BatchBoot: Fast Batched Bootstrapping for TFHE scheme and Practical Applications

Zhihao Li[1], Hongyu Wang[2], Yuan Zhao[1], Lichun Li[1], Zhiwei Wang[3]*
Jiaxing He[1], Changzheng Wei[1], Ying Yan[1], and Lifeng Guo[2]
[1]Ant Digital Technologies, Ant Group [2]Shanxi University
[3]State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS
{lzh458070, caesar.zy, lichun.llc, jiaxing.hjx, changzheng.wcz, fuying.yy}@antgroup.com
{wanghongyu, lfguo}@sxu.edu.cn, wangzhiwei@iie.ac.cn

Abstract

Torus-based Fully Homomorphic Encryption (TFHE) is distinguished by its unique bootstrapping mechanism, which enables arbitrary computation while refreshing the noise budget. However, this mechanism exhibits limited scalability since it can handle only a *single* encrypted message at a time. To address this, recent studies have proposed *batched* bootstrapping schemes that allow TFHE to process ciphertexts in parallel, thereby achieving promising amortization benefits. Despite these advances, this emerging direction remains underexplored, leaving ample room for further investigation.

In this paper, we present BATCHBOOT, an efficient batched bootstrapping framework for TFHE that enables amortized processing of encrypted messages. Specifically, our work makes three key contributions. First, we redesign the core submodule, i.e., homomorphic polynomial multiplication, to substantially reduce the reliance on expensive FFT operations. Second, we propose a sparsity-aware message packing strategy that flexibly supports varying packing scales. Third, we extend functional bootstrapping to circuit bootstrapping, thereby greatly enhancing the expressiveness of supported functions. Together, these contributions enable BATCHBOOT to deliver a 2.4× speedup over the state-of-the-art batched scheme (Guimarães et al., CCS'25) and a 43.8× improvement over the non-batched TFHE-rs implementation.

At the application level, we highlight the versatility of BatchBoot through two practical use cases. First, we present the first TFHE-based PSI protocol under the unbalanced setting, which achieves a 294× reduction in communication cost and a 4.1× speedup compared to the best BFV-based solution (PEPSI, USENIX Security'24). Second, we design an 8-bit FHE instruction set based on the BATCHCBOOT that delivers up to a 5.4× speedup over the existing results (Wang et al., CCS'25).

*Zhiwei Wang is the corresponding author.

1 Introduction

Fully Homomorphic Encryption (FHE) is a cornerstone of privacy-preserving computation, enabling arbitrary operations directly on encrypted data without compromising confidentiality [31]. Over the past decade, two primary classes of FHE schemes have emerged, reflecting different trade-offs between functionality and efficiency. Word-wise schemes, such as BGV [10], BFV [30], and CKKS [19], pack multiple plaintexts into a single ciphertext to enable SIMD-style execution. However, their reliance on high-degree polynomial approximations for non-linear functions (e.g., comparisons) incurs prohibitive computational costs and accuracy loss. In contrast, bit-wise schemes such as FHEW [28, 44] and TFHE [20, 22] are optimized for Boolean and circuit-based computation. By embedding logic gates into the bootstrapping process, these schemes natively support expressive, unbounded non-linear operations with far greater efficiency, positioning TFHE as the preferred choice for general-purpose workloads.

The core strength of TFHE lies in functional bootstrapping (FBS) [9], also known as programmable bootstrapping (PBS) [23], which evaluates arbitrary functions via look-up tables (LUTs) while refreshing noise to sustain long computation chains. For example, TFHE-rs [12] uses FBS to implement complex logic and relational operations. After that, TFHE incorporates multi-valued bootstrapping [13, 23] for computing multiple functions in one call and circuit bootstrapping (CBS) [21, 40, 50] for multi-input functions and higher-precision operations. These advances broaden TFHE's expressiveness and practical applicability.

Despite these innovations, TFHE remains fundamentally limited by the lack of batching and message-packing mechanisms, creating a major bottleneck for high-throughput and vectorized workloads. Some work [27, 34, 45] explores *batch-friendly* algorithms for TFHE schemes that amortize bootstrapping costs across multiple messages, but practical adop-

tion is constrained by large parameter sizes and rapid noise growth. A recent breakthrough, AmortBoot [33], introduced an amortization framework and implementation that improved both computational efficiency and key size, setting a new benchmark for memory efficiency, scalability, and real-world applicability in large-scale homomorphic computation.

1.1 Problems and Challenges

Our investigation reveals that existing batched (or amortized) designs continue to exhibit several critical limitations, which remain pressing challenges for the field. The following critique assumes familiarity with TFHE building blocks and batched bootstrapping. Unfamiliar readers may reference these foundations in Sec. 2 and Sec. 3 before proceeding.

❶ Excessive FFT invocations leads to poor performance. To achieve batched bootstrapping, the prevailing approach leverages the sparsity of the RLWE secret key, attaining a complexity of $O(hn\log n)$, i.e., $O(h\log n)$ per message, where n and h ($h < n$) denote the dimension and Hamming weight of the secret key, respectively. Although this asymptotic improvement represents a significant algorithmic advance, it still depends on FFT-intensive subroutines in the core construction. These subroutines not only limit the realized performance gains but also constitute a fundamental bottleneck to scalability, underscoring the need to rethink algorithmic design to advance the performance frontier.

❷ Sparse message packing remains unexplored. Existing techniques primarily assume a *full-packing* regime, where all n ciphertext slots are simultaneously populated. In practice, however, diverse applications demand varying degrees of sparsity in packing, which cannot be efficiently accommodated under this assumption. This rigidity stems from the inherent structure of RLWE decryption, where the computational complexity scales with the ciphertext dimension n rather than the actual number of active slots. The lack of sparsity-aware optimizations thus reveals a fundamental mismatch between algorithmic design and workload characteristics.

❸ Existing schemes compromise on functional generality. Existing schemes are limited to functional bootstrapping, which supports only single-input LUTs with 8-bit precision. In contrast, for more variants, such as multi-input LUT evaluation, the circuit bootstrapping needs to be introduced to support these functions, rather than merely adjusting the encoding. Extending batched bootstrapping to incorporate these features is not only technically challenging, requiring careful parameter-level optimizations, but also critical for realizing the full practical potential of TFHE-based systems.

1.2 Our contributions

To address these challenges, we propose BATCHBOOT, a novel batched bootstrapping framework that significantly enhances both efficiency and functionality. Specifically, we reduce the overhead of expensive FFTs by restructuring the batched bootstrapping subroutines. Additionally, we introduce an efficient sparse packing method derived from the MLWE decryption paradigm. Finally, we present the first batch circuit bootstrapping extension that broadens bootstrapping functionality. The effectiveness of BATCHBOOT is validated through empirical deployment on PSI and instruction set architectures. Our contributions are as follows:

Efficient batched bootstrapping through significant FFT reduction. To address issue ❶, BATCHBOOT employs two key strategies that substantially reduce the number of costly FFT operations. First, we develop a multi-bit CMux variant for batched bootstrapping, effectively reducing both the number of iterations and the FFTs, compared to the single-bit CMux method used in AmortBoot [33]. Second, we observe that automorphism operations can be post-processed after performing the FFT and redesign the polynomial multiplication pipeline for homomorphic evaluation. This enables FFT computation sharing between automorphism and CMux, thereby removing the additional FFT burden traditionally associated with automorphism. As a result, our approach reduces FFT operations, delivering a 2.4× speedup over the optimal amortization scheme (AmortBoot) and a 43.8× improvement over the non-batched TFHE-rs implementation.

Sparsity-aware message packing leveraging the MLWE decryption paradigm. To overcome issue ❷, BATCHBOOT incorporates an MLWE-based decryption paradigm to enable sparsity-aware optimization. In the sparse packing setting, where an RLWE ciphertext encodes only $\underline{n} < n$ messages, we apply RLWE-to-MLWE extraction to derive a lower-dimensional MLWE ciphertext. This transformation reduces the polynomial degree from n to $\underline{n}$ while preserving the secret key's sparsity, thereby maintaining the efficiency benefits of sparse structures. Consequently, the computational complexity drops from $O(hn\log n)$ to $O(h\underline{n}\log \underline{n})$, yielding performance improvements that scale linearly with the packing sparsity compared to the full-packing setting.

Extended functional expressiveness through batched circuit bootstrapping. We address issue ❸ by extending BATCHBOOT from functional bootstrapping to circuit bootstrapping to deal with multi-input LUT evaluation, resulting in BATCHCBOOT. Our design builds upon the circuit bootstrapping paradigm [40]. Specifically, the architecture proposed in [40] splits the circuit bootstrapping process into two subroutines: the ciphertext conversion from LWE to RGSW, and LUT evaluation by performing external operations within a tree-like structure. We note that in the first phase, the BATCHBOOT can be utilized to efficiently perform batch ciphertext conversions from RLWE to RGSWs, thereby enhancing performance. To the best of our knowledge, this is the first work that supports batch circuit bootstrapping. Benchmark results on 12-bit input/output LUTs demonstrate speedups ranging

from 4.9× to 134.3× over the non-batched method in [50].

Application I: Communication-friendly private set intersection (PSI) protocol. We present the first TFHE-based PSI protocol designed to minimize communication overhead. Leveraging BATCHBOOT, our design supports ciphertext transmission with an exceptionally small modulus (as low as 12 bits), whereas BFV-based schemes typically require larger FHE parameters, such as 100–200-bit moduli. This results in at least a two-order-of-magnitude reduction in communication cost relative to existing FHE-based PSI solutions. Moreover, in certain unbalanced settings, our protocol surpasses the state-of-the-art solution [43]; for example, it achieves a 4.1× speedup on datasets where the client holds 2,048 elements and the server has 2^{28} elements.

Application II: High-performance 8-bit instruction set implementation. We further develop an optimized 8-bit FHE instruction set encompassing fundamental operations such as AND, ADD, MUL, and MAX. Leveraging the BATCHCBOOT algorithm, we enable the evaluation of multiple distinct instruction sets within a single bootstrapping operation, incorporating customized designs tailored to the unique characteristics of each set. Compared to prior solutions limited to a single instruction set [49,50], our approach delivers 3.3×-6.5× amortized performance improvements.

2 Preliminary

We first present some definitions of symbols in Appendix B.1 and then introduce some necessary background for FHE schemes.

2.1 Ciphertexts Types

GLWE ciphertext [4]. Given the secret key $\mathbf{S} = (\mathfrak{s}_0, ..., \mathfrak{s}_{k-1}) \in R_N^k$, the message $m \in R_{N,p}$ can be encrypted into GLWE ciphertext as $(\mathfrak{a}_0, \cdots, \mathfrak{a}_{k-1}, \mathfrak{b}) \in R_{N,Q}^k$, where $\mathfrak{b} = \sum_{i=0}^{k-1} \mathfrak{a}_i \cdot \mathfrak{s}_i + \triangle \mathfrak{m} + \mathfrak{e} \mod Q$, where $\triangle = \left\lfloor \frac{Q}{p} \right\rfloor$, $(\mathfrak{a}_0, \cdots, \mathfrak{a}_{k-1})$ are uniformly sampled from $R_{N,Q}$, and the error $\mathfrak{e}$ is sampled from the Gaussian Distribution χ. Here p, Q are the plaintext and ciphertext modulus, k and N are the dimension and degree of the GLWE, respectively, and we define $\widetilde{\mathfrak{m}} = \triangle \mathfrak{m} + \mathfrak{e}$.

To avoid confusion, a GLWE ciphertext where $N = 1$ is termed an *LWE* ciphertext [47], denoted $(\mathbf{a}, b) \in \mathrm{LWE}_{\mathbf{s},p/Q}^k(m)$. When $k = 1$ and $N > 1$, it is called the ring LWE (*RLWE*) ciphertext [42], that is $(\mathfrak{a}, \mathfrak{b}) \in \mathrm{RLWE}_{\mathfrak{s},p/Q}^N(\mathfrak{m})$. A GLWE with $k > 1$ and $N > 1$ is referred to as $\mathrm{MLWE}_{\mathbf{S},p/Q}^{k,N}(\mathfrak{m})$. These ciphertexts support homomorphic addition and scalar multiplication. In addition, we may omit some parameters for notation simplicity.

Gadget RLWE ciphertext [44]. Given the gadget vector $\mathbf{g} = (B_0, ..., B_{d-1})$, the gadget RLWE ciphertext is defined as $\mathrm{RLWE}'(\mathfrak{m}) = (\mathrm{RLWE}(B_0 \cdot \mathfrak{m}), \cdots, \mathrm{RLWE}(B_{d-1} \cdot \mathfrak{m}))$.

RGSW ciphertext [28]. RGSW encryption of the message $\mathfrak{m}$ is defined as $\mathrm{RGSW}(\mathfrak{m}) = (\mathrm{RLWE}'(-\mathfrak{s} \cdot \mathfrak{m}), \mathrm{RLWE}'(\mathfrak{m}))$, where $\mathsf{half}(\mathrm{RGSW}(\mathfrak{m}))$ denotes ciphertext $\mathrm{RLWE}'(\mathfrak{m})$.

2.2 Building Blocks

Gadget Decomposition (Decom). For a modulus Q, and a base B, the gadget vector is defined as $\mathbf{g} = (B_0, ..., B_{d-1}) = \left(\left\lceil \frac{Q}{B^d} \right\rfloor, \left\lceil \frac{Q}{B^d} \right\rfloor \cdot B, ..., \left\lceil \frac{Q}{B^d} \right\rfloor \cdot B^{d-1} \right)$. Then, given a polynomial $\mathfrak{a} \in R_Q$, the approximate decomposition can be used to output $(\mathfrak{a}_0, ..., \mathfrak{a}_{d-1})$ such that $\mathfrak{a} = \sum_{i=0}^{d-1} \mathfrak{a}_i \cdot \left\lceil \frac{Q}{B^{d-i}} \right\rfloor$. Note that the approximate gadget decomposition introduces an approximate error $\mathfrak{e}$, i.e., $\langle \mathbf{g}^{-1}(\mathfrak{a}), \mathbf{g} \rangle \equiv \mathfrak{a} + \mathfrak{e}$, where $||\mathfrak{e}|| \leq \frac{1}{2} \left\lceil \frac{Q}{B^d} \right\rfloor$.

Gadget Product (GP). Given a polynomial $\mathfrak{t} \in R_Q$, let $(\mathfrak{t}_0, ..., \mathfrak{t}_{d-1}) = \mathsf{Decom}(\mathfrak{t})$ is the result of the gadget decomposition with respect to the gadget vector $\mathbf{g} = (B_0, ..., B_{d-1})$. The Gadget product $\odot$ is defined by:

$$\mathfrak{t} \odot \mathrm{RLWE}'(\mathfrak{m}) := \sum_{i=0}^{d-1} \mathfrak{t}_i \cdot \mathrm{RLWE}(B_i \cdot \mathfrak{m}) = \mathrm{RLWE}(\mathfrak{t} \cdot \mathfrak{m}).$$

External Product (EP). Given ciphertexts $\mathrm{RLWE}(\mathfrak{m}_1) = (\mathfrak{a}, \mathfrak{b})$ and $\mathrm{RGSW}(\mathfrak{m}_2)$, the external product is defined by:

$$\begin{aligned} &\mathrm{RLWE}(\mathfrak{m}_1) \boxdot \mathrm{RGSW}(\mathfrak{m}_2) \\ &= \mathfrak{a} \odot \mathrm{RLWE}'(-\mathfrak{s} \cdot \mathfrak{m}_2) + \mathfrak{b} \odot \mathrm{RLWE}'(\mathfrak{m}_2) \\ &= \mathrm{RLWE}(\mathfrak{m}_1 \cdot \mathfrak{m}_2 + \mathfrak{e}_1 \cdot \mathfrak{m}_2). \end{aligned}$$

The external product outputs encryption of the product of $\mathfrak{m}_1$ and $\mathfrak{m}_2$. Note that the gadget product and external products involve polynomial operations, which can be accelerated with the Fast Fourier Transform (FFT).

CMux Gate (CMux). The CMux gate takes as inputs two RLWE ciphertexts c_0, c_1 and a RGSW ciphertext $C \in \mathrm{RGSW}(m)$, and CMux operation is defined as

$$\mathsf{CMux}(c_0, c_1, C) = c_0 + (c_1 - c_0) \boxdot C = \begin{cases} c_0, & \text{if } m = 0; \\ c_1, & \text{else } m = 1, \end{cases}$$

where m satisfies $m \in \{0, 1\}$. We present the noise growth for V_{GP}, V_{EP}, and $\mathsf{V}_{\mathsf{CMux}}$ in Appendix B.2.

