in Sections 7 and 8, respectively. The latency measurements of running these schemes are reported in Sec. 9. The above implementation is utilized to construct and demonstrate the first end-to-end flow for performing DNN inference over encrypted data in Sec. 10. Finally, Sec. 11 discusses some takeaways from the study, and Sec. 12 concludes the paper.

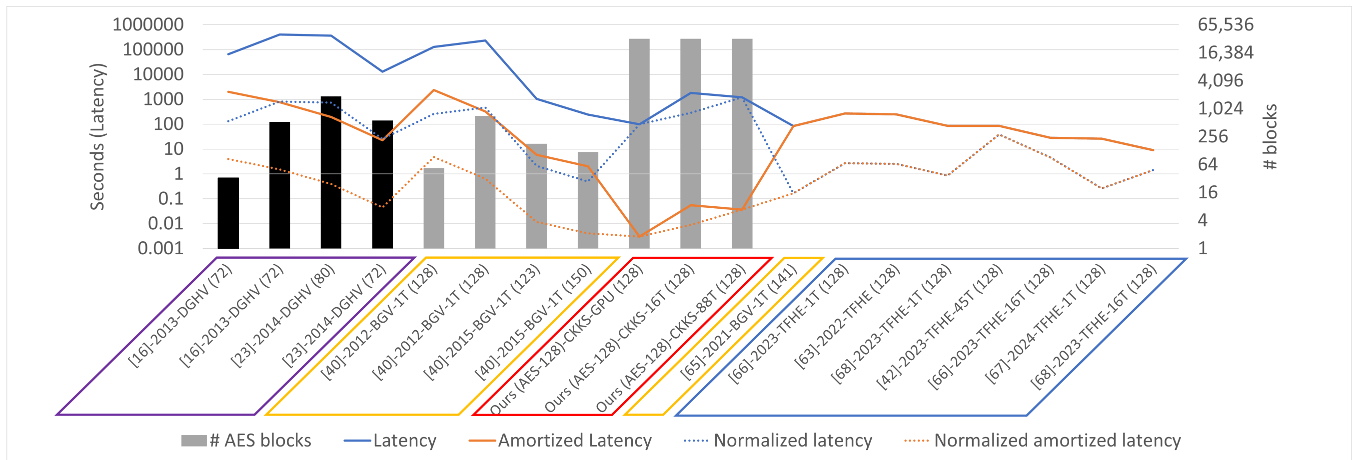


**Figure 2: Performance of AES implementations under FHE. Parallelograms represent different FHE schemes: DGHV (purple), BGV (yellow), CKKS - ours (red), and TFHE (blue). The bars are associated with the right y-axis and count the number of AES blocks that were evaluated in parallel (batch size). The solid lines are associated with the left y-axis and report the latency and amortized latency (latency/batch size) as originally reported per implementation. The x-axis labels contains the original number of CPU threads. The dotted lines represent the original performance normalized for a GPU run (as in our case). Specifically, we assumed a 500× speedup of our GPU compared to "old" CPUs and a factor of $\frac{100}{x}\times$ for an $x$ threads modern CPU implementation.**

| 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 3: Non-standardized block cipher proposals. Each block cipher is described below the FHE scheme that it targets. The order in which the block ciphers is listed is arbitrary.**

## 2 RELATED WORK.

The two principle categories for transciphers' implementation are: a) implementations of transciphers that use standard symmetric encryption e.g., AES-electronic codebook (ECB) [58]; b) implementations of transciphers that offer a new non-standardized symmetric encryption schemes.

The former group of implementations is compared in Fig. 2. DGHV-based implementations offer less than 128-bit security e.g., [17, 24], while others (e.g., [41]) took over an hour to complete. A revision [42] of [41] reported the fastest amortized latency of 2 seconds per AES block when using the BGV FHE scheme, but claimed that no further deep computations are possible, which means that we cannot use it for our end-to-end implementation. Alternatively, [42] also reported a latency of 1,050 seconds and an amortized latency of 5.833 sec, with a bootstrappable BGV context. In contrast, our best implementation has an amortized latency of 3 msec, 1925× faster. We remark that we use a commodity GPU while [42] used a single-thread CPU from 2014. To normalize the results of [42] we used a factor of 500×, see Fig. 2 dotted lines. Still, our implementation is faster. Moreover, there is no known FHE scheme switching ability between BGV and CKKS, and there is no known BGV implementation of modern DNNs, which makes our implementation the only one that fits an end-to-end demonstration.

Fig. 2 also presents TFHE-based implementations [43, 64, 66–68], which operate on one AES block at a time. Thus, their latency equals the amortized latency with minimal results of 9 sec per block (running on 16 CPU threads). Also here, we used the dotted line to normalize the results, this time with a factor of $\frac{100}{x}\times$ for an $x$ threads CPU implementation. With this normalization, the fastest implementation has a latency of 0.262 seconds, which is 87.3× slower compared to our implementation. Our end-two-end demonstration involves an image input of size $224 \times 224 \times 3 = 150{,}528$ bytes or $150{,}528/16 = 9{,}408$ AES blocks, which will take $9{,}408 * 0.262 = 2{,}465$ seconds to decrypt using the fastest TFHE implementation, while only 99 seconds with our implementation.

The second type of implementations is summarized in Fig. 3 and it includes Krevyium [16], FLIP [57] and Elizabeth [26] that target the FHEW/TFHE [21, 35] FHE schemes, LowMC [6], Rasta [30], Dasta [51], Pasta [33], Masta [48], Fasta [23], and Chaghri [7] that target the B/FV [13, 38] and BGV [14] schemes, and Hera [22] and Rubato [49] that target CKKS. However, none of these schemes are yet standardized, or planned to be standardized (a process that takes several years), 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 [44] that present a key recovery attack on Rubato with some recommendations for parameter modifications.

Another type of implementation was conducted to assess the performance of FHE on lightweight stream ciphers that participated

in the NIST lightweight cryptography project [59] before the selection of Ascon as the finalist for standardization. For example, [10] reported the implementations of the stream ciphers Trivium [29], Kreyvium [16], and Grain-128a [45] under TFHE [21]. Additionally, it also implemented Grain128-AEAD [45] and thus initiated the process of studying AT constructions. Sec. 6 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.

# 3 PRELIMINARIES AND NOTATION

Table 1 details the notation used throughout the paper.

