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. This includes a modified batched Circuit Bootstrapping that converts an entire digit with a single Blind Rotation using our novel balanced and monomial selector bases, and a novel batched AES evaluation that halves the per-round Blind Rotations. We further integrate a state-of-the-art noise reduction method into the WoPBS workflow by adopting the centered mean noise reduction for both the PBSmanyLUT and the multi-bit Blind Rotation modulus switching, enabling more efficient parameter sets. All of our parameter sets meet the $\texttt{IND-CPA}^D$ failure budget of 2^{-128}; under it, our workflows evaluate large lookup tables up to 1.9× faster and AES-128 transciphering up to 1.6× faster than the state of the art on the same machine, with our batched AES evaluation running 1.15× faster than our binary workflow. Our CUDA implementation further evaluates one AES-128 block in 331 ms on a single RTX 4090, 43× faster than our single-threaded binary CPU workflow and 1.18× faster than the recent Maverick GPU accelerator, whose parameters admit a failure probability of only about 2^{-40}.

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 [42,43] 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 of 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 knowledge of the secret key or the underlying message. Numerous FHE schemes have been proposed, 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 bootstrapping (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, it is practical only for messages of small precision, typically 4 to 6 bits [11]: its execution time increases exponentially with the precision (more than doubling per additional bit after 4), rendering it impractical for larger messages. 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, arbitrary programs translate into large binary circuits that quickly become unmanageable for mid-sized workloads. More recent advancements, including AES Transciphering [34] and TFHE libraries [41], adopt ciphertexts encrypting small numbers (4-bit precision typically being optimal), representing large-precision messages in a radix format [2]. Circuits then consist of small lookup tables instead of binary gates, significantly reducing their depth and width.

Nonetheless, applying arbitrary functions to large messages exceeds the practical capabilities of PBS. The tree-based method of [23] addresses this, but scales poorly: the number of small PBS computations grows exponentially with the message bit size, and expensive public key switching operations generate the intermediate encrypted LUTs.

A new computing paradigm in TFHE, dubbed *Without Padding PBS (WoPBS)*, has recently emerged [2]. This approach relies on two main oper-

ations, Circuit Bootstrapping (CBS) and Vertical Packing (VP): CBS converts LWE ciphertexts to GGSW ciphertexts (using PBS internally, which refreshes the noise and keeps the paradigm fully homomorphic), while VP computes lookup tables on the converted ciphertexts with CMUX operations, handling tables up to 28 bits relatively efficiently. The original CBS [11] was nearly $10\times$ slower than a PBS; recent refinements [36] bring this overhead below $2\times$.

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 is most efficient on small integers (plus a padding bit), switching between the two contexts is expensive; the WoPBS workflow would also improve significantly if it operated on small integers with fewer PBS operations (the most costly step in CBS).

Large Ciphertext Expansion: The converted GGSW ciphertexts are substantially larger than their LWE counterparts. The corresponding memory overhead can be challenging, as existing parameter sets choose large decompositions to combat the high noise growth of CBS and VP; judicious parameters can minimize this 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 it, with appropriate parameters, improves performance at the same failure probability.

Incompatibility with Existing Applications: Numerous applications are already optimized for the classic PBS, and certain operations are far more efficient in one context than the other: `XOR` (critical in AES transciphering, where mix-columns is pure `XOR`) requires many PBS and public key switching operations in the PBS context, yet costs only native LWE additions in the WoPBS context.

To address these limitations, we optimize the WoPBS workflow through new switching algorithms that convert LWE ciphertexts seamlessly between the two contexts while minimizing PBS operations, and we adopt the *mean compensation* noise reduction [32] in the CBS steps and in our switching algorithms. Overall, we claim the following contributions:

- A *balanced selector* basis with uniform noise factor $2^{\alpha-2}$ for any digit size, an all-monomial family attaining the factor-1 noise floor for 4-bit digits, and a *modified batched CBS* that couples the fused decomposition-and-refresh of [36] with these bases, converting a full 4-bit digit per Blind Rotation (Section 3).
- A *multi-digit recomposition* that returns to the PBS context with α/κ PBS operations instead of α (Section 3).

- A novel *batched AES evaluation* whose redundant spread encoding lets the batched CBS absorb the `XOR` carries, halving the per-round Blind Rotations to 64, which is 1.15× faster than our binary workflow (Section 4).
- CUDA-friendly WoPBS core operations that adopt the centered mean noise reduction [32] with an exact noise model for the multi-bit Blind Rotation modulus switch, evaluating one AES-128 block in 331 ms on an RTX 4090: 43× faster than our binary CPU workflow and 1.18× faster than Maverick [39] at a far stricter failure probability (Section 4).
- Rigorous parameter sets satisfying the $\texttt{IND-CPA}^D$ budget of $P_{\text{fail}} \leq 2^{-128}$, under which our LUT evaluation is up to 1.9× and AES transciphering up to 1.6× faster than [36] on the same machine (Section 4).

Roadmap: Section 2 presents background on FHE, TFHE, and its ciphertext operations. Section 3 introduces our methodology for converting between the PBS and WoPBS contexts and our workflow accelerations, including the noise-reduced `PBSmanyLUT`. Section 4 presents our CPU and GPU evaluation, reporting up to 1.9× faster large-LUT evaluation than the state of the art. Finally, Section 5 discusses prior art, and Section 6 concludes the paper.

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 [31] and its ring-based variant (RLWE) [30]. 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 [6] 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 whose data is initially encrypted using fast symmetric schemes to optimize storage and transmission efficiency. Since FHE ciphertexts incur a significant size expansion, transciphering effectively reduces 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. We detail these representations below, starting with the LWE ciphertext.

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:

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

where q is the ciphertext modulus, $\mathbf{a} \in \mathbb{Z}_q^n$ is a uniformly random vector called the *mask*, $b \in \mathbb{Z}_q$ is the scalar *body*, e is an error term sampled from χ_σ, and $\langle \mathbf{a}, \mathbf{s} \rangle$ is the dot product over $\mathbb{Z}_q$; 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 uniformly random polynomials, $E(X)$ is an error polynomial with small coefficients sampled from χ_σ, and $\langle \mathbf{A}(X), \mathbf{S}(X) \rangle = \sum_{i=0}^{k-1} a_i(X) \cdot s_i(X)$ is the polynomial inner product; all operations are carried out over $\mathbb{Z}_q[X]$ modulo $X^N + 1$, with 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 Δ, recovering the N small-integer coefficients of M, 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:

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

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

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

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

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 Bootstrapping (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 $\mathbf{S}$ that encrypts $f(\mu_{in})$. The Blind Rotation is performed with the homomorphic multiplexer CMUX; further details can be found in [11].

Algorithm 1 Programmable Bootstrapping (PBS)

Input: LWE ciphertext $\texttt{LWE}_{\mathbf{s},q}(\mu_{in} \cdot \Delta_{in})$
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 $\texttt{LWE}_{\mathbf{S},q}(f(\mu_{in}) \cdot \Delta_{out})$
1: $\texttt{ct}' \leftarrow \texttt{ModSwitch}(\texttt{LWE}_{\mathbf{s},q}(\mu_{in} \cdot \Delta_{in}))$
2: `rotated_lut` $\leftarrow \texttt{BlindRotate}(\texttt{ct}', \texttt{bsk}, \texttt{lut})$
3: `ct_out` $\leftarrow \texttt{SampleExtract}(\texttt{rotated_lut})$
4: **return** `ct_out`

`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 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 [7], 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 $\texttt{LWE}_{\mathbf{s}}(m \cdot \Delta)$, 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. In the remainder of this paper and in our parameter tables, these GGSW decomposition parameters are denoted ℓ_{ep} and B_{ep} (for *external product*, the operation that consumes GGSW ciphertexts), to distinguish them from the gadget decompositions of the PBS, key switching, trace, and scheme switching operations. Each GGSW ciphertext is composed of $(k+1)\ell$ GLWE ciphertexts, each encrypting the binary message m under the GLWE secret key, with the extended key convention $\mathbf{S} = (S_0, \ldots, S_{k-1}, S_k = -1)$. 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 $\texttt{GLWE}_{\mathbf{S}}(m \cdot S_i \cdot \frac{q}{\beta^j})$, 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 $\texttt{LWE}_{\mathbf{s}}(m \cdot \Delta)$ into ℓ LWE ciphertexts $\texttt{LWE}_{\mathbf{S}}(m \cdot \frac{q}{\beta^j})$. 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, 38] 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 [38], and [36].

