Seamless Switching Between PBS and WoPBS for Scalable TFHE

Rostin Shokri and Nektarios Georgios Tsoutsos

University of Delaware, Newark DE 19716, USA
{rostinsh, tsoutsos}@udel.edu

Abstract. Fully Homomorphic Encryption (FHE) enables arbitrary computation directly on encrypted data. The TFHE scheme supports encryption of bits or small integers, evaluating any univariate function via programmable bootstrapping (PBS), which also refreshes ciphertext noise. Since both linear and nonlinear functions can be expressed with PBS, arbitrary circuits of unlimited depth can be computed without accuracy loss, aside from a negligible failure probability. However, a key limitation of TFHE is that it processes only single encrypted messages of small size; for larger messages, PBS becomes prohibitively expensive, as its cost grows rapidly with message bit-length. To address this, Without-Padding PBS (WoPBS) allows evaluation of larger lookup tables, and is practical up to 28 bits, offering significantly reduced latency compared to pure PBS. Recent advances in the WoPBS workflow have further optimized Circuit Bootstrapping (CBS) operations, narrowing the latency gap to PBS. Still, minimizing the underlying number of PBS operations remains crucial, as PBS dominates the cost.

In this work, we introduce novel switching algorithms to efficiently convert ciphertexts between the PBS and WoPBS contexts, carefully managing noise growth while minimizing PBS invocations. We further integrate a state-of-the-art noise reduction method into the WoPBS workflow, enabling more efficient parameter sets. Our optimizations reduce AES transciphering latency by up to 18% relative to the best prior work purely through parameter tuning, and deliver a 2× speedup for large lookup tables by reducing PBS overheads. Notably, our CUDA implementation of the optimized WoPBS workflow achieves up to 50× speedup for AES transciphering over single-threaded CPU baselines.

Keywords: Fully Homomorphic Encryption, Programmable Bootstrapping, TFHE, Circuit Bootstrapping.

1 Introduction

A growing number of applications in today's digital ecosystem run in the cloud, spanning domains such as modern machine learning tasks [25] and genomic analysis in healthcare [27]. By processing sensitive user data in the cloud, these applications have become prime targets for recent side-channel attacks [39,40] capable

of exfiltrating private information to adversaries. Thus, clients seek to preserve the privacy of their data from both attackers and the cloud itself while benefiting from the computational capabilities provided by these remote services. Although traditional cryptographic algorithms, such as AES [13], can safeguard users' data during storage and transmission, the encrypted data cannot be processed without accessing the decryption key, leaving it vulnerable to these side-channel threats.

To address these concerns, a promising approach is fully homomorphic encryption (FHE) [18], which enables unlimited and arbitrary computations on encrypted data without requiring knowledge of the secret key or the underlying message. Numerous FHE schemes have been proposed in the literature, each tailored for efficient operations on specific data types, with unique primitives supporting specialized homomorphic computations. One such scheme that offers the ability for arbitrary computation is TFHE [11], which has undergone major improvements in recent years [12], [2]. In particular, the TFHE scheme is suitable for non-linear operations by leveraging the programmable bootstrap (PBS) operation to compute any univariate function on a ciphertext encrypting a small integer. TFHE employs much smaller lattice parameters than other word-wise FHE schemes, making PBS operations highly efficient. Additionally, FHE programs composed of PBS operations are highly parallelizable and can achieve much faster evaluation times on specialized hardware such as the GPU [21].

While PBS is a powerful tool in the realm of encrypted computations, it is practical only for messages with small precision, typically up to 4 to 6 bits [11]. The execution time of PBS increases exponentially with the number of precision bits (more than doubling with each additional bit after 4 bits), rendering it impractical for larger message sizes. Indeed, many early applications of TFHE focused on encrypting individual bits [10], relying solely on gate bootstrapping to evaluate arbitrary homomorphic boolean circuits [14,20]. However, this approach proved impractical in most real-world scenarios, since arbitrary programs translate into very large binary circuits that quickly become unmanageable for mid-sized workloads.[1] More recent advancements, including AES Transciphering [33] and TFHE libraries [38], have adopted ciphertexts encrypting small numbers; in particular, 4-bit precision is shown to yield optimal execution times for their respective use cases. These efforts optimize parameters and introduce highly parallel operations, such as integer addition and multiplication, by representing large-precision messages in a radix format [2]. This enables much faster execution of arbitrary circuits, which are now composed of small lookup tables instead of binary gates, thereby significantly reducing the depth and width of the circuit.[2]

Nonetheless, applying arbitrary functions to large messages remains a major challenge, as it exceeds the practical capabilities of PBS. To address this challenge, the work in [23] proposed a tree-based method for computing univariate

[1] A single 32-bit multiplication takes 32 seconds on 48 CPU cores using the Google Transpiler [22, Figure 6].

[2] The work in [22] reports 33× faster multiplication than the Google Transpiler [20].

functions on large messages. While this tree-based technique sets a powerful theoretical foundation, its main drawback is poor scalability: the required number of small PBS computations still grows exponentially with the bit size of the larger message. In addition, expensive public key switching operations are required to generate intermediate encrypted LUTs.

A new computing paradigm in TFHE, dubbed *Without Padding PBS (WoPBS)*, has recently emerged [2]. This approach relies on two main operations, namely Circuit Bootstrapping (CBS) and Vertical Packing (VP), and provides much stronger computing capabilities than the traditional PBS. The CBS operation converts LWE ciphertexts to GGSW ciphertexts, while VP utilizes CMUX operations to compute lookup tables on the converted ciphertexts. CBS uses PBS as one of its operations, which refreshes the noise and makes this computing paradigm fully homomorphic. With WoPBS, lookup tables up to 28 bits can be computed relatively efficiently. Notably, one major drawback of the original CBS proposal [11] was its high latency, being nearly $10\times$ slower than a PBS operation, and was later iteratively improved, with recent refinements [34] bringing this overhead to less than $2\times$.

While the improved CBS operation holds significant potential, the WoPBS workflow still faces several limitations that hinder its widespread adoption in TFHE-based applications. In this work, we identify key challenges that must be addressed to enhance the practicality and applicability of WoPBS across a broad range of use cases:

Inconsistent Message Domain: The CBS operation requires the input LWE ciphertexts to be in binary format, only containing a single message bit, with no padding bits. Since the PBS workflow maximizes its efficiency while operating on small integers (plus a bit of padding), switching from an LWE ciphertext in the PBS context to LWE ciphertexts encrypting a single bit in the WoPBS context (required by CBS), and vice versa, becomes expensive and challenging. On top of that, the WoPBS workflow would improve significantly if it could operate more efficiently on small integers by utilizing fewer PBS operations, thereby achieving an efficiency comparable to the standard PBS workflow (since PBS is the most costly step in CBS).

Large Ciphertext Expansion: The converted GGSW ciphertexts are a few times larger than their LWE counterparts. The corresponding memory overhead can be challenging, given that existing parameter sets choose a large decomposition to combat the high noise growth of the CBS and VP operations. Thus, judiciously defined parameters can help minimize this ciphertext expansion.

Increased Noise Growth and Failure Probability: A crucial step in CBS, called "`PBSmanyLUT`" in [12], incurs a high probability of failure due to the large noise growth of its modulus switching. Reducing its noise growth in conjunction with choosing appropriate parameters can result in improved performance targeting the same failure probability.

Incompatibility with Existing Applications: Numerous applications and operations are already optimized for the classic PBS, so applying it alongside the WoPBS computation can have major benefits, such as much faster evaluation

of larger LUTs. Additionally, certain operations are noticeably more efficient in one context than in the other. For instance, the `XOR` operation, which is critical in many applications, including AES Transciphering (e.g., mix-columns can be computed with only `XOR`), is computationally expensive in the PBS context since it requires many PBS and expensive public key switching operations. Conversely, in the WoPBS context, if the LWE ciphertexts are binary, `XOR` can be performed using simple native LWE additions that incur negligible cost compared to a PBS.

To address all these limitations, we optimize and improve the WoPBS workflow through new and efficient switching algorithms that convert LWE ciphertexts seamlessly between the PBS and WoPBS contexts, while minimizing the required PBS operations for high efficiency. We also adopt a state-of-the-art noise reduction methodology, called *mean compensation* [31], to further reduce the noise growth of the CBS steps in the WoPBS workflow (e.g., `PBSmanyLUT`), as well as in our proposed switching algorithms. Our approach enables minimizing the total PBS operations while lowering the corresponding memory overheads. Overall, in this work, we claim the following contributions:

- A highly efficient decomposition and recomposition methodology (Alg. 3, 5) that converts LWE ciphertexts between the PBS and WoPBS context with the minimum amount of required PBS operations.
- Formulated the mean noise reduction technique into the WoPBS workflow and our switching algorithms to reduce noise growth in critical core operations, enabling more efficient parameter generation and higher-precision PBSmanyLUT operations.
- Designed CUDA-friendly core operations of the memory-intensive WoPBS workflow with minimal memory overhead, which enables 76× faster runtime in AES transciphering (compared to a CPU baseline).

Roadmap: The rest of the paper is organized as follows: In Section 2, we present essential background information on fully homomorphic encryption (FHE), the TFHE scheme, as well as the structure and primitive operations over ciphertexts. Section 3 introduces our methodology for efficiently converting between the PBS and WoPBS contexts and describes our novel approach for accelerating various workflows, including our noise-reduced `PBSmanyLUT` operation. In Section 4, we present our experimental evaluation based on AES transciphering on both CPU and GPU platforms, and report 2× speedup for large LUT evaluation over the current state-of-the-art. Finally, Section 5 discusses notable prior art, and Section 6 presents our concluding remarks.

2 Background

2.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) enables meaningful computations on encrypted data without the need for decryption during processing. The security of modern FHE schemes relies on the concept of noise, which is intentionally

injected into each ciphertext. With every homomorphic operation, this noise increases and, when it exceeds a certain threshold, decryption yields incorrect results. To mitigate this, an operation known as "bootstrapping" is employed. Bootstrapping homomorphically evaluates the decryption function on a ciphertext, effectively reducing its noise and enabling continued computation. Most FHE schemes rely on hard problems, such as the Learning with Errors (LWE) problem [30] and its ring-based variant (RLWE) [29]. These problems are assumed to be intractable for both classical and quantum algorithms. Furthermore, all FHE schemes support both addition and multiplication, which allows for the composition of arbitrary functions in the encrypted domain.