**Table 1: Notation legend.**

| Notation | Description |
|---|---|
| $0^x$ | A sequence of $x$ bits, where all bits are 0. |
| $a\|\|b$ | A concatenation of two strings $a$ and $b$. |
| $b_i$ | The $i$th bit of a byte $b$, where $b = \sum_{0\le i<8} b_i \cdot 2^i$. |
| $GF(2^a)$ | A Galois fields (GF) of characteristic $2^a$, e.g., $GF(2^8)$ for the AES state elements. |
| $0x$ | Hexadecimal prefix, e.g., $0xe = 14$. |
| $a \oplus b$ | A Boolean XOR operation of two bits, bytes, or 64-bit words dependent on the context. |
| $a \odot b$ | A Boolean AND operation of two bits, bytes, or 64-bit words dependent on the context. |
| $a := f()$ | A deterministic assignment of $f()$ to $a$. |
| $a \leftarrow f()$ | A probabilistic assignment of $f()$ to $a$. |
| $x \ggg k$ | A left rotation of the bits of a value $x$ by $k$. |

## 3.1 Homomorphic Encryption

We start by describing the high-level background and basic concepts of FHE schemes. FHE schemes allow us to perform operations on encrypted data [50]. Modern FHE instantiations such as BGV [14], B/FV [13, 38], and CKKS [18] rely on the complexity of the Ring-LWE problem [56] for security and support single instruction multiple data (SIMD) operations. The FHE system has an encryption operation $\mathrm{FHE.Enc}: \mathcal{R}_1 \rightarrow \mathcal{R}_2$ 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 $\mathrm{FHE.Dec}: \mathcal{R}_2 \rightarrow \mathcal{R}_1$. An FHE scheme is correct if for every valid input $x, y \in \mathcal{R}_1$

$$\mathrm{FHE.Dec}(\mathrm{FHE.Enc}(x)) = x \quad (1)$$

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

$$\mathrm{FHE.Dec}(\mathrm{FHE.Enc}(x) \cdot \mathrm{FHE.Enc}(y)) = x * y \quad (3)$$

and is approximately correct (as in CKKS) if for some small $\epsilon > 0$ that is determined by the key, $|x - \mathrm{FHE.Dec}(\mathrm{FHE.Enc}(x))| \le \epsilon$. Equations 2,and 3 are modified in the same way. In this paper, we used CKKS for the experiments, as state-of-the-art DNN inference studies such as [9, 54] 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 FHE 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 FHE 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 FHE 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, we adopt the methodology proposed by BLEACH [34] and employ a cleanup utility after a specific number of Boolean gates. See further analysis of the error management in Sec. 5.

## 3.2 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 $C$.

The key generation method $k \leftarrow \mathrm{AEAD.KeyGen}$ generates a new (pseudo)random symmetric key $k \in \mathcal{K}$. The encryption function $(c,t) \leftarrow \mathrm{AEAD.Enc}_k(a,n,m)$ 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 $c \in C$ in addition to an authentication tag $t$ over the pair $(a,m)$. The decription method $\{m,\perp\} = \mathrm{AEAD.Dec}_k(a,n,c)$ receives a ciphertext $c \in C$ 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 $\perp$.

# 4 ADVANCED ENCRYPTION STANDARD (AES)

This chapter starts by briefly describing the AES block cipher. Subsequently, we describe our encrypted AES-CTR implementation.

## 4.1 The AES block cipher

AES was officially standardized by NIST in 2001 [58] 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,

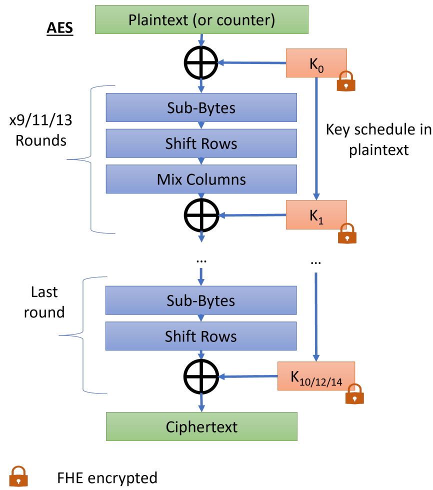


**Figure 4: An illustration of an AES-128/192/256 block cipher. In hybrid encryption, the key is encrypted using FHE.**

which is strongly supported by industry players like IBM in its Z systems [52], and Intel, who have introduced AES-NI processor instructions [46, 47] to enhance AES performance.

We briefly describe the AES encryption algorithm, which we illustrate in Fig. 4, the decryption procedure is described in [58]. The description of the AES steps below is required for comprehending the FHE optimization we implemented and for later understanding of how we addressed the noise challenge when operating under CKKS. 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, …, 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 $RoundKey[j]$ 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), 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 FHE, 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 [61].

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.

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.*

## 4.2 Implementing AES-CTR over HE

Our implementation follows a bit-sliced approach, where each FHE plaintext slot is treated individually per AES block, leveraging the FHE 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 App. B). We consider ciphertexts of size approximately 12 MB, thus the overall state size amounts to $\sim$ 12 MB[1] $*128 = 1.5$ 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 [34]. 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

[1]Every ciphertext involves two polynomials each has 65,536 coefficients and a multiplication depth of $\sim$ 12. Using 8 bytes per coefficient leads to $8 * 12 * 65{,}536 * 2 = 12MB$.

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 [19]. 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 $\epsilon$ denotes an extremely small value, see Sec. 5. 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, see App. B.

Remark 2. *It is feasible to "share" FHE ciphertexts for multiple AES decryptions, employing distinct keys. This can be achieved by either the clients broadcasting the respective keys to the corresponding FHE 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 FHE 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 $4 \times 4$ 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

$$D0 = x \cdot b0 + (x+1) \cdot b1 + b2 + b3$$
$$D1 = b0 + x \cdot b1 + (x+1) \cdot b2 + b3$$
$$D2 = b0 + b1 + x \cdot b2 + (x+1) \cdot b3$$
$$D3 = (x+1) \cdot b0 + b1 + b2 + x \cdot b3$$

As described in [39], these equations can be simplified to:

$$D0 = x \cdot (b_0 + b_1) + b_1 + b_2 + b_3$$
$$D1 = x \cdot (b_1 + b_2) + b_2 + b_3 + b_0$$
$$D2 = x \cdot (b_2 + b_3) + b_3 + b_0 + b_1$$
$$D3 = x \cdot (b_3 + b_0) + b_0 + b_1 + b_2$$

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

$$(a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0) = (a_6, a_5, a_4, a_3 \oplus a_7, a_2 \oplus a_7, a_1, a_0 \oplus a_7, a_7)$$

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:

$$D0 = (x^3+x^2+x) \cdot b0 + (x^3+x+1) \cdot b1 + (x^3+x^2+1) \cdot b2 + (x^3+1) \cdot b3$$
$$D1 = (x^3+1) \cdot b0 + (x^3+x^2+x) \cdot b1 + (x^3+x+1) \cdot b2 + (x^3+x^2+1) \cdot b3$$
$$D2 = (x^3+x^2+1) \cdot b0 + (x^3+1) \cdot b1 + (x^3+x^2+x) \cdot b2 + (x^3+x+1) \cdot b3$$
$$D3 = (x^3+x+1) \cdot b0 + (x^3+x^2+1) \cdot b1 + (x^3+1) \cdot b2 + (x^3+x^2+x) \cdot b3$$

These equations involve multiple serial multiplications, which leads to an increase in the circuit's multiplication depth when executed under FHE. 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 [11, 12, 15, 60]. One prominent approach involves transforming the AES Galois field data to a tower (composite) field with a minimized number of gates. For instance, in [12], a circuit was achieved using only 34 AND gates and a multiplication depth of 4, while [11] 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 FHE 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 FHE 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.