The standard HomTrace amplifies the automorphism key-switching noise by a factor on the order of N^2 (the "cubic noise barrier"), which [36] mitigates with a larger, rank-switched GLWE dimension (high-precision HomTrace). The *reverse-normalized* trace (RevHomTrace) [28] instead applies the $\log_2(N)$ automorphism steps in reverse order with a one-bit rescaling after each, so that each key switch contributes only once, for a variance of $\log_2(N) \cdot V_{auto}$ plus a small rounding term. We adopt RevHomTrace throughout our CBS workflow, removing the need for the high-precision trace and its rank-switching keys.

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. The Blind Rotation in VP consumes the last $\min(m, \log_2 N)$ input bits: the GGSW ciphertext of bit i selects, via one CMUX, between the current LUT polynomial and its rotation by the fixed monomial X^{-2^i} (at most $\log_2(N)$ CMUX operations in total). This is much cheaper than the Blind Rotation in PBS, which performs one CMUX per LWE mask coefficient (n CMUX, with $\log_2(N) \ll n$), rotating by the modulus-switched mask coefficients rather than by fixed powers of two. If $m > \log_2(N)$, the remaining bits select the LUT polynomial out of $2^{m-\log_2(N)}$ candidates through a CMUX tree evaluated before the Blind Rotation, at $2^{m-\log_2(N)} - 1$ CMUX operations, growing exponentially with m. 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 $p = 16$ (the base-16 outputs used throughout this work), we need to compute 2 VP operations.

2.6 Threat Model and Security Budget

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.

Recently, Cheon *et al.* [8] introduced an attack on exact FHE schemes such as CGGI, breaking the $\texttt{IND-CPA}^D$ 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. In this work, all proposed parameter sets are optimized to achieve a probability of failure of at most 2^{-128}, so that they remain secure under this stricter threat model; prior-work sets are listed at their published failure probabilities.

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 $\texttt{LWE}_{\mathbf{s}_p,q}(m \cdot \Delta)$, where $\mathbf{s}_p$ 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 $\texttt{LWE}_{\mathbf{s}}(b_i \cdot \frac{q}{2})$ 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 one-Blind-Rotation decomposition algorithms with provably lower selector noise, culminating in a modified batched CBS (built on our novel balanced and monomial selector bases) that converts an entire digit into GGSW selectors with a single Blind Rotation (Sections 3.2–3.4).

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 [29], and take as input an LWE ciphertext $\texttt{LWE}_{\mathbf{s}}(m_{in} \cdot \Delta_{in})$, and output α LWE ciphertexts $\texttt{LWE}_{\mathbf{S}}(m_i \cdot \Delta_i)$, 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.

The number of lookup-table slots that one Blind Rotation can carry is limited by the `PBSmanyLUT` modulus switching (Equation 2): setting ϑ LSBs to zero amplifies the modulus-switching noise, and the parameters must satisfy

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

A natural instantiation would use `PBSmanyLUT` directly as a bit decomposer, filling the 2^ϑ lanes with per-bit extraction functions; our modified batched CBS of Section 3.3 improves on this by carrying the GGSW levels on the lanes instead, so that decomposition and the CBS refresh share the same single Blind Rotation.

For a no-padding α-bit digit ($\Delta_{in} = \frac{q}{2^\alpha}$) with 2^ϑ lanes, the bound reads $2^\alpha \cdot 2^\vartheta < 2N$, which comfortably holds for our instantiations (e.g., $\alpha = 4$, $\vartheta = 1$, $N = 2048$).

To operate in the WoPBS context, we must apply a key switching operation to each decomposed LWE ciphertext, changing the underlying GLWE secret key from $\mathbf{S}_p$ (the GLWE key of the PBS context, under which sample-extracted PBS outputs are encrypted) to $\mathbf{s}_w$, the secret key of the WoPBS context. During initialization, we generate a key switching key $\mathtt{KSK}_{\mathbf{S}_p \to \mathbf{s}_w}$ that performs this transformation homomorphically. Instead of a coefficient-wise LWE-to-LWE key switch [11], we perform this (and every large-to-small key switch in this work) as a *GLWE key switching*: the input LWE mask is reinterpreted as kN/b_{ks} polynomials of size b_{ks}, key-switched in the polynomial domain with the gadget (B_{ks}, ℓ_{ks}), and the constant coefficient is sample-extracted back into an LWE ciphertext. This replaces the kN scalar gadget products with a few FFT-based polynomial products; the parameters $(\ell_{ks}, B_{ks}, b_{ks})$ in our tables refer to this operation. Throughout, $\mathbf{S}_p$ and $\mathbf{S}_w$ denote the *large* LWE secret keys (under which PBS outputs are encrypted), while $\mathbf{s}_p$ and $\mathbf{s}_w$ denote the *small* LWE secret keys (under which PBS inputs are encrypted).

As the comparison baseline, we use the *multi-bit extraction* of [36], which combines `PBSmanyLUT` with the Multi-Value Bootstrapping (MVB) technique of [7]: one Blind Rotation of the single-box sign polynomial $f(X) = -\frac{q}{4}\big(1 + X + \cdots + X^{N-1}\big)$, followed by clear-polynomial products on the rotated accumulator, extracts an α-bit chunk into the masked MSB-XOR basis $[\, b_{\alpha-1},\ b_{\alpha-1} \oplus b_{\alpha-2},\ \ldots,\ b_{\alpha-1} \oplus b_0 \,]$ at scale $\Delta_{out} = \frac{q}{2}$; the mask is removed with one LWE addition per bit, or absorbed into the subsequent LUT. Written out for general α, the selector extracting the k-th masked bit is a signed sum of $2^{k+1} - 1$ monomials, so the worst-case output noise is $(2^{\alpha-1} - 1) \cdot V_{pbs}$, where V_{pbs} is the output variance of the PBS (for inputs with a padding bit, the worst case doubles to $(2^\alpha - 1) \cdot V_{pbs}$). We note that [36] instantiates only small chunk sizes ($\tau \leq 3$, writing τ for the digit size α converted per Blind Rotation).

3.2 Our Balanced Selectors for Any Digit Size

In the multi-bit extraction baseline above, the selector extracting the k-th masked bit has squared norm $2^{k+1} - 1$, so the output noise is highly unbalanced: the multiplier of the least significant bit dominates with a factor of $2^{\alpha-1} - 1$. Since the parameters must accommodate the worst output, this factor directly constrains the achievable failure probability. The recent Tetris construction [35] rebalances the basis using prefix XORs, lowering the worst-case factor to $2^{\alpha-2} + 1$.