Sample Extraction (SE). Given the ciphertext $c = (\mathfrak{a}, \mathfrak{b}) \in \mathrm{RLWE}_{\mathfrak{s},Q}^N(\mathfrak{m})$, the sample extraction operation can extract N LWE ciphertexts associated with the coefficient terms. For example, for the first term, we can define $\mathsf{SampleExtract}(\mathsf{ct}) = (a_0, -a_{N-1}, ..., -a_1, b_0) \in \mathrm{LWE}_{\mathsf{Coef}(\mathfrak{s}),Q}^N(\mathsf{Coef}(m)[0])$.

RLWE-to-MLWE Extraction (RtM.Ext). Given an RLWE ciphertext $(\mathfrak{a}, \mathfrak{b}) \in \mathrm{RLWE}_{\mathfrak{s},p/Q}^N(\mathfrak{m})$, where $\mathfrak{m} = \sum_{i=0}^{n-1} m_i \cdot X^{i \cdot \frac{N}{n}}$. Let $\mathfrak{m}' = \sum_{i=0}^{n-1} m_i \cdot X^i$, we can extract the MLWE ciphertext as $(\mathfrak{a}'_0, ..., \mathfrak{a}'_{k-1}, \mathfrak{b}') \in \mathrm{MLWE}_{\mathbf{S},p/Q}^{k,n}(\mathfrak{m}')$, where $k = \frac{N}{n}$, $(\mathfrak{a}'_0, ..., \mathfrak{a}'_{k-1}) = \mathsf{RingSwitch}_{k,N}(\mathfrak{a})$, $\mathbf{S} = (\mathfrak{s}'_0, ..., \mathfrak{s}'_{k-1}) =$

$\mathsf{RingSwitch}(\mathfrak{s})$, and $\mathfrak{b}' = \mathsf{RingSwitch}(\mathfrak{b})_{k,N}[0]$. We show the details of RingSwitch in Appendix B.3.

Modulus Switching (MS). Given an RLWE ciphertext $c \in \mathrm{RLWE}^N_{\mathfrak{s},p/Q}(\mathfrak{m})$, the modulus switching $\mathsf{ModSwitch}_{Q\to q}(c)$ can change the ciphertext modulus from Q to q as $\mathrm{RLWE}^N_{\mathfrak{s},p/q}(\mathfrak{m})$ as shown in Appendix B.3.

Key Switching (KS). It can convert a ciphertext encrypted under the secret key $\mathfrak{s}_1$ into a ciphertext encrypted under the new secret key $\mathfrak{s}_2$ with the key switching key. In addition, the RLWE key switching variant scheme switching SchemSwitch [27, 40, 51] can embed the secret key $\mathfrak{s}$ into the message. We show the RLWE key switching (RLWE.KeySwitch) and scheme switching steps [22] in Appendix B.3.

Homomorphic Automorphism (HomoAuto). There are N automorphisms as $\tau_j : \mathfrak{m}(X) \to \mathfrak{m}(X^j)$ for $j \in \mathbb{Z}^*_{2N}$ for $\mathbb{Z}_Q[X]/(X^N+1)$. Given ciphertext $c = (\mathfrak{a},\mathfrak{b}) \in \mathrm{RLWE}^N_{\mathfrak{s},Q}(\mathfrak{m})$, and the RLWE key switching key $\mathsf{Atk}_{\tau_j} \in \mathrm{RLWE}'_{\mathfrak{s}}(\mathfrak{s}(X^j))$, the Homomorphic automorphism $\mathsf{HomAuto}_{\tau_j}(\mathsf{ct},\mathsf{Atk}_{\tau_j})$ is divided into two steps: first performs $\mathsf{Auto} : \tau_j(c) = (\mathfrak{a}(X^j),\mathfrak{b}(X^j)) \in \mathrm{RLWE}^N_{\tau_j(\mathfrak{s}),Q}(\tau_j(\mathfrak{m}))$; and applies the RLWE key switching from the secret key $\mathfrak{s}(X^j)$ to $\mathfrak{s}(X)$.

Homomorphic Trace (HomoTrace). For the polynomial ring R_N and its residue ring $R_{N,Q}$, the tower of finite fields is expressed as $E = K_N \geq K_{N/2} \geq \cdots \geq K_1 = K$, where K_n denotes the $(2n)$-th cyclotomic field for a power-of-two n. Then, the trace Tr_{K_N/K_n} can be defined as $\mathrm{Tr}_{K_N/K_n}(\mathfrak{m}(X)) = \sum_{i=0}^{n-1} \mathfrak{m}_{i\cdot N/n} \cdot X^{i\cdot N/n}$, which can be evaluated through $\log_2 N/n$ automorphism operations. The homomorphic trace algorithm Tr_{K_N/K_n} is described in Appendix B.4.

Repacking (Repack). Given n RLWE ciphertexts $\mathrm{RLWE}^N_{\mathfrak{s},Q}(\mathfrak{m}_j) = (\mathfrak{a}_j,\mathfrak{b}_j)$ for $i \in [0,n-1]$, the repacking algorithm can pack the first items of ciphertexts $(\mathfrak{m}_{0,0},\ldots,\mathfrak{m}_{N-1,0})$ into a new RLWE ciphertext $\widetilde{\mathfrak{m}} = \sum_{i=0}^{n-1} \mathfrak{m}_{i\cdot N/n} \cdot X^{i\cdot N/n}$, where $n|N$ $(n < N)$ are referred to as sparse packing (SRepack). This paper uses the methods [14, 38] that need to perform the HomoAuto and Homotrace operations. Detailed algorithms and noise growth are presented in Appendix B.4.

2.3 Functional Bootstrapping

Functional bootstrapping (FBS) [9] can evaluate the arbitrary function in bootstrapping. Specifically, given a ciphertext $c \in \mathrm{LWE}^n_{\mathbf{s},p/q}(m)$, FBS can output a new ciphertext that encrypts $f(m)$ as shown in Alg. 1. For the arbitrary function $f : \mathbb{Z}_p \to \mathbb{Z}_{p_{\mathsf{out}}}$, we can embed it into a polynomial testP as shown in line 1. Note that we set $p_{\mathsf{out}} = p$ by default, unless otherwise specified. Then, the modulus switching is used to convert the ciphertext from q to $2N$. After performing the lines 3-6 through n CMux operations (referred to as blind rotation in [20]), we obtain the RLWE ciphertext $\mathrm{RLWE}(\mathsf{testP}\cdot X^{\widetilde{m}})$, where $\widetilde{m} = \triangle \cdot m + e$ and $\triangle = \frac{2N}{p}$. Then, the desired result is preserved in the constant term of the polynomial. Finally,

Algorithm 1 Functional Bootstrapping (FBS).

Input:
LWE ciphertext $c = (\mathbf{a},b) \in \mathrm{LWE}^n_{\mathbf{s},p/q}(m)$.
Bootstrapping key $\mathsf{Bsk} : \{\mathrm{RGSW}_{\mathfrak{s},Q}(s_i)\}$ for $i \in [0,n-1]$.
Output:
LWE ciphertext of $f(m)$.
1: Set $\mathsf{testP} = -\sum_{i=0}^{N-1} \left\lfloor \frac{Q}{p_{\mathsf{out}}} \right\rceil \cdot f\left(\left\lfloor \frac{ip}{2N} \right\rceil\right) \cdot X^{N-i-1}$
2: $(\mathbf{a},b) = \mathsf{ModSwitch}_{q\to 2N}(c)$
3: $\mathsf{acc} = \mathsf{testP}(X)\cdot X^b$,
4: **for** $i = 0$ to $n-1$ **do**
5: $\quad \mathsf{acc} = \mathsf{CMux}(\mathsf{acc}, X^{-a_i}\cdot\mathsf{acc}, \mathsf{Bsk}_i)$,
6: **end for**
7: **return** $\mathsf{SampleExtraction}(\mathsf{acc})$

we can use the sample extraction to obtain the LWE ciphertext of $f(m)$. Essentially, the bootstrapping performs LWE decryption on the exponent of X.

3 Revisiting Previous Batched Bootstrapping

Guimarães et al. [33] present AmortBoot that aims to bootstrap an RLWE ciphertext instead of the LWE ciphertext, where the RLWE encrypts a polynomial $\mathfrak{m}(X) = \sum_{i=0}^{n-1} m_i \cdot X^i$. In detail, given the ciphertext $\mathrm{RLWE}^n_{\mathfrak{s},p/q}(\mathfrak{m})$, this procedure can generate n ciphertexts, that is $\mathrm{RLWE}(\mathsf{testP}_i \cdot X^{\widetilde{m}_i})$ for $i \in [0,n-1]$. This multiplication $\mathsf{testP}_i \cdot X^{\widetilde{m}_i}$ subsequently results in the encryption of $f(m_0),\ldots,f(m_{n-1})$.

To obtain n ciphertext $\mathrm{RLWE}(\mathsf{testP}_i \cdot X^{\mathfrak{m}_i})$ simultaneously, one needs to compute $\mathfrak{m}(X) = \mathfrak{b}(X) - \mathfrak{a}(X)\cdot\mathfrak{s}(X)$ on the exponent of X in a single bootstrapping. AmortBoot exploits the sparsity of the RLWE secret key, which accelerates the polynomial multiplication process. When the secret key is chosen from a sparse binary key distribution with a Hamming weight of h $(h < n)$[1], RLWE decryption formula can be expressed as

$$\mathfrak{b} - \mathfrak{a}\cdot\mathfrak{s} = \mathfrak{b} - \mathfrak{a}\sum_{j\in\mathsf{idx}(\mathfrak{s})} X^j = \mathfrak{b} - \sum_{j\in\mathsf{idx}(\mathfrak{s})} \left(\sum_{i=0}^{n-1} \mathfrak{a}(X)\right) \widetilde{\odot} X^j, \tag{1}$$

where $j \in \mathsf{idx}(\mathfrak{s})$ represents the set containing the degrees of nonzero coefficients of $\mathfrak{s}$, n is the degree of RLWE, and $\widetilde{\odot}$ represents a monomial-by-polynomial multiplication. For instance, if $\mathfrak{s}(X) = 1 + 1\cdot X^5 + 1\cdot X^7$, we have $j = (0,5,7)$. For the cleartext, the Eq. 1 can be computed in $O(hn)$ operations. However, since the index j is related to the secret key, it is encrypted into the bootstrapping key. After that, AmortBoot [33] rewrites the Eq. 1 as

$$\left(\left(\left(\mathfrak{b}\widetilde{\odot}X^{-j_0} - \mathfrak{a}\right)\widetilde{\odot}X^{j_0-j_1} - \mathfrak{a}\right)\widetilde{\odot}X^{j_1-j_2}\cdots - \mathfrak{a}\right)\widetilde{\odot}X^{j_h}, \tag{2}$$

which can be efficiently computed in the exponent of X by using external products. Note that since $\mathfrak{a}$ is a public poly-

[1] As many notable FHE schemes [1, 19, 33, 37] have done.

nomial, subtracting it in the exponent can be achieved by multiplying all X^{-a_i}. Moreover, AmortBoot [33] constructs a sub-procedure MPmul to evaluate $\widetilde{\odot}$. This process can be generalized to compute the multiplication $\mathfrak{u} \cdot X^v \mod X^n+1$ in the exponent, and v can be instantiated for all possible values of $j_i - j_{i+1}$. After that, Eq. 2 can be reduced to

$$\mathsf{MPmul}\left(\ldots \mathsf{MPmul}\left(\mathsf{MPmul}\left(\mathfrak{b}, X^{-\tilde{j}_0}\right) - \mathfrak{a}, X^{\tilde{j}_0-\tilde{j}_1}\right) \cdots - \mathfrak{a}, X^{\tilde{j}_h}\right). \tag{3}$$

Note that, during the homomorphic computation process, all coefficients of $\mathfrak{u}$ are encrypted into $\mathrm{RLWE}(X^{u_i})$ as inputs, while $v \in [-n, n]$ is encrypted and encapsulated within the bootstrapping key, denoted as Bsk. Essentially, multiplying by X^v is to rotate the coefficients of $\mathfrak{u}$ by v positions. For this positive v rotations[2], the multiplication $\mathfrak{u} \cdot X^v \mod X^n+1$ has the following map

$$(u_0, u_1, \ldots, u_{n-1}) \mapsto \underbrace{(-u_{n-v}, -u_{n-v+1}, \ldots, -u_{n-1}}_{\text{negate } v \text{ positions}}, u_0, u_1, \ldots, u_{n-v-1}).$$

Because the RLWE is instantiated in the cyclotomic polynomial $\mathbb{Z}[X]/\langle X^n+1\rangle$, we also need to multiply v of them by -1 in some positions.

In this process, one can process each coefficient in sequence in this map. More precisely, MPmul further decomposes v as $v = \sum_{i=0}^{\ell-1} 2^i \cdot v_i$, and sequentially multiplies $X^{2^0 v_0}, \ldots, X^{2^{\ell-1} v_{\ell-1}}$ for each coefficient (position). It is clear that multiplying by $X^{2^i v_i}$ is equal to evaluating the following equation:

$$f(u_0, \ldots, u_{n-1}) \mapsto \begin{cases} (u_0, u_1, \ldots, u_{n-1}) & \text{if } v_i = 0, \\ (-u_{n-2^i}, \ldots, -u_{n-1}, u_0, \ldots, u_{n-2^i-1}) & \text{if } v_i = 1. \end{cases}$$

Then, according to the input $v_i \in \{0,1\}$, one can use the CMux to pick out the corresponding value. Hence, for the output coefficient range $2^i \le j \le n-1$, given the $\mathsf{Bsk}_i = \mathrm{RGSW}(v_i)$, the homomorphic computation for this step is shown as follows

$$\hat{c}_j = (c_{j-2^i} - c_j) \boxdot \mathsf{Bsk}_i + c_j = \begin{cases} c_{j-2^i}, & \text{if } v_i = 0\,; \\ c_j, & \text{else } v_i = 1. \end{cases}$$

For the first 2^i coefficients, we first apply the $\mathsf{HomoAuto}_{\tau_{-1}}$: $X \mapsto X^{-1}$ to get the encryption of $X^{-u_{n-2^i+j}}$, and perform the CMux as

$$\hat{c}_j = (\mathsf{HomoAuto}_{-1}(c_{n-2^i+j}) - c_j) \boxdot \mathsf{Bsk}_i + c_j. \tag{4}$$

This process involves $n\ell$ CMux and $\ell(2^\ell - 1)$ automorphism operations as shown in [33], where $2^l < n$ in practical parameters.

[2]Followed by a similar method for handling negative values.

4 Design Overview of BatchBoot

In this section, we propose some novel batched bootstrapping algorithms that support functional evaluations for multiple messages. Afterwards, we conduct a comprehensive evaluation of the correctness and noise growth of these algorithms.

4.1 Efficient MPmul Procedure

This section proposes a more efficient MPmul (EMPmul) procedure. At a fine-grained level, external product $\boxdot$ and HomoAuto involve gadget decomposition and FFT (used in accelerating polynomial multiplication). For example, the evaluation of $\hat{c}_j = (c_{j-2^i} - c_j) \boxdot \mathsf{Bsk}_i + c_j$ can be broken down into five steps. 1) Apply the gadget decomposition on $c_{j-2^i} - c_j$ with the coefficient representation; 2) Perform FFTs on the result of the decomposition; 3) Evaluate Hadamard multiplication with Bsk_i, where Bsk_i is preprocessed into FFT form; 4) Perform the inverse FFT (iFFT) to recover the coefficient representation; 5) Compute homomorphic addition with c_j. In addition, in Eq. 4, i.e., $\hat{c}_j = (\mathsf{HomAuto}_{-1}(c_{n-2^i+j}) - c_j) \boxdot C_i + c_j$, $\mathsf{HomAuto}_{-1}$ also needs to perform additional FFTs (see Sec. 2.2) , and then this implementation can be summarized as

$$\mathsf{HomoAuto}_{-1}(c) \boxdot \mapsto \underbrace{\mathsf{Auto} + \mathsf{Decom} + \mathsf{FFT} + \mathsf{Hadm} + \mathsf{iFFT}}_{\mathsf{HomoAuto}} + \underbrace{\mathsf{Decom} + \mathsf{FFT} + \mathsf{Hadm} + \mathsf{iFFT}}_{\text{External product}}, \tag{5}$$

Since FFTs exhibit a computational complexity of $O(n \log n)$, as opposed to the $O(n)$ complexity of Hadamard multiplication, FFT operations represent the main computational bottleneck in Eq. 5. Overall, the method of Sec. 3 requires approximately $2(2d+2) \cdot n\ell$ FFTs, where d is the length of the gadget decomposition.