In Appendix A we show how all 255 indicators can be computed using 255 multiplications and depth 3. In our implementation we used a similar technique, however, to have a more readable code our implementation used 272 multiplication and the same depth.

As noted above, since the tables are binary and given in plaintext. The output of each table can be computed as a sum of a subset of the indicators with no additional multiplication.

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 $h_1$ (Sec. 5): 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 when the underlying hardware supports it. 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.

# 5 BINARY CIRCUITS OVER CKKS

We chose CKKS [18] because it is the leading FHE scheme when considering state-of-the-art inference applications, for example, [9, 54]. For our implementation, we leveraged a recent technique called BLEACH [34], which has demonstrated that executing binary circuits over CKKS is practical. Specifically, it showed (Lemma 5.1) that performing XOR ($\oplus$), AND ($\wedge$), or OR ($\vee$) operations on two encrypted bits, followed by the cleanup function $h_1(x) = -2x^3+3x^2$ [20], does not introduce any significant increase in the ciphertext error. This allows us to execute these operations while maintaining the desired level of accuracy in the ciphertext.

Lemma 5.1 ([34] 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 $h_1(x \oplus y)$*

However we found that bleaching after every boolean gate is not efficient enough, and thus we extend this BLEACH 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 5.2 ([34][Lemma 2 ). *For $x = b_x + e_x$ and $y = b_y + e_y$, where $b_x, b_y \in \{0, 1\}$ and $|e_x|, |e_y| < e < 0.25$,*

$$|(x \wedge y) - (b_x \wedge b_y)| < 5e,$$
$$|(x \vee y) - (b_x \vee b_y)| < 5e,$$
$$|(x \oplus y) - (b_x \oplus b_y)| < 2.25e.$$

Using this gate error bounds (5,5,2.25) we state Lemma 5.3.

Lemma 5.3. *Let $0 < e < 0.25$ be the bound on the initial error of the inputs, 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 $z = h_1(f(x_1, \ldots, x_n)) = b_z + e_z$, 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 $w = b_w + e_w = f(x_1, \ldots, x_n)$, 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 $Be+e_{ckks} < (B+0.25)e$, 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:

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

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 Fig. 5. 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.*

# 6 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 FHE scenario, there exists a user and an untrusted *semi-honest* environment, such as the cloud. The user generates the FHE 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 FHE service, the user encrypts data using the FHE 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 DNN inference, and returns the encrypted results to the user. The user can then decrypt and view the results using their private key.

Modern FHE schemes are designed to be either IND-CPA secure (e.g., BGV [14] or B/FV [13, 38]) or IND-CPA$^D$ secure [55] like

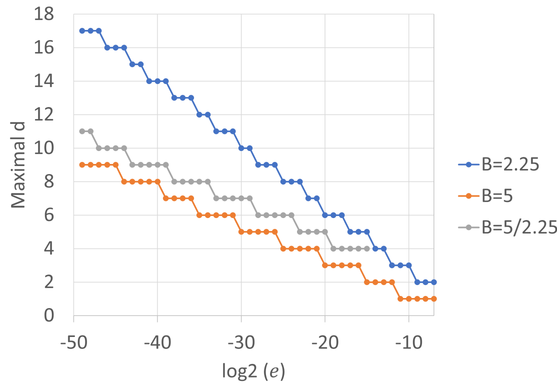


**Figure 5: Maximum $d$ (y-axis) as a function of $\log_2(e)$ (x-axis). Three functions ($f$) are considered: $f$ with AND gates only ($B = 5$), $f$ with XOR gates only ($B = 2.25$), and $f$ with XOR gates except for the last three levels ($B = 5/2.5$).**

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 FHE 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 FHE service, specifying a list of keys to be utilized for accessing data from the database. The FHE 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 FHE service to authenticate the data on behalf of the user. We refer to the combination of AEAD and FHE construction as AT. The concept of AT was also explored in [10] with the Grain128-AEAD implementation. Below we provide further discussion on AT that leads to our near-standardized implementations in Sections 7, 8.

The fundamental concept underlying AT is to ensure that the authentication tag propagates seamlessly from the AEAD ciphertexts to the FHE 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 FHE service and the tags generated during the AEAD decryption under FHE 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 FHE 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 FHE from any potential adversary. Even though the key is encrypted under FHE, 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 FHE service itself since we assume it to be semi-honest. Moreover, regardless of the situation, the FHE 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 FHE secret key and thus cannot decrypt FHE ciphertexts.

**Alternatives to AT.** There exist alternatives to the aforementioned construction, such as employing asymmetric encryption instead of symmetric encryption in conjunction with the FHE 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 FHE 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. FHE 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 FHE ciphertexts, such as using verifiable computation (VC) or trusted execution environments (TEEs) like Intel® SGX [53] or ARM® Trustzone [25], 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 FHE 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 FHE 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 FHE computation. The client can ask the FHE 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 FHE, it can combine them in the evaluation process.*

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

## 7 AES-GCM

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

### 7.1 Background

The Galois / counter mode (GCM) [36] 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 [61] and is highly recommended by prominent companies libraries like AWS encryption SDK [8]. 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. Fig. 6 illustrates the AES-GCM scheme, where it highlights the parts encrypted under FHE.

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 FHE is presented in Fig. 6. The figure provides a visual representation of the components that are encrypted with AEAD, encrypted with FHE, 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 FHE. Moreover, because $H = AES_k(0^{128})$ is encrypted, the entire tag computation must be done under FHE.

### 7.2 An implementation of AES-GCM

Our implementation of the AES-CTR mode is discussed in detail in Sec. 4.2. This implementation serves as the foundation for our