We introduce a *balanced selector* basis that reduces the worst-case factor to $2^{\alpha-2}$ and makes it uniform across all α outputs. Let $a = N/2^{\alpha-1}$ and let

$B_a(X) = \sum_{r=0}^{a-1} X^r$ be a box of width a. Instead of the single-box test polynomial f, we Blind Rotate the two-box accumulator

$$F(X) \;=\; -\frac{\Delta_{out}}{2}\, X^{-a/2}\left(1 + X^{N/2}\right) B_a(X),$$

and obtain the j-th output by multiplying the rotated accumulator with a selector polynomial

$$G_j \;=\; \sum_{v=0}^{2^{\alpha-1}-1} g_{j,v}\, X^{va}, \qquad g_{j,v} \in \{-1, 0, 1\},$$

followed by an addition of the trivial LWE ciphertext $(\mathbf{0}, \Delta_{out}/2)$. The factor $(1 + X^{N/2})$ replicates every message cell half a period away; folding the target ± 1 sign pattern of each output across the two half-periods realizes it with exactly $2^{\alpha-2}$ nonzero coefficients, i.e., $\|G_j\|_2^2 = 2^{\alpha-2}$ for *every* j. The outputs form the same MSB-XOR basis $[\, b_{\alpha-1},\ b_{\alpha-1} \oplus b_{\alpha-2},\ \ldots,\ b_{\alpha-1} \oplus b_0 \,]$ as the MVB baseline above, and are decoded identically (one LWE addition per bit, or absorbed into the subsequent LUT). For instance, for $\alpha = 3$ (so $a = N/4$), the three multipliers are

$$G_{b_2} = -(1 + X^a), \qquad G_{b_2 \oplus b_1} \;= X^{2a} + X^{3a}, \qquad G_{b_2 \oplus b_0} = -1 + X^a,$$

each with squared norm 2. An exhaustive search over all block-aligned accumulators and integer multipliers (for $\alpha \in \{3, 4\}$) found no full-rank XOR basis with a worst-case factor below $2^{\alpha-2}$, so the balanced basis is minimax-optimal in this model.

3.3 Modified Batched CBS: One Blind Rotation per Digit

In the WoPBS workflow, every decomposed bit must be converted to a GGSW ciphertext by CBS, whose refresh step computes ℓ_{ep} scaled versions of the bit (Section 2). The state of the art [36] already fuses the two for small chunks: one `PBSmanyLUT` converts a $\tau \in \{2, 3\}$-bit chunk, its lanes carrying the ℓ_{ep} GLev levels (the bit's GLWE encryptions at the gadget scales $\frac{q}{B_{ep}^{u}}$, jointly a *GLev* ciphertext [12]), while the chunk's selectors are derived from the same rotated accumulator by clear polynomial products; the worst-case factor $2^{\tau-1} - 1$ of its masked MSB-XOR selectors caps the chunk size. Our modified batched CBS keeps this structure but uses the selector bases of Sections 3.2 and 3.4, whose lower noise factors raise the converted chunk to a full 4-bit digit at the 2^{-128} target (Algorithm 2).

The accumulator interleaves the levels: coefficient i carries the cell sign of F scaled by $\frac{q}{2B_{ep}^{u(i)}}$, where $u(i)$ is the level assigned to lane $i \bmod 2^{\vartheta}$. Since the `PBSmanyLUT` modulus switching (Equation 2, denoted $\mathtt{MS}_{\vartheta}$ in Algorithm 2) rotates in multiples of 2^{ϑ}, lanes never mix, and the cell-aligned selectors G_j act on all lanes simultaneously; the level-u GLWE ciphertext is recovered by dividing

the rotated accumulator by the monomial X^t for lane t. The GLev ciphertexts of each selector are then converted to GGSW ciphertexts with HomTrace and Scheme Switching, exactly as in the standard CBS workflow. In total, an α-bit digit is converted into α GGSW selectors with only $\lceil \ell_{ep}/2^{\vartheta} \rceil$ Blind Rotations (decomposition included) and the selector noise factors of Section 3.2 apply unchanged at every level.

Algorithm 2 Our Modified Batched CBS via `PBSmanyLUT`.

Input: LWE ciphertext $\mathtt{LWE}_{\mathbf{s},q}(m \cdot \Delta_{in})$, where $m = \sum_{i=0}^{\alpha-1} b_i 2^i$ with no padding bit ($\Delta_{in} = \frac{q}{2^\alpha}$)
Input: Accumulator F and selectors $\{G_j\}_{j=0}^{\alpha-1}$ (Section 3.2 or 3.4); GGSW gadget (B_{ep}, ℓ_{ep}); lane count 2^ϑ
Input: Bootstrapping key BSK, automorphism keys ATK, scheme switching key SSK
Output: GGSW ciphertexts $\{\mathtt{GGSW}(s_j)\}_{j=0}^{\alpha-1}$ of the selector bits: for the XOR bases, $s_0 = b_{\alpha-1}$ and $s_j = b_{\alpha-1} \oplus b_{\alpha-1-j}$; for the monomial basis, the bits of its selector code (Section 3.4)

for $c = 0$ to $\lceil \ell_{ep}/2^\vartheta \rceil - 1$ **do**
  Build A_c from F, scaling coefficient i by $\frac{q}{2B_{ep}^{u(i)}}$, where $u(i) = \min\big(c\,2^\vartheta + (i \bmod 2^\vartheta) + 1,\ \ell_{ep}\big)$ ▷ *lanes beyond ℓ_{ep} are unused*
  $\mathtt{ACC}_c \leftarrow \mathtt{BlindRotate}(\mathtt{MS}_\vartheta(\mathtt{ct}), \mathtt{BSK}, A_c)$
  for $t = 0$ to $\min(2^\vartheta,\ \ell_{ep} - c\,2^\vartheta) - 1$ **do**
    $C_{c2^\vartheta+t+1} \leftarrow \mathtt{ACC}_c \cdot X^{-t}$ ▷ *lane extraction*
  end for
end for
for $j = 0$ to $\alpha - 1$ **do**
  $\mathtt{GLev}_j \leftarrow \big\{ G_j \cdot C_u + (\mathbf{0}, \frac{q}{2B_{ep}^u}) \big\}_{u=1}^{\ell_{ep}}$
  $\mathtt{GGSW}(s_j) \leftarrow \mathtt{SS}(\mathtt{HomTrace}(\mathtt{GLev}_j))$
end for
return $\{\mathtt{GGSW}(s_j)\}_{j=0}^{\alpha-1}$

3.4 Monomial Selectors for 4-Bit Digits

For $\alpha = 4$, we can improve upon the balanced factor $2^{\alpha-2} = 4$: we introduce a selector family in which every multiplier is a *single monomial*, attaining the one-Blind-Rotation noise floor $\|G_j\|_2^2 = 1$. The accumulator cells (of width $a = N/8$) are filled with the anti-periodic sign sequence

$$w = (-1,\ -1,\ +1,\ -1,\ -1,\ -1,\ -1,\ -1),$$

extended anti-periodically over the 16 cells of the no-padding grid, and the four selectors are the monomials

$$G_j = X^{a \cdot t_j}, \qquad (t_0, t_1, t_2, t_3) = (0,\ 10,\ 12,\ 14).$$

Table 1. Worst-case selector noise factor (multiplying V_{pbs}) of one-Blind-Rotation decomposition bases for α-bit digits without padding.

Selector basis	**Any** α	$\alpha = 3$	$\alpha = 4$
MSB-XOR (multi-bit extraction [36])	$2^{\alpha-1} - 1$	3	7
Prefix-XOR (Tetris [35])	$2^{\alpha-2} + 1$	3	5
Balanced (ours, Sect. 3.2)	$2^{\alpha-2}$	**2**	4
Monomial (ours, Sect. 3.4)	1 (exists for $\alpha \in \{4, 5\}$)	–	**1**

After the Blind Rotation by message m, output j equals $y_j(m) = 1$ if and only if $w_{m-t_j} > 0$. The joint map $m \mapsto (y_0, \ldots, y_3)$ is a bijection on $[0, 16)$; however, unlike the XOR bases above, it is *not* affine over GF(2). This is harmless in the WoPBS workflow: the subsequent Vertical Packing absorbs the inverse map by permuting its LUT entries, at zero cost. The selector noise is thus exactly V_{pbs} at every level (4× lower than the balanced basis and 7× lower than the MSB-XOR basis of [36]), which directly enlarges the feasible parameter space at a target failure probability. We found this family by an exhaustive search over all anti-periodic sign sequences and monomial tap sets. The same search proves that no all-monomial family exists for $\alpha = 3$: consecutive taps are ruled out by a parity obstruction of the anti-periodic sequence, which the stride-2 taps above evade, and for $\alpha = 3$ no admissible tap set remains; the balanced basis of Section 3.2 is therefore used for 3-bit digits. The search also finds all-monomial families for $\alpha = 5$; in this work, however, we instantiate and parameterize only the $\alpha = 4$ family, since a no-padding 5-bit digit halves the message cell and correspondingly tightens the modulus-switching failure budget. Table 1 summarizes the worst-case selector noise factors of the different bases.