Various FHE schemes have been proposed that support *integer* operations. The BFV [16] and BGV [5] are two such schemes where integer values are batched together into a single RLWE ciphertext, permitting both addition and multiplication on these ciphertexts. The CKKS scheme was also introduced with capabilities similar to BFV and BGV, but it focuses on real numbers [9]. One of the first *bit-wise* schemes is FHEW [15], which is based on the LWE and RLWE problems, and operates on single-bit messages by encrypting each bit in a separate LWE ciphertext. This design enables low-latency, gate-level logic computations. FHEW employs a noise-management process called *gate bootstrapping*, which can perform logical operations, such as `XOR` and `AND`, while simultaneously refreshing the noise in the ciphertext.

Building on FHEW, the TFHE scheme [11] introduced a more efficient bootstrapping technique known as programmable bootstrapping (PBS). TFHE employs binary secret keys and introduces a faster Blind Rotation technique, which is a core operation for PBS, and can evaluate any univariate function while refreshing ciphertext noise. Using standard parameters, TFHE can perform PBS in roughly 10 milliseconds on a CPU. Additionally, TFHE can encrypt small integers (i.e., 4-6 bits) by utilizing significantly smaller lattice parameters than those required in word-wise schemes like CKKS.

2.2 The Transciphering Technique

Transciphering enables secure and efficient homomorphic processing of data that was originally encrypted under a standard (non-homomorphic) scheme, by adding a homomorphic encryption layer before "peeling off" the original encryption. For example, transciphering can convert a ciphertext generated using a symmetric cipher such as AES [13] into an FHE ciphertext by *homomorphically evaluating the decryption circuit* of the underlying symmetric scheme [19]. Following this conversion, FHE operations can be performed directly on the transciphering output. In this example, the AES secret key needs to be encrypted under FHE.

This approach is particularly beneficial for resource-restricted clients in scenarios where data is initially encrypted using fast symmetric schemes to optimize storage and transmission efficiency. Since FHE ciphertexts incur a significant size expansion compared to their plaintext counterparts, transciphering offers

an effective method to reduce storage and communication costs (before any processing), albeit at the expense of additional execution time [22].

2.3 TFHE Ciphertexts

In TFHE, ciphertexts are structured around two fundamental representations that form the backbone of its homomorphic operations: the standard LWE ciphertext and its polynomial counterpart, the GLWE ciphertext. In the following paragraphs, we detail these representations, starting with the LWE ciphertext, which serves as a building block for more advanced constructions.

LWE ciphertexts These ciphertexts operate over the finite ring $\mathbb{Z}_q$, where a message $\mu \in \mathbb{Z}_q$ is embedded in the MSBs using a scaling factor $\Delta \in \mathbb{Z}_q$, and encrypted under a binary secret key $\mathbf{s} \in \{0,1\}^n$. An error e, sampled from a discretized Gaussian distribution χ_σ and rounded to an integer, is added to the LSBs. Concretely, an LWE ciphertext is defined as a pair:

$$\texttt{LWE}_s(\mu \cdot \Delta) = (\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + \mu \cdot \Delta + e) \in \mathbb{Z}_q^n \times \mathbb{Z}_q,$$

where:

- q is the ciphertext modulus,
- $\mathbf{a} \in \mathbb{Z}_q^n$ is a vector sampled uniformly at random, called the *mask*,
- $b \in \mathbb{Z}_q$ is the scalar component, called the *body*,
- $e \in \mathbb{Z}_q$ is an error term sampled from χ_σ,
- $\langle \mathbf{a}, \mathbf{s} \rangle$ denotes the dot product over $\mathbb{Z}_q$.

In this case, all operations are performed modulo q. Decryption proceeds by computing $b - \langle \mathbf{a}, \mathbf{s} \rangle \mod q$, before rounding to the nearest $\mu \cdot \Delta$ value, where μ is a small integer. The decryption algorithm will return correct results assuming the following inequality holds: $|e| < \frac{\Delta}{2}$.

GLWE ciphertexts. The Generalized Learning With Errors (GLWE) scheme extends LWE to the polynomial ring $\mathbb{Z}_q[X]/(X^N + 1)$, enabling the encryption of polynomial messages. A message $M(X) \in \mathbb{Z}_q[X]/(X^N + 1)$ is scaled by a factor $\Delta \in \mathbb{Z}_q$ and embedded in the most significant bits of its coefficients. The secret key is a vector of binary polynomials $\mathbf{S}(X) = (s_0(X), \ldots, s_{k-1}(X)) \in (\{0,1\}[X]/(X^N + 1))^k$. The error term is a polynomial vector with small coefficients sampled from a discretized Gaussian distribution χ_σ. A GLWE ciphertext is defined as:

$$\texttt{GLWE}_{\mathbf{S}}(M \cdot \Delta) = \big(\mathbf{A}(X),\ B(X) = \langle \mathbf{A}(X), \mathbf{S}(X) \rangle + M(X) \cdot \Delta + E(X)\big),$$

where:

- $\mathbf{A}(X) = (a_0(X), \ldots, a_{k-1}(X)) \in (\mathbb{Z}_q[X]/(X^N + 1))^k$ is a vector of polynomials sampled uniformly at random,
- $E(X) \in \mathbb{Z}_q[X]/(X^N + 1)$ is an error polynomial with small coefficients sampled from χ_σ,
- $\langle \mathbf{A}(X), \mathbf{S}(X) \rangle = \sum_{i=0}^{k-1} a_i(X) \cdot s_i(X)$ is the polynomial inner product,

Here, all operations are carried out over $\mathbb{Z}_q[X]$ with reduction modulo $X^N + 1$ and coefficient-wise reduction modulo q. Decryption proceeds by computing $B(X) - \langle \mathbf{A}(X), \mathbf{S}(X) \rangle$, before rounding each coefficient to the nearest multiple of $\Delta \cdot M$, where M consists of N small integers, under the assumption that all noise coefficients of $E(X)$ are bounded by $\Delta/2$.

Modulus Switching The PBS operation requires the LWE ciphertexts to be reduced modulo $2N$, where $N \ll q$. The modulus switching step is given by:

$$a'_i = \left\lfloor a_i \cdot \frac{2N}{q} \right\rceil \bmod 2N, \tag{1}$$

where:

- a_i denotes the i-th coefficient (ring element) of the LWE ciphertext,
- $\lfloor \cdot \rceil$ denotes rounding to the nearest integer.

The `PBSmanyLUT` operation introduced in [12] utilizes a different modulus switching step, given by:

$$a' = \left\lfloor \frac{a \cdot 2N \cdot 2^{-\vartheta}}{q} \right\rceil \cdot 2^{\vartheta} \bmod 2N, \tag{2}$$

where ϑ is the number of LSBs set to 0. Note that the modulus switching algorithm in [12] also depends on a parameter $\varkappa$, representing the number of MSBs omitted from the PBS. In this work, we set $\varkappa = 0$, as we do not use this feature.

Programmable Bootstrap (PBS) A PBS is a three-stage operation in TFHE that simultaneously refreshes a noisy LWE ciphertext and evaluates an arbitrary function f on the encrypted message μ_{in}. The general procedure is outlined in Algorithm 1: First, a modulus switching operation is performed to reduce the modulus to $2N$, where N is the ring dimension of the GLWE ciphertext. Next, the Blind Rotation algorithm is applied to the modulus-switched ciphertext using a GLWE accumulator that encodes the desired lookup table corresponding to the function f. Finally, the first coefficient of the rotated GLWE ciphertext is extracted, yielding an LWE ciphertext under the GLWE secret key S that encrypts $f(\mu_{in})$. A homomorphic Multiplexer operation CMUX, defined in [11], is used to perform the Blind Rotation operation. Additional details on the Blind Rotation step can be found in [11].

`PBSmanyLUT` follows the same steps as Algorithm 1, except that it uses the modulus switching step in Equation 2. The LUT can encode 2^{ϑ} functions in the

Algorithm 1 Programmable Bootstrapping (PBS)

Input: LWE ciphertext $\texttt{LWE}_{s,q}(\mu_{in} \cdot \Delta_{in})$
Input: Bootstrapping key $\texttt{bsk} = \{\text{GGSW}(s_i)\}_{i=0}^{n-1}$
Input: GLWE ciphertext `lut` encoding function f with scaling factor Δ_{out}
Output: LWE ciphertext $\texttt{LWE}_{S,q}(f(\mu_{in}) \cdot \Delta_{out})$
1: $\texttt{ct}' \leftarrow \texttt{ModSwitch}(\texttt{LWE}_{s,q}(\mu_{in} \cdot \Delta_{in}))$
2: $\texttt{rotated\_lut} \leftarrow \texttt{BlindRotate}(\texttt{ct}', \texttt{bsk}, \texttt{lut})$
3: $\texttt{ct\_out} \leftarrow \texttt{SampleExtract}(\texttt{rotated\_lut})$
4: **return** `ct_out`

same accumulator, from which the first 2^ϑ LWE ciphertexts of the rotated LUT are extracted using `SampleExtract`. Since the ϑ LSBs are set to zero during modulus switching, when rotating the accumulator, the LUT will rotate in steps of 2^ϑ, instead of 1 in the normal PBS. Therefore, at the end of the Blind Rotation, the first 2^ϑ slots will hold the outputs of the target functions.

Multi-Value Bootstrapping (MVB) The MVB operation, introduced in [6], works by first evaluating a base function f through a Blind Rotation, producing a GLWE ciphertext C that encodes f without Sample Extraction. For $i = 0, \ldots, k-1$, let

$$f_i \;=\; G_i \cdot f$$

be the i-th target function, where $G_0, G_1, \ldots, G_{k-1}$ are fixed polynomials and k corresponds to the number of additional functions to evaluate. Given the ciphertext C encoding f, the ciphertext encoding f_i is obtained as

$$C_i \;=\; G_i \cdot C.$$

One downside to MVB is the additional noise generated by this multiplication, which is quantified by the squared ℓ_2 norm of G_i. In particular, if the input ciphertext C has variance V_{pbs}, then the output variance is

$$V_{\text{out}} \;=\; \|G_i\|_2^2 \cdot V_{\text{pbs}},$$

where $\|G_i\|_2^2 = \sum_j g_{i,j}^2$ is the squared ℓ_2 norm of the coefficient vector of G_i.