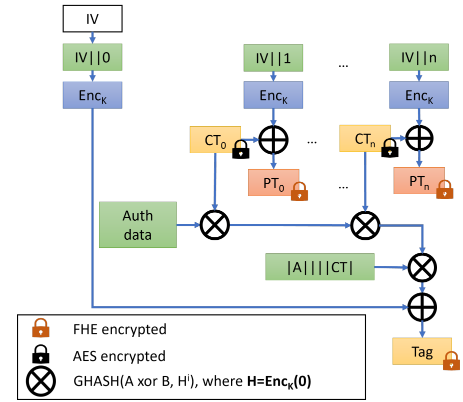


**Figure 6: An illustration of the AES-GCM AEAD scheme within the context of FHE. Green blocks represent plaintext blocks, blue blocks represent AES-GCM encryption methods, yellow blocks represent AES-GCM encrypted blocks, and red blocks represent the AES-GCM decrypted plaintext that remains encrypted under FHE.**

```
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 7: An illustration of our GHASH implementation using SageMath and Numpy.**

AES-GCM implementation, as well as an additional implementation of the GHASH function under FHE.

The code presented in Fig. 7 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 FHE 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 FHE rotate operation (*np.roll*) to be executed without incurring additional computational cost.

**HTBL.** Consider the AAD data $A = a_1, \ldots, a_m$ and ciphertext data $C = c_1, \ldots, c_n$ as elements from $GF(2^{128})$ on which we apply the function GHASH$(A, C, H)$, defined as:

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

Fig. 6 presents an alternative approach to compute the GHASH tag, utilizing Horner's rule, which states that

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

This technique is commonly employed to avoid the expensive computation of powers of $H$. However, due to the SIMD nature of FHE, 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 FHE 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 FHE 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 FHE 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 $HTBL$ is similar to the size of an AES ciphertext under FHE encryption, i.e., $\sim$ 12 MB $*128 = 1.5$ 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.

## 8 ASCON

**Background.** Ascon [32] stands as an alternative to AES-GCM in the presence of lightweight and low-end devices. Recently, it was selected by NIST for standardization [59]. Additionally, Ascon emerged as the top choice for AE in the CAESAR competition [31]. 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 high-end CPUs but also with FHE. Another advantage 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 natively support 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 $(x_0, \ldots, x_5)$, resulting in a total of 320 bits. The process also 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 Fig. 8), and a linear diffusion layer described by equations 4 to 8. This systematic approach ensures the secure transformation of data during encryption.

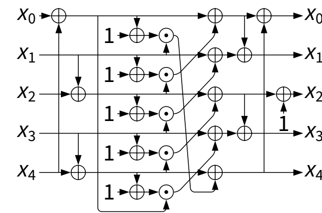


**Figure 8: Schematic representation of the Ascon s-box, image was taken from [32].** $x_0, \ldots, x_4$ **are 64-bit word elements.**

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

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

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

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

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

**Implementing Ascon.** For the purpose of Ascon decryption within the context of FHE, we made a decision to employ a 64-bit word sliced implementation instead of a bit-sliced implementation as we did for AES. This choice was motivated by the fact that the 320-bit state would require the utilization of 320 ciphertexts, resulting in a total size of approximately 12 MB $\times$ 320 $\approx$ 3.84 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 FHE 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 $h_1()$: 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.

## 9 EXPERIMENTS

**Experimental setup.** We considered two platforms for the experiments: a) **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), where we used a single CPU thread by setting

**Table 2: A comparison of decryption methods under FHE.**

| Process unit | Sec bits | Cipher | Size (KB) | Latency (min) | Block size (bits) | Blocks (#) | Amortized lat. (mSec/block) | Peak RAM usage (GB) |
|---|---|---|---|---|---|---|---|---|
| CPU | 128 | Ascon | 4 | 21 | 64 | $2^9$ | 2,460 | 20 |
| | | CTR | 512 | 20 | 128 | $2^{15}$ | 37 | 69 |
| | 256 | CTR | 512 | 28.58 | 128 | $2^{15}$ | 52 | 74 |
| GPU | 128 | Ascon | 4 | 0.43 | 64 | $2^9$ | 50.8 | 22 |
| | | Ascon | 512 | 11.5 | 64 | $2^{16}$ | 10.49 | 23 |
| | | CTR | 512 | **1.65** | 128 | $2^{15}$ | **3** | 14 |
| | | GCM | 512 | **6.95** | 128 | $2^{15}$ | **13** | 14 |
| | 256 | CTR | 512 | 2.3 | 128 | $2^{15}$ | 4.2 | 14 |
| | | GCM | 512 | 7.66 | 128 | $2^{15}$ | 14 | 15 |

OMP_NUM_THREADS=1. and; b) **CPU**: An Intel® Xeon® CPU E5-2699 v4 @ 2.20GHz machine with 44 cores (88 threads) and 750GB memory. 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 FHE. Each experiment underwent 10 repetitions, and the presented outcome reflects the minimum recorded running time. We configured HElayers to utilize a bootstrappable HEaaN context, leveraging the CryptoLab HEaaN library, with a security level of 128 bits. The employed ciphertexts had a capacity of 32,768 slots, facilitating concurrent processing of multiple data elements. The multiplication depth was set to 12, with the fractional part precision (scale) configured to 42 bits, and the integer part precision (number of additional bits in the first prime) set to 18 bits. Furthermore, the chain index after bootstrap was set to 12, while the minimal chain index for bootstrap was established at 3.

**Experiments results.** Table 2 presents[2] the benchmark results for the decryption process of various block ciphers in different modes of operation. For comparison with SotA see Fig. 2 and the discussion in Sec. 2. 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 fully utilizes the GPU. 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. Fig. 9 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 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 FHE 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

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

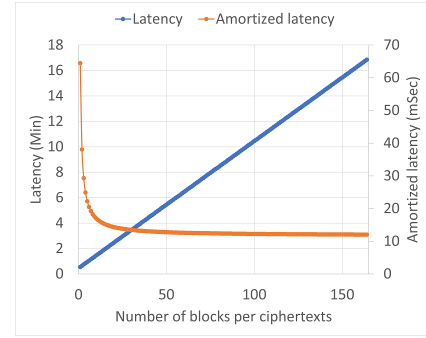


**Figure 9: Latency and amortized latency of Ascon for a batch of 512 ciphertexts and different number of blocks (either AAD or ciphertext data).**

of the CPU implementation for AES-CTR. As depicted in the table, even with 88 threads, the CPU implementation is nearly 12.12× 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 Fig. 2. 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 we implemented Ascon flavor that operates on a block size of 8 bytes. When comparing amortized latency per 16 bytes, the reported value for Ascon 20.98 mSec (by doubling the 10.49 mSec latency per 8 bytes) is higher than that for AES-GCM (13 mSec). Another observation is that in our experiment we used a GPU with a single CPU thread, in practice the computation of the AES-CTR and the GHASH functions can be parallelized, which will result in latency of $6.95 - 1.65 = 5.33$ minutes and amortized latency of 9.76 milli-seconds.