Our method. We design the EMPmul to evaluate two-input bits for v instead of a single-input bit, which can reduce the number of time-consuming FFTs from $2(2d+2) \cdot n\ell$ to $(d+1) \cdot n\ell$. For simplicity, we focus on the case $v > 0$ ($v < 0$ is shown in Appendix C.1), and then the sequential multiplication $X^{2^i v_i}$ and $X^{2^{i+1} v_{i+1}}$ has the map as:

$$f(u_0, \ldots, u_{n-1}) \mapsto \begin{cases} (u_0, \ldots, u_{n-1}) & \text{if } v_{i:i+1} = (0,0) \\ (-u_{n-2^i}, \ldots, -u_{n-1}, u_0, \ldots, u_{n-2^i-1}) & \text{if } v_{i:i+1} = (1,0) \\ (-u_{n-2^{i+1}}, \ldots, -u_{n-1}, u_0, \ldots, u_{n-2^{i+1}-1}) & \text{if } v_{i:i+1} = (0,1) \\ (-u_{n-2^i-2^{i+1}}, \ldots, -u_{n-1}, u_0, \ldots, u_{n-2^i-2^{i+1}-1}) & \text{if } v_{i:i+1} = (1,1) \end{cases} \tag{6}$$

Let us first consider these non-negative positions, i.e., $2^i + 2^{i+1} \le j \le n-1$. The multi-bit CMux operation can be employed directly to evaluate the function as

Table 1: Setup of bootstrapping keys for multi-bit CMux.

Bsk	$\mathsf{Bsk}_{i,0}$	$\mathsf{Bsk}_{i,1}$	$\mathsf{Bsk}_{i,2}$	$\mathsf{Bsk}_{i,3}$	Value
Bsk^+	RGSW(0)	RGSW(0)	RGSW(0)	RGSW(1)	$v_i=0; v_{i+1}=0$
	RGSW(1)	RGSW(0)	RGSW(0)	RGSW(0)	$v_i=1; v_{i+1}=0$
	RGSW(0)	RGSW(1)	RGSW(0)	RGSW(0)	$v_i=0; v_{i+1}=1$
	RGSW(0)	RGSW(0)	RGSW(1)	RGSW(0)	$v_i=1; v_{i+1}=1$
Bsk^-	$\mathrm{RGSW}^{\tau_{-1}}(0)$	$\mathrm{RGSW}^{\tau_{-1}}(0)$	$\mathrm{RGSW}^{\tau_{-1}}(0)$	RGSW(1)	$v_i=0; v_{i+1}=0$
	$\mathrm{RGSW}^{\tau_{-1}}(1)$	$\mathrm{RGSW}^{\tau_{-1}}(0)$	$\mathrm{RGSW}^{\tau_{-1}}(0)$	RGSW(0)	$v_i=1; v_{i+1}=0$
	$\mathrm{RGSW}^{\tau_{-1}}(0)$	$\mathrm{RGSW}^{\tau_{-1}}(1)$	$\mathrm{RGSW}^{\tau_{-1}}(0)$	RGSW(0)	$v_i=0; v_{i+1}=1$
	$\mathrm{RGSW}^{\tau_{-1}}(0)$	$\mathrm{RGSW}^{\tau_{-1}}(0)$	$\mathrm{RGSW}^{\tau_{-1}}(1)$	RGSW(0)	$v_i=1; v_{i+1}=1$

$$\begin{aligned}\hat{c}_j &= c_{j-2^i} \boxdot \mathsf{Bsk}^+_{i,0} + c_{j-2^{i+1}} \boxdot \mathsf{Bsk}^+_{i,1} \\ &\quad + c_{j-2^i-2^{i+1}} \boxdot \mathsf{Bsk}^+_{i,2} + c_j \boxdot \mathsf{Bsk}^+_{i,3} \\ &= \begin{cases} c_j, & \text{if } v_i=0; v_{i+1}=0\,; \\ c_{j-2^i}, & \text{if } v_i=1; v_{i+1}=0\,; \\ c_{j-2^{i+1}}, & \text{if } v_i=1; v_{i+1}=0; \\ c_{j-2^i-2^{i+1}}, & \text{if } v_i=1; v_{i+1}=1 \end{cases}\end{aligned} \tag{7}$$

to obtain the desired result, where bootstrapping keys are set up in Tab. 1. However, the situation is more complex for the first 2^i+2^{i+1} indexes, since the additional automorphism $X \mapsto X^{-1}$ needs to be performed. To eliminate the time-consuming FFT induced by automorphism, we introduce the following method.

Parametrized External Product. We first review the parametrized external product proposed by Bernard et al. [7]. Specifically, the variant RGSW ciphertext associated with the automorphism τ_{-1} is defined as

$$\mathrm{RGSW}^{\tau_{-1}}_{\mathfrak{s}}(\mathfrak{m}_1) = (\mathrm{RLWE}'_{\mathfrak{s}}(-\tau_{-1}(\mathfrak{s})\cdot\mathfrak{m}_1), \mathrm{RLWE}'_{\mathfrak{s}}(\mathfrak{m}_1)).$$

Then, given ciphertext $c=(\mathfrak{a},\mathfrak{b}) \in \mathrm{RLWE}_{\mathfrak{s}}(\mathfrak{m}_2)$, the variant external product associated with τ_{-1}, denote as $\boxdot_{\tau_{-1}}$, is evaluated as:

$$\begin{aligned}&\mathrm{RLWE}(\mathfrak{m}_2) \boxdot_{\tau_{-1}} \mathrm{RGSW}^{\tau_{-1}}_{\mathfrak{s}}(\mathfrak{m}_1) \\ &= \tau_{-1}(\mathfrak{a}) \odot \mathrm{RLWE}'(-\tau_{-1}(\mathfrak{s})\cdot\mathfrak{m}_2) + \tau_{-1}(\mathfrak{b}) \odot \mathrm{RLWE}'(\mathfrak{m}_2) \\ &= \mathrm{RLWE}(-\tau_{-1}(\mathfrak{a})\cdot\tau_{-1}(\mathfrak{s})\cdot\mathfrak{m}_2 + \tau_{-1}(\mathfrak{b})\cdot\mathfrak{m}_2) \\ &= \mathrm{RLWE}(\tau_{-1}(\mathfrak{m}_1)\cdot\mathfrak{m}_2 + \tau_{-1}(\mathfrak{e}_1)\cdot\mathfrak{m}_2).\end{aligned}$$

In fact, this step integrates the automorphism τ_{-1} and multiplication into a single external product operation, thereby avoiding the additional RLWE key switching (including FFTs) induced by the automorphism. Nevertheless, even so, the problem remains fundamentally unsolved. This is because both operations $c \boxdot$ and $\mathsf{HomoAuto}_{-1}(c) \boxdot$ may be applied simultaneously to the input ciphertext c, such as $c_{n-2^i}, c_{n-2^{i+1}}, c_{n-2^i-2^{i+1}}$ in Eq. 6. We observe that the Auto_{-1} can penetrate both gadget decomposition and FFT operations, enabling us to deal with Auto_{-1} in the FFT domain.

Then, we can perform Eq. 8 instead of Eq. 5, where the external product and automorphism share the instance of the Decom and FFT operations.

$$\mathsf{Auto}_{-1}(c) \boxdot_{\tau_{-1}} \mapsto \mathsf{Decom} + \mathsf{FFT} + \begin{cases} \mathsf{Auto} + \mathsf{Hadm}(\mathsf{Bsk}^-) \\ \mathsf{Hadm}(\mathsf{Bsk}^+) \end{cases} + \mathsf{iFFT} \tag{8}$$

Note that the penetration technique (termed as Hoisting [8]) has been applied in the BFV and CKKS schemes, but we integrate this technique into TFHE for the first time to minimize FFT computations. Moreover, for the remaining intervals of the map of Eq. 6, we can further break it down into three intervals $0 \sim 2^i-1, 2^i \sim 2^{i+1}-1, 2^{i+1} \sim 2^i+2^{i+1}-1$. For instance, for the first 2^i indexes, by integrating the multi-bit CMux and the parametrized external product technique, we can perform

$$\begin{aligned}\hat{c}_j &= c_{n-2^i+j} \boxdot_{\tau_{-1}} \mathsf{Bsk}^-_{i,0} + c_{n-2^{i+1}+j} \boxdot_{\tau_{-1}} \mathsf{Bsk}^-_{i,1} \\ &\quad + c_{n-2^i-2^{i+1}+j} \boxdot_{\tau_{-1}} \mathsf{Bsk}^-_{i,2} + c_j \boxdot \mathsf{Bsk}^+_{i,3} \\ &= \begin{cases} c_j, & \text{if } v_i=0; v_{i+1}=0\,; \\ -c_{n-2^i+j}, & \text{if } v_i=1; v_{i+1}=0\,; \\ -c_{n-2^{i+1}+j}, & \text{if } v_i=1; v_{i+1}=0; \\ -c_{n-2^i-2^{i+1}+j}, & \text{if } v_i=1; v_{i+1}=1. \end{cases}\end{aligned} \tag{9}$$

to deal with this case. The remaining indices and complete procedural steps are outlined in Alg. 2, where the symbol $\boxtimes$ represents the external product computed within the FFT domain. Note that our approach executes the (i)FFT only once for all input and output ciphertexts $c_0,\ldots,c_{n-1}$ as shown in lines 3 and 18, thus minimizing the FFTs in the iterative layers and eliminating the extra FFTs induced by automorphisms. Finally, we show the noise growth in the lemma 4.1.

Lemma 4.1. Input a list $c_i \in \mathrm{RLWE}(X^{u_i})$ for $0 \le i \le n$ with error variance $\mathsf{Var}(c_{\mathsf{in}})$, bootstrapping $\mathsf{Bsk}^{\pm}_{i,j}$ for $0 \le i \le \ell$, Alg. 2 outputs a list for the encryption for $\mathfrak{t}(X) = \mathfrak{u}(X)\cdot X^v \bmod X^n+1$, and its variance satisfies

$$\mathsf{V}_{\mathsf{EMP}} < \mathsf{Var}(c_{\mathsf{in}}) + 2\ell \cdot \mathsf{V}_{\mathsf{EP}},$$

where $\mathsf{V}_{\mathsf{CMux}}$ is associated with Bsk. The complete proof is present in Appendix C.2.

Table 2: Computation cost for different input bits of EMPmul.

Input-bits	$\delta=1$	$\delta=2$	$\delta=3$	$\delta=4$
FFT	$2(d+1)n\ell$	$(d+1)n\ell$	$2(d+1)n\ell/3$	$(d+1)n\ell/2$
Hadamard-Mult	$4dn\ell$	$8dn\ell$	$12dn\ell$	$16dn\ell$

Remark 1. Our Alg. 2 (EMPmul) can be extended to larger δ-input bits for v inputs by using the δ-bit CMux operation. Note that larger inputs reduce FFT calls but increase Hadamard

Algorithm 2 Efficient Monomial-polynomial multiplication (EMPmul).

Input:

A list $c_i \in \mathrm{RLWE}(X^{u_i})$ for $0 \le i \le n$ representing a polynomial $\mathfrak{u}(X)$ encrypted in the exponent.

The bootstrapping key $\mathsf{Bsk}_{i,j}$ for $0 \le i < \ell/2-1$ and $0 \le j \le 3$ as shown in Tab 1, which is associated with an encrypted value v.

Output:

A list $\hat{c}_i \in \mathrm{RLWE}(X^{w_i})$ for polynomial $\mathfrak{t}(X) = \mathfrak{u}(X) \cdot X^v \bmod X^n+1$, where $0 \le i \le n$.

1: for $i = 0$ to $\ell/2-1$ do
2: for $j = 0$ to $n-1$ do
3: $\hat{c}_j = \mathsf{FFT} \circ \mathsf{Decom}(c_j)$
4: end for
5: for $j = 0$ to $2^{2i}-1$ do
$\hat{c}_j = c_{n-2^{2i}+j} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,0}$
6: $+ c_{n-2^{2i+1}+j} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,1}$
$+ c_{n-2^{2i}-2^{2i+1}+j} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,2} + c_j \boxtimes \mathsf{Bsk}^{+}_{2i,3}$
7: end for
8: for $j = 2^{2i}$ to $2^{2i+1}-1$ do
$\hat{c}_j = c_{j-2^{2i}} \boxtimes \mathsf{Bsk}^{+}_{2i,0}$
9: $+ c_{n-2^{2i+1}+j} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,1}$
$+ c_{n-2^{2i}-2^{2i+1}+j} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,2} + c_j \boxtimes \mathsf{Bsk}^{+}_{2i,3}$
10: end for
11: for $j = 2^{2i+1}$ to $2^{2i}+2^{2i+1}-1$ do
$\hat{c}_j = c_{j-2^{2i}} \boxtimes \mathsf{Bsk}^{+}_{2i,0}$
12: $+ c_{j-2^{2i+1}} \boxtimes \mathsf{Bsk}^{+}_{2i,1}$
$+ c_{n-2^{2i}-2^{2i+1}+j} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,2} + c_j \boxtimes \mathsf{Bsk}^{+}_{2i,3}$
13: end for
14: for $j = 2^{2i}+2^{2i+1}$ to $n-1$ do
$\hat{c}_j = c_{j-2^{2i}} \boxtimes \mathsf{Bsk}^{+}_{2i,0} + c_{j-2^{2i+1}} \boxtimes \mathsf{Bsk}^{+}_{2i,1}$
15: $+ c_{j-2^{2i}-2^{2i+1}} \boxtimes \mathsf{Bsk}^{+}_{2i,2} + c_j \boxtimes \mathsf{Bsk}^{+}_{2i,3}$
16: end for
17: for $j = 0$ to $n-1$ do
18: $\hat{c}_j = \mathsf{iFFT}(c_j)$
19: end for
20: $(c_0, ..., c_{n-1}) \leftarrow (\hat{c}_0, ..., \hat{c}_{n-1})$
21: end for
22: **return** $(c_0, ..., c_{n-1})$

multiplications and key sizes, and the computation cost analysis is provided in Tab. 2. In consideration of the trade-off between performance and key sizes, we use $\delta = 2$ as the default parameter in all experimental evaluations (corresponding to Alg. 2).

4.2 Sparse Batched Bootstrapping

Building on the EMPmul procedure, it is easy to construct the BATCHBOOT by the Eq. 3 as shown in Alg. 3, which bootstraps n messages under the full packing case. However, scenarios may arise where bootstrapping a subset of $\underline{n}$ messages ($\underline{n} \mid n$) is required, analogous to the thin bootstrapping mechanism used in BFV/CKKS [17, 35]. We note that the computational cost of thin bootstrapping will decrease if $\underline{n}$ is reduced. However, in the bootstrapping philosophy of Sec. 3, the computational overhead does not change under the sparse packing case.

To address this issue, we propose an efficient batched bootstrapping for the sparse-aware packing. In detail, given the ciphertext $\mathrm{RLWE}^{n}_{\mathfrak{s},q}(\mathfrak{m})$ for $\mathfrak{m} = \sum_{i=0}^{\underline{n}-1} \mathfrak{m}_{i\cdot n/\underline{n}} \cdot X^{i\cdot n/\underline{n}}$, we can use the RLWE-to-MLWE extraction to generate the MLWE ciphertext as $(\mathfrak{a}_0, ..., \mathfrak{a}_{k-1}, \mathfrak{b}) \in \mathrm{MLWE}^{k,\underline{n}}_{\mathbf{s},q}(\mathfrak{m})$, where $k = n/\underline{n}$, and $\mathbf{S} = (\mathfrak{s}_0, ..., \mathfrak{s}_{k-1}) = \mathsf{RingSwitch}(\mathfrak{s})$. According to the MLWE decryption, we have $\mathfrak{m} = \mathfrak{b} - \sum_{i=0}^{k-1} \mathfrak{a}_i \cdot \mathfrak{s}_i$. Note that this extraction process preserves the sparsity for the secret key, i.e., $h = \sum_{i=0}^{k-1} hw(\mathfrak{s}_i)$. Then, we can rewrite the decryption formula as

$$\mathfrak{b} - \sum_{i=0}^{k-1} \mathfrak{a}_i \cdot \mathfrak{s}_i = \mathfrak{b} - \sum_{i=0}^{k-1} \sum_{j \in \mathsf{idx}(\mathfrak{s}_i)} \left(\sum_{t=0}^{\underline{n}-1} \mathfrak{a}(X) \right) \widetilde{\odot} X^j.$$

This process enables sequentially performing $h+k$ EMPmul through the MLWE decryption, but with a lower degree of the polynomial $\underline{n} = n/k$. To elaborate further, let the output of $i-1$-th iteration be **acc**, the i-th iteration can be evaluated as follows

$$\left(\left((\mathbf{acc} \widetilde{\odot} X^{-j_{i,0}} - \mathfrak{a}) \widetilde{\odot} X^{j_{i,0}-j_{i,1}} - \mathfrak{a} \right) - \widetilde{\odot} X^{j_{i,1}-j_{i,2}} \ldots - \mathfrak{a} \right) \widetilde{\odot} X^{j_{i,hw(\mathfrak{s}_i)}}. \tag{10}$$

After that, we can set $\mathbf{v}_i = (v_{i,0}, ..., v_{i,hw(\mathfrak{s}_i)}) = (-j_{i,0}, j_{i,0} - j_{i,1}, ..., j_{i,hw(\mathfrak{s}_i)})$ for $0 \le i \le k-1$, and encapsulate all $\mathbf{v}_i$ as the bootstrapping BSK_i as shown in Tab. 1, where $\mathsf{BSK}_{i,j} = \left\{ \mathsf{Bsk}^{\pm}_{i,j,0}, ..., \mathsf{Bsk}^{\pm}_{i,j,3} \right\}$. We show the detailed steps in Alg. 3, and analyze the computational complexity for the full packing and sparse packing cases as follows.