Circuit Bootstrapping (CBS) The CBS operation requires the input bits to be encrypted in separate LWE ciphertexts, which are then converted to GGSW ciphertexts. These ciphertexts serve as inputs to an operation called `Vertical Packing` [11], which consists of a Blind Rotation step and, if needed, a CMUX tree, and produces an LWE ciphertext containing the encrypted output. The CBS operation then takes as input $\texttt{LWE}_s(m \cdot \Delta)$, where m is a single bit and $\Delta = \frac{q}{2}$ (with q being the ciphertext modulus), and outputs a GGSW ciphertext.

A GGSW ciphertext is defined by the decomposition parameters ℓ and β, which denote the number of decomposition levels and the decomposition base,

respectively. Each GGSW ciphertext is composed of $(k+1)\ell$ GLWE ciphertexts, each encrypting the binary message m under the GLWE secret key $S = (S_0, \ldots, S_k)$. Note that the message m does not necessarily have to be binary, but for the CBS algorithm, we assume this to be the case. We represent a GGSW ciphertext as $\texttt{GLWE}_S(m \cdot S_i \cdot \frac{q}{\beta^j})$, where $1 \leq j \leq \ell$ and $0 \leq i \leq k$. Here, $j \in \{1, \ldots, \ell\}$ indexes the decomposition level, and β^j denotes the base raised to the j-th power.

The original CBS, proposed in [11], uses ℓ PBS operations to convert an LWE ciphertext of the form $\texttt{LWE}_s(m \cdot \Delta)$ into ℓ LWE ciphertexts $\texttt{LWE}_s(m \cdot \frac{q}{\beta^j})$. Next, the algorithm computes $(k+1)\ell$ private functional key switching operations (see [11, Alg. 3]) to multiply the underlying value $\frac{q}{\beta^j}$ of these LWE ciphertexts by the corresponding element of the secret key S_i. This produces the final `GGSW` ciphertext.

2.4 State-of-the-Art CBS

Recent work in [36,34] has significantly improved the CBS latency by replacing the costly private functional key switching with more efficient key switching and scheme switching operations. This approach divides the CBS into two steps: The first is a refresh step, where the PBS operation is used to compute scaled LWE ciphertexts of the same message bit. The second is a conversion step, where the scaled LWE ciphertexts are converted to a GGSW ciphertext. Here are these steps in more detail:

- **PBSmanyLUT**: Evaluates the scaled LWE ciphertexts using only a single Blind Rotation to accelerate the refresh step. Sample Extraction is not used in this case.
- **Homomorphic Trace (HomTrace)**: This step utilizes GLWE key switching to preserve only the monomial terms that contain our message bits, zeroing out all other terms.
- **Scheme Switching (SS)**: Uses an external product to multiply the secret key by the previous GLWE ciphertexts, generating valid GGSW ciphertexts.

Additional details about these algorithms are available in [36], and [34].

2.5 Vertical Packing (VP)

Given GGSW ciphertexts, each holding a single message bit, a lookup table (LUT) can be computed using CMUX operations via VP. If the number of input GGSW ciphertexts m is less than or equal to $\log_2(N)$, where N is the ring dimension of the underlying polynomials, only a Blind Rotation is used to compute the lookup table. Note that the Blind Rotation in VP is much cheaper than in PBS, as VP uses $\log_2(N)$ CMUX operations, whereas PBS uses n CMUX operations, where $\log_2(N) \ll n$. If the number of input bits is larger than $\log_2(N)$, a CMUX tree is performed before the Blind Rotation step. The number of CMUX operations in the CMUX tree is $2^{m-\log_2(N)}$, and therefore it grows exponentially. The output LWE ciphertext can hold $\log_2(p)$ bits of the LUT output, where p is

the message modulus for a given parameter set. Therefore, to compute an m-to-m bit LUT, we must compute $\frac{m}{\log_2(p)}$ separate VP operations. For example, to compute an 8-to-8 bit LUT, with $p = 2$, we need to compute 8 VP operations.

2.6 Our Assumed Threat Model

We employ the TFHE scheme [10] as our target FHE scheme. By default, TFHE ciphertexts are indistinguishable under a chosen-plaintext attack (IND-CPA), which is a widely adopted threat model in nearly all FHE applications. The programmable bootstrapping (PBS) in TFHE, however, incurs a small probability of failure, denoted by P_{fail}, during decryption due to significant noise growth in the modulus switching step. In this work, we primarily optimize parameters to achieve a probability of failure of about 2^{-35} to 2^{-40}, which is commonly targeted in the literature to ensure realistic comparisons.

Recently, Cheon *et al.* [8] introduced an attack on exact FHE schemes such as CGGI, breaking the IND-CPA^D security notion by exploiting a decryption oracle and leveraging the scheme's probability of decryption failure to mount key-recovery attacks. Consequently, the probability of failure becomes a critical security parameter and must be reduced to at least 2^{-128} for applications employing this threat model to remain secure. This tighter requirement necessitates choosing larger parameters to accommodate the lower probability of failure, thereby increasing the execution time of both PBS and WoPBS operations. We also provide optimized parameters corresponding to this stricter failure probability, which allows users to select configurations based on their specific use cases and threat models.

3 Switching between PBS and WoPBS

To enable the computation of large lookup tables using the CBS + VP approach, the LWE ciphertexts converted to GGSW must encode the message bit located in its MSB. Consider an LWE ciphertext $\texttt{LWE}_{s_p,q}(m \cdot \Delta)$, where s_p is the secret LWE key in the PBS context and q is the ciphertext modulus. We need to decompose it into α LWE ciphertexts of the form $\texttt{LWE}_s(b_i \cdot \frac{q}{2})$ each holding a single message bit, such that

$$m = \sum_{i=0}^{\alpha-1} b_i \cdot 2^i.$$

The work in [2] performs α PBS operations, requiring α Blind Rotation operations to decompose a ciphertext encrypting an α-bit message. In this work, we propose two new and efficient decomposition algorithms that use only one Blind Rotation operation.

3.1 Our Bit Decomposition Algorithms

Two decomposition algorithms have been introduced in the literature that take as input an LWE ciphertext encrypting a message m and then decompose it

into LWE ciphertexts, each encrypting a digit of the original message (note, the digit need not be a bit, as it can be in any base B). These decomposition algorithms are described in [12] and [28], and take as input an LWE ciphertext $\texttt{LWE}_s(m_{in} \cdot \Delta_{in})$, and output α LWE ciphertexts $\texttt{LWE}_s(m_i \cdot \Delta_i)$, where α is the digit count of m_{in} in base B. These algorithms require α bootstrapping operations, and although they scale well for a large base B, they quickly become inefficient when the B is relatively small, such as $B = 2$ in our case.

Our first decomposition technique, introduced in Algorithm 2, reduces the number of bootstraps from α to 1 by adapting the `PBSmanyLUT` introduced in [12]. This technique enables the computation of multiple lookup tables in a single LWE ciphertext at the cost of one Blind Rotation, which is the main time-consuming operation in PBS. In more detail, `PBSmanyLUT` leverages the generalized PBS (`GenPBS`), which allows PBS evaluation on a specific chunk of the underlying message space rather than on the entire space [12]. In this case, PBS employs a special modulus switching operation (different from the basic modulus switching in Algorithm 1), which ignores $\varkappa$ bits of the MSB of the message space in the upcoming Blind Rotation and sets the ϑ LSBs of the modulus-switched values to 0. Setting these bits to 0 guarantees that the accumulator rotates in multiples of 2^ϑ. Therefore, we can now encode our lookup table into the GLWE accumulator in chunks of size 2^ϑ, with each slot encoding a function; for ϑ bits set to 0, we can compute 2^ϑ functions at the cost of only one Blind Rotation. At the end of the Blind Rotation, we *Sample-Extract* the first 2^ϑ coefficients of the rotated accumulator, yielding LWE ciphertexts that are encrypted under the GLWE secret key. Note that applying the `PBSmanyLUT` technique *does not introduce any additional noise* during the Blind Rotation, as it behaves similarly to a normal PBS. However, the noise growth from its modulus switching is proportional to ϑ; this leads to a potential restriction on the number of functions that can be computed with one `PBSmanyLUT`, which depends on the ring dimension N. Therefore, when increasing ϑ, the probability of failure increases significantly. In particular, the parameters must satisfy:

$$\frac{q \cdot 2^\vartheta}{\Delta_{\text{in}} \cdot 2^\varkappa} < 2N. \tag{3}$$

To use `PBSmanyLUT` in our decomposition algorithm, we need to fill the accumulator with α distinct functions f_i. Each such function extracts the i-th bit (b_i) of the input message and is scaled with $\Delta_{out} = \frac{q}{2}$; the GLWE accumulator is filled with the corresponding scaled function outputs so that $f_i(m_{in} \cdot \Delta_{in}) = b_i \cdot \Delta_{out}$. We employ the special modulus switching from `GenPBS`, where $\vartheta = \lceil \log_2(\alpha) \rceil$ and $\varkappa = 0$, since we want the Blind Rotation to cover the entire message space.

The output of Algorithm 2 is α binary LWE ciphertexts, each encrypting a single message bit stored in the MSB. To demonstrate correctness, note that since m_{in} is of bitsize α and we assume one bit of padding, the scaling factor of the input LWE ciphertext is $\Delta_{in} = \frac{q}{2^{\alpha+1}}$. Substituting into Equation 3 gives $2^{\alpha+1} \cdot 2^{\lceil \log_2(\alpha) \rceil} < 2N$. Since we assume α to be between 4 and 6 and the ring dimension for common TFHE parameter sets lies in the range $1024 \leq N \leq 8192$, the inequality holds.

Algorithm 2 Our Bit Decomposition via `PBSmanyLUT`.