## 10 AN END2END IMPLEMENTATION

Our end-to-end process is illustrated in Fig. 10. In this demonstration, we utilize our implementations of AES-GCM-256 and CKKS. The objective is to perform an inference operation on a DNN, 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 FHE. 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, here, 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, in our case, it was our local file system.

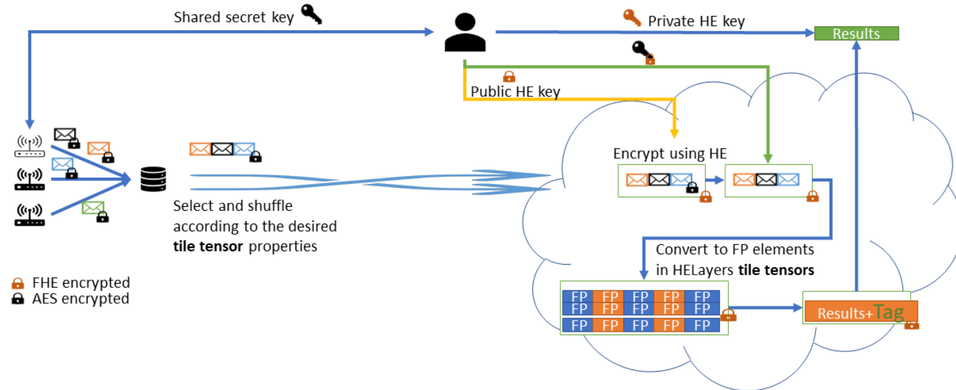


**Figure 10: An illustration of an end-to-end flow using AES-GCM and CKKS.**

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, $[128, \frac{150{,}528*4}{32{,}768}]$, see App. B for more info on tile tensors). In other words, 2 blocks of 128 FHE 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 FHE 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 $IsEq()$ FHE 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 ($AVG(|pt - \text{FHE.Dec}(ct)|)$) observed was $1.16 \times 10^{-10}$, with an even smaller standard deviation of $7.30562 \times 10^{-17}$. These measurements confirm the expected behavior discussed in Sec. 5.

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 discuss later in greater details. 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 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. 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 more insights below

To execute the inference operation, we leveraged the existing AI over FHE capability provided by HElayers [2], as documented in [9]. Specifically, we utilized their pre-trained ResNet-50 model that is compatible with FHE computations and trained on the ImageNet dataset. Notably, the latency and accuracy achieved in our implementation closely aligned with the results reported in [9]. 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.

**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.

Note that not all conversions are feasible due to the inherent error involved in the restoration process. For instance, if the FHE 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.

Alg. 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 FHE configuration and specifically, the error bound $e$.

**Algorithm 1** Constructing numbers from bits

**Input:** $(in)$ an array of $n$ encrypted bits, $(e)$ the error bound.
**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: $b = \left(2^{\lfloor i/2 \rfloor} \cdot in_i\right)^2$
6: $b = IsOdd(i)\ ?\ 2b : b$
7: **else**
8: $b = 2^{i-(4\lfloor i/4 \rfloor)} \cdot \left(2^{\lfloor i/4 \rfloor} \cdot in_i\right)^4$
9: $out = out + b$ ▷ Here, $out = out + 2^i in_i$
10: **return** $out$

**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 due to masking techniques. Consequently, additional bootstraps are typically unnecessary.

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, this approach will not 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.

## 11 DISCUSSION

Our demonstration establishes the feasibility and practicality of an end-to-end AT approach that enables an 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, FHE, and AES encrypted under FHE 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 FHE. 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 [13, 38] instead of CKKS [18] 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 research. We did not explored this option, as such scheme switching implementation is not yet available in modern FHE libraries.

**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 [61]. 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 FHE. In this case the data is encrypted twice once with AES-CTR and another time with the AES-GCM.

**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 also a TLS 1.3 recommended AEAD scheme. 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.

## 12 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 DNN 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 FHE 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 FHE standardization process is finalized, users will be able to adopt standardized hybrid encryption, eliminating certain barriers associated with the adoption of FHE in general use cases.

## REFERENCES


[1] 2018. California Consumer Privacy Act. Statutes of California. https://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201720180AB375 California Assembly Bill 375.

[2] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, et al. 2023. HElayers: A tile tensors framework for large neural networks on encrypted data. *PoPETs* 2023 (2023), 325–342. Issue 1. https://doi.org/10.56553/popets-2023-0020

[3] Ehud Aharoni, Nir Drucker, Gilad Ezov, Eyal Kushnir, Hayim Shaul, and Omri Soceanu. 2023. E2E near-standard hybrid encryption. https://homomorphicencryption.org/6th-homomorphicencryption-org-standards-meeting/ Poster session at 6th HomomorphicEncryption.org Standards Meeting, Seoul, South korea.

[4] Ehud Aharoni, Nir Drucker, Gilad Ezov, Eyal Kushnir, Hayim Shaul, and Omri Soceanu. 2023. Poster: Efficient AES-GCM Decryption Under Homomorphic Encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 3567–3569. https://doi.org/10.1145/3576915.3624377

[5] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2018. *Homomorphic Encryption Security Standard*. Technical Report. HomomorphicEncryption.org, Toronto, Canada. https://HomomorphicEncryption.org

[6] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. 2015. Ciphers for MPC and FHE. In *Advances in Cryptology – EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 430–454. https://doi.org/10.1007/978-3-662-46800-5_17

[7] Tomer Ashur, Mohammad Mahzoun, and Dilara Toprakhisar. 2022. Chaghri - A FHE-Friendly Block Cipher. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 139–150. https://doi.org/10.1145/3548606.3559364

[8] AWS. 2023. AWS Encryption SDK. https://docs.aws.amazon.com/encryption-sdk/latest/developer-guide/faq.html Last accessed Jun 2023.

[9] Moran Baruch, Nir Drucker, Gilad Ezov, Eyal Kushnir, Jenny Lerner, Omri Soceanu, and Itamar Zimerman. 2023. Sensitive Tuning of Large Scale CNNs for E2E Secure Prediction using Homomorphic Encryption. arXiv:2304.14836 [cs.LG]

[10] Adda-Akram Bendoukha, Aymen Boudguiga, and Renaud Sirdey. 2022. Revisiting Stream-Cipher-Based Homomorphic Transciphering in the TFHE Era. In *Foundations and Practice of Security*, Esma Aïmeur, Maryline Laurent, Reda Yaich, Benoît Dupont, and Joaquin Garcia-Alfaro (Eds.). Springer International Publishing, Cham, 19–33. https://doi.org/10.1007/978-3-031-08147-7_2

[11] Joan Boyar, Philip Matthews, and René Peralta. 2013. Logic Minimization Techniques with Applications to Cryptology. *Journal of Cryptology* 26, 2 (2013), 280–312. https://doi.org/10.1007/s00145-012-9124-7

[12] Joan Boyar and René Peralta. 2012. A Small Depth-16 Circuit for the AES S-Box. In *Information Security and Privacy Research*, Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 287–298. https://doi.org/10.1007/978-3-642-30436-1_24