- **Full packing** case necessitates $h+1$ EMPmul, and each EMPmul involves $O(v \log n)$ external products. Since $v < n$, the overall computation overhead result in $O((h+1)n \log n)$ external product operations.
- **Sparse packing** case needs to perform the $h+k$ EMPmul, but with a lower degree of $\underline{n}$, resulting in a total of $O((h+k) \cdot \underline{n} \log \underline{n})$ external product operations. Note that in general settings, we have $k < h < n$.

Remark 2. After performing the accumulation process, it outputs $\underline{n}$ RLWE ciphertext as $\mathrm{RLWE}^{N}_{\widetilde{\mathfrak{s}},Q}(\mathfrak{m}_j) = (\mathfrak{a}_j, \mathfrak{b}_j)$ as shown in line 7 and line 17. If one wants to proceed with the next batched bootstrapping, these $m_{0,0}, \ldots, m_{\underline{n}-1,0}$ need to be

Algorithm 3 Batched Bootstrapping

Input:

The RLWE ciphertext $c \in \mathrm{RLWE}^{n}_{\mathfrak{s},q}(\mathfrak{m})$ under a sparse secret key $\mathfrak{s}$, where $\mathfrak{m}(X) = \sum_{i=0}^{\underline{n}-1} m_i X^{i\cdot\frac{n}{\underline{n}}}$. The bootstrapping key BSK encrypting the binary decomposition for $\mathbf{v_0}, ..., \mathbf{v}_h$ as shown in Eq. 10 and Tab 1 under a secret key $\widetilde{\mathfrak{s}}$. The automorphism keys $\mathsf{Auk}_i : \mathrm{RLWE}'_{\widetilde{\mathfrak{s}},Q}(\tau_i(\widetilde{\mathfrak{s}}))$ used in RLWE packing.

The RLWE key switching key $\mathsf{Rlk} : \mathrm{RLWE}'_{\mathfrak{s}}(\widetilde{\mathfrak{s}})$. $\underline{n}$ test polynomials testP_i that encode the functions $f_0, ..., f_{\underline{n}-1}$ as shown in Alg. 1.

Output:

The RLWE ciphertext that encrypts new message $\mathfrak{m}' = \sum_{i=0}^{\underline{n}-1} f_i(m_i) X^{i\cdot\frac{n}{\underline{n}}}$.

/* Full Packing Case $\underline{n} = n$*/

$(\mathfrak{a}, \mathfrak{b}) \leftarrow \mathsf{ModSwitch}_{q\to 2N}(c)$
Let $\mathbf{acc}_0 = (c_0, ..., c_{n-1})$, where $c_i = \mathsf{testP}_i \cdot X^{b_i}$
for $i = 0$ to h do
  $\mathbf{acc}_i \leftarrow \mathsf{EMPmul}(\mathbf{acc}_i, \mathsf{BSK}_i)$
  $\mathbf{acc}_i = (\mathsf{acc}_{i,0} \cdot X^{-a_0}, ..., \mathsf{acc}_{i,n-1} \cdot X^{-a_{n-1}})$
end for
$\mathbf{acc}_h \leftarrow \mathsf{EMPmul}(\mathbf{acc}_h, \mathsf{BSK}_h)$
$c' = \mathsf{Repack}(\mathbf{acc}_h, \mathsf{Auk})$
$c' = \mathsf{RLWE.KeySwitch}_{\widetilde{\mathfrak{s}}\to\mathfrak{s}}(c', \mathsf{Rlk})$
return c'

/* Sparse Packing Case $\underline{n} | n$ */

$(\mathfrak{a}_0, ..., \mathfrak{a}_{k-1}, \mathfrak{b}) \leftarrow \mathsf{RtM.Ext}_{n\to k,\underline{n}}(\mathsf{ModSwitch}_{q\to 2N}(c))$
Let $\mathbf{acc}_{0,0} = (c_0, ..., c_{\underline{n}-1})$, where $c_i = \mathsf{testP}_i \cdot X^{b_i}$
for $i = 0$ to $k-1$ do
  for $j = 0$ to $hw(\mathfrak{s}_i)$ do
    $\mathbf{acc}_{i,j} \leftarrow \mathsf{EMPmul}(\mathbf{acc}_{i,j}, \mathsf{BSK}_{i,j})$
    $\mathbf{acc}_{i,j} = (\mathsf{acc}_{i,j,0} \cdot X^{-a_{i,0}}, ..., \mathsf{acc}_{i,j,n-1} \cdot X^{-a_{i,n-1}})$
  end for
  $\mathbf{acc}_{i,hw(\mathfrak{s}_i)} \leftarrow \mathsf{EMPmul}(\mathbf{acc}_{i,hw(\mathfrak{s}_i)}, \mathsf{Bsk}_{i,hw(\mathfrak{s}_i)})$
$c' = \mathsf{SRepack}(\mathbf{acc}_{k-1,hw(\mathfrak{s}_{k-1})}, \mathsf{Auk})$
$c' = \mathsf{RLWE.KeySwitch}_{\widetilde{\mathfrak{s}}\to\mathfrak{s}}(c', \mathsf{Rlk})$
return c'

repacked into a new RLWE ciphertext under the sparse secret key. We adopted the sparse-secret encapsulation technique of AmortBoot [33]. In detail, the RLWE repacking procedure first packs these ciphertexts into an RLWE encryption under secret key $\widetilde{\mathfrak{s}}$, and subsequently performs key-switching to the secret key $\mathfrak{s}$ associated with the input RLWE, which possesses a lower Hamming weight. Since the computational overhead of this step has a negligible effect on the overall performance, we offer the algorithm description in Appendix B.4. Finally, we show the noise growth in Lemma 4.2.

Lemma 4.2. Input a ciphertext $\mathrm{RLWE}^{n}_{\mathfrak{s},q}(\mathfrak{m})$, some function f_i, and the associated key BSK, Auk, Rlk, the batched bootstrapping outputs a new RLWE, which can be further categorized into the following two cases:

1. Full packing outputs encryption of $\mathfrak{m}' = \sum_{i=0}^{n-1} f_i(m_i) X^i$, and the error variance satisfies $\mathsf{V}_{\mathsf{Boot}} < (h+1)\cdot \mathsf{V}_{\mathsf{EMP}} + \mathsf{V}_{\mathsf{Pack}} + \mathsf{V}_{\mathsf{RKS}}$.
2. Sparse packing outputs encryption of $\mathfrak{m}' = \sum_{i=0}^{\underline{n}-1} f_i(m_i) X^{i\cdot\frac{n}{\underline{n}}}$, where the error variance of satisfies $\mathsf{V}_{\mathsf{SBoot}} < (h+k)\cdot \mathsf{V}_{\mathsf{EMP}} + \mathsf{V}_{\mathsf{SPack}} + \mathsf{V}_{\mathsf{RKS}}$.

Here $\mathsf{V}_{\mathsf{EMP}}$, $\mathsf{V}_{\mathsf{Pack}}$, $\mathsf{V}_{\mathsf{SPack}}$, $\mathsf{V}_{\mathsf{RKS}}$ are associated with BSK, Auk, and Rlk. Note that in the sparse packing, the sparsity parameter k increases noise growth. However, in practical parameter settings where $k \ll h$, the resulting additional noise growth is negligible compared to that under the full packing case. The complete proof is present in Appendix C.2.

4.3 Batched Circuit Bootstrapping

Existing batched bootstrapping, including Alg. 3, focuses on the functional bootstrapping that can only evaluate the single-input function, i.e., $f(x) : \mathbb{Z}_p \to \mathbb{Z}_{p_{\mathsf{out}}}$, and has low message precision ($p < 2^8$). When higher precision is required, its efficiency decreases significantly due to the necessity of enlarging the system parameters. We note that the circuit bootstrapping [21] and emerging variants [38, 40, 51] effectively support a wider variety of functions without parameter inflation, such as the higher precision via multi-input function, e.g, $f(x_1, x_2) : \mathbb{Z}_p \times \mathbb{Z}_p \to \mathbb{Z}_{p_{\mathsf{out}}}$. We propose the BatchCBoot, a batched circuit bootstrapping that integrates the method of [40] (called EP-CBS). In particular, the workflow of EP-CBS can be divided into two subroutines, ciphertext conversion from $\mathrm{LWE}(m)$ to $\mathrm{RGSW}(X^{\widetilde{m}})$ and LUT evaluation with the external product tree. Our main insight is that BatchBoot can be used in the first stage to obtain better amortized results. Our approach is presented in the following two steps.

Step 1: Half.BatchCBS. The ciphertext conversion from $\mathrm{LWE}(m)$ to $\mathrm{RGSW}(X^{\widetilde{m}})$ is constructed in EP-CBS through an expensive blind rotation as shown in Alg. 1. Based on the BatchBoot, we construct a batched ciphertext conversion from $\mathrm{RLWE}^{n}_{\mathfrak{s},q}(\mathfrak{m})$ to $\mathrm{RGSW}(X^{\widetilde{m}_i})$, for $i \in [0, n-1]$, denoted as $\mathsf{Half.BatchCBS}$. Specifically, we begin by converting the ciphertext modulus from q to $\frac{2N}{\overline{d}}$ and scaling the ciphertext by a factor of $\overline{d}$, adhering to the encoding technique presented in EP-CBS [40]. This operation clears the $\log_2 \overline{d}$ least significant bits (LSBs) of the encoding space. These zeroed bits subsequently allow for the gadget vector to be embedded within the initial accumulator. To illustrate, for $\overline{d} = 2$, the n initial accumulators are configured as $\mathsf{acc}_i = \overline{B}_0 \cdot X^{b_i} + \overline{B}_1 \cdot X^{b_i+1}$.

By performing BatchBoot without repacking, it outputs n RLWE ciphertexts, that are $c_i = \mathrm{RLWE}^{N}_{\widetilde{\mathfrak{s}},Q}(\overline{B}_0 X^{\widetilde{m}_i} + \overline{B}_1 X^{\widetilde{m}_i+1})$, for $0 \le i \le n-1$. After that, we can perform the ciphertext conversion of EP-CBS [40]

$$\mathrm{RLWE}(\overline{B}_0 X^{\widetilde{m}_i} + \overline{B}_1 X^{\widetilde{m}_i+1}) \xrightarrow{\text{HomoTrace}} \mathrm{RLWE}'(X^{\widetilde{m}_i}) \xrightarrow{\text{SchemeSwitch}} \mathrm{RGSW}(X^{\widetilde{m}_i}).$$

Algorithm 4 Batched Circuit Bootstrapping

Input:

The RLWE ciphertext $c \in \mathrm{RLWE}^{n}_{\mathfrak{s},p/q}(\mathfrak{m})$ under a sparse secret key $\mathfrak{s}$, where $\mathfrak{m}(X) = \sum_{i=0}^{n-1} m_i X^i$.
The bootstrapping key BSK used in Alg. 3.
The automorphic key Auk as shown in Alg. 5.
The scheme switching key $\mathsf{Ssk} : \mathrm{RLWE}'_{\widetilde{\mathfrak{s}}}(\widetilde{\mathfrak{s}}^2)$.
Gadget vector $\mathbf{g} = (\overline{B}_0, ..., \overline{B}_{\overline{d}-1})$ corresponding to the output RGSW.
A multi-input function $f : [0, p-1]^{\beta} \to [0, p-1]^{\gamma}$.

Output:

Some ciphertexts of $f(m_0, ..., m_{\beta-1})$.

Let $(\mathfrak{a}, \mathfrak{b}) = \mathsf{ModSwitch}_{q \to 2N/\overline{d}}(c) \cdot \overline{d} \mod 2N$
for $i = 0$ to $n-1$ do
  $\mathsf{acc}_i = \overline{B}_0 \cdot X^{\mathfrak{b}_i} + \cdots + \overline{B}_{\overline{d}-1} \cdot X^{\mathfrak{b}_i + \overline{d} - 1}$
end for
$(c_0, ..., c_{n-1}) \leftarrow \mathsf{BatchBoot}(\mathsf{acc}_0, ..., \mathsf{acc}_{n-1}, \mathsf{BSK})$ /* Perform lines 2-6 of Alg. 3 */
for $i = 0$ to $n-1$ do
  for $j = 0$ to $\overline{d}-1$ do
    $c_{i,j} = c_i \cdot X^{-j}$
    $\overline{C}_{i,j+\overline{d}} = \mathsf{HomoTrace}.\mathrm{Tr}_{R_N/R_{N/\overline{d}}}(c_{i,j}, \mathsf{Auk})$
    $\overline{C}_{i,j} = \mathsf{Scheme.Switch}(\overline{C}_{i,j+\overline{d}}, \mathsf{Ssk}))$
  end for
end for
for $i = 0$ to $\gamma - 1$ do
  $c'_i = \mathsf{EP.Tree}(\overline{C}_0, ..., \overline{C}_{\beta-1})$. /* Perform external product tree in Alg. 8 *, where $\beta < n$./
end for
return $c'_0, ..., c'_{\gamma-1}$

as shown in lines 6-12 of Alg. 4. Here, since $\overline{B}_0$ and $\overline{B}_1$ fall into distinct equivalence classes, the HomoTrace (HT) $\mathrm{Tr}_{R_N/R_{N/2}}$[3] is employed to generate the $\mathrm{RLWE}(\overline{B}_0 \cdot X^{\widetilde{m}_i})$. Subsequently, the term $\overline{B}_1$ is rotated to align with $\overline{B}_0$ (via multiplication by X^{-i}), and the HomoTrace is repeated to produce a second RLWE ciphertext for $\overline{B}_1$. This procedure generalizes to any length $\overline{d}$, ultimately yielding a gadget RLWE ciphertext $\mathrm{RLWE}'(X^{\widetilde{m}_i})$. Finally, the secret key $\widetilde{\mathfrak{s}}$ is embedded into the ciphertext $\mathrm{RLWE}(-B_i\widetilde{\mathfrak{s}} \cdot X^{\widetilde{m}_i})$ through $\overline{d}$ times scheme switching as shown in Appendix B.3, and the $2\overline{d}$ ciphertexts make up the ciphertext $\mathrm{RGSW}(X^{\widetilde{m}_i})$.

Step 2: External product tree. Upon receiving RGSW ciphertexts $\mathrm{RGSW}(X^{\widetilde{m}_i})$ generated by $\mathsf{Half.BatchCBS}$, we use the external product tree proposed in EP-CBS to compute the multi-input function, such as $f(m_0, m_1)$. Given the ciphertexts $\mathrm{RGSW}(X^{\widetilde{m}_0})$ and $\mathrm{RGSW}(X^{\widetilde{m}_1})$ through the batched circuit bootstrapping, one can first perform p external products at the first level. The results are then repacked into a new RLWE ciphertext, corresponding to the second level of the test polynomial form. The final result is obtained after completing the last level.

Alg. 8 formalizes this for a generic function $f : [0, p-1]^{\beta} \to [0, p-1]^{\gamma}$. For each sub-function $f : [0, p-1]^{\beta} \to [0, p-1]$, the multi-valued bootstrapping method [13] (as shown in Appendix A) can be used at the first level,[4] which reduces $p^{\beta-1}$ external products. Hence, this process involves $\gamma \cdot \sum_{i=0}^{\beta-2} p^i = \gamma \cdot \frac{p^{\beta-1}-1}{p-1}$ external products, and $\gamma \cdot \sum_{i=0}^{p-2} p^i = \gamma \cdot \frac{p^{\beta-1}-1}{p-1}$ repacking operations. We omit the associated correctness analysis, which can be obtained directly from the external product and repacking steps. For a more detailed analysis, please refer to scheme [40], and the noise growth is presented in Appendix C.2. Finally, we adopt the notation $\mathsf{BatchCBS}_{p,\beta,\gamma}$ for configurations parameterized by message precision p, input size β in EP.Tree, and the number invocations γ for EP.Tree step.