3.5 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_{ep} PBS operations (where ℓ_{ep} is the GGSW decomposition level), we can use the `PBSmanyLUT` technique to compute ℓ_{ep} different functions (details on these ℓ_{ep} PBS functions can be found in Section 2). Our modified batched CBS (Algorithm 2) goes further: the 2^{ϑ} lanes carry the ℓ_{ep} GGSW levels, and one Blind Rotation converts an entire α-bit digit (decomposition and refresh included) into α GGSW selectors. 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], lookup tables of up to 28 bits can be computed using the WoPBS approach.

For small lookup tables (8 to 16 bits), the Vertical Packing time 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 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., $\Delta = \frac{q}{2}$ for binary CBS inputs, or $\Delta = \frac{q}{2^\alpha}$ for the α-bit digits of our batched CBS) 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 (parameter-dependent) can be performed before the noise 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 below.

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 over the input domain $[0, 2N)$ is defined by $f(x + N) = -f(x)$. Since the error in an LWE ciphertext can be negative (it is drawn from a Gaussian), we must center it so that only the message value affects the accumulator rotation. 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 the constant LUT encodes the negacyclic extension of the constant $-\frac{q}{4}$: the evaluated function equals $-\frac{q}{4}$ on the first half of the input domain and $+\frac{q}{4}$ on the second half, satisfying $f(x + N) = -f(x)$.

3.6 Our Proposed Recomposition Methodology

To switch back to the PBS context, the Vertical Packing outputs must be recombined homomorphically into ciphertexts of the target PBS encoding. Our recomposition operates on the κ-bit padded digits produced by VP (Section 3.6). Before applying it, we key switch the underlying secret key from the WoPBS context to the PBS context with a key switching key $\mathtt{KSK}_{\mathbf{S}_w \to \mathbf{s}_p}$ generated at

initialization; the inputs are encrypted under the GLWE secret key $\mathbf{S}_w$ of the WoPBS context, and a key $\mathtt{KSK}_{\mathbf{s}_w \to \mathbf{s}_p}$ can likewise be generated for ciphertexts under the small key $\mathbf{s}_w$.

Multi-Digit Recomposition. A bit-by-bit recomposition would cost one PBS per *bit*. When the switching target is an integer encoding with $\kappa \geq 2$ message bits (e.g., the standard 2-bit message, 2-bit carry, 1-bit padding encoding of TFHE-rs [41]), the PBS count can be reduced by recomposing κ bits at a time. To this end, we exploit the programmability of the VP output encoding, as noted in [2], whose WoP-PBS re-adds the padding bit inside the LUT: we let each VP produce a κ-bit digit with one bit of padding, i.e., $\mathtt{LWE}_{\mathbf{S},q}(d_j \cdot \frac{q}{2^{\kappa+1}})$ with $m = \sum_j d_j \, 2^{j\kappa}$. Because these ciphertexts already carry a padding bit, a standard PBS applies directly, and no negacyclic offset is required. As before, a key switch to the PBS context precedes the PBS operations. Each digit is bootstrapped once with a rescaling LUT that moves it to its target position, and the results are summed (Algorithm 3). This requires α/κ PBS operations instead of α: for $\kappa = 2$, the PBS cost of switching back is *halved*. Alternatively, omitting the final summation leaves the α/κ digits as separate radix blocks $\mathtt{LWE}_{\mathbf{S},q}(d_j \cdot \frac{q}{2^{2\kappa+1}})$ of the target integer encoding (each digit in the message slots, with empty carry slots), ready for radix arithmetic in the PBS context.

The digit size κ trades PBS count against failure probability: the recomposition PBS receives $\kappa + 1$ bits of precision, so its modulus-switching noise grows with κ. For the AES parameter sets of Section 4, padded $\kappa = 4$ outputs overrun the budget, whereas $\kappa = 2$ keeps the failure probability within 2^{-128} while halving the PBS cost; the large-LUT set, whose margins are tuned for 4-bit cells, affords $\kappa = 4$ directly (Table 4).

As an illustrative end-to-end example, consider a privacy-preserving neural network: its linear layers run in the PBS context on radix integers, while a 16-bit activation table (impractical for PBS) is evaluated by converting the four digits with the batched CBS, applying Vertical Packing, and recomposing back to the radix encoding, deferring the recomposition when several nonlinear functions follow each other.

Algorithm 3 Our Multi-Digit Recomposition.

Input: LWE ciphertexts $\{\mathtt{ct}_j\}_{j=0}^{\alpha/\kappa-1}$ of form $\mathtt{LWE}_{\mathbf{s},q}(d_j \cdot \frac{q}{2^{\kappa+1}})$, where $m = \sum_j d_j \, 2^{j\kappa}$ (output by VP with one padding bit, after key switch)
Input: Lookup tables $\{\mathtt{lut}_j\}$, where $\mathtt{lut}_j$ encodes $f_j : x \mapsto x \cdot \frac{q}{2^{\alpha+1-j\kappa}}$
Input: Bootstrapping key $\mathtt{BSK}$
Output: LWE ciphertext $\mathtt{LWE}_{\mathbf{S},q}(m \cdot \frac{q}{2^{\alpha+1}})$
1: **for** $j = 0$ to $\alpha/\kappa - 1$ **do**
2: $\quad \mathtt{ct}'_j \leftarrow \mathtt{PBS}(\mathtt{ct}_j, \mathtt{BSK}, \mathtt{lut}_j)$ ▷ *standard PBS; the input has a padding bit*
3: **end for**
4: **return** $\mathtt{ct_out} \leftarrow \sum_{j=0}^{\alpha/\kappa-1} \mathtt{ct}'_j$

3.7 Modulus-Switching Noise Reduction

PBSmanyLUT. The drift technique of [3] squares the failure probability of a given parameter set (from 2^{-r} to 2^{-2r}); in this work we employ the Centered Mean Noise Reduction [32]: as the mask is public, the mean of its rounding errors is computed in the clear and removed from the body before the modulus switch. The work in [32] already pairs the compensation with `PBSmanyLUT` at $\vartheta = 1$; we extend it to the special modulus switching of Equation 2 at arbitrary ϑ and, below, to the multi-bit Blind Rotation, while the CBS and Vertical Packing noise follows the analysis of [36]. The `PBSmanyLUT` modulus-switching noise (adopted from [12]) is:

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

where:

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

Here, we have $w = 2N\cdot 2^{-\vartheta}$, which is the term multiplied with the LWE element during modulus switching, q' equals $q\cdot 2^{-\varkappa}$ ($= q$ in this work, since $\varkappa = 0$), and σ_{in} is the noise standard deviation of the input LWE ciphertext. By applying the Centered Mean preprocessing, the $\mathbb{E}(s_i)$ contribution is eliminated from the variance term in Eq. 4. Therefore:

$$\mathrm{Var}(E_{res}) = \frac{w^2\sigma_{in}^2}{q'^2} + \frac{1}{12} - \frac{w^2}{12q'^2} + \frac{n}{48} + \frac{nw^2}{24q'^2}.$$

The correctness condition $\Gamma\sqrt{\mathrm{Var}(E_{res})} < \frac{w\Delta_{in}}{2q'}$, with Γ the target margin-to-noise ratio, then rearranges, after multiplying through by $\frac{q'^2}{\Gamma^2 w^2}$, to the parameter bound

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

For practical parameters ($w \ll q'$), the dominant mask term $n\cdot\mathrm{Var}(\bar{a}_i)\big(\mathrm{Var}(s_i)+\mathbb{E}^2(s_i)\big) \approx \frac{n}{24}$ is halved to $\approx \frac{n}{48}$: the compensation eliminates its $\mathbb{E}^2(s_i)$ part, halving the overall variance in the regime where this term dominates and reducing the noise standard deviation by $1/\sqrt{2}$. The freed budget buys a $\sqrt{2}$ larger σ_{in}, half a bit of extra precision (a $\sqrt{2}$ smaller Δ_{in}), or a squared failure probability (Γ, the ratio of the decryption margin to the noise standard deviation, grows by $\sqrt{2}$, with $P_{\text{fail}} = 1 - \mathrm{erf}(\frac{\Gamma}{\sqrt{2}})$). Since increasing ϑ by 1 quadruples the q'^2/w^2 terms in the bound, the reduction alone does not pay for an extra `PBSmanyLUT` slot; the higher-precision configurations of Section 4 additionally rely on the reduced selector factors of Sections 3.2 and 3.4.

