

FHE for ALU over Large Prime Moduli and Application to One-Round Threshold ECDSA

Yuchen Wei¹, Kaisheng Ma¹(✉), Mingyu Gao^{1,2}, and Hongren Zheng¹

¹ Tsinghua University

² Shanghai Qi Zhi Institute

{weiyu22,zhenghr22}@mails.tsinghua.edu.cn

{kaisheng,gaomy}@tsinghua.edu.cn

Abstract. Fully Homomorphic Encryption (FHE) has served as a theoretical building block for cryptographic primitives, but concrete instantiations remain limited by the cost of generic homomorphic computation. One example is the universal thresholdizer that compiles any (deterministic) signature scheme into a multi-party threshold signature scheme with a one-round signing protocol (Boneh et al., Crypto’18), where the signing algorithm is homomorphically evaluated without communication with other parties. Practically, however, the signing algorithm itself often involves different types of computation. For example, ECDSA requires a hash function, point arithmetic over the high-precision defining field $\mathbb{F}_p$, and base-point group arithmetic over $\mathbb{F}_u$. Generic schemes like DM/CGGI take days to complete the homomorphic evaluation. Recent schemes like Boneh–Kim (Crypto’25), Cha–Park–Lee (Eurocrypt’26) and Gao–Zheng (Crypto’26) either support only one computation type, or lack efficient conversion methods.

We propose a new FHE scheme whose arithmetic logic unit supports both large prime modulus arithmetic and bit-level operations. We then specialize it to the $\mathbb{F}_p$ mode, the $\mathbb{F}_u$ mode and the Boolean mode for ECDSA. Our key component is a homomorphic canonicalization operation that restores the unique bit representation of the encrypted messages. It requires a constant number of CKKS bootstrapping operations regardless of the bit-width of the prime moduli, and it therefore enables efficient high-precision conversions like $\mathbb{F}_p$ -to- $\mathbb{F}_u$ modular reduction and $\mathbb{F}_u$ -to-Boolean conversion. The conversion to Boolean mode is essential to allow noise flooding in the threshold decryption process.

For the workload of $\mathbb{F}_u$ modular multiplication followed by conversion to bit representation in GPU, we estimate a $14.5\times$ and $\sim 19000\times$ better amortized latency compared with Montgomery-based approach in Cha–Park–Lee and TFHE-rs based approach, respectively. Our end-to-end homomorphic ECDSA signing experiment takes 56s on an NVIDIA RTX Pro 6000 GPU. To the best of our knowledge, it is the first concrete FHE-based construction towards threshold ECDSA with one-round signing.

1 Introduction

Since Gentry’s groundbreaking bootstrapping technique [Gen09], Fully Homomorphic Encryption (FHE) has been utilized for both theoretical primitives and practical implementations. From a theoretical perspective, FHE has been used to construct various cryptographic primitives like indistinguishability obfuscation, functional encryption, round-efficient multi-party computation, the universal thresholdizer and the universal blinder [AJL⁺12, GGH⁺13, BGG⁺18, BK26]. On the other hand, their concrete instantiations could be prohibitively expensive when using FHE in a generic and black-box way, and their designs could be constrained by the underlying FHE schemes, such as the supported operation types and the available plaintext moduli.

Practically, cryptographic computation often requires a mixture of drastically different computation types. In elliptic curve cryptography, both high-precision modular arithmetic and bit-level logic operations are simultaneously needed. For example, the elliptic curve secp256k1 has a defining field $\mathbb{F}_p$, and its base point also forms a (cyclic) group $\mathbb{F}_u$, with both p and u being large 256-bit primes. At the same time, the bit representation of a scalar is required for fast “exponentiation” on the elliptic curve.

In this work, we instantiate the universal thresholdizer for the ECDSA signature scheme [BGG⁺18, EY24]. A universal thresholdizer uses threshold FHE to compile any deterministic signature scheme into a threshold version with a one-round signing protocol. The high-level idea is that each party uses a homomorphically encrypted signing key to compute the signature, and the threshold decryption process takes only *one round* of communication. The threshold version of ECDSA signature is of particular interest in scenarios where the trust should be distributed across multiple parties, such as blockchain. There has been a long line of research, and existing constructions take at least two rounds of communication (including preprocessing rounds) [Lin17, LN18, CGG⁺20, LLZD25, NP26]. We acknowledge that the FHE-based solution cannot directly compete with them in all aspects, as they have much lower computation and storage costs. Nevertheless, we think it is still interesting to concretely report the performance of FHE-based one-round threshold ECDSA.