5 Application for Batched Bootstrapping

In this section, we show two practical applications for the proposed batch bootstrapping: private set intersection and 8-bit FHE instruction sets.

5.1 Application I: Private Set Intersection

5.1.1 Universal Techniques for PSI

In PSI settings, sender holds a set X of size N_x, and the receiver holds a set of size N_y. Both sets consist of σ-bit strings. Like most FHE-based PSI protocols, our protocol is particularly powerful against unbalanced PSI, where the sender's set is much larger than the receiver's set, i.e. $N_y \ll N_x$. The ideal PSI functionality outputs $X \cap Y$ to the receiver and nothing to the sender. Hashing-to-bins can reduce the total number of comparisons and the cost of each comparison in a PSI protocol [25,43]. Elements are distributed into a predetermined number of bins, and comparisons are only performed between client and server elements in the same bin. Since the bin in which an element is placed is determined using hashes of that element, it is necessary to pad each bin with dummy elements up to a predetermined size ω to prevent the load of the bins from leaking information about the dataset. The hash functions required in PSI are as follows:

- *Cuckoo hashing* [46]. Cuckoo hashing can place N_y elements in $b = \varsigma N_y$ bins, for some constant ς, and each bin holds at most one element. Typically, the placement of each element is determined using $k = 3$ hash functions, and $b \approx 1.27 N_y$.

[3] We show the algorithm for general case Tr_{R_N/R_n} and noise growth in Appendix B.4, and more details can be referred to [38, 40].

[4] Details of this part can be found in Sec. 4 of [40].

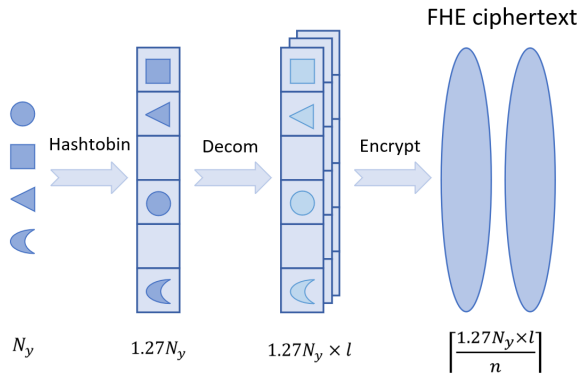

Figure 1: Dataset Preprocessing for client.

- *Permutation-based Hashing* [25, 43]. Permutation hashing can reduce the length of the elements being compared from σ to $\sigma' = \sigma - c$, where $c = \lceil \log_2 b \rceil$.

5.1.2 PSI protocol with Batched Bootstrapping

Our PSI protocol involves three stages: dataset preprocessing, server computation, and client decryption.

Step 1: Dataset Preprocessing. The parties first agree on FHE and PSI parameters such as $n, p, p_{\mathsf{out}}, q = 2N$, and b, ω, σ, σ'. Then, the client and server can asynchronously preprocess their datasets to prepare them for the next stage. Specifically, the preprocessing of the server is as follows: use permutation hash and insert these items into a two-dimensional hash table $\mathcal{B}^x$ using h hash functions $H_1, \ldots, H_h$, where $\mathcal{B}^x$ has b bins and each bin has ω elements, labeled as $B^x[i][j] \in \{0,1\}^{\sigma'}$ for $i \in [0, b-1]$, $j \in [0, \omega - 1]$; and then decompose all elements of $\mathcal{B}^x$ with the base p, i.e., $x = \sum_{i=0}^{\ell-1} p^i \cdot x_i$ ($\ell = \log_p 2^{\sigma'}$) to obtain ℓ sub-tables $\mathcal{B}_0^x, \ldots, \mathcal{B}_{\ell-1}^x$, where p is the plaintext modulus of FHE; finally, encode each $B^x[i][j]$ as a polynomial $\mathsf{TB}_{i,j}^k = X^{-\triangle \cdot B_k^x[i][j]} \sum_{i=-\triangle/2}^{\triangle/2} X^{-i}$, where $\triangle = \frac{N}{p}$.

On the other hand, the client first performs the permutation with Cuckoo hash that outputs a one-dimensional hash table $\mathcal{B}^y$ with $1.27N_y$ elements. After that, it decomposes all elements to obtain ℓ sub-tables $\mathcal{B}_0^y, \ldots, \mathcal{B}_{\ell-1}^y$. Finally, the client concatenates each sub-table and fills it with $\lceil \ell 1.27 \cdot N_y / n \rceil$ plaintext polynomials, then encrypts them into RLWE ciphertext with modulus q as shown in Fig. 1. Note that through the decomposition, we can deal with large data sizes $\sigma' > \log_2 p$, which also means that we need to compare x and y with a match being successful when all corresponding x_i and y_i are equal for $i \in [0, \ell - 1]$.

Step 2: PSI Computation. In this phase, the client generates the evaluation keys as shown in Alg. 3, and then transmits the evaluation keys and ciphertexts to the server. The server receives the ciphertexts and performs batched bootstrapping for all ciphertexts. This process produces the intermediate ciphertexts corresponding to the elements of sub-tables $\mathcal{B}_0^y, \ldots, \mathcal{B}_{\ell-1}^y$ as $\mathrm{RLWE}(X^{\widetilde{\mathcal{B}_k^y[i]}})$ as shown in lines 1-7 of Alg. 3, where $\widetilde{\mathcal{B}_k^y[i]} = \triangle \cdot \mathcal{B}_k^y[i] + e$, and e is noise of RLWE that satisfies $|e| < \frac{\triangle}{2}$. Then, the server evaluates the equivalence operation

$$\begin{aligned}
&\mathrm{RLWE}(X^{\widetilde{\mathcal{B}_k^y[i]}}) \cdot \mathsf{TB}_{i,j}^x = \mathrm{RLWE}(X^{\widetilde{\mathcal{B}_k^y[i]}}) \cdot X^{-\triangle \cdot B^x[i][j]} \sum_{i=-\triangle/2}^{\triangle/2} X^{-i} \\
&= \mathrm{RLWE}\left(X^{\triangle \cdot \mathcal{B}_k^y[i] - \triangle \cdot B^x[i][j]} \sum_{i=-\triangle/2}^{\triangle/2} X^{e-i} \right) \\
&= \mathrm{RLWE}(m_b + *).
\end{aligned}$$

for all $j \in [0, \omega - 1]$, $i \in [0, b-1]$, $k \in [0, \ell - 1]$, and uses the sample extraction to obtain LWE ciphertexts of m_b. Note that we can conclude that if $\mathcal{B}_k^x[i]$ equals $\mathcal{B}_k^y[i][j]$, $m_b = 1$; otherwise, it equals 0.

Subsequently, the server carries out the aggregation operations by LWE additions. The first step is to sum all elements of each bin as $c_{k,i} = \sum_{j=0}^{\omega-1} c_{k,i,j}$ with the result being an encrypted value of either 0 or 1. The second aggregation step sums all sub-tables as $c_i = \sum_{k=0}^{\ell-1} c_{k,i}$ to get b LWE ciphertexts. At this moment, the encrypted value m_i satisfies $m_i \in \{0, \ell - 1\}$, where $m_i = \ell$ corresponds to a successful intersection result. In addition, to prevent information leakage, the server selects a random $r_i \leftarrow \mathbb{Z}_{p_{\mathsf{out}}}^*$ [5] and computes $c' = (c_i - \ell) \cdot r_i$, for $i \in [0, n-1]$. This step serves to mask the other information, and a successful intersection implies that the encryption yields 0. To reduce the communication, the server uses Repack operation to integrate these LWE ciphertexts into an RLWE ciphertext. Finally, we apply bootstrapping once again to refresh the noise, ensuring that the final output noise is independent of the noise generated during the PSI computation step, as shown in [48]. Notably, we avoid the use of noise flooding techniques [15, 16, 43], as it results in an undesirable expansion of the system parameters.

Step 3: Decryption. In the decryption stage, the client decrypts the ciphertext received from the server to obtain the indicator vector. The indicator vector identifies whether a match has been found in the server set for each element in the client set. The client then extracts the indices encrypted as 0 and generates the intersection result.

5.1.3 Analysis of Our PSI Protocol

Security Analysis. Our PSI protocol operates in the semi-honest model, i.e., the client and server follow the protocol but may try to infer extra information. Client input privacy is guaranteed due to the use of homomorphic encryption. On the one hand, the random number r is selected to prevent information leakage of each bin. On the other hand, considering that the noise level in homomorphic encryption texts may reveal additional information about the server dataset, we use the bootstrapping method (also known as sanitisation, see [48]) to refresh the noise level, thereby protecting circuit privacy.

[5] $r_i \in [0, p_{\mathsf{out}} - 1]$, and p_{out} is set to a prime number greater than ℓ.

Communication Complexity. The total communication complexity consists of the request and response messages. In the request process, we have $\ell = \left\lceil \frac{2\sigma'}{p} \right\rceil$ tables, each containing $1.27 \cdot N_y$ elements. These elements are then encrypted, resulting in a total of $\mathsf{n}_{\mathsf{req}} = \lceil \ell 1.27 \cdot N_y / n \rceil$ ciphertexts. In addition, since we use RLWE symmetric encryption, the part of $\mathfrak{a}$ can be replaced by random seeds, thus reducing the ciphertext size by half. In the response phase, the server only returns $\mathsf{n}_{\mathsf{res}} = \lceil 1.27 \cdot N_y / n \rceil$ ciphertexts, after applying modulus switching to scale the ciphertext modulus down to an input modulus q. A key advantage of our protocol is the compact FHE parameter set, notably a 12-bit ciphertext modulus q. Consequently, the overall communication amount is minimized to about 15 KB. This optimization is a unique feature of our TFHE bootstrapping design and is infeasible for any past or future PSI implementation relying on BFV.

Computation Complexity. The computational cost of our protocol comprises $\mathsf{n}_{\mathsf{req}} + \mathsf{n}_{\mathsf{res}}$ BATCHBOOT processes, $\ell\omega \cdot b$ scalar multiplications for RLWE, and $\ell\omega \cdot b + \ell b$ LWE ciphertext additions. In addition, we set the temporary ciphertext modulus to the machine word size (i.e., $Q = 2^{64}$), leading to a substantial improvement in computational efficiency. We present a detailed performance and comparison analysis in Sec. 6.2.

5.2 Application II: FHE Instruction Sets

The FHE instruction set can handle addition, multiplication arithmetic operations, and logical operations. Zama [12] achieves these universal instruction sets based on the non-batch TFHE scheme, and further designs a privacy-preserving fhEVM coprocessor [26] for blockchain. Currently, several TFHE-based techniques can be utilized to construct these processors, including gate bootstrapping [12], tree-based bootstrapping [49], and circuit bootstrapping [50]. Nevertheless, these methods are limited to processing only one encrypted and one instruction set at a time. We employ BATCHCBOOT for batch computation of the instruction set, specifically targeting 8-bit messages as in state-of-the-art methods [49, 50]. We customize each instruction according to its operational characteristics, thereby ensuring efficient execution following the approach adopted in [49]. The proposed methodology comprises the following steps.

Message encoding and encryption. For the 8-bit instruction set, we aim to compute $f(u,v)$, such as the $\mathsf{ADD}(u,v) = u+v \pmod{2^8}$, where $u,v \in \mathbb{Z}_{2^8}$. Given $2n$ messages $\mathbf{u} = (u_0,\ldots,u_{n-1})$ and $\mathbf{v} = (v_0,\ldots,v_{n-1})$, we first decompose each message into two 4-bit, i.e., $u_i = u_{i,L} + 16 \cdot u_{j,M}$, which means that $p = 2^4$ in FHE system parameters. After that, we can encrypt these decomposed messages into four RLWE ciphertexts $(c_{u,L}, c_{u,M}, c_{v,L}, c_{v,M})$.

Instruction Sets Evaluation. In the evaluation of the general 16-bit-to-8-bit LUT, i.e. $\mathsf{BatchCBS}_{4,4,2}$, we perform Half.BatchCBS on these ciphertexts to obtain n ciphertext tuples, and each tuple containing ciphertexts $\mathrm{RGSW}(X^{\widetilde{u}_{i,L}}), \mathrm{RGSW}(X^{\widetilde{u}_{i,M}}), \mathrm{RGSW}(X^{\widetilde{v}_{i,L}}), \mathrm{RGSW}(X^{\widetilde{v}_{i,M}})$. Then, we can construct two external product trees, i.e., $\beta = 4$ and $\gamma = 2$ in Alg. 8, to evaluate the general LUT. This process demands four Half.BatchCBS invocations, along with $2 \times 16^2 \times n = 512n$ instances of repacking and an equal number ($512n$) of external product operations. The

Trama et al. [49] point out that the number of homomorphic operations can be reduced based on the characteristics of the instruction set. For example, the AND instruction can be defined as $\mathsf{AND}(u,v) = (u_M \,\&\, v_M, u_L \,\&\, v_L)$. Then, we can use two sub-procedures

$$\mathsf{BatchCBS}_{4,2,1}(c_{u,M}, c_{v,M}, \mathsf{AND})$$

$$\mathsf{BatchCBS}_{4,2,1}(c_{u,L}, c_{v,L}, \mathsf{AND})$$

to evaluate $u_M \& v_M$ and $u_L \& v_L$, respectively. Since the level of the external product tree is two (i.e., $\beta = 2$), we can reduce the number of external products operations and repacking steps from $512n$ to $2n$. We show the computational costs for different instruction sets in Tab 3, with detailed computation steps for each instruction outlined in [49].

Table 3: Computational costs for various 8-bit instruction sets.

Instructions	Half.BatchCBS	External product	Repacking
AND/XOR/OR	4	$2n$	$2n$
ADD/SUB	6	$4n$	$4n$
MUL	6	$4n$	$4n$
EQ	6	$3n$	$3n$
LT(E)/GT(E)	8	$5n$	$5n$
MIN/MAX	12	$10n$	$10n$

6 Evaluation

6.1 Methodology

Our algorithms mainly involve two ciphertext types: RLWE and RGSW, which serve as input ciphertext and bootstrapping keys. By default, the ciphertext modulus is set to 2^{64}, unless stated otherwise. The bootstrapping keys include bootstrapping key Bsk, automorphism key Auk, scheme switching key Ssk, and RLWE sparse switching key Rlk, each associated with the length d and base B used in the approximate gadget decomposition. We show the standard deviation of noise for different evaluation keys in Tab. 9 of Appendix C.3. We use the lattice estimator[6] to evaluate the security, where the security levels of these parameters exceed 128 bits. For the sparse secret key, we adopt the strategy [33] where the key generation

[6] https://github.com/malb/lattice-estimator.

process has a high probability of satisfying $j_i - j_{i+1} < n/h$. The security level is determined by using the Lattice Estimator [2] using the default reduction model and dual hybrid attack with meet-in-the-middle optimizations from [18]. Note that Rlk involves a larger standard deviation because it uses sparse keys for encryption. Then, in Appendix D.1, we show the parameters of BatchBoot in Tab. 10 and BatchCBoot in Tab. 11, respectively. Finally, under these parameters, we can calculate a concrete decryption failure rate. The detailed calculation process can be found in Appendix C.4.

6.2 Performance for Batched Bootstrapping

We show some experimental results for BatchBoot, and all of our algorithms are executed in the MOSFHE [32] library. The benchmarking experiments were conducted on a cloud server equipped with an Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz and 32 GB of RAM. The operating system was Ubuntu 22.04.4 LTS, and the code was compiled with g++ 14.2.0, which supports AVX-512 instruction set optimization. The single-threaded performance results of our Alg. 3 are presented in Tab. 4. We compare it with the state-of-the-art batched [33] and non-batched [12] implementations across various levels of message precision.

Table 4: Performance, key sizes, and decryption failure rate. Note that set $\mathsf{SBoot}_{i,k}$ is associated with the same set of parameters Boot_i and $\underline{n} = n/k$, which is used in sparse packing case.