[13] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology – CRYPTO 2012*, Reihaneh Safavi-Naini and Ran Canetti (Eds.), Vol. 7417 LNCS. Springer Berlin Heidelberg, Berlin, Heidelberg, 868–886. https://doi.org/10.1007/978-3-642-32009-5_50

[14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Transactions on Computation Theory* 6, 3 (Jul 2014), 1–36. https://doi.org/10.1145/2633600

[15] D. Canright. 2005. A Very Compact S-Box for AES. In *Cryptographic Hardware and Embedded Systems – CHES 2005*, Josyula R. Rao and Berk Sunar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 441–455. https://doi.org/10.1007/11545262_32

[16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. 2018. Stream Ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression. *Journal of Cryptology* 31, 3 (2018), 885–916. https://doi.org/10.1007/s00145-017-9273-9

[17] Jung Hee Cheon, Jean-Sébastien Coron, Jinsu Kim, Moon Sung Lee, Tancrède Lepoint, Mehdi Tibouchi, and Aaram Yun. 2013. Batch Fully Homomorphic Encryption over the Integers. In *Advances in Cryptology – EUROCRYPT 2013*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 315–335. https://doi.org/10.1007/978-3-642-38348-9_20

[18] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, Springer, Cham, 409–437. https://doi.org/10.1007/978-3-319-70694-8_15

[19] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. 2020. Efficient Homomorphic Comparison Methods with Optimal Complexity. In *Advances in Cryptology – ASIACRYPT 2020*, Shiho Moriai and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 221–256. https://doi.org/10.1007/978-3-030-64834-3_8

[20] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. 2020. Efficient homomorphic comparison methods with optimal complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, Springer, Cham, 221–256. https://doi.org/10.1007/978-3-030-64834-3_8

[21] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology* 33, 1 (2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x

[22] Jihoon Cho, Jincheol Ha, Seongkwang Kim, Byeonghak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. 2021. Transciphering Framework for Approximate Homomorphic Encryption. In *Advances in Cryptology – ASIACRYPT 2021*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 640–669. https://doi.org/10.1007/978-3-030-92078-4_22

[23] Carlos Cid, John Petter Indrøy, and Håvard Raddum. 2022. FASTA – A Stream Cipher for Fast FHE Evaluation. In *Topics in Cryptology – CT-RSA 2022*, Steven D. Galbraith (Ed.). Springer International Publishing, Cham, 451–483. https://doi.org/10.1007/978-3-030-95312-6_19

[24] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. 2014. Scale-Invariant Fully Homomorphic Encryption over the Integers. In *Public-Key Cryptography – PKC 2014*, Hugo Krawczyk (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 311–328. https://doi.org/10.1007/978-3-642-54631-0_18

[25] –. ARM Corporation. 2009. 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.

[26] Orel Cosseron, Clément Hoffmann, Pierrick Méaux, and François-Xavier Standaert. 2022. Towards Case-Optimized Hybrid Homomorphic Encryption. In *Advances in Cryptology – ASIACRYPT 2022*, Shweta Agrawal and Dongdai Lin (Eds.). Springer Nature Switzerland, Cham, 32–67.

[27] Ronald Cramer and Victor Shoup. 2003. Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack. *SIAM J. Comput.* 33, 1 (2003), 167–226. https://doi.org/10.1137/S0097539702403773

[28] CryptoLab. 2023. HEaaN: Homomorphic Encryption for Arithmetic of Approximate Numbers, version 0.2.0. https://www.cryptolab.co.kr/products/heaan-he/

[29] Christophe De Cannière and Bart Preneel. 2008. *Trivium*. Springer Berlin Heidelberg, Berlin, Heidelberg, 244–266. https://doi.org/10.1007/978-3-540-68351-3_18

[30] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. 2018. Rasta: A Cipher with Low ANDdepth and Few ANDs per Bit. In *Advances in Cryptology – CRYPTO 2018*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer International Publishing, Cham, 662–692. https://doi.org/10.1007/978-3-319-96884-1_22

[31] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. 2016. Ascon v1.2. Submission to Round 3 of the CAESAR competition. https://competitions.cr.yp.to/round3/asconv12.pdf

[32] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. 2019. Ascon v1.2. Submission to Round 1 of the NIST Lightweight Cryptography project. https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/ascon-spec.pdf

[33] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. 2021. Pasta: A Case for Hybrid Homomorphic Encryption. Cryptology ePrint Archive, Paper 2021/731. https://eprint.iacr.org/2021/731

[34] Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. 2022. BLEACH: Cleaning Errors in Discrete Computations over CKKS. Cryptology ePrint Archive, Paper 2022/1298. https://eprint.iacr.org/2022/1298

[35] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *Advances in Cryptology – EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 617–640. https://doi.org/10.1007/978-3-662-46800-5_24

[36] Morris Dworkin. 2007. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. https://doi.org/10.6028/NIST.SP.800-38d

[37] EU General Data Protection Regulation. 2016. 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* 119 (2016). http://data.europa.eu/eli/reg/2016/679/oj
[38] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *Proceedings of the 15th international conference on Practice and Theory in Public Key Cryptography* (2012), 1–16. https://eprint.iacr.org/2012/144
[39] Hayato Fujii, Félix Carvalho Rodrigues, and Julio López. 2020. Fast AES Implementation Using ARMv8 ASIMD Without Cryptography Extension. In *Information Security and Cryptology – ICISC 2019*, Jae Hong Seo (Ed.). Springer International Publishing, Cham, 84–101. https://doi.org/10.1007/978-3-030-40921-0_5
[40] Gartner. 2021. *Gartner Identifies Top Security and Risk Management Trends for 2021*. Technical Report. Gartner. https://www.gartner.com/en/newsroom/press-releases/2021-03-23-gartner-identifies-top-security-and-risk-management-t
[41] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. In *Advances in Cryptology – CRYPTO 2012*, Reihaneh Safavi-Naini and Ran Canetti (Eds.), Vol. 7417 LNCS. Springer Berlin Heidelberg, Berlin, Heidelberg, 850–867. https://doi.org/10.1007/978-3-642-32009-5_49
[42] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2015. Homomorphic Evaluation of the AES Circuit. Cryptology ePrint Archive, Paper 2012/099. https://eprint.iacr.org/archive/2012/099/20150103:190644
[43] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. 2023. HELM: Navigating Homomorphic Encryption through Gates and Lookup Tables. Cryptology ePrint Archive, Paper 2023/1382. https://eprint.iacr.org/2023/1382
[44] Lorenzo Grassi, Irati Manterola Ayala, Martha Norberg Hovd, Morten Øygarden, Håvard Raddum, and Qingju Wang. 2023. Cryptanalysis of Symmetric Primitives over Rings and a Key Recovery Attack on Rubato. Cryptology ePrint Archive, Paper 2023/822. https://eprint.iacr.org/2023/822