The key challenge in achieving one-round threshold ECDSA using FHE is the homomorphic execution of the signing algorithm. The signing algorithm has the following steps that are typically hard to execute in FHE: (i) large-characteristic finite field arithmetic in both $\mathbb{F}_p$ and $\mathbb{F}_u$, (ii) modular reduction for a scalar from $\mathbb{F}_p$ to $\mathbb{F}_u$, (iii) hash functions for generating the nonce, and (iv) conversion to a bit representation for a scalar in $\mathbb{F}_u$ to allow noise flooding in threshold decryption. The situation therefore motivates the following question:

Is there an FHE scheme that natively supports both large prime moduli arithmetic and bit-level operations, and efficient conversions between them?

Previous works such as [BK25, CPL26] have explored homomorphic computation of *multiplication* in P-384 and Curve25519. However, they either lack modular reduction to another modulus and to a bit representation, or their

Table 1. Capability comparison of different FHE schemes for large prime moduli. n is the bit-width of the prime moduli. “A→B Conversion” refers to the cost of enabling Boolean operations after multiplication. “PBS” refers to DM/CGGI-style programmable bootstrapping, and “BTS” refers to CKKS bootstrapping. While the two bootstrapping types are not directly comparable and “ $O(1)$ BTS” does not necessarily dominate “ $O(n)$ PBS”, CKKS-style BTS enables SIMD and could have better amortized latency.

Scheme	Mult (Large Prime)	Boolean	A→B Conversion	Type
DM/CGGI	$O(n^2)$ PBS	$O(n)$ PBS	N/A	Scalar
REFHE [BFG ⁺ 26]	Montgomery	$O(n)$ PBS	$O(n)$ PBS	Scalar
[CPL26]	Montgomery	$O(1)$ BTS	$O(\log(n/4))$ BTS	SIMD
[GZ26]	Montgomery	$O(1)$ BTS	Amort. $O(1)$ BTS	SIMD
dBFV [PZZ26]	Native	$O(n)$ PBS	$O(n)$ PBS	Per Scalar
Ours	Native	$O(1)$ BTS	$O(1)$ BTS [†]	SIMD

[†] We require additional $\log(n)$ CKKS level consumption.

Montgomery-based simulation of arithmetic modulo an arbitrary modulus is expensive because of the reduction process after each multiplication. dBFV [PZZ26] natively supports arithmetic over an arbitrary modulus through a relaxed digit representation, and uses a modulo-by-lattice operation to refresh the relaxed representation. To recover the exact representation, its bootstrapping count remains linear in the bit width.

We additionally remark that end-to-end homomorphic signing exercises several capabilities within a single application, such as a large number of Boolean operations and sequential high-precision modular multiplications. It therefore provides a comprehensive benchmark, consistent with the call for benchmarks by the FHE benchmarking project.³

1.1 Our Contributions

A homomorphic canonicalization operation with $O(1)$ CKKS bootstrapping. We use the concept of “homomorphic canonicalization operation” to describe the common pattern in homomorphic computation-type conversion. To support certain computation types, messages are often homomorphically encrypted in some “convenient” format, usually with redundancies. However, to support other computation types, the messages usually need to be restored to their exact representations through a “canonicalization” operation. Intuitively, the cost of the canonicalization operation is related to the bit-width of the message precision.

Our major contribution is a homomorphic canonicalization operation that uses only a *constant* number of CKKS bootstrapping operations and additionally $\log(n)$ CKKS levels. It supports converting messages encoded for either large prime moduli or power-of-two moduli to their bit representations. More concretely, its cost is 3 CKKS bootstrapping operations for power-of-two moduli and 5 CKKS bootstrapping operations for large prime moduli.

³ <https://fhe-benchmarking.github.io/>

Additionally, we use the canonicalization operation to achieve a modular reduction operation with $O(1)$ CKKS bootstrapping operations from one modulus to another modulus. We can also use it to achieve a homomorphic ciphertext-plaintext division algorithm for a prime divisor.

A new FHE scheme supporting large prime modulus ALU, and showcased for ECDSA. We extend the triangle encoding in Gao–Zheng [GZ26] to support arithmetic over large prime moduli by proving that there exists small-norm polynomial whose evaluation at 2 gives the prime modulus. We then use the homomorphic canonicalization operation to instantiate its arithmetic-to-Boolean conversion, obtaining a large prime modulus arithmetic logic unit (ALU) whose refreshing and conversion operations both take only $O(1)$ CKKS bootstrapping operations.