Scheme	Sets	$\log_2 p$	Packed Mess.	Total Time	Amort. Time	Failure Prob.	Key Sizes	Speedup
TFHE-rs [12]	Def	2	1	9.8 ms	9.8ms	2^{-53}	76.3 MB	**2.74**×
AmortBoot [33]	Boot_2	2	2048	16.57s	8.09 ms	2^{-120}	17.1 MB	**2.2**×
BatchBoot	Boot_2	2	2048	7.32s	**3.57** ms	2^{-120}	59.6 MB	–
BatchBoot	$\mathsf{SBoot}_{2,2}$	2	1024	**3.83** s	3.74 ms	2^{-120}	70.5 MB	–
TFHE-rs [12]	Def	4	1	13.6 ms	13.6 ms	2^{-53}	52.1 MB	**3.77**×
AmortBoot [33]	Boot_4	4	2048	17.72s	8.65 ms	2^{-94}	18.4 MB	**2.4**×
BatchBoot	Boot_4	4	2048	7.39s	**3.6** ms	2^{-94}	64.1 MB	–
BatchBoot	$\mathsf{SBoot}_{4,2}$	4	1024	**3.86** s	3.71 ms	2^{-94}	75.8 MB	–
TFHE-rs [12]	Def	6	1	131.1 ms	131.1 ms	2^{-53}	488.5 MB	**15.55**×
AmortBoot [33]	Boot_6	6	4096	36.53s	8.92 ms	2^{-64}	18.6 MB	**1.05**×
BatchBoot	Boot_6	6	4096	34.57s	**8.43** ms	2^{-64}	120.1 MB	–
BatchBoot	$\mathsf{SBoot}_{6,2}$	6	2048	**18.31** s	8.94 ms	2^{-64}	142 MB	–
TFHE-rs [12]	Def	8	1	1.1 s	1.1 s	2^{-53}	3.25 GB	**43.8**×
AmortBoot [33]	Boot_8	8	4096	234.88 s	57.34 ms	2^{-62}	76.6 MB	**2.27**×
BatchBoot	Boot_8	8	4096	102.86s	**25.11** ms	2^{-62}	205.3 MB	–
BatchBoot	$\mathsf{SBoot}_{8,2}$	8	2048	**54.54** s	26.63 ms	2^{-62}	242.6 MB	–

Compared to TFHE-rs (using their default parameters[7]), we achieve gains of up to 43.8 times under 8-bit precision. After that, we implement the open source codes[8] corresponding to scheme [33] in the same test environment. Our method achieves a speedup of up to 2.4× compared to [33] due to a drastic reduction in FFT operations. Interestingly, we can reduce the latency from 7.39 s to 3.83 s for 4-bit LUT while maintaining approximately the same amortization results through the sparse packing case, which is not feasible with [33]. Tab. 4 also shows the comparison of the bootstrapping key sizes for different schemes. Although the key sizes of BatchBoot are larger than [33], the performance gains of BatchBoot are substantial. Additionally, the bootstrapping key is generated only once and therefore does not become a bottleneck in practical applications. On the other hand, we note that BatchBoot exhibits excellent scalability in multi-core CPU environments, since FFT and external products in the EMPmul procedure are inherently independent operations. Multi-threading experiments show that latency decreased from 102.8 s to 19 s under the 8-bit message precision by using 8 threads as shown in Tab.12, and more detailed performance results and comparisons are present in Appendix D.2.

Table 5: Performance comparison with the CKKS functional bootstrapping.

Scheme	Packed Mess.	4-4 LUT		6-6 LUT	
		Total Time	Amort. Time	Total Time	Amort. Time
CKKS [3]	65536	9.93 s	0.15 ms	12.7 s	0.19 ms
	2048	5.58 s	2.72 ms	6.97 s	3.4 ms
	512	5.35 s	10.4 ms	6.57 s	12.8 ms
BatchBoot	2048	7.39 s	3.6 ms	18.31 s	8.94 ms
	512	2.11 s	4.13 ms	5.4 s	10.7 ms

In addition, we compare BatchBoot with the state-of-the-art CKKS functional bootstrapping [3] by executing their open-source code[9], where the performance results are summarized in Tab. 5. From the result, we can derive the fact that CKKS bootstrapping performs better in the fully packing mode. However, under some sparse packing (e.g., 512 slots), BatchBoot achieves 2.5× improvement under 4-bit LUT evaluation by leveraging better sparse packing techniques. Moreover, our scheme still offers two additional advantages over CKKS-based bootstrapping. Firstly, CKKS relies on a super-polynomial modulus, while BatchBoot is built on a polynomial-modulus RLWE assumption with stronger security as shown in Appendix A. Secondly, CKKS bootstrapping entails huge key sizes, typically on the order of gigabytes in [3, 5, 6, 29, 36]. Even though HERMES [4] incorporates some optimization techniques to reduce bootstrapping key sizes, it still requires a key size of 673 MB under functional bootstrapping parameters. In contrast, BatchBoot achieves a 3.2−11.3× reduction in key sizes. Finally, we also present the peak memory usage in single-threaded setting in Tab. 13 of Appendix D. For 2-bit and 4-bit LUT parameters, BatchBoot utilizes 360 MB of memory, while CKKS-based scheme [29] demands a minimum of 13.8 GB. This significant disparity underscores the superior memory efficiency of

[7] https://github.com/zama-ai/tfhe-rs

[8] https://github.com/antoniocgj/Fast-Amortized-Bootstrapping

[9] https://github.com/openfheorg/openfhe-development/tree/main/src/pke

Table 6: Performance comparison for PSI protocols.

N_y	N_x	PEPSI [43]			BatchBoot		
		ω	Time (s)	Comm.	ω	Time (s)	Comm.
2^{24}	1024	6710	9.6	4.21 MB	25616	13.82/**0.7**×	15 KB/**287**×
	2048	6710	10.5	4.21 MB	20288	13.85/**0.75**×	30 KB/**143**×
	4096	6710	19	8.32 MB	10367	13.92/**1.36**×	51 KB/**170**×
2^{28}	1024	100565	152.9	4.32 MB	397376	66.89/**2.28**×	15 KB/**294**×
	2048	100565	152.9	4.32 MB	313363	67.26/**2.27**×	30 KB/**147**×
	4096	100565	279.9	8.32 MB	157572	67.34/**4.1**×	51 KB/**170**×
2^{32}	1024	N/A	N/A	N/A	6308095	870.93	15 KB
	2048	N/A	N/A	N/A	4968820	873.67	30 KB
	4096	N/A	N/A	N/A	2487973	874.98	51 KB

BATCHBOOT.

6.3 Performance for PSI Protocol

For the PSI protocol, we run experiments with $\sigma = 32$ bit elements. By using the permutation-based hashing, we can obtain $\kappa = \lceil \log_2 b \rceil$. Thus, the hash size of elements is $\sigma' = \sigma - \kappa$ bits, where $b \geq 2^{11}$ in our setting. We use the parameter set $\mathsf{PSI.Boot}_6$, and split each element of N_y into four digits as detailed in Sec. 5.1. Tab. 6 shows a comparison of runtime and communication costs with the scheme [43]. Here the size N_y is set to 1024, 2048, and 4096, and the size of N_x is chosen as 2^{24}, 2^{28} and 2^{32}. The server bin load ω is determined by the server set size and the number of bins, where we adopt the method [43, 46] to obtain the upper bounds for ω [43, 46] and validate them experimentally via binning simulations. The results indicate that our protocol achieves a dramatic reduction of two orders of magnitude (143–294×). In addition, we also show the runtimes for PSI under 32 threads. From the experimental results, we observe that for some smaller parameter sets (e.g., $N_x = 2^{24}$ and $N_y = 2048$), our scheme incurs higher latency. In contrast, a performance gain of 4.1 times is achieved over PEPSI [43] when N_x is scaled to 2^{28}. In addition, our solution can also support larger datasets, such as $N_x = 2^{32}$, whereas existing solutions do not show execution results in this case. Therefore, our proposed method is particularly advantageous for deployment in bandwidth-sensitive environments or for processing substantial data volumes. Finally, we present further comparative analyses (such as comparisons with [25]) in Appendix A.

6.4 Performance for General LUT Evaluation

For the batched circuit bootstrapping, since Alg. 4 introduces larger noise growth than Alg. 3 (as shown in Lemma C.2), we need to reset the parameters. We add the parameters $\overline{B}$, $\overline{d}$, and the scheme switching key Ssk in Tab. 11. These elements are used in Alg. 4 and can support a message precision of $p = 2^4$ for each sub-block. We first show the performance with general LUTs using our batched circuit bootstrapping as shown in Tab. 7. Under the 8-bit case, i.e., $\beta = \gamma = 2$ in Alg. 4, our method is **8.5-47**× faster than the previous circuit bootstrapping methods (TBC+ [49], WHS+ [50], LSL+ [40]) in terms of amortized results based on the benchmarks they provided. Finally, we evaluate 8-bit FHE instruction sets using the methodology from Sec. 5.2. We implement 6 practical instruction sets, including AND, ADD, MUL, etc., where the computational overhead of instruction evaluation can be optimized according to its circuit complexity. The amortization analysis reveals a **3.3-6.5**× performance improvement across these instruction sets.

Table 7: Amortization performance of general LUT.

Method	8-to-8 LUT	12-to-12 LUT	Speedup
TBC+ [49]	160.15 ms	4.6 s	**12.06/134.3** ×
WHS+ [50]	112.74 ms	170.62 ms	**8.5/4.9** ×
LSL+ [40]	630 ms	1.12 s	**47/35** ×
BatchBoot	**13.27**ms	**34.24**ms	–

Table 8: Performance of 8-bit instruction sets.

Instructions	TBC+ [49]	WHS+ [50]	**BatchCBoot**	**Speedup**
AND/OR/XOR	184 ms	227 ms	28.14 ms	**6.5**×
ADD/SUB	493 ms	227 ms	42.68 ms	**5.3**×
MUL	504 ms	227 ms	42.68 ms	**5.3**×
EQ	276 ms	227 ms	42.21 ms	**5.4**×
LT(E)/GT(E)	437 ms	227 ms	56.75 ms	**4**×
MIN/MAX	825 ms	227 ms	68.3 ms	**3.3**×

7 Conclusion

This paper presents BATCHBOOT, a batched bootstrapping method for TFHE that processes multiple ciphertexts simultaneously. We design efficient homomorphic polynomial multiplication to reduce the number of FFT operations, and introduce an MLWE-based decryption paradigm for sparse packing, ensuring linear complexity in packing density. We extend the functionality through circuit bootstrapping, enabling the evaluation of more general functions. Overall, BATCHBOOT yields a 2.4× speedup over state-of-the-art batched schemes and a 43.8× improvement over non-batched implementations. Practical applications include a PSI protocol and an 8-bit FHE instruction set; our approach achieves a 294× reduction in communication overhead and a 4.1× performance gain over previous BFV-based PSI protocols, and up to 6.5× computational speedup for the instruction set.

Ethical Considerations

We have assessed the value of publishing this work against potential implications. 1. Identification of Stakeholders. We performed a comprehensive analysis of all parties who could be affected by our proposed BatchBoot algorithm, PSI protocols, and FHE instruction set design. The following stakeholders were identified: Researchers in privacy-preserving and cryptography: These individuals may use and extend our efficient BatchBoot algorithm. Users or devices requiring private Set Intersection can utilize our proposed PSI to reduce latency and communication overhead. Compiler and systems developers targeting encrypted computing: FHE instruction set helps separate programming from cryptographic details, reduces latency for encrypted operations, and can lower barriers to practical privacy-preserving analytics in regulated domains. 2. Potential Impacts on Stakeholders. Impacts during the research process: The research process poses minimal ethical risk because: All experiments use public benchmark datasets or random number which contain no personally identifiable information. No sensitive organizational or user data were accessed. These properties ensure that the research caused no direct harm to any stakeholder. 3. Mitigations. We release only the artifacts essential for benchmark reproduction, documenting all underlying assumptions and limitations. Production deployments should enforce rigorous key management, fine-grained access control, rate limiting, comprehensive audit logging, and periodic compliance reviews to mitigate residual risks. 4. Rationale for Conducting and Publishing the Research. The motivations for publishing our work are as follows: Advancing batched TFHE has tangible practical value, as it can reduce both computational and communication overhead in PSI protocols. Making the technology publicly available enables the academic community to validate, optimize, and securely build upon its foundations. Given these benefits—along with the minimal associated risks and the presence of robust mitigation measures—we believe that the overall ethical impact of conducting and publishing this research is substantially positive.

Open Science

Following the open science policy, we are dedicated to guaranteeing that our work is transparent and easily available. Our work facilitates the analysis of batch bootstrapping while ensuring data confidentiality and security, thus promoting open science and collaboration. The code for Batchboot is available at https://doi.org/10.5281/zenodo.17936945.

References

[1] Albrecht, M., Chase, M., Chen, H., Ding, J., Goldwasser, S., Gorbunov, S., Halevi, S., Hoffstein, J., Laine, K., Lauter, K., et al.: Homomorphic encryption standard. In: Protecting privacy through homomorphic encryption, pp. 31–62. Springer (2022)

[2] Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. Journal of Mathematical Cryptology **9**(3), 169–203 (2015)

[3] Alexandru, A., Kim, A., Polyakov, Y.: General functional bootstrapping using ckks. In: Annual International Cryptology Conference. pp. 304–337. Springer (2025)

[4] Bae, Y., Cheon, J.H., Kim, J., Park, J.H., Stehlé, D.: HERMES: efficient ring packing using mlwe ciphertexts and application to transciphering. In: Annual International Cryptology Conference. pp. 37–69. Springer (2023)

[5] Bae, Y., Cheon, J.H., Kim, J., Stehlé, D.: Bootstrapping bits with CKKS. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 94–123. Springer (2024)

[6] Bae, Y., Kim, J., Stehlé, D., Suvanto, E.: Bootstrapping small integers with CKKS. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 330–360. Springer (2025)

[7] Bernard, O., Joye, M.: Bootstrapping (t)FHE ciphertexts via automorphisms: Closing the gap between binary and gaussian keys. Cryptology ePrint Archive, Paper 2025/163 (2025), https://eprint.iacr.org/2025/163

[8] Bossuat, J.P., Mouchet, C., Troncoso-Pastoriza, J., Hubaux, J.P.: Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 587–617. Springer (2021)

[9] Boura, C., Gama, N., Georgieva, M., Jetchev, D.: Simulating homomorphic evaluation of deep learning predictions. In: International Symposium on Cyber Security Cryptography and Machine Learning. pp. 212–230. Springer (2019)

[10] Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Annual Cryptology Conference. pp. 868–886. Springer (2012)

[11] Brakerski, Z., Langlois, A., Peikert, C., Regev, O., Stehlé, D.: Classical hardness of learning with errors. In: Proceedings of the forty-fifth annual ACM symposium on Theory of computing. pp. 575–584 (2013)

[12] Brenna, L., Singh, I.S., Johansen, H.D., Johansen, D.: TFHE-rs: A library for safe and secure remote computing using fully homomorphic encryption and trusted execution environments. Array **13**, 100118 (2022)

[13] Carpov, S., Izabachène, M., Mollimard, V.: New techniques for multi-value input homomorphic evaluation and applications. In: Topics in Cryptology–CT-RSA 2019: The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4–8, 2019, Proceedings. pp. 106–126. Springer (2019)

[14] Chen, H., Dai, W., Kim, M., Song, Y.: Efficient homomorphic conversion between (ring) LWE ciphertexts. In: International Conference on Applied Cryptography and Network Security. pp. 460–479. Springer (2021)

[15] Chen, H., Huang, Z., Laine, K., Rindal, P.: Labeled psi from fully homomorphic encryption with malicious security. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1223–1237 (2018)

[16] Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1243–1255 (2017)

[17] Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for approximate homomorphic encryption. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 360–384. Springer (2018)

[18] Cheon, J.H., Hhan, M., Hong, S., Son, Y.: A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret lwe. IEEE Access **7**, 89497–89506 (2019)

[19] Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 409–437. Springer (2017)

[20] Chillotti, I., Gama, N., Georgieva, M., Izabachene, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22. pp. 3–33. Springer (2016)

[21] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 377–408. Springer (2017)

[22] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. Journal of Cryptology **33**(1), 34–91 (2020)

[23] 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)

[24] Chung, H., Kim, H., Kim, Y.S., Lee, Y.: Amortized large look-up table evaluation with multivariate polynomials for homomorphic encryption. Cryptology ePrint Archive (2024)

[25] Cong, K., Moreno, R.C., da Gama, M.B., Dai, W., Iliashenko, I., Laine, K., Rosenberg, M.: Labeled PSI from homomorphic encryption with reduced computation and communication. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 1135–1150 (2021)

[26] Dahl, M., Danjou, C., Demmler, D., Frederiksen, T., Ivanov, P., Joye, M., Rotaru, D.: fhevm: Confidential evm smart contracts using fully homomorphic encryption (2023)

[27] De Micheli, G., Kim, D., Micciancio, D., Suhl, A.: Faster amortized fhew bootstrapping using ring automorphisms. p. 322–353. Springer-Verlag, Berlin, Heidelberg (2024)