Multi-bit Blind Rotation. Our GPU evaluation uses the multi-bit Blind Rotation [5, 44] which, as implemented in TFHE-rs [41], processes the mask in n/d groups of d secret bits and modulus-switches each of the 2^d subset sums $\sum_{i \in S} a_i \bmod q'$ of a group directly (rounding a sum does not commute with summing rounded terms). The rounding error of a group is that of the *selected* subset sum: zero for the empty subset, and distributed as $\bar{a}_i$ otherwise, since subset sums of a uniform mask are uniform. As naive per-coefficient mean compensation is counterproductive under this rounding [32] (which analyzes compensated unrolled rotations for $d \leq 3$ in the average case, under a modified rounding rule), we instead subtract from the body, per group, the average of all 2^d subset-sum rounding errors: the exact conditional mean of the group error given the (public) mask, requiring no change to the TFHE-rs rounding. Any two distinct nonempty subset sums are independent for a uniform mask: the two subsets differ in at least one mask coefficient, and that coefficient's fresh uniform value re-randomizes one sum without affecting the other, so the pair is jointly uniform. The law of total variance then yields, with $p = 1 - 2^{-d}$:

$$\begin{aligned} \mathrm{Var}(E_{res}) = \frac{w^2 \sigma_{in}^2}{q'^2} + \mathrm{Var}(\bar{b}) \\ + \frac{n}{d}\Big(p^2 \,\mathrm{Var}(\bar{a}_i) + p(1-p)\,\mathbb{E}^2(\bar{a}_i)\Big). \end{aligned} \tag{5}$$

At $d = 1$ ($p = \frac{1}{2}$) this reduces exactly to the centered variance derived above; we additionally confirmed Eq. 5 by exhaustive enumeration over small tori. For the $d = 4$ grouping used in our GPU experiments, the dominant mask term $\frac{n}{d} p^2 \,\mathrm{Var}(\bar{a}_i) \approx \frac{225}{256} \cdot \frac{n}{48}$ falls slightly below its $d = 1$ counterpart, and our GPU parameter set (Section 4) is evaluated under this model.

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]). All our CPU-based experiments are performed on an Ubuntu workstation with an AMD Ryzen Threadripper 7970X CPU (32 cores) and 157 GB of RAM. We remark that all CPU experimental results are based on *single-threaded execution*. For our GPU experiments, we employ a single NVIDIA GeForce RTX 4090 GPU with 24 GB of memory. 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 [36] and the noise reduction technique from [32], we demonstrate that it is possible to generate more efficient parameters at the

$\texttt{IND-CPA}^D$ failure budget of 2^{-128}, as shown in Table 2. 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.[1] We remark that [36] 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 [36]), which increases the failure probability, but lowers the execution time by a few seconds, depending on the parameters. We do not employ HalfCBS: every round of our evaluation, including the first, runs the full CBS. In our evaluation, all of our parameter sets target a failure probability below 2^{-128}, as required by the $\texttt{IND-CPA}^D$ threat model (Section 2). Table 2 consolidates all parameter sets used in this work, i.e., the full binary AES workflow (*AES-BIN*, combined with its GPU variant) and the batched AES workflow (*AES-BATCH*) of Section 4.2, together with the large-LUT sets of Section 4.3, next to the corresponding sets of [36] and Maverick [39]. The single-threaded CPU runtimes are reported in Table 3, and the GPU evaluation in Section 4.5 (Table 5).

Since the AES circuit requires `XOR` operations (implemented as LWE additions) between SBox evaluations, a packed digit cannot simply be carried from one round to the next; the batched AES workflow of Section 4.2 handles these additions in a redundant encoding, keeping every round uniform. Concretely, one binary AES round proceeds as follows: the 8 bits of each state byte are converted to GGSW selectors by CBS (128 CBS per round), and Vertical Packing evaluates 8-bit LUTs that fuse the SBox with the MixColumns constant multiplications, returning $S(x)$, $2 \cdot S(x)$, and $3 \cdot S(x)$ as binary LWE ciphertexts. ShiftRows is a free reordering of ciphertexts; MixColumns reduces to three `XOR`s (LWE additions) per output byte over the precomputed multiples, and AddRoundKey to one further addition of the encrypted round key. The initial AddRoundKey is a plain `XOR` with the first round key, and the final round omits MixColumns.

4.2 Batched AES Evaluation

The binary workflow converts all 128 state bits with CBS in every round, i.e., 128 Blind Rotations per round per block. The MixColumns and AddRoundKey `XOR`s cannot be embedded entirely into the lookup tables, as this would require 32-bit LUTs, which is impractical. We instead halve the Blind Rotation count by keeping the state in a redundant *spread* encoding between rounds and letting the batched CBS of Algorithm 2 absorb the `XOR` carries.

Spread encoding. Each pair of `XOR` output bits is packed into a single LWE ciphertext with three slots $[\, p_1 \,|\, c_0 \,|\, p_0 \,]$ at scales $\frac{q}{2}$, $\frac{q}{4}$, and $\frac{q}{8}$, where p_0, p_1 are the two result parities and c_0 is a garbage carry slot that is never extracted. The SBox and the MixColumns constants are folded into pairwise 16-bit VP LUTs: for each output pair, one VP outputs $T_1 = \text{spread}(2\,S(a) \oplus 3\,S(b))$ and a second outputs $T_2 = \text{spread}(S(c) \oplus S(d))$, where $\text{spread}(\cdot)$ places its two bits at the $\frac{q}{2}$ and $\frac{q}{8}$ slots. The MixColumns and AddRoundKey `XOR`s of one column then

[1] While AES *decryption* is used for transciphering, the AES *encryption* is also widely researched and implemented in the literature [34].

Table 2. All parameter sets used in this work and in the compared prior works, at 128 bits of security. The *AES-BIN* row combines the CPU and GPU variants of one set: "x / y" entries give the CPU and GPU values, which differ only in the multi-bit Blind-Rotation grouping d and the resulting P_{fail} (grouped centered model, Section 3.7). (ℓ_{ks}, B_{ks}) is the GLWE key switch to the small key, with common polynomial sizes $b_{ks} = 256$, 128, 32, and 64 for the [36] AES, *AES-BIN*, *AES-BATCH*, and LUT rows; the LUT set in [36] uses a standard LWE-to-LWE key switch. The LUT set in [36] and Maverick's CBS-II are published at $P_{\text{fail}} \approx 2^{-40}$; Maverick's "128-bit" figure is the security level λ. A "–" marks unreported values: Maverick [39] reports only its dimensions and levels, omitting all decomposition bases and keyswitch parameters.

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	d	P_{fail}
[36] (AES, Set-II)	768	1024	2	3	2^4	2	2^{15}	6	2^7	2	2^{17}	4	2^4	2	–	$2^{-368.3}$
[36] (LUT)	769	2048	1	3	2^4	2	2^{15}	7	2^7	2	2^{16}	4	2^4	2	–	2^{-40}
Maverick (CBS-II) [39]	676	2048	–	–	–	2	–	6	–	2	–	–	–	–	–	$\approx 2^{-40}$
Ours (AES-BIN, CPU / GPU)	640	1024	2	4	2^3	2	2^{15}	3	2^{12}	2	2^{16}	2	2^7	1	–/4	$2^{-184.8}$ / $2^{-169.7}$
Ours (AES-BATCH)	736	1024	2	7	2^2	3	2^{12}	3	2^{12}	2	2^{17}	2	2^8	1	–	$2^{-136.2}$
Ours (LUT, $\tau = 4$, monomial)	832	2048	1	5	2^3	3	2^{12}	4	2^{10}	3	2^{13}	2	2^8	1	–	2^{-144}

Table 3. Single-threaded CPU runtimes of AES-128 transciphering (three verified runs each). The [36] row is our same-machine implementation of its full-CBS binary workflow at its Set-II parameters (reported at 19.4 s on its own platform [36]); *AES-BIN* is the same workflow with the reverse-normalized trace and centered-mean noise reduction, enabling its cheaper parameter set.