As an example, we instantiate the extended Gao–Zheng scheme for the primes p and u in the elliptic curve secp256k1 and show its application to ECDSA. We also show its applicability to Curve25519.

GPU implementation of end-to-end homomorphic ECDSA signing. We built a GPU implementation⁴ using OpenFHE [BAB⁺22] and FIDESlib [ADOVLG⁺25], and report the end-to-end computation time of generating one signature to be 56 seconds. Our homomorphic computation can be combined with existing techniques to enable *one-round threshold* ECDSA [MHP⁺25, PPS26].

1.2 Technical Overview

The general idea of supporting high-precision integer arithmetic in FHE is to decompose the integers into smaller units and use algebraic constraints to preserve the target arithmetic. More concretely, a large integer is either decomposed into digits [CPL26, PZZ26], factored into smaller RNS limbs [BK25], or “flattened” into a polynomial with small coefficients [CLPX18, GV25, CHM⁺25, BFG⁺26, GZ26].

The actual difficulty, however, lies in the treatment of additional “overflow” and “noise” terms introduced during the homomorphic computation, which is tightly coupled with the smaller units. For example, the multiplication of two 64-bit integers produces a 128-bit integer. While only the lower 64 bits might be needed for later computation, the higher 64 bits are not automatically removed and are also encoded somewhere. Typically, we can either *refresh* them to a small enough size to enable further computation of the same type, or *remove* them to obtain the bit representation for logic operations and conversion to other encodings. We denote the first operation as the *homomorphic refreshing operation* and the second operation as the *homomorphic canonicalization operation*.

More formally, we study the isomorphism $\mathbb{Z}_q \cong \mathbb{Z}[X]/\langle F_q(X), X - 2 \rangle$ where $F_q(2) = q$. Let $[m]_t = \sum_{i=0}^{n-1} b_i X^i$ be the “flattened” polynomial for a scalar $m \in \mathbb{Z}_q$, with b_i being the i -th bit of m and n being the bit-width of q . After homomorphic encryption, a ciphertext encrypts an expression of the form $[m]_t +$

⁴ <https://github.com/RyanWei0224/FHE-ALU-ECDSA>

$(X - 2)I + \alpha F_q(X)$, with I and α being integer polynomials. The refreshing operation resets I and α to a smaller size so that later computation can continue with small noise growth. The canonicalization operation removes both I and α so that we obtain the bit representation.

The cost of both operations appears to be related to the bit-width of the high-precision plaintext modulus q . Intuitively, this is because the size of both I and α is related to the bit-width of q . Indeed, the Cha–Park–Lee scheme [CPL26] requires *iterative* refreshing after arithmetic operations; after the refreshing operation, its canonicalization operation LCtoC takes only a constant number of bootstrapping operations through the carry-lookahead technique. dBfV [PZZ26] can execute a modulo-by-lattice operation to restore a relaxed digit representation and enable further arithmetic operations, but it still requires $O(n)$ DM/CGGI-style PBS to restore the exact representation. Gao–Zheng [GZ26] can use a “truncate” operation (A2A- I) to reset the overflow terms when the plaintext modulus can form a “triangle encoding”, but its canonicalization cost can only be amortized to $O(1)$ through a large batch of ciphertexts. Note that the constant bootstrapping cost can be achieved for either operation, but not for both.

The question is therefore whether we can achieve constant bootstrapping cost for both operations, and our answer is affirmative for the Gao–Zheng scheme. Technically, an idea that naturally arises is combining both the “truncate” operation and LCtoC to obtain an $O(1)$ CKKS bootstrapping count for the canonicalization operation. For $q = 2^n$ with $F_q(X) = X^n - X + 2$, this is indeed the case: we can merge the treatment of α into that of I using $X^n \equiv X - 2$, and the removal of I can be regarded as a carry-propagation problem (i.e., replacing the multiple of 2 in lower bits with $1 \cdot X$ in higher bits). Note that the “truncate” operation (A2A- I) can reset the overflow term I to be small, and LCtoC can propagate carries in one bootstrapping when the base B of its digit vector is properly chosen.⁵

For general prime moduli q , however, we need to treat α and I separately. When we view the coefficients of the polynomial $[m]_t + I(X - 2) + \alpha F_q(X)$ as a digit vector, the removal of I is a carry-propagation to higher bits, whereas the removal of α is a *wrap-around* operation that moves higher bits into lower bits (i.e., replacing $A \cdot 2^n$ with the bit string of $A \cdot 2^n \pmod{q}$). Our key observation is that the efficient wrap-around can be implemented as a LUT function that outputs different bits of the multiplication result, because A is bounded.