Input: LWE ciphertext ct of form $\texttt{LWE}_{s,q}(m_{in} \cdot \Delta_{in})$ where $m_{in} = \sum_{i=0}^{\alpha-1} b_i \cdot 2^i$
Input: Bootstrapping keys BSK
Output: α LWE ciphertexts $\{\text{ct}_i\}_{i=0}^{\alpha-1}$ of the form $\texttt{LWE}_{S,q}(b_i \cdot \Delta_{out})$, where $\Delta_{out} = \frac{q}{2}$

Define accumulator L with scaling factor Δ_{out}.
for $m = 0$ to $2^\alpha - 1$ **do**
  for $j = 0$ to $\frac{N}{2^\alpha} - 1$ **do**
    for $i = 0$ to $\alpha - 1$ **do**
      Define function $f_i : x \mapsto i$-th bit of x
      $L[m \cdot \frac{N}{2^\alpha} + j] \leftarrow f_i(m) \cdot \Delta_{\text{out}}$
    end for
  end for
end for
$\{\text{ct}_i\}_{i=0}^{\alpha-1} \leftarrow \text{PBSmanyLUT}(\text{ct}, \text{BSK}, L, \Delta_{out}, \varkappa = 0, \vartheta = \lceil \log_2(\alpha) \rceil)$
return $\{\text{ct}_i\}_{i=0}^{\alpha-1}$

To operate in the WoPBS context, we must apply a key switching operation to each binary LWE ciphertext. This operation changes the underlying GLWE secret key from S_p (used in the PBS context, as the output LWE ciphertexts in Algorithm 2 are encrypted under the GLWE key) to s_w, which is the secret key in the WoPBS context. During initialization, we generate a key switching key $\texttt{KSK}_{S_p \to s_w}$ that performs this transformation homomorphically. We then utilize a LWE-to-LWE key switching algorithm as in [11]. Here, S_p and S_w denote the *large LWE secret key* under which the ciphertexts are encrypted from the output of PBS, while s_p and s_w denote the *small LWE secret key* under which the ciphertexts are encrypted when they are given as input to the PBS.

The main drawback of the `PBSmanyLUT` operation is the high probability of failure induced by its modulus switching. While we can significantly decrease its noise growth, as later discussed in Section 3.4, we cannot use it to decompose more than two or three bits, and the ring dimension N must be increased to decompose more bits with one Blind Rotation.

To work around this limitation, we propose our second decomposition algorithm that is based on the Multi-Value Bootstrapping (MVB) technique [6]. This introduces additional noise that must be accounted for when optimizing parameters. Our decomposition algorithm will work for both padded and non-padded ciphertexts, with the only difference being that the worst-case noise of the output for the padded ciphertexts is double that of the non-padded variant. In this case, the base function f is computed using

$$f(X) \;=\; -\frac{q}{4}\left(1 + X + \cdots + X^{N-1}\right).$$

This function, which is negacyclic (to support the no-padding case), outputs only the MSB bit, zeroing all other bits. To extract each of the remaining message bits by moving it into the MSB position for standalone processing using MVB,

we use the G polynomials given below:

$$G_k(X) \;=\; -\sum_{j=1}^{2^{k+1}-1} (-1)^{j+1} X^{jN/2^{k+1}}, \qquad k = 0, \ldots, \alpha - 2,$$

where α is the number of bits in the input message. A detailed description of our second bit decomposition method is described in Algorithm 3.

Algorithm 3 Our Bit Decomposition via MVB.

Input: LWE ciphertext $\mathtt{LWE}_{s,q}(m_{\text{in}} \cdot \Delta_{\text{in}})$ where $m_{\text{in}} = \sum_{i=0}^{\alpha-1} b_i \cdot 2^i$
Input: Bootstrapping keys BSK
Output: α LWE ciphertexts $\{\text{ct}_i\}_{i=0}^{\alpha-1}$ of the form $\mathtt{LWE}_{S,q}(b_i \cdot \Delta_{\text{out}})$, where $\Delta_{\text{out}} = \frac{q}{2}$

1: **Define** the base test polynomial

$$f(X) \;=\; -\frac{q}{4}\left(1 + X + \cdots + X^{N-1}\right),$$

which extracts the most significant bit $b_{\alpha-1}$ after Blind Rotation.
2: **Perform** a Blind Rotation of the accumulator using f and the input ciphertext.
3: **Obtain** the first output ciphertext

$$\text{ct}_{\alpha-1} \;\leftarrow\; \mathtt{LWE}_{S,q}(b_{\alpha-1} \cdot \Delta_{\text{out}})\,.$$

4: **for** $k = \alpha - 2$ to 0 **do** ▷ *Process remaining bits from next-MSB down to LSB*
5: **Multiply** the Blind Rotation output by the selector polynomial

$$G_k(X) \;=\; -\sum_{j=1}^{2^{k+1}-1} (-1)^{j+1} X^{jN/2^{k+1}}.$$

6: **Obtain** the ciphertext

$$\text{ct}_{k-\alpha-2} \;\leftarrow\; G_k \cdot f \;=\; \mathtt{LWE}_{S,q}\Big((b_{\alpha-1} \oplus b_{k-\alpha-2}) \cdot \Delta_{\text{out}}\Big).$$

▷ *Selector $G_k(X)$ yields parity w.r.t. $b_{\alpha-1}$, so $\oplus$ is needed*
7: **end for**
8: **return** $\{\text{ct}_i\}_{i=0}^{\alpha-1}$

In the worst case, the additional noise generated for an input LWE ciphertext of an α bit message is $(2^{\alpha-1} - 1) \cdot V_{pbs}$, where V_{pbs} is the output variance of the PBS. Note that the message bits are masked (XOR-ed) by the MSB bit due to the negacyclic property. For the case that there is a bit of padding, $b_\alpha = 0$, and each output bit is correct on its own. However, worst-case noise would double and be $(2^\alpha - 1) \cdot V_{pbs}$. Since all bits except the MSB are XORed with the MSB, we can simply XOR them again with the MSB (implemented as an LWE addition) to remove the XOR and proceed with the subsequent operations. Alternatively, when performing CBS and Vertical Packing to evaluate a LUT, we can incorporate the XOR of the MSB with the other bits directly into the

LUT definition, so that this operation is handled as part of the function being evaluated.

3.2 Operations in the WoPBS context

The most useful operation that can be performed in the WoPBS context is the large lookup table computation, which requires CBS and Vertical Packing evaluation. The authors of [12] propose using the `PBSmanyLUT` to accelerate the circuit bootstrapping phase. Since each binary LWE ciphertext requires ℓ PBS operations (where ℓ is the level of decomposition of the GGSW ciphertexts), we can use the `PBSmanyLUT` technique to compute ℓ different functions (details on the ℓ PBS functions can be found in Section 2). To optimize this further, we can use PBSmanyLUT to compute the ℓ PBS operations and use Algorithm 3 to compute it for each bit of the input; this approach is elaborated in Section 4.2. For each bit b_i we require ℓ functions, so we have $\vartheta = \log_2(\ell)$.

Our MVB decomposition methodology (Alg. 3 can reduce the number of PBS required for the WoPBS technique, effectively reducing the total cost by α PBS operations. We remark that more than α bits can be used to compute large WoPBS lookup tables by applying CBS on all the bits and performing Vertical Packing. According to [2], up to $n = 28$ bit lookup tables can be computed using the WoPBS approach (here, n is the number of bits).

For relatively small lookup table sizes (i.e., 8 to 16 bits), the computation time of Vertical Packing is negligible compared to the cost of CBS. However, after a certain size (namely, 25 bits), the cost of Vertical Packing becomes even higher than the CBS step because the CMUX tree that is evaluated uses many more CMUX operations than the PBS operations in CBS. Therefore, for relatively small bit sizes, multiple lookup tables can be computed with negligible impact on the total execution time. To compute more WoPBS lookup tables on the output LWE ciphertexts of Vertical Packing, we must use the same scaling factor as the input ciphertexts (i.e., $\Delta = \frac{q}{2}$) in the lookup table entries evaluated in the CMUX tree. This way, the output LWE ciphertexts from Vertical Packing have the same encoding as the inputs.

In addition to the large lookup table evaluation in the WoPBS context, another important operation is the homomorphic `XOR` gate. Since the binary LWE ciphertexts hold the message bit in the MSB slot, the `XOR` operation can be performed by simply adding the two LWE ciphertexts together. The cost of LWE additions is negligible compared to PBS, which makes `XOR` in the WoPBS context much more efficient. Moreover, since each LWE addition increases the noise, only a certain number of additions (depending on the chosen parameters) can be performed before the noise exceeds the threshold and causes decryption errors. To address this, we can bootstrap the noisy ciphertexts to refresh the noise after a certain number of additions. More details about this bootstrapping operation are provided in Algorithm 4.

Bootstrapping bits with no bit of padding. The PBS operation introduced in [10] assumes a bit of padding set to 0 so that any univariate function can be

computed without restriction. This is due to the negacyclicity of the ring used in the polynomial modulus $(X^N + 1)$. To bootstrap an LWE ciphertext without a padding bit, our lookup table must be negacyclic, where a negacyclic function is defined as $f(x + 2N) = -f(x)$. Since the error in an LWE ciphertext can be negative (recall, this is drawn from a Gaussian distribution), we must center the error so that only the value of the message affects the rotation of the accumulator. We add a constant LWE ciphertext $(\mathbf{0}, q/4)$ to the input, where the mask is 0 and the body is $q/4$. Then, we apply a normal PBS, as in Algorithm 1. Examining the lookup table, if $b = 0$, the output will be $-q/4$, while if $b = 1$, the output will be $q/4$. To map the PBS output back to the original encoding, we add it to the same constant LWE ciphertext that was added prior to the PBS. The result is a refreshed LWE ciphertext holding the same message bit. One can easily confirm that LUT encodes the univariate function $f(x) = -\frac{q}{4}$, which is negacyclic.