[45] Martin ?gren, Martin Hell, Thomas Johansson, and Willi Meier. 2011. Grain-128a: a new version of Grain-128 with optional authentication. *International Journal of Wireless and Mobile Computing* 5, 1 (2011), 48–59. https://doi.org/10.1504/IJWMC.2011.044106
[46] Shay Gueron. 2009. Intel's New AES Instructions for Enhanced Performance and Security. In *Fast Software Encryption*, Orr Dunkelman (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 51–66. https://doi.org/10.1007/978-3-642-03317-9_4
[47] Shay Gueron. 2010. Intel® Advanced Encryption Standard (AES) New Instructions Set Rev. 3.01. urlhttps://www.intel.com.bo/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf. *Intel Software Network* (2010).
[48] Jincheol Ha, Seongkwang Kim, Wonseok Choi, Jooyoung Lee, Dukjae Moon, Hyojin Yoon, and Jihoon Cho. 2020. Masta: An HE-Friendly Cipher Using Modular Arithmetic. *IEEE Access* 8 (2020), 194741–194751. https://doi.org/10.1109/ACCESS.2020.3033564
[49] Jincheol Ha, Seongkwang Kim, Byeonghak Lee, Jooyoung Lee, and Mincheol Son. 2022. *Rubato: Noisy Ciphers for Approximate Homomorphic Encryption (Full Version)*. Technical Report Report 2022/537. https://eprint.iacr.org/2022/537
[50] Shai Halevi. 2017. Homomorphic Encryption. In *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, Yehuda Lindell (Ed.). Springer International Publishing, Cham, 219–276. https://doi.org/10.1007/978-3-319-57048-8_5
[51] Phil Hebborn and Gregor Leander. 2020. Dasta – Alternative Linear Layer for Rasta. *IACR Transactions on Symmetric Cryptology* 2020, 3 (Sep. 2020), 46–86. https://doi.org/10.13154/tosc.v2020.i3.46-86
[52] IBM. 2020. IBM z15 Performance of Cryptographic Operations. https://www.ibm.com/downloads/cas/6K2653EJ
[53] Simon Johnson, Vincent Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. 2016. Intel ® Software Guard Extensions: EPID Provisioning and Attestation Services. *White Paper* (April 2016). https://software.intel.com/sites/default/files/managed/ac/40/2016%20WW10%20sgx%20provisioning%20and%20attesatation%20final.pdf
[54] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, 12403–12422. https://proceedings.mlr.press/v162/lee22e.html
[55] Baiyu Li and Daniele Micciancio. 2021. On the Security of Homomorphic Encryption on Approximate Numbers. In *Advances in Cryptology – EUROCRYPT 2021*, Anne Canteaut and François-Xavier Standaert (Eds.). Springer International Publishing, Cham, 648–677. https://doi.org/10.1007/978-3-030-77870-5_23
[56] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2010. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology – EUROCRYPT 2010*, Henri Gilbert (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–23. https://doi.org/10.1007/978-3-642-13190-5_1
[57] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. 2016. Towards Stream Ciphers for Efficient FHE with Low-Noise Ciphertexts. In *Advances in Cryptology – EUROCRYPT 2016*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 311–343. https://doi.org/10.1007/978-3-662-49890-3_13
[58] NIST. 2001. FIPS PUB 197: Advanced encryption standard (AES). , 311 pages. https://doi.org/10.6028/NIST.FIPS.197
[59] NIST. 2023. Lightweight Cryptography. https://csrc.nist.gov/Projects/lightweight-cryptography Last accessed 28 June 2023.
[60] Chester Rebeiro, David Selvakumar, and A. S. L. Devi. 2006. Bitslice Implementation of AES. In *Cryptology and Network Security*, David Pointcheval, Yi Mu, and Kefei Chen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 203–212. https://doi.org/10.1007/11935070_14
[61] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://doi.org/10.17487/RFC8446
[62] N P Smart and F Vercauteren. 2014. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography* 71, 1 (2014), 57–81. https://doi.org/10.1007/s10623-012-9720-4
[63] ICT standardization committee. 2019. Homomorphic Encryption for Arithmetic of Approximate Numbers. https://committee.tta.or.kr/data/standard_view.jsp?secondDepthCode=PG501&firstDepthCode=TC5&pk_num=TTAK.KO-12.0347&commit_code=PG501 TTAK.KO-12.0347, Telecommunications Association Standard (TTAS).
[64] Roy Stracovsky, Rasoul Akhavan, and Florian Mahdavi Kerschbaum. 2022. Faster Evaluation of AES using TFHE. https://drive.google.com/file/d/1WMBjjM416BXGoiLf16gPn6q5aLt4zZqi/view FHE.org 2022, Last accessed July 2023.
[65] The HEBench Organization. 2022. HEBench. https://hebench.github.io/
[66] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. 2023. A Homomorphic AES Evaluation in Less than 30 Seconds by Means of TFHE. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (, Copenhagen, Denmark,) *(WAHC '23)*. Association for Computing Machinery, New York, NY, USA, 79–90. https://doi.org/10.1145/3605759.3625260
[67] Ruida Wang, Yundi Wen, Zhihao Li, Xianhui Lu, Benqiang Wei, Kun Liu, and Kunpeng Wang. 2024. Circuit Bootstrapping: Faster and Smaller. Cryptology ePrint Archive, Paper 2024/323. https://eprint.iacr.org/2024/323
[68] Benqiang Wei, Ruida Wang, Zhihao Li, Qinju Liu, and Xianhui Lu. 2023. Fregata: Faster Homomorphic Evaluation of AES via TFHE. In *Information Security*, Elias Athanasopoulos and Bart Mennink (Eds.). Springer Nature Switzerland, Cham, 392–412.

# 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 FHE, even when the binary values $x_0, \ldots, x_{\lceil \log_2 n \rceil}$ are encrypted.

A straightforward approach to calculate $v_x$ is as follows:

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

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 $\overline{x_i} = 1 - x_i$ then

$$v_x = \left(\overline{x_{\log n}} \cdots \overline{x_1}\,\overline{x_0}\right) v_0 + \left(\overline{x_{\log n}} \cdots \overline{x_1} x_0\right) v_1 +$$
$$\left(\overline{x_{\log n}} \cdots \overline{x_2} x_1 \overline{x_0}\right) v_2 + \left(\overline{x_{\log n}} \cdots \overline{x_2} x_1 x_0\right) v_3 +$$
$$\ldots + \left(x_{\log n} \cdots x_1 x_0\right) v_n$$
$$= \sum c_{m_i} m_i,$$

where $m_1, \ldots, m_n$ represent $n$ monomials, namely

$$1, x_0, x_1, \ldots, x_{\log n}, x_0 x_1, \ldots, x_0 x_1 \ldots x_{\log n}$$

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.1 Computing all monomials efficiently

To compute all the monomials

$$1, x_1, \ldots, x_{\log n}, x_1 x_2, x_1 x_3, \ldots, x_1 x_2 \ldots x_{\log n}$$

, a recursive approach can be employed.

We start with the given monomials $1, x_1, \ldots, x_{\log n}$ 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 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.

## A.2 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 Sec. 4.2. 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.

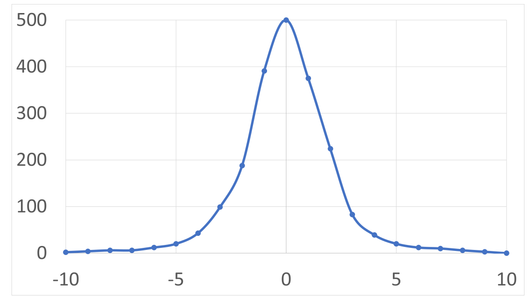


**Figure 11: The number of monomials (y-axis) with a coefficient $c_m$ (x-axis) dervied from the values of the AES S-box.**

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 Sec. 4.2, 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. 11 shows the distribution of the coefficients.