We now explain the technical steps of our homomorphic canonicalization operation for general prime moduli q ; the overview is shown in Fig. 1:

- We apply A2A- I to refresh $[m]_t + I(X - 2) + \alpha F_q(X)$ with smaller I and α .
- To eliminate I , we use one *multi-value functional bootstrapping* to extract all bits of $[m]_t + I(X - 2) + \alpha F_q(X)$, and get $O(\log K)$ bit strings. K is the constant bound on $|I|$. We then contract the bit strings into a digit vector with entries in $\{0, 1, 2\}$, because we can compress three bit strings into two with a full-adder-like circuit using two CKKS levels. We then convert the $\{0, 1, 2\}$ digits into a single bit string $\mathbf{b}$ using LCtoC in Cha–Park–Lee [CPL26].

⁵ This is one way of explaining the concurrent works [SLWL26, LMWW26].

$$\begin{array}{ccccccc} \text{Input} & \xrightarrow{\text{A2A-}I} & tI + [m]_t & \xrightarrow{\text{MV-LUT}} & O(\log K) \text{ bit str} & \xrightarrow{\text{Contraction}} & \text{Small alphabet} \xrightarrow{\text{LCtoC}} \mathbf{b} \\ & & \text{collected overflow } A & \xrightarrow{\text{SP-LUT}} & \mathbf{d}^{(0)} & \xrightarrow{\text{LCtoC}} & [m]_t \end{array}$$

Fig. 1. The overview of the homomorphic canonicalization operation A2B_{CLA} . Here we do not draw additional bit strings like $\mathbf{d}^{(1)}$ for ease of understanding.

- To handle the wrap-around, we need to collect the additional bits A above 2^n during the above carry-propagation process, and “multiply” them by the constant 2^n modulo q to obtain another bit string $\mathbf{d}^{(0)}$. This multiplication can be carried out through a functional bootstrapping whose i -th output slot is the i -th bit of $A_0 := [A \cdot 2^n]_q$; we call it *slot-parallel* functional bootstrapping. Ignoring the redundant representation problem, we can directly apply LCtoC on $(\mathbf{d}^{(0)} + \mathbf{b})$ to produce $[m]_t$. To resolve that problem, we employ additional LUTs that can be fused into previous steps with moderate overhead.

An FHE scheme for ALU over large prime moduli. We show that for any large prime q , we can find a polynomial F_q with distinct complex roots, and we can use it to construct the triangle encoding as

$$\mathbb{C}^{n/2} \text{ or } \mathbb{C}^{(n-1)/2} \times \mathbb{R} \longleftarrow \mathbb{R}[X]/\langle F_q(X) \rangle \xleftarrow{\text{Trig}} \mathbb{Z}[X]/\langle F_q(X), X-2 \rangle$$

The exact complex embedding depends on the parity of n . This extends the triangle encoding in Gao–Zheng [GZ26] to arbitrary prime moduli. We use the aforementioned homomorphic canonicalization operation as its arithmetic-to-Boolean operation, and therefore we obtain an FHE scheme for prime modulus ALU with $O(1)$ bootstrapping.

We specialize the generalized triangle encoding to the elliptic curve secp256k1 with defining field $\mathbb{F}_p$ and the group $\mathbb{F}_u$ of its base point, with $p = 2^{256} - 2^{32} - 977$. Their exact polynomials can be found in Section 5.1. The procedure is as follows: we apply A2B_{CLA} to obtain a bit string, and apply B2A to get the triangle encoding for $\mathbb{F}_u$. Such modular reduction process actually works for any prime modulus p and u , and it implies homomorphic ciphertext-plaintext division algorithm.

Using the FHE scheme above, we implemented the ECDSA signing process with standard ECC techniques such as homogeneous projective coordinates and the modular inverse chain [Det14, Smi17]. We use SHA3 as the hash function because the required Boolean operations are more easily implemented using discrete CKKS. After the signature generation, we convert the result to a bit representation to allow noise flooding, in a way similar to THED [PPS26].