Algorithm 4 Negacyclic Bootstrapping with Constant LUT

Input: LWE ciphertext $\mathtt{ct}_{\text{in}}$ of the form $\mathtt{LWE}_{s,q}(b \cdot \frac{q}{2})$, where $b \in \{0, 1\}$
Output: Refreshed LWE ciphertext $\mathtt{ct}_{\text{out}}$ of the same form
1: Define constant LUT of size N, where each entry is $-q/4$
2: $\mathtt{ct}_{\text{adj}} \leftarrow \mathtt{ct}_{\text{in}} + (\mathbf{0}, q/4)$ ▷ Centers the noise positively
3: $\mathtt{ct}_{\text{boot}} \leftarrow \mathtt{PBS}(\mathtt{ct}_{\text{adj}}, \mathtt{LUT})$
4: $\mathtt{ct}_{\text{out}} \leftarrow \mathtt{ct}_{\text{boot}} + (\mathbf{0}, q/4)$ ▷ Maps to original encoding
5: **return** $\mathtt{ct}_{\text{out}}$

Other Boolean Operations. In addition to `XOR`, the `AND` gate can also be implemented using the `LWEMult` operation introduced in [12]. The general outline of this operation is to apply a Public Functional Key switch, introduced in [11], to each binary LWE ciphertext and pack them into a single GLWE ciphertext. Then, we perform a multiplication of GLWE ciphertexts (similar to the polynomial multiplication in BFV [16]) and extract the outputs as LWE ciphertexts with the same scaling factor. The difference between this operation and `XOR` is that *it can be batched* (i.e., multiple LWE ciphertexts can be multiplied at the cost of one GLWE multiplication) and it is much more expensive in terms of latency. The key switching itself is the most time-consuming operation, while the noise growth is much larger compared to simple LWE additions. Accordingly, larger parameters must be chosen to accommodate the noise growth of `LWEMult` and bootstrapping must be performed more frequently.

3.3 Our Proposed Recomposition Methodology

Now, to switch back to the PBS context, we need to recombine the binary LWE ciphertexts homomorphically into a new ciphertext with an additional bit of padding to handle negacyclicity. Before applying our recomposition algorithm on binary LWE ciphertexts in the form $\mathtt{LWE}_{s_w}(m \cdot \frac{q}{2})$, we first need to key switch

the underlying secret key from the WoPBS context to the PBS context. We generate the key switching key $\texttt{KSK}_{S_w \to s_p}$ in an initialization step (along with other key switching keys and bootstrapping keys) to change the secret key from the WoPBS context to the PBS context. This assumes the input LWE ciphertexts are encrypted under the GLWE secret key S_w in the WoPBS context. We could also generate a key switching key $\texttt{KSK}_{s_w \to s_p}$ if we want to key switch from LWE ciphertexts encrypted under the smaller LWE key s_w.

Our recomposition technique is given in Algorithm 5. The core idea is similar to the negacyclic bootstrap method given in Algorithm 4: We first center the error on the input binary LWE ciphertexts, and then apply a normal PBS operation to each ciphertext with a specific lookup table as outlined in the algorithm. Next, we map to the original encoding by adding the offset value to the resulting ciphertexts. Thus, each ciphertext encrypts its bit in a different position of the message space (from position $\frac{q}{4}$ to $\frac{q}{2^{\alpha+1}}$). We finally add all the ciphertexts together to compute the output.

Algorithm 5 Recomposition of Binary LWE Ciphertexts

Input: List of LWE ciphertexts $\{\texttt{ct}_i\}_{i=0}^{\alpha-1}$ of form $\texttt{LWE}_{s,q}(b_i \cdot \frac{q}{2})$
Input: Lookup tables $\{\texttt{lut}_i\}_{i=0}^{\alpha-1}$ where each $\texttt{lut}_i$ is initialized with $-\frac{q}{2^{i+3}}$
Input: Bootstrapping key BSK
Output: LWE ciphertext encrypting $\texttt{LWE}_{S,q}(m \cdot \frac{q}{2^{\alpha+1}})$

1: **for** $i = 0$ to $\alpha - 1$ **do**
2: $\quad \texttt{ct}_i \leftarrow \texttt{ct}_i + (\mathbf{0}, \frac{q}{4})$
3: $\quad \texttt{ct}'_i \leftarrow \texttt{PBS}(\texttt{ct}_i, \texttt{BSK}, \texttt{lut}_i)$
4: $\quad \texttt{ct}'_i \leftarrow \texttt{ct}'_i + (\mathbf{0}, \frac{q}{2^{i+3}})$
5: **end for**
6: $\texttt{ct\_out} \leftarrow \sum_{i=0}^{\alpha-1} \texttt{ct}'_i$
7: **return** ct_out

3.4 Our Modulus-Switching Noise Reduction

To enable an efficient parameter set that achieves a P_{fail} of 2^{-128}, we can adapt the modulus switching reduction technique introduced in [3]. With a cost of around 50 LWE additions, they can decrease the probability of failure for a given parameter set from 2^{-r} to 2^{-2r}. The cost of the extra LWE additions compared to Blind Rotation is negligible, making it highly desirable to use it for the strong $\texttt{IND\_CPA}^D$ threat model. However, we must ensure the Circuit Bootstrapping and Vertical Packing operations do not cause excessive noise growth for $P_{\text{fail}} = 2^{-128}$. We use the noise analysis given in [34] to ensure our parameters achieve the targeted failure probability.

Another approach to reduce the noise growth in the modulus switching operation was recently introduced in [31]; this technique computes the mean of the rounding errors of the LWE mask and removes it from the LWE ciphertext

before computing the modulus switch. The rounding errors can be computed publicly since the mask is public. This allows us to achieve a lower probability of failure for the same cryptographic parameters, which can be useful when choosing more efficient parameters to target a specific P_{fail} threshold. The work in [31] also employs a method to mitigate the noise growth of key switching operations, which involves decomposing and rounding the LWE *mask* $\mathbf{a}$ and *body* b (Section 2.3) in the first step. In our work, we will adapt the Centered Mean Noise Reduction [31] to reduce the special modulus switching employed in the `PBSmanyLUT` operation introduced in [12]. For the modulus switching operation presented in Equation 2, we can perform the following noise analysis (adopted from [12]):

$$\begin{aligned}\mathrm{Var}(E_{res}) = \frac{w^2\sigma_{in}^2}{q'^2} + \mathrm{Var}(\bar{b}) + n\cdot\mathbb{E}^2(\bar{a}_i)\,\mathrm{Var}(s_i) \\ + n\cdot\mathrm{Var}(\bar{a}_i)\left(\mathrm{Var}(s_i) + \mathbb{E}^2(s_i)\right)\end{aligned} \tag{4}$$

where:

$$\begin{aligned}\mathrm{Var}(\bar{b}) = \mathrm{Var}(\bar{a}_i) = \tfrac{1}{12} - \tfrac{w^2}{12q'^2}, \\ \mathbb{E}(\bar{a}_i) = -\tfrac{w}{2q'}, \quad \mathrm{Var}(s_i) = \tfrac{1}{4}.\end{aligned}$$

Here, we have $w = 2N\cdot 2^{-\vartheta}$, which is the term multiplied with the LWE element during modulus switching, while q' equals $q\cdot 2^{-\varkappa}$. By applying the Centered Mean preprocessing, the $\mathbb{E}(s_i)$ contribution is eliminated from the variance term in Eq. 4. Therefore:

$$\begin{aligned}\mathrm{Var}(E_{res}) &= \tfrac{w^2\sigma_{in}^2}{q'^2} + \tfrac{1}{12} - \tfrac{w^2}{12q'^2} \\ &\quad + n\Big(\big(\tfrac{1}{12} - \tfrac{w^2}{12q'^2}\big)\cdot\tfrac{1}{4}\Big) + n\Big(\big(-\tfrac{w}{2q'}\big)^2\cdot\tfrac{1}{4}\Big) \\ &= \frac{w^2\sigma_{in}^2}{q'^2} + \frac{1}{12} - \frac{w^2}{12q'^2} + n\left(\frac{1}{48} - \frac{w^2}{48q'^2}\right) + n\left(\frac{w^2}{16q'^2}\right) \\ &= \frac{w^2\sigma_{in}^2}{q'^2} + \frac{1}{12} - \frac{w^2}{12q'^2} + \frac{n}{48} + \frac{nw^2}{24q'^2}.\end{aligned}$$

In this case, the correctness condition would be:

$$\Gamma\sqrt{\mathrm{Var}(E_{res})} < \frac{w\Delta_{in}}{2q'},$$

$$\Gamma^2\left(\frac{w^2\sigma_{in}^2}{q'^2} + \frac{1}{12} - \frac{w^2}{12q'^2} + \frac{n}{48} + \frac{nw^2}{24q'^2}\right) < \frac{w^2\Delta_{in}^2}{4q'^2}.$$

Multiplying both sides by $\frac{q'^2}{w^2}$, we have:

$$\Gamma^2\sigma_{in}^2 + \frac{\Gamma^2 q'^2}{12w^2} - \frac{\Gamma^2}{12} + \frac{\Gamma^2 n q'^2}{48w^2} + \frac{\Gamma^2 n}{24} < \frac{\Delta_{in}^2}{4},$$

$$\boxed{\sigma_{in}^2 < \frac{\Delta_{in}^2}{4\Gamma^2} - \frac{q'^2}{12w^2} - \frac{nq'^2}{48w^2} + \frac{1}{12} - \frac{n}{24}}.$$

In the variance expression, the largest positive contribution is $\frac{nw^2}{24q'^2}$, arising from the bias term $\mathbb{E}^2(\bar{a}_i)\,\mathrm{Var}(s_i)$. The Centered Mean Noise Reduction halves this contribution, effectively halving the overall variance in the regime where it dominates. Halving the variance reduces the noise standard deviation by a factor of $1/\sqrt{2}$, which lowers P_{fail} for the same parameters. This freed noise budget can be spent in different ways: for the same ring dimension N, we can either increase ϑ by 1 (doubling the number of functions in a single `PBSmanyLUT`) or increase Δ_{in} by a factor of $\sqrt{2}$ while keeping P_{fail} approximately unchanged. Here P_{fail} is defined as $\mathrm{erf}(\frac{\Gamma}{\sqrt{2}})$.