[28] Ducas, L., Micciancio, D.: FHEW: bootstrapping homomorphic encryption in less than a second. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 617–640. Springer (2015)

[29] Dumezy, J., Alexandru, A., Polyakov, Y., Clet, P.E., Chakraborty, O., Boudguiga, A.: Evaluating larger lookup tables using ckks. IACR Cryptol. ePrint Arch. **2025**, 1301 (2025), https://api.semanticscholar.org/CorpusID:280038428

[30] Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive (2012)

[31] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the forty-first annual ACM symposium on Theory of computing. pp. 169–178 (2009)

[32] Guimarães, A., Borin, E., Aranha, D.F.: Mosfhet: Optimized software for fhe over the torus. Journal of Cryptographic Engineering **14**(3), 577–593 (2024)

[33] Guimarães, A., Pereira, H.V.: Fast amortized bootstrapping with small keys and polynomial noise overhead. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 2967–2981 (2025)

[34] Guimarães, A., Pereira, H.V., Van Leeuwen, B.: Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 3–35. Springer (2023)

[35] Halevi, S., Shoup, V.: Bootstrapping for helib. Journal of Cryptology **34**(1), 7 (2021)

[36] Kim, J., Noh, T.: Modular reduction in CKKS. Cryptology ePrint Archive (2024)

[37] Kluczniak, K., Schild, L.: FDFB: Full domain functional bootstrapping towards practical fully homomorphic encryption. IACR Transactions on Cryptographic Hardware and Embedded Systems **2023, Issue 1**, 501–537 (2022)

[38] Lee, K.H., Yoon, J.W.: Homomorphic field trace revisited : Breaking the cubic noise barrier. Cryptology ePrint Archive, Paper 2025/1088 (2025), https://eprint.iacr.org/2025/1088

[39] Lee, Y., Micciancio, D., Kim, A., Choi, R., Deryabin, M., Eom, J., Yoo, D.: Efficient FHEW bootstrapping with small evaluation keys, and applications to threshold homomorphic encryption. In: Advances in Cryptology–EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part III. pp. 227–256. Springer (2023)

[40] Li, Z., Shen, X., Lu, X., Wang, R., Zhao, Y., Wang, Z., Wei, B.: Leveled functional bootstrapping via external product tree. Cryptology ePrint Archive, Paper 2025/022 (2025), https://eprint.iacr.org/2025/022

[41] Liu, Z., Wang, Y.: Amortized functional bootstrapping in less than 7 ms, with o˜(1) polynomial multiplications. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 101–132. Springer (2023)

[42] Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. Journal of the ACM (JACM) **60**(6), 1–35 (2013)

[43] Mahdavi, R.A., Lukas, N., Ebrahimianghazani, F., Humphries, T., Kacsmar, B., Premkumar, J., Li, X., Oya, S., Amjadian, E., Kerschbaum, F.: Pepsi: Practically efficient private set intersection in the unbalanced setting. In: 33rd USENIX Security Symposium. pp. 6453–6470 (2024)

[44] Micciancio, D., Polyakov, Y.: Bootstrapping in FHEW-like cryptosystems. In: Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography. pp. 17–28 (2021)

[45] Micciancio, D., Sorrell, J.: Ring packing and amortized FHEW bootstrapping. Cryptology ePrint Archive (2018)

[46] Pagh, R., Rodler, F.F.: Cuckoo hashing. Journal of Algorithms **51**(2), 122–144 (2004)

[47] Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM (JACM) **56**(6), 1–40 (2009)

[48] Smart, N.P., Walter, M.: Error-simulatable sanitization for TFHE and applications. IACR Communications in Cryptology **2**(3) (2025)

[49] Trama, D., Boudguiga, A., Clet, P.E., Sirdey, R., Ye, N.: Designing a general-purpose 8-bit (t) fhe processor abstraction. IACR Transactions on Cryptographic Hardware and Embedded Systems **2025**(2), 535–578 (2025)

[50] Wang, R., Ha, J., Shen, X., Lu, X., Chen, C., Wang, K., Lee, J.: Refined tfhe leveled homomorphic evaluation and its application. In: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security. pp. 2309–2323 (2025)

[51] 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)

A Related Work

This section presents related work and provides some comparison results.

BFV/CKKS-based functional bootstrapping. A recent series of works [3, 5, 6, 24, 36, 41] develop BFV and discrete CKKS schemes to support amortized functional bootstrapping. These schemes use polynomial interpolation to evaluate arbitrary functions while reducing error based on the noise cleaning technique. Note that BFV/CKKS schemes are characterized by significant noise growth and large parameters, which often necessitate stronger security assumptions. More precisely, if one can efficiently solve (R)LWE instances in dimension N with modulus Q and discrete Gaussian error with parameter σ, then one can also solve hard problems such as the approximate shortest vector problem, GapSVP_γ, in dimension N with approximation factor $\gamma = O(N \cdot Q/\sigma)$ [11, 47]. To ensure correctness under higher noise in BFV/CKKS schemes, the ciphertext modulus Q must be chosen considerably larger than the error bound e, leading to increased γ for the underlying lattice problem that governs security. As γ grows, the associated lattice problems such as GapSVP_γ become easier to solve. Consequently, assuming the worst-case hardness of GapSVP_γ becomes a stronger security assumption as γ increases.

FHEW/TFHE-based batch functional bootstrapping. In Se. 6.2, we provide the comparison and analysis of the state-of-the-art scheme [33]. We note that there is another route for batch functional bootstrapping for FHEW/TFHE schemes [27, 34, 45]. These schemes use the NTT to evaluate polynomial multiplication in RLWE decryption without utilizing the sparsity of the secret key. This method can reduce the complexity to $O(n \log n)$. However, to keep the noise in approximate polynomial factors, the recursive depth is limited to a small constant. A concrete implementation is presented in [34], with the amortization result of 851 ms under 7 message precision bits. Our method significantly outperforms this, with a 34× speedup. Furthermore, their approach requires very large bootstrapping keys (65.3 GiB in total), our scheme reduces it to just 205.3 MB, a reduction by a factor of 584.

Other FHE-based PSI. The FHE-based PSI protocol [16] was proposed by Chen et al. in 2017, which focuses on unbalanced settings by using the BFV scheme. Their foundational idea was to represent the equality operation via an interpolation polynomial. This evaluation reveals an intersection if the client's element exists in the server's set, result equals to zero; otherwise, the result is a random number. To improve performance, Cong et al. [25] introduce some optimizations, such as incorporating extremal postage-stamp bases and Frobenius operation, to reduce multiplicative depth and support for elements of arbitrary bit length. Regarding parameter settings $n = 2^{28}$ and $m = 4096$, the overall latency of the protocol is 2133 s under 32 threads, which is 31 times slower than our PSI solution. However, their protocol achieves faster server-side online computation. Specifically, their method requires only 2.5 s for online time, which is 20.4× faster than our method (51 s in our method). Moreover, our approach yields a 357× reduction in communication overhead, substantially alleviating network bandwidth constraints and enhancing scalability for large-scale deployments.

B Fundamental Algorithms and Noise Growth

B.1 Notations

The lower-case regular bold letters indicate vectors in this paper. For a real number r, we write the floor, ceiling, and round functions as $\lfloor r \rfloor$ $\lceil r \rceil$ $\lfloor r \rceil$, respectively. We use $\mathbb{Z}_Q$ to denote the Qring of integers $\mathbb{Z}/Q\mathbb{Z}$ in the range $[-Q/2, Q/2) \cap \mathbb{Z}$. In addition, we use the cyclotomic ring $R_{N,Q} = \mathbb{Z}_Q[X]/\langle X^N + 1 \rangle$, where N is an integer to the power of 2, and its coefficients are from $\mathbb{Z}_Q$. The notation a_i or $a[i]$ is used to index the vector and polynomial coefficients. We use $\mathfrak{s}$ or $\mathfrak{s}(X)$ to denote polynomials, and s_i represents the i-th coefficient of the polynomial, where its coefficient vector is $\mathsf{Coef}(\mathfrak{s}) = (s_0, \ldots, s_{N-1}) \in \mathbb{Z}_Q^N$. Furthermore, we use $\mathsf{Var}(\mathfrak{a})$ to denote the variance of the Gaussian distribution for $\mathfrak{a} \in R$, and $\mathsf{V}_{\mathsf{step}}$ to indicate the variance accumulated during step execution.

B.2 Noise Growth for GP, EP, and CMux

For the gadget product with the gadget vector $\mathbf{g} = (B_0, ..., B_{d-1})$, the error variance of the gadget product result is bounded by $\mathsf{V}_{\mathsf{GP}} = \frac{B^2}{12} dN \cdot \mathsf{Var}(c_{\mathsf{in}}) + \frac{N}{12} \cdot \left(\left\lceil \frac{Q}{B^d} \right\rceil \right)^2$ as shown in [22], where $\mathsf{Var}(c_{\mathsf{in}})$ is the error variance of the input gadget RLWE. Since the external product requires two gadget products, we can derive the noise variance as $\mathsf{V}_{\mathsf{EP}} = \frac{B^2}{6} dN \cdot \mathsf{Var}(C_{\mathsf{in}}) + \frac{N}{12} \cdot \left(\left\lceil \frac{Q}{B^d} \right\rceil \right)^2$, where $\mathsf{Var}(C_{\mathsf{in}})$ is the error variance of the input RGSW ciphertext. Furthermore, the CMux has an additional subtraction from the external product, so the noise variance is twice that of the first term of external product, i.e., $\mathsf{V}_{\mathsf{CMux}} = \frac{B^2}{3} dN \cdot \mathsf{Var}(C_{\mathsf{in}}) + \frac{N}{12} \cdot \left(\left\lceil \frac{Q}{B^d} \right\rceil \right)^2$.

B.3 Homomorphic Operations

Modulus Switching (MS). Given an RLWE ciphertext $c = (\mathfrak{a}, \mathfrak{b}) \in \mathrm{RLWE}^n_{\mathbf{s},p/Q}(m)$ with error variance $\mathsf{Var}(c_{\mathsf{in}})$, the modulus switching algorithm computes $\mathsf{ModSwitch}_{Q \to q}(c) = \left(\left\lfloor \frac{q}{Q} \cdot \mathfrak{a} \right\rceil, \left\lfloor \frac{q}{Q} \cdot \mathfrak{b} \right\rceil \right)$. It is easy to verify that the ciphertext is a correct ciphertext under modulo q, and its variance is $\mathsf{V}_{\mathsf{MS}} \leq (\frac{q}{Q})^2 \cdot \mathsf{Var}(c_{\mathsf{in}}) + \frac{||\mathbf{s}||_2^2 + 1}{12}$, where the factor $\frac{1}{12}$ is the standard deviation of a uniform distribution in $[-1/2, 1/2]$. For the binary secret key with h Hamming Weight, we have $\mathsf{V}_{\mathsf{MS}} \leq (\frac{q}{Q})^2 \cdot \mathsf{Var}(c_{\mathsf{in}}) + \frac{h+1}{12}$.

*RLWE key switching (*RKS*).* The RLWE key switching can convert the ciphertext $\mathrm{RLWE}_{\mathfrak{s}_1,Q}(\mathfrak{m})$ to a new ciphertext $\mathrm{RLWE}_{\mathfrak{s}_2,Q}(\mathfrak{m})$. Given the $c = (\mathfrak{a}, \mathfrak{b}) \in \mathrm{RLWE}_{\mathfrak{s}_1,Q}(\mathfrak{m})$, and the key switching key $\mathsf{Rlk} = \mathrm{RLWE}'_{\mathfrak{s}_2,Q}(\mathfrak{s}_1)$, the RLWE key switching operation performs $\mathsf{RLWE.KeySwitch}(c, \mathsf{Rlk}) = (0, \mathfrak{b}) - \mathfrak{a} \odot \mathsf{Rlk}$, which outputs the ciphertext $\mathrm{RLWE}_{\mathfrak{s}_2,Q}(\mathfrak{m})$, where the correctness can be found in [22, 39]. The error variance $\mathsf{V}_{\mathsf{RKS}}$ is the same as that of the gadget product V_{GP}.

*Scheme switching (*SS*).* We can use the scheme switching [27, 40, 51] to embed the secret key $\mathfrak{s}$ into the message. Specifically, given the key switching key $\mathsf{Ssk} = \mathrm{RLWE}'_{\mathfrak{s},Q}(\mathfrak{s}^2)$, and the ciphertext $c = (\mathfrak{a}, \mathfrak{b}) \in \mathrm{RLWE}_{\mathfrak{s},Q}(m)$, we can compute $\mathsf{Scheme.Switch}(c, \mathsf{Ssk}) = \mathfrak{a} \odot \mathsf{Ssk} + (\mathfrak{b}, 0)$, which outputs ciphertext $\mathrm{RLWE}_{\mathfrak{s},Q}(-\mathfrak{s} \cdot \mathfrak{m})$ as shown in [27, 51], where the error variance satisfies $\mathsf{V}_{\mathsf{SS}} \leq \frac{N}{2} \cdot \mathsf{Var}(c_{\mathsf{in}}) + \mathsf{V}_{\mathsf{RKS}}$.

RingSwitch (RS). Ring switching can switch the polynomial ring $R_{N,Q}$ to the subset and coset for polynomial $R_{N/k,Q}$, where both k and N are powers of 2, and $k|N$. Given a polynomial $\mathfrak{m}(X) = m_0 + m_1 X + \cdots + m_{N-1} X^{N-1}$, we define the ring switching as $(\mathfrak{m}_0, ..., \mathfrak{m}_{k-1}) \leftarrow \mathsf{RingSwitch}(\mathfrak{m})$, where $\mathfrak{m}_i(X) = m_i X^i + m_{k+i} X + \cdots + m_{(N/k-1)k+i} X^{N/k-1} \in R_{N/k,Q}$. This process does not increase noise, and please refer to work [4] for more details.

Algorithm 5 HomoTrace for Tr_{K_N/K_n}

Input:
The RLWE ciphertext $c = (\mathfrak{a}, \mathfrak{b}) \in \mathrm{RLWE}^N_{\mathfrak{s},Q}(\mathfrak{m}(X))$.
The automorphism keys $\mathsf{Auk}_i = \mathrm{RLWE}'_{\mathfrak{s},Q}(\tau_i(\mathfrak{s}))$, where $i \in \{2^{\log N-k+1}+1\}$ and $k \in [1, \log N]$.
Output:
RLWE ciphertext $c' \in \mathrm{RLWE}(\mathrm{Tr}_{K_N/K_n}(\mathfrak{m}))$.

1: for $i = \log_2 n + 1$ to $\log_2 N$ do
2: $\quad c' = \mathsf{ModSwitch}_{Q \to Q/2}(c')$
3: $\quad c' = \mathsf{ModRaise}_{Q/2 \to Q}(c')$
4: $\quad c' = c' + \mathsf{HomoAuto}_{\tau_{2^i+1}}(c', \mathsf{Auk}_{2^i+1})$,
5: end for
6: **return** c'.

Algorithm 6 RLWE ciphertext repacking (RLWE.Repack).

Input: n RLWE samples $c_j \in \mathsf{RLWE}^N_{\mathfrak{s},Q}(\mathfrak{m}_i)$, $j \in [0, n-1]$, where $n|N$.
The automorphism keys $\mathsf{Auk}_i = \mathrm{RLWE}'_{\mathfrak{s},Q}(\tau_i(\mathfrak{s}))$, where $i \in \{2^{\log N-k+1}+1\}$ and $k \in [1, \log N]$.
Output: An RLWE ciphertext $\mathsf{ct}' \in \mathsf{RLWE}_{\mathfrak{s}}(\mathfrak{m}))$.

1: for $i = 1$ to $\log n$ do
2: $\quad$ for $j = 0$ to $n/2^i - 1$ do
3: $\quad\quad \widetilde{c}_j = c_j - X^{\frac{N}{2^i}} \cdot c_{j+n/2^i}$,
4: $\quad\quad \widetilde{c}_j = \mathsf{ModSwitch}_{Q \to Q/2}(c')$
5: $\quad\quad \widetilde{c}_j = \mathsf{ModRaise}_{Q/2 \to Q}(c')$
6: $\quad\quad c_j = \widetilde{c}_j + \mathsf{HomoAuto}_{\tau_i}(\widetilde{c}_j, \mathsf{Auk}) + X^{\frac{N}{2^i}} \cdot c_{j+n/2^i}$,
7: $\quad$ end for
8: end for
9: if $n < N$, **return** $\mathsf{ct}' = \mathsf{Eval.Tr}_{K_N/K_n}(c_0)$.
10: if $n = N$, **return** c_0.