Based on our evaluation, our proposed methodology exhibited a moderate advancement over the implementation of Sec. 4.2 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 Sec. 9, and include it solely as an appendix for the sake of completeness.

# B TILE TENSORS

Ciphertext packing involves how plaintext data is packed within one or more FHE ciphertexts before encryption. This packing can range from simple major-row packing to more complex methods like tile tensors [2]. The chosen packing method significantly affects the speed of function evaluation. For instance, if a plaintext vector $x = (1, 2, 3, 4, 5, 6, 7, 8)$ is packed into one ciphertext $c$ encrypting all 8 elements in parallel in a SIMD fashion, evaluating $f(x) = x^2$ only requires one multiplication, $res = c^2$. Conversely, if each element of $x$ is packed into a separate ciphertext $c_1, \ldots, c_8$, the number of required multiplications increases with the number of ciphertexts, $res = (c_1^2, c_2^2, \ldots, c_8^2)$.

Tile tensors [2] aims to address the optimized packing challenge. It is a data structure that involves various packing algorithms, metadata for describing these algorithms, and generalized operators for performing operations on packed data. Tile tensors consist of a tensor packed within tiles, where each tile is a one-dimensional vector operated on in SIMD fashion. In the context of FHE, every tile is a ciphertext. The shape of a tile tensor describes both the shape of the packed tensor inside it and the packing details. For instance, a tile tensor $T_M$ with the shape $[\frac{5}{2}, \frac{6}{4}]$ signifies that $T_M$ packs a tensor (matrix) $M$ of shape $5 \times 6$ in multiple tiles, each has

a shape of $2 \times 4$. In the example above the required number of tiles is six arranged in a two dimensional array of size $\lceil \frac{5}{2} \rceil \times \lceil \frac{6}{4} \rceil = 3 \times 2$.

As another example, consider a tile tensor $S_M$ of shape $[\frac{5}{1}, \frac{6}{8}]$ that also has $M$ as its logical packed tensor but this time in 5 tiles of shape $1 \times 8$, one tile per row of $M$. Here, each row of $M$ is stored in its own separate tile, and because the tile size is 8, only six slots are filled with data from the original tensor, where the rest are set to 0.

Finally, a tile tensor $T_V$ of shape $[\frac{5}{2}, \frac{*}{4}]$ means that a vector $V$ of shape $5 \times 1$ is packed in tiles of shape $2 \times 4$, where the $*$ indicates that the vector is duplicated 4 times over the second dimension. Consequently, the total number of tiles is $\lceil \frac{5}{2} \rceil \times 4 = 3 \times 4$. To understand tile tensor and its notation better we refer the reader [2].

## C HANDLING AUTHENTICATION TAGS UNDER CKKS.

In a CKKS-based AT, handling the AEAD tag requires comparing the recomputed tag with the input tag under FHE. This comparison yields an encryption of an approximated selector $c_{sel} = \text{FHE.Enc}(sel)$, where $sel \in [-\epsilon, \epsilon] \cup [1-\epsilon, 1+\epsilon]$, i.e., in $\{0, 1\}$ with some small noise $\epsilon$. Subsequently, the AT should return

$$c_{auth} = \begin{cases} \text{FHE.Enc}(0) & \text{FHE.Dec}(c_{sel}) \in [\epsilon, \epsilon] \\ \text{FHE.Enc}(m) & \text{FHE.Dec}(c_{sel}) \in [1-\epsilon, 1+\epsilon] \end{cases}$$

where $m$ is the decrypted AES message, and $c = \text{FHE.Enc}(m)$.

In exact schemes, where $\text{FHE.Dec}(c_{sel}) \in \{0, 1\}$ such as TFHE [21] or BGV [14], this is simple, by just performing $c_{auth} = c \cdot c_{sel}$. However, in CKKS the situation is more complex. Here, when $\text{FHE.Dec}(c_{sel}) \in [1-\epsilon, 1+\epsilon]$ then $\text{FHE.Dec}(c_{auth}) = \text{FHE.Dec}(c) + \epsilon'$, which reveals the expected and allowed AES decryption data up to some small $\epsilon$. However, when $\text{FHE.Dec}(c_{sel}) \in [-\epsilon, \epsilon]$ then $c_{auth} = \text{FHE.Enc}(0 + \epsilon')$, where $\epsilon'$ may leak information on $m$ (the corrupted AEAD plaintext), which is forbidden by the AEAD definition.

To prevent the above leakage we took a different approach, instead of zeroising $c$ we have decided to add a large uniform random data $r$ (e.g., in the range $[0, 2^{64}]$) independent of $c$ to mask it:

$$c_{auth} = \begin{cases} c + r & c_{sel} = \text{FHE.Enc}(0) \\ c + 0 & c_{sel} = \text{FHE.Enc}(1) \end{cases}$$

One way to perform this operation is by multiplying $r$ with $1 - c_{sel}$. Unfortunately, this approach does not work. When $c_{sel} = \text{FHE.Enc}(1)$ then $1 - c_{sel} = \text{FHE.Enc}(0)$ and when $\epsilon$ is larger than a particular bound, e.g., $\epsilon > 2^{-32}$, the result $r \cdot \text{FHE.Enc}(0)$ is an ancrpytion of a number in $[2^{-32}, 2^{32}]$ which when added to $c$ can destroy the value of the underlying plaintext message $m$. In addition, when $c_{sel} = \text{FHE.Enc}(0)$ then $1 - c_{sel} = \text{FHE.Enc}(1)$, also here a large $\epsilon$ may leak information on the key used during the tag computation.

Consequently, we decided to use the following alternative. We first generate a random number $r$ represented as a vector of bits. We multiply every bit of $r$ by $1 - c_{sel}$ and clean it using BLEACH to maintain the error bounds. Subsequently, we apply Algorithm 1 to construct $r$ from its masked bits. The results are FHE.Enc(0) when $c_{sel} = 1$ and FHE.Enc($r$) otherwise as expected. The fact that Alg. 1 may yield approximated results is not an issue as $r$ is expected to be chosen uniformly at random.

Remark 6. *An AT system should return an indicator to the user whether the data is valid or not because even after zeroising or destructing corrupted AEAD data, the user cannot necessarily distinguish an FHE decrypted 0 or an FHE decrypted random value from valid data. Nevertheless, the AT system should also aim to nullify the data before using it in further computation to avoid leaking information through the accumulated error values. It might seem that because the data is FHE encrypted, no harm can be done by leaving the nullification process to the user. However, when the same FHE key pair is used within many different processes an adversary may capture an FHE ciphertext that includes corrupted AEAD data, move it to a different process, and integrate it into the system using the malleability property of FHE. Consequently, it is desired to nullify the FHE ciphertexts as soon as the tag-checking process ends, to minimize the potential attack surface.*

*In contrast, when multiple FHE ciphertexts are required to be nullified, it may be preferred to defer the nullification process, until the evaluated circuits reach a point where it has a smaller number of ciphetexts to nullify. The situation gets even more complicated when multiple tags are involved, e.g., in a system that involves multiple AEAD ciphertext inputs. This security-latency tradeoff as well as other tradeoffs should be considered in the threat model of every different applications.*