Set	Runtime (s)
[36] (Set-II)	20.076
Ours (AES-BIN)	**14.198**
Ours (AES-BATCH)	**12.342**

reduce to the LWE addition $T_1 + T_2 + rk$, with rk the (encrypted) round key in the same spread layout. Since at most three addends meet in each slot, every slot sum lies in $[0, 3]$: each parity stays in place, the low-slot carry is confined to the garbage slot, and the top-slot carry wraps modulo q. Consequently, bit 0 of the packed chunk *is* p_0 and bit 2 *is* p_1; the `XOR` results are read off as plain bits, and no carry-propagation LUTs are needed.

Round structure. Each 3-bit chunk is converted by the batched CBS with the balanced selector basis of Section 3.2, extracting only the two parity selectors $[\, b_2,\ b_2 \oplus b_0 \,]$; the middle (garbage) bit is skipped, together with its HomTrace and Scheme Switching. One Blind Rotation per chunk yields both selectors at all ℓ_{ep} levels, so each AES round performs 64 Blind Rotations per block instead of 128.

The *AES-BATCH* row of Table 2 reports our batched AES parameter set, which meets the 2^{-128} failure target while accounting for the pair-VP noise (two 16-bit VP outputs plus one fresh round-key addition per chunk) entering the next batched CBS.

The resulting 1.15× speedup of AES-BATCH over AES-BIN (Table 3) matches the Blind-Rotation accounting: the rotation count halves (64 vs. 128 per round), while each rotation is $\approx 1.7\times$ more expensive ($\ell_{pbs} = 3$ vs. 2 and $n = 736$ vs. 640, Table 2), for a theoretical $2 \div 1.7 \approx 1.16\times$.

4.3 Faster LUT Evaluation

Fast LUT evaluation on base-β inputs is obtained directly from the batched CBS of Algorithm 2: each α-bit digit is converted to α GGSW selectors with $\lceil \ell_{ep}/2^{\vartheta} \rceil$ Blind Rotations, using the monomial selectors for $\alpha = 4$ (base $\beta = 16$) and the balanced selectors for $\alpha = 3$ (base $\beta = 8$). The gadget scaling factors $\Delta_i = \frac{q}{B_{ep}^i}$, $i \in [1, \ell_{ep}]$, ride the `PBSmanyLUT` lanes, so no separate decomposition step is needed.

We remark that the Mean Noise Reduction (Section 3.7) is what makes the single-Blind-Rotation conversion of a 4-bit digit feasible: without it, 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 $\frac{B}{\log_2(\beta_{out})}$ Vertical Packing operations, where B is the output bit-length of the evaluated LUT, to maximize efficiency. Then, to evaluate more LUTs, we apply the same procedure repeatedly, 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 [12, 26, 29] 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 becomes very high, around 2^{-34} for the $\tau = 4$ LUT set of Table 2, which is impractical. Here, the FD PBS requires two separate Blind Rotation operations each time. Our multi-digit recomposition (Section 3.6) resolves this trade-off: the VP emits padded κ-bit digits to which a standard (single-Blind-Rotation) PBS applies directly, with the failure probability within the 2^{-128} budget ($\kappa = 2$ for the AES parameter sets and $\kappa = 4$ for the LUT set in Table 4), reducing the recomposition PBS count by $\kappa\times$. Moreover, the batched CBS of Algorithm 2 doubles the digit size of the fused MVB extraction of [36]: it produces all selectors of a full 4-bit digit, at all ℓ_{ep} levels, from $\lceil \ell_{ep}/2^{\vartheta} \rceil$ Blind Rotations, thanks to the lower selector noise factors of Table 1.

4.4 Large LUT Parameters and Experimental Evaluation

In Table 2 (the LUT row), we report an optimized parameter set for large LUT evaluation with the batched CBS of Algorithm 2, meeting $P_{\text{fail}} \leq 2^{-128}$: a $\tau = 4$ set using the monomial selector basis of Section 3.4 ($\vartheta = 1$, so a single Blind

Rotation covers both GGSW levels of a 4-bit digit). The set operates on input LWE ciphertexts without a padding bit; if a padding bit is present, it can be removed by adding the ciphertext to itself (with this addition accounted for in the noise analysis). Whereas the selector noise of [36] limits its fused conversion to two-bit chunks per Blind Rotation, our monomial basis converts an entire 4-bit digit (decomposition and refresh included) with a single Blind Rotation, halving the rotation count. Since the Blind Rotation is by far the most expensive operation in the workflow, this reduction yields substantial performance gains. We further replace the high-precision HomTrace of [36] with the reverse-normalized trace evaluation [28] (Section 2), which avoids the cubic noise growth of the standard trace without switching to a larger GLWE dimension. Combined with the noise-factor-1 monomial selectors and the mean-compensated modulus switching (Section 3.7), the $\tau = 4$ set attains a modeled per-digit failure probability of 2^{-144} in chained (LUT-after-LUT) operation.

For our evaluation, we compute 8-to-8, 12-to-12, and 16-to-16 LUTs and compare the execution times against [36] in Table 4. Table 4 also isolates the switching costs and Blind-Rotation counts. Switching *in* is folded into the CBS: decomposition adds no separate operations, and the isolated pre-PBS key switch costs 0.3 ms for the four digits of a 16-bit input, versus 17.1 ms for its sixteen bit-level switches in [36]. For switching *back*, our VP emits padded 4-bit digits (which is affordable since the padded digit lands with the same decryption margin as the chained no-padding digits). Consequently, recomposition takes one rescaling PBS per digit (Section 3.6), whereas the bit-granular pipeline of [36] recomposes at one PBS per bit, as in the Recomposer of [1] (Section 5). Our individual PBS is more expensive ($\ell_{pbs} = 3$ vs. 2, $n = 832$ vs. 769: $\approx 1.4\times$), which is why the recomposition speedup is $16/4 \div 1.4 \approx 2.9\times$. The same accounting explains the evaluation speedups: half the Blind Rotations at $\approx 1.5\times$ the per-rotation cost yields the theoretical $1.33\times$ on the Blind-Rotation share (measured 112.0 $\rightarrow$ 85.6 ms), and the rest of the observed $1.85\times$ comes from the stages this estimate ignores: HomTrace (55.5 $\rightarrow$ 12.3 ms: half the GLev levels per selector, each traced by a cheaper level-4 reverse-normalized trace instead of a level-7 standard one) and the per-bit conversion overhead (pre-PBS key switches, selector products, GGSW packing; 17.0 $\rightarrow$ 1.4 ms).

Notably, the parameter set tolerates the noise generated by a 16-bit LUT, which is $m \cdot \mathrm{V}_{cbs}$ with $m = 16$, where V_{cbs} is the output noise variance of one converted GGSW selector, while remaining below the 2^{-128} failure target.

Reaching $P_{\mathrm{fail}} \leq 2^{-128}$ with 4-bit digits. With the MSB-XOR selector basis of [36], meeting $P_{\mathrm{fail}} \leq 2^{-128}$ requires either decreasing the digit size to $\alpha = 3$ or doubling the ring dimension N, since the worst-case selector factor $2^{\alpha-1} - 1$ consumes most of the noise budget. The factor-1 monomial selectors remove this restriction: as Table 2 shows, $\tau = 4$ digits meet the target directly at $N = 2048$ with $\vartheta = 1$, avoiding the more-than-doubled execution time of a larger ring dimension. For applications requiring an even lower failure probability, the balanced basis with $\tau = 3$ digits (Section 3.2) provides additional margin.