B.4 HomoTrace and Repacking

Homomorphic Trace (HomoTrace). We show the HomoTrace operation for Tr_{K_N/K_n} $(n|N)$ as shown in Alg. 5. Specifically, given the ciphertext $\mathrm{RLWE}(\mathfrak{m}(X))$ for $\mathfrak{m}(X) = m_0 + \cdots + m_{N-1}X^{N-1}$, this algorithm outputs a new ciphertext $\mathrm{RLWE}(\mathrm{Tr}_{K_N/K_n}(\mathfrak{m}))$, where $\mathrm{Tr}_{K_N/K_n}(\mathfrak{m}(X)) = \sum_{i=0}^{n-1} \mathfrak{m}_{i\cdot N/n} \cdot X^{i\cdot N/n}$. Note that this algorithm also incurs an expansion by a factor of N/n. Here, we use the method of RevHomTrace [38] to remove the effect of this factor inflation by evaluating the change in ModSwitch and ModRaise between Q and $Q/2$. In addition, the error variance of HomoTrace is bound by $\mathsf{V}_{\mathsf{HT}} \le \log_2(\frac{N}{n}) \cdot \mathsf{V}_{\mathsf{Auto}} + 4\log_2(\frac{N}{n}) \cdot \mathsf{V}_{\mathsf{MS}} + \mathsf{Var}(\mathsf{Auk})$. For more details, please refer to [38].

Repacking. (Repack) Given n LWE ciphertexts or RLWE ciphertexts as $\mathrm{RLWE}^N_{\mathfrak{s},Q}(\mathfrak{m}_j) = (\mathfrak{a}_j, \mathfrak{b}_j)$, where $i \in [0, n-1]$ and $n|N$. We can pack these ciphertexts into an RLWE ciphertexts as shown in Alg. 6. Here, we consider two scenarios based on the number of ciphertexts n to be packed.

- Sparse packing case $n|N$. In this case, all ciphertexts are first integrated into an RLWE that encrypts $\sum_{j=0}^{n-1} m_{j,0} \cdot X^{j\cdot\frac{N}{n}} \cdot \sum_{k=0}^{\frac{N}{n}-1} *X^k$ through the repacking. Next, the Tr_{K_N/K_n} is used to eliminate redundancies Alg. 5. Therefore, the repacking process requires $\frac{n}{2} + \frac{n}{4} + \cdots + 1 + \log(N/n) = n - 1 + \log(N/n)$ automorphisms, and the error variance can be summarized as $\mathsf{V}_{\mathsf{Pack}} = (\log_2 n + \log_2(N/n)) \cdot (\mathsf{V}_{\mathsf{Auto}} + 4\mathsf{V}_{\mathsf{MS}}) + \mathsf{Var}(c_{\mathsf{in}})$.
- Full packing case $n = N$. In this case, we apply the automorphism technique to combine them into an RLWE ciphertext, and this process requires $\frac{N}{2} + \frac{N}{4} + \cdots + 1 = N - 1$ automorphisms. The error variance is bound by $\mathsf{V}_{\mathsf{Pack}} = \log_2 N \cdot (\mathsf{V}_{\mathsf{Auto}} + 4\mathsf{V}_{\mathsf{MS}}) + \mathsf{Var}(c_{\mathsf{in}})$ as shown in [38].

C Algorithm and Analysis for BatchBoot

C.1 EMPmul for negative number

Recall that the Alg. 2 presents the EMPmul procedure with the positive integer v. Here, the Alg. 7 shows the case for negative v, i.e, $v \in [-n, 0]$. For the negative, the encrypted message will be rotated to the left. This means the sub-loop order and output ciphertext indices must be reversed, while its noise growth remains the same as Alg. 2.

C.2 Correctness and noise growth

Proof of Lemma 4.1 Firstly, Alg. 2 performs $\ell/2$ iterations, each iteration involves four external products. Its correctness can be obtained directly from the multi-bit CMux gates and the parameterized external product. In addition, for the noise growth, Alg. 2 removes the noise from the automorphism, but introduces the noise growth of four external products. Since the noise variance of the external product is additive, after $\ell/2$ iterations, we can conclude that $\mathsf{V}_{\mathsf{EMp}} < \mathsf{Var}(c_{\mathsf{in}}) + 2\ell \cdot \mathsf{V}_{\mathsf{EP}}$.

Proof of Lemma 4.2. Alg. 3 first performs $h + k$ EMPmul operations (Alg. 2), with additive noise as stated in Lemma 4.2. The resulting ciphertexts are then repacked into a new RLWE ciphertext, with error variance analyzed in Sec. B.4. Finally, RLWE key switching adjusts the key to match the original secret key input, bounding the error variance by

$$\begin{aligned}\mathsf{V}_{\mathsf{Boot}} &< (h+k) \cdot \mathsf{V}_{\mathsf{EMP}} + \mathsf{V}_{\mathsf{Pack}} + \mathsf{V}_{\mathsf{RKS}} \\ &< 2(h+k)\ell \cdot \mathsf{V}_{\mathsf{CMux}} + \log_2 N \cdot (\mathsf{V}_{\mathsf{Auto}} + 4\mathsf{V}_{\mathsf{MS}}) + \mathsf{V}_{\mathsf{RKS}}.\end{aligned}$$

Lemma C.1. Input an RLWE ciphertext $\mathsf{ct} \in \mathrm{RLWE}^n_{\mathfrak{s},q}(\mathfrak{m})$, the bootstrapping key Bsk, automorph key Auk, and RLWE key switching key Rvk. Alg. 4 outputs a n encryption for $\mathrm{RGSW}^N_{\widetilde{\mathfrak{s}},Q}(X^{m_i})$ for $i \in [0, n-1]$, and its variance satisfies

$$V_{\mathsf{CBoot}} \le N(h+k)\ell \cdot V_{\mathsf{CMux}} + \frac{N \cdot \log_2 \overline{d}}{2} \cdot V_{\mathsf{Auto}} + V_{\mathsf{RKS}}.$$

Proof of Lemma C.1. Alg. 4 first performs BatctBoot as shown in Alg. 3 without repacking, which outputs n RLWE

Algorithm 7 Efficient MPmul for negative number

Input:
A list $c_i \in \mathrm{RLWE}(X^{u_i})$ for $0 \le i \le n$ representing a polynomial $u(X)$ encrypted in the exponent.
The bootstrapping key $\mathsf{Bsk}_{i,j}$ for $0 \le i < \ell/2-1$ and $0 \le j \le 3$ as shown in Tab 1, which is associated with an encrypted value $-v, (v > 0)$.

Output:
A list $\hat{c}_i \in \mathrm{RLWE}(X^{w_i})$ for polynomial $w(X) = u(X) \cdot X^{-v} \mod X^n+1$, where $0 \le i \le n$.

for $i = 0$ to $\ell/2-1$ do
  for $j = 0$ to $n-1$ do
    $\hat{c}_j = \mathsf{FFT} \circ \mathsf{Decom}(c_j)$
  end for
  for $j = 0$ to $n-2^{2i}-2^{2i+1}-1$ do
    $\hat{c}_j = c_{j+2^{2i}} \boxtimes \mathsf{Bsk}^{+}_{2i,0} + c_{j+2^{2i+1}} \boxtimes \mathsf{Bsk}^{+}_{2i,1} + c_{j+2^{2i}+2^{2i+1}} \boxtimes \mathsf{Bsk}^{+}_{2i,2} + c_j \boxtimes \mathsf{Bsk}^{+}_{2i,3}$
  end for
  for $j = n-2^{2i}-2^{2i+1}$ to $n-2^{2i+1}-1$ do
    $\hat{c}_j = c_{j+2^{2i}} \boxtimes \mathsf{Bsk}^{+}_{2i,0} + c_{j+2^{2i+1}} \boxtimes \mathsf{Bsk}^{+}_{2i,1} + c_{j+2^{2i}+2^{2i+1}-n} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,2} + c_j \boxtimes \mathsf{Bsk}^{+}_{2i,3}$
  end for
  for $j = n-2^{2i+1}$ to $n-2^{2i}-1$ do
    $\hat{c}_j = c_{j+2^{2i}} \boxtimes \mathsf{Bsk}^{+}_{2i,0} + c_{j+2^{2i+1}-n} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,1} + c_{j+2^{2i}+2^{2i+1}-n} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,2} + c_j \boxtimes \mathsf{Bsk}^{+}_{2i,3}$
  end for
  for $j = n-2^{2i}$ to $n-1$ do
    $\hat{c}_j = c_{j+2^{2i}-n} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,0} + c_{j+2^{2i+1}-n} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,1} + c_{j+2^{2i}+2^{2i+1}-n} \boxtimes \tau_{-1}\mathsf{Bsk}^{-}_{2i,2} + c_j \boxtimes \mathsf{Bsk}^{+}_{2i,3}$
  end for
  for $j = 0$ to $n-1$ do
    $\hat{c}_j = \mathsf{iFFT}(c_j)$
  end for
  $(c_0, ..., c_{n-1}) \leftarrow (\hat{c}_0, ..., \hat{c}_{n-1})$
end for
return $(c_0, ..., c_{n-1})$

Algorithm 8 External Product Tree (EPT).

Input: β RGSW ciphertexts $C_k = \mathsf{RGSW}(X^{\widetilde{m}_k})$, for $k \in [0, \beta-1]$.
$p^{\beta-1}$ polynomials testP_i that encode the LUT $f : [0, p_{\mathsf{in}}-1]^{\beta} \to [0, p_{\mathsf{out}}-1]$, where $t \in [0, p^{\beta}-1]$, where $\mathsf{testP}_i = -\sum_{j=0}^{N-1} \left\lfloor \frac{Q}{p_{\mathsf{out}}} \right\rceil \cdot f\left(t, \left\lfloor \frac{jp}{2N} \right\rceil\right) \cdot X^{N-\frac{N}{p}-j-1}$

Output: An RLWE ciphertext, where its zeroth term corresponds to $f(m_0, ..., m_{\beta-1})$.

for $k = 0$ to $\beta-1$ do
  for $i = 0$ to $p^{\beta-k-1}-1$ do
    $c_{k,i} = \mathsf{testP}_i \boxdot C_k$,
  end for
  for $i = 1$ to $p^{\beta-k-2}$ do
    $\mathsf{testP}_{i-1} = \mathsf{RLWE.Repack}(c_{k,(i-1)p}, ..., c_{k,ip-1}) \cdot \sum_{i=0}^{2N/p-1} -X^i$
  end for
end for
return testP_0.

ciphertexts. After that, it uses HomoTrace to extract the ciphertext associated with B_i. This step can generate the gadget RLWE ciphertext. Finally, the scheme switching is used to embed the secret key. The noise in these steps is additive, so we have

$$\begin{aligned}\mathsf{V}_{\mathsf{CBoot}} &< \frac{N}{2}((h+k)\cdot \mathsf{V}_{\mathsf{EMp}} + \mathsf{V}_{\mathsf{HT}}) + \mathsf{V}_{\mathsf{RKS}} \\ &< N(h+k)\ell \cdot \mathsf{V}_{\mathsf{CMux}} + \frac{N \cdot \log_2 \overline{d}}{2} \cdot \mathsf{V}_{\mathsf{Auto}} + \mathsf{V}_{\mathsf{RKS}}.\end{aligned}$$

Lemma C.2. Input β RGSW ciphertext $C_k \in \mathrm{RGSW}(X^{\widetilde{m}_i})$ with the noise variance V_{in}, and the automorphism key Auk, Alg. 8 outputs the ciphertext of $f(m_0, ..., m_{\beta-1})$, where the variance satisfies

$$\mathsf{V}_{\mathsf{EPT}} \le \beta \cdot \mathsf{V}_{\mathsf{EP}} \cdot V_{\mathsf{in}} + (\beta-1) \cdot \mathsf{V}_{\mathsf{Pack}}.$$

Proof of Lemma C.2. The correctness of Alg. 8 can be directly obtained from the correctness of the external product and functional bootstrapping. It is worth noting that this algorithm introduces a multiplicative noise growth. By carrying the variance of the CBS (lemma C.1) into the variance of the initial external product, we can obtain the final error variance batched circuit bootstrapping as follows $\mathsf{V}_{\mathsf{out}} < \beta \cdot \mathsf{V}_{\mathsf{EP}} \cdot (N(h+k)\ell \cdot \mathsf{V}_{\mathsf{CMux}} + \frac{N \cdot \log_2 \overline{d}}{2} \cdot \mathsf{V}_{\mathsf{Auto}} + \mathsf{V}_{\mathsf{RKS}}) + (\beta-1) \cdot \mathsf{V}_{\mathsf{Pack}}$. For a more detailed noise analysis, please refer to [40].

C.3 Security and standard deviations

In this subsection, we show the standard deviations of the initial noise corresponding to various bootstrapping keys and ciphertexts in Tab. 9. All parameter settings satisfy the 128-bit security level through the lattice estimator.

Table 9: Security and standard deviations for FHE parameters using the lattice estimator.

Sets	h	N	Q	Std	λ
Boot_2	39	2048	2^{64}	2^{-15}	>128
Boot_2	512	2048	2^{64}	2^{-53}	>128
Boot_4	42	2048	2^{64}	2^{-17}	>128
Boot_4	512	2048	2^{64}	2^{-53}	>128
Boot_6	33	4096	2^{64}	2^{-21}	>128
Boot_6	512	2048	2^{64}	2^{-53}	>128
Boot_8	34	4096	2^{64}	2^{-24}	>128
Boot_8	512	4096	2^{64}	2^{-56}	>128

C.4 Failure Probability

Given a ciphertext with Gaussian-distributed noise e with standard deviation Std, we can compute the probability of decryption failure. Notice that we are leaving one bit of padding for the message to avoid the negacyclicity as done in [33], i.e., $m_i \in \mathbb{Z}_p$ and $\triangle = q/2^{p+1}$. Therefore, the message $m_i \in \mathbb{Z}_p$ is correctly bootstrapped with probability $\mathsf{Pr}[e < q/2^{p+2}] = \mathsf{erf}\left(\frac{q/2^{p+2}}{\sqrt{2}\cdot \mathsf{Std}_\mathsf{M}}\right)$, where Std_M denotes the maximum standard deviation of noise throughout the evaluation process. For example, in Alg. 3, the maximum standard deviation satisfies $\mathsf{Std}_\mathsf{M} = (\mathsf{V}_{\mathsf{Boot}} + \mathsf{V}_{\mathsf{MS}})^{\frac{1}{2}}$. Finally, we present more detailed results for decryption failure rates in Tab. 4.

D Parameter Settings and Performance Results

D.1 FHE Parameter Settings

we present the parameters of BATCHBOOT in Tab. 10 and BATCHCBOOT in Tab. 11, respectively.

Table 10: Parameters of BATCHBOOT of Alg. 3 where the set $\mathsf{PSI.Boot}_6$ is used in PSI.

Para. Sets	p	n	h	N	B_{Bsk}	d_{Bsk}	B_{Auk}	d_{Auk}	B_{Rlk}	d_{Rlk}
Boot_2	2^2	2^{11}	39	2^{11}	2^{23}	1	2^{23}	1	2	12
Boot_4	2^4	2^{11}	42	2^{11}	2^{23}	1	2^{23}	1	2	14
Boot_6	2^6	2^{12}	33	2^{12}	2^{23}	1	2^{23}	1	2	17
Boot_8	2^8	2^{12}	34	2^{13}	2^{23}	1	2^{23}	1	2	17
$\mathsf{PSI.Boot}_6$	2^6	2^{11}	39	2^{11}	2^{15}	2	2^{15}	2	2	12

Table 11: Parameters of BATCHCBOOT of Alg. 4.

Sets	p	n	h	N	B_{Bsk}	d_{Bsk}	B_{Auk}	d_{Auk}	B_{Rvk}	d_{Rvk}	$\overline{B}$	$\overline{d}$
CBoot_4	2^4	2^{11}	39	2^{11}	2^{15}	2	2^{15}	2	2^{15}	2	2^4	4

D.2 Performance Results

This section shows more detailed implementation results. We first present the performance results for multithreaded execution in Tab. 12. Our experimental findings indicate that our algorithm directly benefits from an increased number of threads. Finally, we present the peak memory usage in Tab. 13.

Table 12: Multi-threaded execution result for Batchboot, where T represents the number of threads.

Params.	Packed Mess.	$T=1$	$T=2$	$T=4$	$T=8$	$T=16$
Boot_8	4096	102.8 s	55.3 s	29.2 s	19 s	14.7 s

Table 13: Peak memory usage for the evaluation of LUTs in the single-threaded setting, where the parameters use the fully packing parameters corresponding to Tab. 4.

	2-bit LUT	4-bit LUT	6-bit LUT	8-bit LUT
Peak RAM	360 MB	360 MB	1425 MB	3743 MB