4 Experimental Evaluation of our Methodology

To highlight the practicality and efficiency of our techniques, we demonstrate the implementation of a homomorphic version of AES encryption and decryption. Homomorphic AES is a widely used benchmark in the FHE literature and has received considerable attention in recent works that advance TFHE primitives (e.g., [1]), as it can help showcase new methodologies on homomorphic evaluation of this circuit. All our CPU-based experiments are performed on an Ubuntu workstation with a i9-12900K CPU and 128 GB of RAM. We remark that all CPU experimental results are based on *single-threaded execution*. For our GPU experiments, we employ an NVIDIA H100 80 GB instance. We use `tfhe-rs` for all our evaluations and select parameter sets corresponding to 128 bits of security, while reporting the associated failure probability P_{fail}.

4.1 Improving AES Transciphering with Noise Reduction

Given the AES evaluation from [34] and the noise reduction technique from [31], we demonstrate that it is possible to generate more efficient parameters targeting the same probability of failure, as shown in Table 1. In our analysis, we execute AES-128-CTR using full CBS in each of the 10 rounds, to get consistent execution times for both AES encryption and decryption.[3] We remark that [34] skips the PBS part of the CBS in the first round of AES transciphering and computes the first Inverse SBox in a leveled manner (dubbed HalfCBS in [34]), which increases the failure probability, but lowers the execution time by a few seconds,

[3] While AES *decryption* is used for transciphering, the AES *encryption* is also widely researched and implemented in the literature [33].

depending on the parameters. In our evaluation, we target a maximum failure probability P_{fail} of about 2^{-35}, and we report up to 18% runtime improvement by optimizing the parameter set (Table 1).

Since the AES circuit requires `XOR` operations (implemented as LWE additions) between SBox operations (large LUTs), the combination of `PBSmanyLUT` + MVB cannot be used in every round to compute the GLWE-like intermediate ciphertexts for multiple bits using a single Blind Rotation. Instead, we adopt a hybrid approach, described in detail in Section 4.3. In this approach, we alternate each AES round between computing a single Blind Rotation/`PBSmanyLUT` for every 4-bit ciphertext (i.e., integer iterations), and computing a Blind Rotation/`PBSmanyLUT` for each bit (i.e., binary iterations). Here, the Vertical Packing in the binary iterations yields ciphertexts in base $\beta_{out} = 16$, while in the integer iterations the output ciphertexts are in base $\beta_{out} = 2$. Moreover, in the integer iterations, the additions are performed in the GGSW level, and in the binary iterations, the additions are performed in the LWE level.

Table 1. Comparison of the AES transciphering benchmark between [34] running on a single CPU thread, and our approach. *OPT* corresponds to our optimized parameter set with noise reduction on a single CPU thread, where we employ the Mean Noise Reduction methodology before each modulus switching (Section 3.4). *HYB* corresponds to our hybrid LUT evaluations on a single CPU thread for different parameter pairs (Section 4.3); the different parameters for the *binary* iterations appear in parentheses. *GPU* corresponds to our CUDA-accelerated implementation on a single H100 GPU, accounting for the noise increase by the parallel Blind Rotation with $d = 4$ (Section 4.5).

Set	n	N	k	ℓ_{ks}	B_{ks}	ℓ_{pbs}	B_{pbs}	ℓ_{tr}	B_{tr}	ℓ_{ss}	B_{ss}	ℓ_{ep}	B_{ep}	ϑ	**Runtime (s)**	P_{fail}
[34]	768	1024	2	3	4	1	23	3	12	2	17	6	2	3	16.008	$2^{-34.86}$
Ours (OPT)	640	1024	2	3	4	1	23	3	12	2	17	5	2	3	**13.078**	$2^{-34.24}$
Ours (HYB)	768	1024	2	5(2)	3(5)	2	15	6(4)	7(12)	2	17	3(2)	5(7)	2(1)	**14.544**	$2^{-34.24}$
Ours (HYB2)	768	1024	2	7(2)	2(5)	2	15	6(4)	7(12)	2	17	4(2)	4(7)	2(1)	**15.315**	$2^{-41.92}$
Ours (GPU)	768	1024	2	3	4	2	15	4	10	2	17	2	6	1	**0.32**	2^{-726}

4.2 Faster LUT Evaluation

The ability to compute LUTs faster can be met by a combination of the `PBSmanyLUT` + MVB, the mean compensation noise reduction, and the CBS and Vertical Packing algorithms. Consider input LWE ciphertexts that are in base β, for which we wish to compute a LUT. The basic solution entails setting $\beta = 2$, and using `PBSmanyLUT` for the refresh step of CBS. To decrease the number of `PBSmanyLUT` required, we would need to increase β. In order to do this, we also need to utilize the digit decomposition from Algorithm 3, which uses MVB to decompose the bits. In Algorithm 3, the scaling factor is set to $\Delta = \frac{q}{4}$. For the case of CBS, the scaling factors depend on the decomposition parameters B_i

and ℓ_i, where $i \in [1, \ell_{ep}]$ and $\Delta_i = \frac{q}{B_{ep} \cdot i}$. Since no parameter set can be found for $\ell_{ep} = 1$ (due to noise blowup), we must use `PBSmanyLUT` to compute the output bits with different scaling factors Δ_i, in the first ℓ_{ep} slots, while the rest of the process remains similar to Algorithm 3.

The approach in [34] sequentially extracts digits from the input LWE ciphertexts and then performs a Blind Rotation on one or two message bits at a time, depending on the parameters. In our case, with the help of the Mean Noise Reduction technique, we can generate parameters that can use a single Blind Rotation on 4 bits of message, which makes it suitable for basis $\beta = 16$ ciphertexts, which is highly parallelizable (since digit extraction is sequential). We remark that without this noise reduction, the P_{fail} of the `PBSmanyLUT` on a 4-bit message becomes very high and impractical. Notably, if the input LWE ciphertexts contain a bit of padding, we can first add each ciphertext to itself to shift the message bits towards the MSBs. The output LWE ciphertexts from the Vertical Packing can also be in base $\beta_{out} = 16$, which means we only need to execute $\frac{B}{\beta_{out}}$ Vertical Packing operations, where B is the size of the LUT that is evaluated, to maximize efficiency. Then, to evaluate more LUTs, we can apply the same procedure and repeat for unbounded depth. Returning to the PBS context after LUT evaluation involves two steps: (1) performing a key-switching operation to convert the secret key to the PBS context, and (2) executing a full-domain (FD) PBS operation [26,12,28] to operate on the ciphertexts in the absence of padding bits. Note that while it is possible to set the output scaling factor of the Vertical Packing LUT to $\Delta_{out} = \frac{q}{\beta \cdot 2}$ so the output LWE ciphertexts have a bit of padding, the probability of failure P_{fail} just before the next PBS operation (measured right after the modulus switching operation) becomes very high, around 2^{-13} for the parameters in Table 2, which is impractical for almost any application. Here, the FD PBS requires two separate Blind Rotation operations each time.

4.3 Proposed Hybrid LUT Evaluation

The LUT evaluation mode we propose does not naturally support XOR operations, as the LWE ciphertexts returned from the LUT are not in the LWE($\frac{q}{2} \cdot m$) format, which is required for XOR operations. Ideally, our goal is to embed the XOR operations in the lookup tables. However, this is not possible for many applications, such as the mix columns step of AES, which is essentially a 32-bit LUT and becomes very costly given that the required CMUX tree will be very large. Based on our estimations, the AES evaluation would be around 50× slower if we were to use 32-bit LUTs, and the corresponding noise growth has to be accounted for as well.

To address this challenge, we propose a hybrid approach. When XOR operations are required before the LUT evaluation and cannot be embedded directly into the LUT (e.g., in the MixColumns step), we perform these XORs in the GGSW domain (after computing the CBS with minimized `PBSmanyLUT` cost of one operation per $\beta = 16$), and then evaluate the LUT on the resulting GGSW ciphertexts. In this case, the output LWE ciphertexts must be binary due to

overflow in the external product. To return to the PBS context, we apply a key switch followed by our recomposition procedure (Algorithm 5).

Conversely, if we continue evaluating LUTs, we compute a PBSmanyLUT on each binary LWE (adding LWE additions if XORs are still needed) and then evaluate the LUT. Since there is no overflow in the GGSW ciphertexts, the output LWE ciphertexts from Vertical Packing can be in base $\beta = 16$, allowing us to proceed with the hybrid approach. For the binary iterations, we can generate smaller Bootstrapping and key-switching keys while keeping the failure probability P_{fail} within the target bound. A minor trade-off of this approach is a modest increase in the size and generation cost of the evaluation keys.

4.4 Large LUT Parameters and Experimental Evaluation

In Table 2, we report our optimized parameter set for large LUT evaluation, in comparison to the recommended parameter set from [34]. Both parameter sets are optimized to work with LWE ciphertexts with $\beta = 16$, without a padding bit. The main advantage of our approach is that the scaled output LWE ciphertexts fed into HomTrace can be computed with a single `PBSmanyLUT` call, whereas the method in [34] requires applying multiple `PBSmanyLUT` operations sequentially. Since `PBSmanyLUT` is the most expensive operation in the workflow, reducing its usage yields substantial performance gains. Although we assume no padding bits, if the input LWE ciphertexts have a bit of padding, this can removed by adding the ciphertext to itself (and this addition should be accounted for in the noise analysis).

The most notable difference between these two parameter sets is our use of *high-precision HomTrace*. This is achieved by first switching to a larger-dimension GLWE ciphertext, evaluating the HomTrace function in that higher dimension, and then switching back to the original dimension. Additionally, we use a higher level ℓ_{ks} for the key switching operation after the LUT evaluation. This key switching key is necessary since we can compute an unbounded number of lookup tables in consecutive order, and we need to switch to the original keys to compute the next `PBSmanyLUT`. In this case, we compute the probability of failure P_{fail} just after the modulus switching operation and before the next `PBSmanyLUT`.

While our key switching and the high precision HomTrace operations are slightly slower compared to [34], the number of `PBSmanyLUT` operations required *is reduced by half*, which is traded for an increased ℓ_{pbs}. We set the GGSW level $\ell_{ep} = 2$, so we can compute the two levels with $\vartheta = 1$, to minimize the growth of P_{fail}. For our evaluation, we compute 8-to-8 and 12-to-12, and 16-to-16 LUTs, and compare the execution times against [34] in Table 3, highlighting the benefits of our novel decomposition algorithm.