1.3 Concurrent Works

Two *concurrent* works also have arithmetic-to-Boolean conversion with $O(1)$ bootstrapping and $O(\log n)$ additional level consumption for the Gao–Zheng scheme. Yet, the technical details are *substantially different* from our work.

The most significant difference is our formulation of the problem. While we formulate the problem as a *canonicalization* problem from $[m]_t + I(X-2) + \alpha F_q(X)$ to $[m]_t$, a viewpoint that works for any modulus, they instead recover some coefficients of I from the *triangle encoding* for power-of-two moduli.

In the triangle encoding, the i -th coefficient c_i can be understood as $c_i = I_i - \overline{0.b_i b_{i-1} \cdots b_0}$ when c_0 is processed in a special way. Shen, Li, Wang and Lu [SLWL26] observed that by a proper choice of stride and scaling, the difference of two coefficients (e.g., c_i and c_{i+5}) in the triangle encoding gives $c_i - 32c_{i+5} = I_i - 32I_{i+5} + \overline{b_{i+5} \cdots b_{i+1}}$. Because the range of I_i is $[-15, 15]$ for each i after truncation, we can rewrite this as $(I_i + \overline{b_{i+5} \cdots b_{i+1}}) - 32I_{i+5}$. By a modulo-by-32 operation, they can *roughly* separate the quotient I_{i+5} and the residue, with at most *one* bit difference caused by the potential carry. The carry is then recovered through LToC in [CPL26]. On the other hand, their method in the current form could not be applied to arbitrary moduli. When counting the A2A- I preprocessing, the bootstrapping count is the same as ours.

Liu, Ma, Wang and Wang [LMWW26] made a similar observation and generalized the set of carry statuses to $\mathcal{C}$. The main idea is to enumerate all possible carry statuses and use indicators to select the only correct guess. Their method does not rely on the triangle encoding for $\mathbb{Z}_{2^n}$ and can be extended to arbitrary moduli through either a general triangle encoding or Montgomery form. For $\mathbb{Z}_{2^n}$, their bootstrapping count is also the same as ours when counting the A2A- I preprocessing. Their bootstrapping count remains 3. for prime moduli supported by triangle encoding. However, the final functional bootstrapping would produce hundreds of indicator ciphertexts. For example, according to their Appendix C.2, when using $|A| = 33$, $\kappa = 0$, $|\mathcal{C}| = 2$ and $n = 256$, they require 198 indicator ciphertexts, and each intermediate carry composition *level* consumes 264 ct-ct multiplications and 132 CKKS rotations. There are $\lceil \log \lceil n/5 \rceil \rceil = 6$ intermediate levels. A key-switching takes roughly 0.1s in our single-threaded CPU setting, and we therefore estimate that the intermediate indicator combination process alone takes roughly 237s.

Compared with [SLWL26], our solution uses a LUT with a larger width ($w = 7$) and our CKKS multiplicative depth is $L = 29$, while they can exploit a LUT of slightly smaller bit width ($w = 6$) and obtain a slightly smaller CKKS multiplicative depth ($L = 26$ for $n = 64$ and $L = 28$ for $n = 256$). On the other hand, their method can only handle the case of $\mathbb{Z}_{2^n}$.

Compared with [LMWW26], we would require a bootstrapping count of 5 instead of 3 to support the general triangle encoding, but our running time is around 70s in Table 4. For the case of $\mathbb{Z}_{2^n}$, our performance is also comparable. Our method is also conceptually much simpler.

Jang and Lee [JL26] discussed the support for large-characteristic extension fields through bivariate polynomials. They do not discuss the conversion to Boolean mode and are therefore orthogonal to our work.

1.4 Related Works

Small Characteristic Finite-Field Extensions to Gao-Zheng FHE Scheme. Seuré and Suvanto [SS26b] and Kim et al. [KLN⁺26] extended the homomorphism chain in Gao-Zheng to support finite-field arithmetic in the form of $\mathbb{F}_{a^b}$, with the idea that $\mathbb{F}_{a^b} \cong \mathbb{F}_a[X]/\langle F(X) \rangle$ for some appropriate polynomial $F(X)$. For noise cleaning and arithmetic-to-Boolean conversion, they rely on the $\mathbb{F}_a$ -module isomorphism $\mathbb{F}_{a^b} \cong \mathbb{F}_a^b$ to embed entries in $\mathbb{F}_a$ into the complex slots and use discrete CKKS for noise cleaning; this approach is thereby only applicable to a relatively small characteristic a . Our work instead focuses on finite-field arithmetic $\mathbb{F}_p$ for large prime p , and we can support arithmetic-to-Boolean conversion.