Table 4. Comparison of execution times for different LUT sizes between our modified batched CBS ($\tau = 4$, monomial selectors, $P_{\text{fail}} \leq 2^{-128}$) and [36] at its published parameters (reported at $P_{\text{fail}} \approx 2^{-40}$ in [36]; Table 2). Both workflows are measured single-threaded on the same machine, averaged over five runs. BR counts the Blind Rotations of each phase; recomposition is the isolated switching-back cost.

		Evaluation			Recomposition		
LUT Size	**Work**	**BR**	**ms**	**Speedup**	**BR**	**ms**	**Speedup**
8-to-8	[36]	4	90.92	1.0×	8	121.21	1.0×
	Ours	**2**	**50.03**	**1.82**×	**2**	**42.46**	**2.85**×
12-to-12	[36]	6	138.97	1.0×	12	180.52	1.0×
	Ours	**3**	**73.69**	**1.89**×	**3**	**63.71**	**2.83**×
16-to-16	[36]	8	189.92	1.0×	16	243.32	1.0×
	Ours	**4**	**102.77**	**1.85**×	**4**	**83.90**	**2.90**×

Table 5. GPU evaluation of AES-128 transciphering on a single RTX 4090, per-block latency amortized over a batch of 8 blocks. Our row is the GPU variant of the *AES-BIN* set in Table 2 ($d = 4$); the Maverick row is its CBS-II configuration evaluating a single block, at $P_{\text{fail}} \approx 2^{-40}$ [39].

Set	**GPU**	**Latency (ms/block)**	P_{fail}
Maverick (CBS-II) [39]	RTX 4090	390	$\approx 2^{-40}$
Ours (GPU-BIN)	RTX 4090	**331.1**	$2^{-169.7}$

4.5 GPU Acceleration of WoPBS

The main bottleneck of CBS is the Blind Rotation inside PBS/`PBSmanyLUT`, which parallelizes well on GPUs: decrypting c AES-128 blocks requires $128 \cdot c$ `PBSmanyLUT` operations per round, and the subsequent HomTrace, Scheme Switching, Vertical Packing, key switching, and modulus switching (with its Mean Compensation preprocessing) are likewise perfectly parallelizable.

We designed our core operations in CUDA, leveraging the CUDA FFT engine of TFHE-rs [41], to evaluate AES transciphering fully on the GPU. Table 5 reports our GPU evaluation on a single NVIDIA GeForce RTX 4090, alongside the recent Maverick GPU bootstrapping accelerator [39] on the same GPU. Our binary workflow evaluates one AES-128 block in 331 ms amortized over a batch of eight blocks, while Maverick's higher-level CBS-II configuration requires 390 ms for a single block. We emphasize that Maverick's parameters correspond to a decryption failure probability of approximately 2^{-40} [39], whereas our parameter sets target the `IND-CPA`^{D} failure budget of 2^{-128}.

The multi-bit (parallel) Blind Rotation [24] increases the bootstrapping key size and output variance by $\frac{m^d-1}{d}$, where m is the secret-key cardinality ($m = 2$

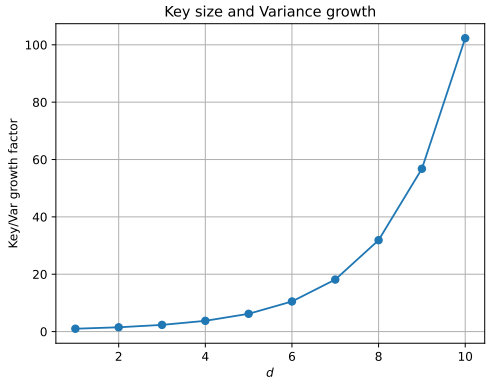

Fig. 1. Noise growth and key expansion factor for the parallel variant of Blind Rotation [24].

for the binary/GINX keys [17]) and d the grouping size; it suits GPUs, where the original Blind Rotation is sequential. Figure 1 shows the growth ratio.

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 $r = 3.75$. The resulting $3.75 \cdot \mathrm{V}_{pbs}$ growth is accounted for in the GPU variant of the *AES-BIN* set (the $d = 4$ column of Table 2).

Compared to the AES Set-II parameters of [36] (Table 2), our sets use a smaller GGSW level ℓ_{ep} and trace level ℓ_{tr}, which results in smaller GGSW ciphertexts and a smaller trace evaluation key. As GPU memory limits how many ciphertexts can be evaluated at a time, and GGSW ciphertexts are a few times larger than their LWE counterparts, minimizing ℓ_{ep} is crucial: for equal inputs, our GGSW ciphertexts are $2\times$ smaller than with the set of [36].[2] This lets us batch more inputs, maximizing throughput ($\ell_{ep} = 1$ is infeasible: the error blows up).

4.6 Secret Key Distribution

The Mean Noise Reduction [32] applies only to binary secret keys, which the most practical TFHE implementations assume [41]; our approach therefore targets the binary-key variant of TFHE.

5 Related Works

A body of work evaluates AES under TFHE with PBS-centric workflows. Trama *et al.* [34] rely exclusively on PBS (the SBox via the tree-based method of [23] with MVB [7], and even `XOR`s via PBS and packing key switching), requiring 270 s

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

single-threaded at $P_{\text{fail}} = 2^{-23}$. Bon *et al.* [4] compute `XOR`s as LWE additions using p-encodings, but every `AND` gate of the SBox Boolean circuit still invokes a PBS (105 s at 2^{-40}); [1] combines the two approaches under an unconventional odd plaintext modulus $p = 17$; and [33] evaluates variable-sized LUTs on no-padding ciphertexts via packing key switching and CMUX trees. All four scale poorly to larger lookup tables or lower failure probabilities, whereas our workflow computes `XOR`s as plain LWE additions in both contexts and evaluates large LUTs via CBS and Vertical Packing at $P_{\text{fail}} \leq 2^{-128}$ (Section 4).

The Hippogryph framework of [1] also includes a bit-to-digit *Recomposer*: each bit is bootstrapped into an intermediary p-encoding and the results are summed, at a cost of one PBS *per bit*; their Property 1 shows that a recomposer using linear operations and one PBS per bit exists only for an *odd* output modulus, which forces the nonstandard $p = 17$. Our multi-digit recomposition differs on both counts: the Vertical Packing outputs κ-bit digits that already carry a padding bit, so standard power-of-two TFHE encodings apply and a single ordinary PBS recomposes an entire digit, requiring α/κ PBS instead of one per bit (Section 3.6).

Closest to our work is the CBS line of Wei *et al.* [40] and Wang *et al.* [36–38], which successively accelerates CBS with `PBSmanyLUT`, Scheme Switching, and HomTrace-based conversions; the state-of-the-art [36] adds a HomTrace-noise-managing preprocessing step and a high-precision HomTrace, yielding the fastest prior AES evaluation. Its workflow remains bit-oriented, however: the `PBSmanyLUT` failure probability forces it to convert only 1–2 bits per Blind Rotation, which is costly in our setting of consecutive large-LUT evaluations and context switches. [2] engineers compatible parameter couples so the padded-PBS and WoP-PBS patterns can interoperate, leaving dedicated switching procedures as future work; our algorithms complete this step.

More recently, Tetris [35] introduced a versatile TFHE LUT evaluation framework geared towards an FHE instruction set architecture; like our work, it builds a batched CBS on the fused `PBSmanyLUT`+MVB conversion of [36]. Most relevant to our work, it redesigns the selector polynomials of the MVB-based bit decomposition into a prefix-XOR basis $[\, b_{\tau-1},\ b_{\tau-1} \oplus b_{\tau-2},\ b_{\tau-1} \oplus b_{\tau-2} \oplus b_{\tau-3},\ \ldots]$, lowering the worst-case selector noise factor from $2^{\tau-1}-1$ to $2^{\tau-2}+1$. Our selector bases strictly improve upon this: the balanced basis attains a *uniform* factor of $2^{\alpha-2}$, shown minimax-optimal among full-rank XOR bases by an exhaustive search, and the monomial family reaches the factor-1 floor for 4-bit digits. Since selector bases are interchangeable within our modified batched CBS, the prefix-XOR basis is dominated in this setting by both (Table 1).