Overall, our methodology achieves around $2\times$ improvement in execution time, which is consistent across different LUT sizes. Notably, the parameter set we introduced can tolerate the noise generated by a 16-bit LUT, which is $m \cdot \mathrm{V}_{cbs}$, where $m = 16$.

Table 2. Parameter sets for LUT evaluation using CBS and Vertical Packing. In both sets, we assume the input LWE ciphertexts are in base 16.

Set	n	N	k	ℓ_{ks}	B_{ks}	ℓ_{pbs}	B_{pbs}	ℓ_{tr}	B_{tr}	b_{tr}	ℓ_{ss}	B_{ss}	$\ell_{k\to k'}$	$B_{k\to k'}$	$b_{k\to k'}$	$\ell_{k'\to k}$	$B_{k'\to k}$	$b_{k'\to k}$	ℓ_{ep}	B_{ep}	ϑ	P_{fail}
[34]	769	2048	1	3	2^4	2	2^{15}	7	2^7	35	2	2^{16}	–	–	–	–	–	–	4	2^4	2	2^{-40}
Ours	768	2048	1	5	2^3	3	2^{12}	4	2^{12}	40	2	2^{17}	3	2^{15}	42	3	2^{12}	40	2	2^7	1	2^{-41}

Table 3. Comparison of execution times for different LUT sizes between our work and [34], highlighting the benefits of our novel decomposition algorithm.

LUT Size	**Work**	**Execution Time (ms)**	**Speedup**
8-to-8	[34]	112.74	1.0×
	Ours	**54.24**	**2.07×**
12-to-12	[34]	170.62	1.0×
	Ours	**82.87**	**2.05×**
16-to-16	[34]	227.05	1.0×
	Ours	**108.03**	**2.10×**

LUT Evaluation with Lower P_{fail}. In case an even lower failure probability is needed (e.g., $P_{\text{fail}} < 2^{-128}$ to meet $\texttt{IND-CPA}^D$ security requirements), one could either decrease the number of bits that are decomposed, or double the ring dimension N. This would be needed since the noise growth may not be lowered without also increasing ℓ_{ep}. For our proposed parameter set in Table 2, $\ell_{ep} = 2$, and additional increases would require a subsequent increase of ϑ, which creates a strict bound on P_{fail}, which cannot be lowered any further.[4] Likewise, since doubling N more than doubles the execution time, it is recommended to lower P_{fail} by decreasing the message bits α. Our analysis indicates that $\alpha = 3$ would be sufficient for $P_{\text{fail}} < 2^{-128}$.

4.5 GPU Acceleration of WoPBS

The main bottleneck of the CBS operation is the Blind Rotation, used in the PBS or `PBSmanyLUT` operation. Fortunately, this step is easily parallelizable and will yield very low amortized latency with GPU acceleration. In the case of AES transciphering, many different AES ciphertexts can be homomorphically decrypted in parallel. Consider, for example, the case of AES-128 decryption, and assume there are c different ciphertexts to homomorphically decrypt; each AES round requires $128 \cdot c$ `PBSmanyLUT` operations, which can be parallelized. The subsequent HomTrace, Scheme Switching, Vertical Packing, and auxiliary operations, such as Mean Compensation preprocessing before modulus switch-

[4] For instance, assume we have $\vartheta = 2$, $N = 2048$, $k = 1$ and $\alpha = 4$ bits of message, the minimum P_{fail} will be $2^{-49.2996}$.

ing, modulus switching, and all the key switching operations, are also perfectly parallelizable in this case.

To evaluate our technique with GPU acceleration, we designed our core operations and algorithms using CUDA, and leverage the CUDA FFT engine from TFHE-rs [38]; this allows us to perform the PBS and `PBSmanyLUT` operations while evaluating AES transciphering on a GPU. Table 1 summarizes the performance gains achieved by our CUDA-accelerated primitives over a single-threaded CPU baseline for the AES transciphering benchmark. Our results underscore the substantial advantage of our approach, which can harness the massive parallelism of modern GPUs. Therefore, the observed improvements reflect not only algorithmic efficiency but also the transformative impact of GPU acceleration, which is absent in prior works.

A parallel variant of the Blind Rotation operation was introduced in [24], incurring an increase in bootstrapping key size and in the output variance of the Blind Rotation. The ratio governing the increase in key size and noise is $\frac{m^d-1}{d}$, where m denotes the cardinality of the secret key, and d the number of digits grouped for parallel execution. Since we are working with the binary key version of TFHE (also known as GINX [17]), we set $m = 2$. This technique is suitable for high-performance computing hardware such as GPUs, where utilizing parallel-friendly algorithms results in a major boost in performance and throughput, since the original Blind Rotation is sequential. The growth ratio for different group sizes is presented in Figure 1.

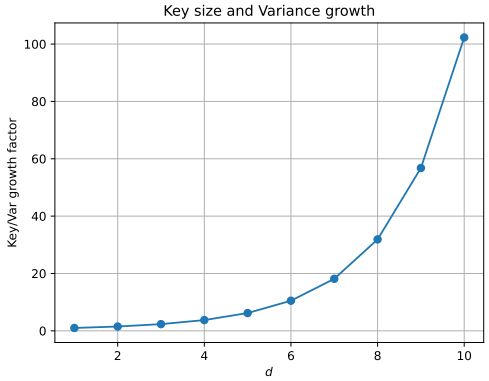

Fig. 1. Noise growth and key expansion factor for the parallel variant of Blind Rotation [24].

Due to the exponential growth of both noise and key sizes, small values of d are chosen for practical applications. For our experiments, we choose $d = 4$, with an expansion factor of $r = 3.75$. We also provide new parameters for our GPU evaluations to account for the $3.75 \cdot \mathrm{V}_{pbs}$ in the AES evaluation. These optimized parameters are summarized in Table 1.

The difference between the two parameter sets is the smaller GGSW level ℓ_{ep} and trace level ℓ_{tr}, which will result in smaller GGSW ciphertexts and trace evaluation key size. This is imperative as the GPU memory is limited and only a limited number of ciphertexts can be evaluated at a time inside the GPU. Since GGSW ciphertexts are a few times larger than their LWE counterpart, minimizing ℓ_{ep} is crucial for optimal throughput. Compared to the parameter set from [34], for an equal number of input LWE ciphertexts, the resulting GGSW ciphertexts for our parameter set are 2x smaller.[5] Therefore, we can batch many more input ciphertexts at a time, maximizing throughput. Note that if we set $\ell_{ep} = 1$, the error blows up and decryption will fail.

4.6 Secret Key Distribution

Although alternative distributions can be chosen for the secret key (resulting in implementation differences in the Blind Rotation), the Mean Noise Reduction technique proposed in [31] is applicable only to binary keys. The most practical implementations of TFHE also assume binary keys [38]. Consequently, our approach focuses on and is optimized for the binary-key variant of TFHE.

5 Related Works

A substantial body of work has examined the homomorphic evaluation of AES encryption and decryption under TFHE, introducing and applying a range of homomorphic operations and optimizations aimed at reducing the scheme's inherent high latency. The first demonstration of an AES implementation in TFHE was presented in [33], where the authors propose an approach that relies exclusively on PBS operations. In their work, all AES operations, including both the SBox and `XOR` operations, are implemented using standard PBS. For the SBox step, the tree-based method from [23] is employed; this method performs well given that the operation is limited to 8 bits. Additionally, the Multi-value Bootstrapping (MVB) optimization introduced in [7] is employed to compute the first level of the tree, while packing functional key switching is utilized to evaluate an encrypted lookup table in the second level. Their implementation of AES-128 encryption requires 270 seconds on a single-threaded system and achieves a failure probability of $P_{\text{fail}} = 2^{-23}$. The primary bottleneck of this approach is the use of PBS and packing key switching for `XOR` operations, which is extremely costly. In contrast, our approach offers novel methodologies and techniques to seamlessly switch between binary and integer contexts, where `XOR` can be very efficiently performed with simple LWE additions, thereby enhancing computational efficiency.

In [4], contrary to the aforementioned study, LWE additions are used to compute the `XOR` operations. However, because the authors employ a message space of $p = 2$ for their ciphertexts, effectively encrypting the message bits

[5] A GGSW ciphertext consists of $\ell_{ep} \cdot (k + 1)$ GLWE ciphertexts.

in the MSB without any padding, it becomes infeasible to compute the `AND` operation between two ciphertexts using PBS, which is required for implementing a Boolean circuit of the AES SBox. Therefore, this work proposes a new encoding scheme, referred to as p-encodings, and adopts a larger message space p to enable the computation of `AND` operations via PBS, while still using LWE additions for `XOR` operations. This approach requires 105 seconds on a single-threaded system and exhibits a failure probability of $P_{\text{fail}} = 2^{-40}$. Furthermore, the corresponding SBox circuit relies on numerous `AND` gates, each invoking a PBS function, which substantially increases the computational cost. Likewise, that approach suffers from limited scalability when applied to larger lookup tables, as these entail many additional nonlinear gates that require PBS. Conversely, our approach can evaluate the AES circuit in about 15.32 seconds on a single CPU thread a similar failure probability.

The work presented in [1] integrates the approaches of the two studies discussed above to achieve a more efficient evaluation of AES encryption. In particular, the tree-based lookup table evaluation with MVB from [33] is combined with the p-encoding strategy of [4] to compute homomorphic `XOR` operations via LWE additions. Notably, this approach is mostly limited to the p-encoding strategy, using an unconventional odd $p = 17$ (which falls outside of standard TFHE implementations), to circumvent negacyclicity, whereas our methods and techniques are fully compatible with existing solutions and methodologies, rendering our approach considerably more practical and applicable to a broader range of applications. Moreover, the approach proposed in [1] exhibits poor scalability as P_{fail} increases. For applications requiring larger lookup tables, that method becomes rapidly inefficient because the execution time of the lookup table increases exponentially with the bit size (tree-based method), whereas our novel decomposition approach offers highly efficient LUT evaluation times (Table 3).