Other Arithmetic-to-Boolean Conversions. Kim [Kim26] focused on performing non-arithmetic operations on the BFV and GBFV [GV25, CHM⁺25] schemes from a discrete CKKS perspective. They start from a coefficient-encoded (G)BFV scheme and reinterpret it as a coefficient-encoded CKKS scheme. If the encoded coefficient is large, they iteratively use MSB bootstrapping [KN25] to decompose it into digits in the format of [Kim25]. They then rely on homomorphic modular reduction from Cha-Park-Lee [CPL26] to reduce the (G)BFV message into its unique representation and perform non-arithmetic operations. This approach has two limitations: first, the available choice of plaintext modulus in (G)BFV is limited due to algebraic constraints, SIMD concerns and noise growth, and the approach does not directly apply to the primes in our elliptic curve application. Second, assuming the bit-width of the (G)BFV plaintext is n , the reduction process also relies on $O(n \log n)$ or $O(\log n)$ CKKS bootstrapping operations, compared with $O(1)$ in our case.

Threshold Signature from Threshold FHE. THED [PPS26] also constructed a threshold signature using threshold FHE. They constructed a two-round signing protocol for threshold Dilithium from CKKS. With some preprocessing, its online phase is a one-round signing protocol. Their total CKKS bootstrapping count is 13 and their scheme performs decently on GPU. As we focus on different applications, the performance is not directly comparable. Our main goal is to achieve a one-round signing protocol, demonstrate the versatility of the encoding framework, and serve as a potential benchmark.

Homomorphic SHA3. Seuré and Suvanto [SS26a] evaluate the Keccak permutation by exploiting the free-XOR property in XBOOT [NHY⁺25]. They enjoy a smaller parameter of $N = 2^{15}$ and they require one XBOOT for each round. We instead compute XOR in discrete CKKS due to our sufficient level budget and larger parameter $N = 2^{16}$ required by later workloads. We consume 16 B2B for 24 rounds.

Homomorphic ECDSA. There is also an ECDSA signing instance from TFHE.⁶ It covers the point arithmetic part $R = [k]G$ and the signature generation

⁶ <https://github.com/zama-ai/bounty-ecdsa-signature>

$s = k^{-1}(\mathcal{H}(\mathcal{M}) + r \cdot \text{sk})$, requiring the user to provide an encrypted nonce k . The running time for generating one signature on secp256k1 is approximately one or two days on a 64-core CPU. Compared with them, our estimated CPU running time is 2.5 hours.

2 Preliminaries

Notation. For a positive integer m , we denote by $[n]_m$ the representative of the integer n modulo m in the range $[0, m)$. We deliberately use $[m]_t$ with $t = X - 2$ to denote its flattened polynomial [GV25]. We use bold font like $\mathbf{v}$ to denote a vector of elements, and we use v_i to denote the i -th element of vector $\mathbf{v}$. For a polynomial $f(X)$, we use $\mathbf{f}$ to denote its coefficient vector, and f_i will be the coefficient of X^i in $f(X)$. When we define a polynomial $f(X) \in \mathbb{Z}[X]/\langle g(X) \rangle$, we use the representation of f with minimal degree by default. We use $\llbracket m \rrbracket$ for a message m to represent the homomorphic encryption of m .

2.1 RLWE, CKKS, and Discrete CKKS

RLWE. The Ring Learning-With-Errors (Ring-LWE) problem is a hard problem formulated in [LPR10]. We use the ring $\mathcal{R}_Q = \mathbb{Z}[X]/\langle X^N + 1, Q \rangle$ to instantiate the RLWE problem, where N is a power of two and Q is a large integer [BCC⁺24]. The RLWE problem allows the encryption of a polynomial $m \in \mathcal{R}_Q$ as follows

$$(-as + m + e, a) \in \mathcal{R}_Q^2$$

where a, e, s are polynomials in $\mathcal{R}_Q$. The decryption is an inner product with $(1, s)$ and results in $m + e \in \mathcal{R}_Q$; how to extract m from the noise is scheme-specific.