On the hardware side, Maverick [39] accelerates TFHE bootstrapping (including the CBS of [36]) through GPU algorithm–hardware co-design: the length-n Blind Rotation chain is split into $\sqrt{n}$ parallel sub-chains whose GLWE accumulators are re-merged through per-sub-chain GLWE-to-GGSW conversions (HomTrace and Scheme Switching), and the NTT is terminated at an intermediate stage to feed Tensor-Core matrix kernels. This design, however, is tied to specific dimensions and hardware: the sub-chain decomposition requires an

LWE dimension with an integer square root (otherwise padded to the next perfect square), while the NTT termination depth and Tensor-Core data layouts are tuned per parameter set. Each sub-chain merge also adds HomTrace and Scheme-Switching noise, and its parameters follow the $\approx 2^{-40}$ failure methodology of [36]. Our GPU workflow instead exploits the abundant *independent* Blind Rotations of the batched AES and LUT workloads together with the multi-bit Blind Rotation (Section 4.5), imposes no structural constraint on n, and meets the $\texttt{IND-CPA}^D$ budget of 2^{-128} at comparable per-block latency on the same GPU (Table 5).

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 balanced and monomial selector bases that lower the worst-case noise of one-Blind-Rotation decomposition to its floor, a modified batched CBS that converts an entire digit into GGSW selectors with a single Blind Rotation (doubling the two-bit fused chunks of [36]), and a multi-digit recomposition that reconstructs ciphertexts with a bit of padding while reducing the PBS count to α/κ. We further optimize the CBS workflow by adopting the centered mean noise reduction for both the `PBSmanyLUT` and the multi-bit Blind Rotation modulus switching, significantly reducing its noise growth and thus the failure probability for the same parameters. This yields up to 1.9× faster large-LUT evaluation than the state of the art at the 2^{-128} failure budget. Furthermore, our binary and batched workflows evaluate AES-128 transciphering in 14.2 s and 12.3 s single-threaded, which is 1.4× and 1.6× faster than the workflow of [36] at its Set-II parameters on the same machine, with the batched evaluation 1.15× faster than our binary one. 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, evaluating one AES-128 block in 331 ms on a single RTX 4090, which is 43× faster than our single-threaded binary CPU workflow, and 1.18× faster than the recent Maverick GPU accelerator [39] (390 ms per block at a failure probability of only about 2^{-40}). In addition, we introduce a novel batched AES evaluation that keeps the state in a redundant spread encoding and halves the per-round Blind Rotations, and all of our parameter sets meet the $\texttt{IND-CPA}^D$ failure target of 2^{-128}. Overall, our framework lays a robust foundation for future advancements in scalable fully homomorphic encryption.

Acknowledgments

This work has been supported by NSF Award #2239334.

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: EUROCRYPT. pp. 181–211. Springer (2025)
4. Bon, N., Pointcheval, D., Rivain, M.: Optimized homomorphic evaluation of boolean functions. IACR Trans. CHES **2024**(3), 302–341 (2024)
5. Bourse, F., Minelli, M., Minihold, M., Paillier, P.: Fast homomorphic evaluation of deep discretized neural networks. In: CRYPTO. pp. 483–512. Springer (2018)
6. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. ACM TOCT **6**(3), 1–36 (2014)
7. Carpov, S., Izabachène, M., Mollimard, V.: New techniques for multi-value input homomorphic evaluation and applications. In: CT-RSA. 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: ACM CCS. pp. 2505–2519 (2024)
9. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: ASIACRYPT. pp. 409–437. Springer (2017)
10. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: ASIACRYPT. pp. 3–33. Springer (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: ASIACRYPT. 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: EUROCRYPT. pp. 617–640. Springer (2015)
16. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR ePrint 2012/144 (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: EUROCRYPT. pp. 528–558. Springer (2016)
18. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: ACM STOC. p. 169–178. 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: CRYPTO. 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. IACR ePrint 2021/811 (2021)
21. Gouert, C., Joseph, V., Dalton, S., Augonnet, C., Garland, M., Tsoutsos, N.G.: Hardware-accelerated encrypted execution of general-purpose applications. PoPETs (2025)
22. Gouert, C., Mouris, D., Tsoutsos, N.G.: HELM: Navigating homomorphic encryption through gates and lookup tables. IEEE TIFS (2025)
23. Guimarães, A., Borin, E., Aranha, D.F.: Revisiting the functional bootstrap in TFHE. IACR Trans. CHES pp. 229–253 (2021)
24. Joye, M., Paillier, P.: Blind rotation in fully homomorphic encryption with extended keys. In: CSCML. 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. IACR ePrint 2021/1135 (2021)
27. Langmead, B., Nellore, A.: Cloud computing for genomic data analysis and collaboration. Nature Reviews Genetics **19**(4), 208–219 (2018)
28. Lee, K.H., Yoon, J.W.: Homomorphic field trace revisited: Breaking the cubic noise barrier. IACR Trans. CHES **2026**(1), 105–134 (2026)
29. Liu, Z., Micciancio, D., Polyakov, Y.: Large-precision homomorphic sign evaluation using FHEW/TFHE bootstrapping. In: ASIACRYPT. pp. 130–160. Springer (2022)
30. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: EUROCRYPT. pp. 1–23. Springer (2010)
31. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM **56**(6), 1–40 (2009)
32. de Ruijter, T., D'Anvers, J.P., Verbauwhede, I.: Don't be mean: Reducing approximation noise in TFHE through mean compensation. IACR ePrint 2025/809 (2025)
33. Schild, L., Abidin, A., Preneel, B.: Fast transciphering via batched and reconfigurable lut evaluation. IACR Trans. CHES **2024**(4), 205–230 (2024)
34. Trama, D., Clet, P.E., Boudguiga, A., Sirdey, R.: A homomorphic AES evaluation in less than 30 seconds by means of TFHE. In: WAHC. pp. 79–90 (2023)
35. Wang, R., Bai, J., Shen, X., Lu, X., Li, Z., Xiang, B., Wang, Z., Wang, H., Zhao, L., Wang, K., Hou, R.: Tetris: Versatile TFHE LUT and its application to FHE instruction set architecture. IACR ePrint 2025/1623 (2025)
36. Wang, R., Ha, J., Shen, X., Lu, X., Chen, C., Wang, K., Lee, J.: Refined TFHE leveled homomorphic evaluation and its application. In: ACM CCS. pp. 2309–2323 (2025)
37. Wang, R., Wei, B., Li, Z., Lu, X., Wang, K.: TFHE Bootstrapping: Faster, Smaller and Time-Space Trade-Offs. In: ACISP. pp. 196–216. Springer (2024)
38. Wang, R., Wen, Y., Li, Z., Lu, X., Wei, B., Liu, K., Wang, K.: Circuit bootstrapping: faster and smaller. In: EUROCRYPT. pp. 342–372. Springer (2024)
39. Wang, Z., He, H., Zhao, L., Niu, Q., Meng, D., Hou, R.: Maverick: Rethinking TFHE bootstrapping on GPUs via algorithm-hardware co-design. In: ACM ASPLOS. pp. 1168–1184 (2026)
40. Wei, B., Wang, R., Li, Z., Liu, Q., Lu, X.: Fregata: Faster homomorphic evaluation of AES via TFHE. In: ISC. pp. 392–412. Springer (2023)

41. Zama: TFHE-rs: A Pure Rust Implementation of TFHE (2022), `https://github.com/zama-ai/tfhe-rs`
42. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: ACM CCS. pp. 305–316 (2012)
43. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-tenant side-channel attacks in PaaS clouds. In: ACM CCS. pp. 990–1003 (2014)
44. Zhou, T., Yang, X., Liu, L., Zhang, W., Li, N.: Faster bootstrapping with multiple addends. IEEE Access **6**, 49868–49876 (2018)