In [32], the authors propose methodologies for computing large lookup tables that differ from the CBS + Vertical Packing approach employed in our work. In particular, their approach aims to compute variable-sized lookup tables with LWE ciphertexts without padding by introducing a modified version of classical PBS for such ciphertexts. That method is further extended to compute larger lookup tables, such as the SBox in AES, through the use of packing key switching and CMUX operations. This is different from the CBS and Vertical Packing techniques that we utilize, and scales poorly as the failure probability P_{fail} decreases.

Lastly, the works from Wei *et al.* [37] and Wang *et al.* [35], [36], [34] iteratively improve the CBS operation, which is a crucial building block in our work. In [37], `PBSmanyLUT` is employed to increase the efficiency of CBS by reducing the number of PBS operations, while in [35], Scheme Switching is employed in conjunction with a public functional key switching technique to reduce the key switching execution time in CBS. In [36], the key switching operation in CBS is massively improved by removing the need for time-consuming public functional key switching operations, while introducing HomTrace Automorphism evaluation, in conjunction with Scheme Switching to compute the CBS. These methods

achieve fast AES execution times, but with very high failure probability due to the high noise growth of the HomTrace function.

The follow-up work in [34] improves the workflow of the CBS technique, proposing a cheap preprocessing step to manage the noise growth of the Trace function, along with a high-precision HomTrace evaluation for LWE integers. This work resulted in state-of-the-art AES evaluation times, improving over earlier works by a significant margin. This work also proposes using LWE integers with `PBSmanyLUT` and MVB to compute large LUTs. Nevertheless, a key limitation of this approach is that it is mostly focused on optimizing the binary ciphertexts workflow, while the main efficiency of modern TFHE comes from working with LWE encrypting small integers. Additionally, the high probability of failure incurred by the `PBSmanyLUT` used in that approach forces the algorithm to extract only 1-2 bits at a time, requiring more than 1 Blind Rotation operation in sequence. This would be time-consuming for our workflow, where we strive to compute many LUTs in consecutive order and switch between PBS and WoPBS contexts, so our goal is to minimize the PBS operations, which is the most time-consuming operation by a large margin. This is imperative since the PBS context is highly optimized and works well with small integers.

In our work, we address these challenges and achieve substantial improvements over previous techniques by introducing new switching algorithms and proposing new methodologies for obtaining efficient parameters with low P_{fail}. Our techniques leverage `PBSmanyLUT` to support larger digit sizes, and we introduce optimized CBS and Vertical Packing workflows for modern GPU targets.

6 Concluding Remarks

In this work, we introduce novel switching algorithms that enable seamless transitions between the PBS and WoPBS contexts in TFHE. Our approach features an efficient bit decomposition algorithm that reduces the number of bootstrapping operations and a recomposition algorithm that reconstructs ciphertexts with an extra bit of padding for handling negacyclicity in the classical PBS operation. We further optimize the CBS workflow by utilizing the new centered mean noise reduction technique, which significantly reduces the noise growth of the modulus switching operation for the `PBSmanyLUT`, resulting in a lower failure probability for the same parameters. This also results in a lower minimum probability of failure for integer CBS, leading to a near $2\times$ improvement in large LUT evaluation with CBS. Furthermore, our approach achieves up to an 18% reduction in execution time for the AES Transciphering benchmark through judicious optimization of the noise parameters. We are also able to utilize `PBSmanyLUT` for larger digit sizes, reducing the number of PBS operations required for our switching algorithms and our large LUT evaluation workflow. Lastly, we design and optimize the memory-heavy CBS workflow for GPU parallelization, achieving up to $50\times$ improvement over a single-threaded CPU baseline. Overall, our framework lays a robust foundation for future advancements in scalable fully homomorphic encryption.

References

1. Belaïd, S., Bon, N., Boudguiga, A., Sirdey, R., Trama, D., Ye, N.: Further Improvements in AES Execution over TFHE. IACR Communications in Cryptology **2**(1) (2025)
2. Bergerat, L., Boudi, A., Bourgerie, Q., Chillotti, I., Ligier, D., Orfila, J.B., Tap, S.: Parameter optimization and larger precision for (T) FHE. Journal of Cryptology **36**(3), 28 (2023)
3. Bernard, O., Joye, M., Smart, N.P., Walter, M.: Drifting towards better error probabilities in fully homomorphic encryption schemes. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 181–211. Springer (2025)
4. Bon, N., Pointcheval, D., Rivain, M.: Optimized homomorphic evaluation of boolean functions. IACR Transactions on Cryptographic Hardware and Embedded Systems **2024**(3), 302–341 (2024)
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) **6**(3), 1–36 (2014)
6. Carpov, S., Izabachène, M., Mollimard, V.: New techniques for multi-value input homomorphic evaluation and applications. In: Cryptographers' Track at the RSA Conference. pp. 106–126. Springer (2019)
7. Carpov, S., Izabachène, M., Mollimard, V.: New techniques for multi-value input homomorphic evaluation and applications. In: Cryptographers' Track at the RSA Conference. pp. 106–126. Springer (2019)
8. Cheon, J.H., Choe, H., Passelègue, A., Stehlé, D., Suvanto, E.: Attacks against the IND-CPAD security of exact FHE schemes. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 2505–2519 (2024)
9. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017. pp. 409–437. Springer International Publishing, Cham (2017)
10. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: Cheon, J.H., Takagi, T. (eds.) Advances in Cryptology – ASIACRYPT 2016. pp. 3–33. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
11. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (2020)
12. Chillotti, I., Ligier, D., Orfila, J.B., Tap, S.: Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for TFHE. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 670–699. Springer (2021)
13. Daemen, J., Rijmen, V.: AES proposal: Rijndael (1999)
14. Dai, W., Sunar, B.: XLS: Accelerated HW Synthesis. `https://github.com/google/xls` (2020)
15. Ducas, L., Micciancio, D.: Fhew: Bootstrapping homomorphic encryption in less than a second. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015. pp. 617–640. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
16. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive (2012)

17. Gama, N., Izabachène, M., Nguyen, P.Q., Xie, X.: Structural lattice reduction: generalized worst-case to average-case reductions and homomorphic cryptosystems. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 528–558. Springer (2016)
18. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing. p. 169–178. STOC '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1536414.1536440
19. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the aes circuit. In: Annual Cryptology Conference. pp. 850–867. Springer (2012)
20. Gorantala, S., Springer, R., Purser-Haskell, S., Lam, W., Wilson, R., Ali, A., Astor, E.P., Zukerman, I., Ruth, S., Dibak, C., Schoppmann, P., Kulankhina, S., Forget, A., Marn, D., Tew, C., Misoczki, R., Guillen, B., Ye, X., Kraft, D., Desfontaines, D., Krishnamurthy, A., Guevara, M., Perera, I.M., Sushko, Y., Gipson, B.: A General Purpose Transpiler for Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2021/811 (2021), https://eprint.iacr.org/2021/811
21. Gouert, C., Joseph, V., Dalton, S., Augonnet, C., Garland, M., Tsoutsos, N.G.: Hardware-accelerated encrypted execution of general-purpose applications. Proceedings on Privacy Enhancing Technologies (2025)
22. Gouert, C., Mouris, D., Tsoutsos, N.G.: Helm: Navigating homomorphic encryption through gates and lookup tables. IEEE Transactions on Information Forensics and Security (2025)
23. Guimarães, A., Borin, E., Aranha, D.F.: Revisiting the functional bootstrap in tfhe. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 229–253 (2021)
24. Joye, M., Paillier, P.: Blind rotation in fully homomorphic encryption with extended keys. In: International Symposium on Cyber Security, Cryptology, and Machine Learning. pp. 1–18. Springer (2022)
25. Khan, T., Tian, W., Zhou, G., Ilager, S., Gong, M., Buyya, R.: Machine learning (ML)-centric resource management in cloud computing: A review and future directions. Journal of Network and Computer Applications **204**, 103405 (2022)
26. Kluczniak, K., Schild, L.: Fdfb: full domain functional bootstrapping towards practical fully homomorphic encryption. arXiv preprint arXiv:2109.02731 (2021)
27. Langmead, B., Nellore, A.: Cloud computing for genomic data analysis and collaboration. Nature Reviews Genetics **19**(4), 208–219 (2018)
28. Liu, Z., Micciancio, D., Polyakov, Y.: Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 130–160. Springer (2022)
29. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 1–23. Springer (2010)
30. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM (JACM) **56**(6), 1–40 (2009)
31. de Ruijter, T., D'Anvers, J.P., Verbauwhede, I.: Don't be mean: Reducing approximation noise in tfhe through mean compensation. Cryptology ePrint Archive (2025)
32. Schild, L., Abidin, A., Preneel, B.: Fast transciphering via batched and reconfigurable lut evaluation. IACR Transactions on Cryptographic Hardware and Embedded Systems **2024**(4), 205–230 (2024)

33. Trama, D., Clet, P.E., Boudguiga, A., Sirdey, R.: 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. pp. 79–90 (2023)
34. Wang, R., Ha, J., Shen, X., Lu, X., Chen, C., Wang, K., Lee, J.: Refined tfhe leveled homomorphic evaluation and its application. Cryptology ePrint Archive (2024)
35. Wang, R., Wei, B., Li, Z., Lu, X., Wang, K.: Tfhe bootstrapping: Faster, smaller and time-space trade-offs. In: Australasian Conference on Information Security and Privacy. pp. 196–216. Springer (2024)
36. Wang, R., Wen, Y., Li, Z., Lu, X., Wei, B., Liu, K., Wang, K.: Circuit bootstrapping: faster and smaller. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 342–372. Springer (2024)
37. Wei, B., Wang, R., Li, Z., Liu, Q., Lu, X.: Fregata: Faster homomorphic evaluation of aes via tfhe. In: International Conference on Information Security. pp. 392–412. Springer (2023)
38. Zama: TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data (2022), `https://github.com/zama-ai/tfhe-rs`
39. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-vm side channels and their use to extract private keys. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 305–316 (2012)
40. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-tenant side-channel attacks in paas clouds. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 990–1003 (2014)