CKKS. The CKKS scheme [CKKS17] further connects complex vectors to polynomials using the ring isomorphism from $\mathbb{C}^{N/2}$ to $\mathbb{R}[X]/\langle X^N + 1 \rangle$. By properly scaling the complex vector using a scaling factor Δ , CKKS can approximately support complex arithmetic.

We use the term “slot” for an entry in the complex vector, and this encoding is called slot encoding. In the bootstrapping process [CHK⁺18], the complex numbers need to move between the slots and the polynomial coefficients, using the linear transformations `CoeffsToSlots` and `SlotsToCoeffs`. The modulus raising operation `ModRaise` and the homomorphic sine operation `EvalSine` introduce and remove the extra $q_0 I$ in the coefficient encoding, respectively. Within this process, a technique called sparse key encapsulation can be used to reduce the size of $|I|$.

Lemma 1 ([BTH22]). *After the `ModRaise` operation, the new overflow I satisfies $\|I\|_\infty \leq K$ except with probability p_{fail} .*

By using the sparse-key encapsulation technique, where the encapsulated sparse key has Hamming weight $\hat{h} = 32$, we have $p_{\text{fail}} \approx 2^{-85}$ for $K = 15$ and p_{fail} is negligible for $K = 16$.

Discrete CKKS. More recently, discrete CKKS [DMPS24, BKSS24, AKP25, AKPZ26] focuses on encoding discrete numbers into the complex slots. It can encode a vector of discrete messages $m_i \in \mathbb{Z}_p$, with p being a power of two 2^w .

The key feature of discrete CKKS is the support for functional bootstrapping with an arbitrary look-up table.

Definition 1 (CKKS Functional Bootstrapping). For a vector of discrete messages $\mathbf{m} \in \mathbb{Z}_p^{N/2}$, the discrete CKKS functional bootstrapping (FBT) evaluates an arbitrary look-up table $f : \mathbb{Z}_p \rightarrow \mathbb{Z}_p$

$$\text{FBT}_f \left(\Delta \frac{\mathbf{m}}{p} + \mathbf{e} \right) = \Delta \frac{f(\mathbf{m})}{p} + \mathbf{e}'$$

with $\|\mathbf{e}'\|$ generally smaller than $\|\mathbf{e}\|$. For convenience, we use the notation $f(\mathbf{m}) = (f(m_0), f(m_1), \dots, f(m_{N/2-1}))$.

It is observed that certain ciphertexts in the functional bootstrapping process can be reused to efficiently support *multi-value* functional bootstrapping.

Definition 2 (CKKS Multi-Value Functional Bootstrapping). For a vector of discrete messages $\mathbf{m} \in \mathbb{Z}_p^{N/2}$, for J look-up table functions $f^{(j)} : \mathbb{Z}_p \rightarrow \mathbb{Z}_p$, the discrete CKKS multi-value functional bootstrapping supports

$$\text{MV-FBT}_{\{f^{(j)}\}_{j \in [J]}} \left(\Delta \frac{\mathbf{m}}{p} + \mathbf{e} \right) = \left[\Delta \frac{f^{(j)}(\mathbf{m})}{p} + \mathbf{e}^{(j)} \right]_{j=0}^{J-1}$$

Another observation is that different CKKS slots are independent and can evaluate different LUTs in one functional bootstrapping [Rov26]. We call this *slot-parallel* functional bootstrapping and it is defined as follows

Definition 3 (CKKS Slot-Parallel Functional Bootstrapping). For a vector of discrete messages $\mathbf{m} \in \mathbb{Z}_p^{N/2}$, for $N/2$ look-up table functions $f^{(j)} : \mathbb{Z}_p \rightarrow \mathbb{Z}_p$, the discrete CKKS slot-parallel functional bootstrapping supports

$$\text{SP-FBT}_{\{f^{(j)}\}_{j \in [N/2]}} \left(\Delta \frac{\mathbf{m}}{p} + \mathbf{e} \right) = \Delta \frac{(f^{(0)}(m_0), \dots, f^{(N/2-1)}(m_{N/2-1}))}{p} + \mathbf{e}'$$

with $\|\mathbf{e}'\|$ generally smaller than $\|\mathbf{e}\|$.

As they exploit different forms of parallelism in the discrete CKKS functional bootstrapping process, we can combine them to support *slot-parallel multi-value functional bootstrapping*. The definition of SP-MV-FBT is straightforward.

2.2 The Gao–Zheng Scheme

The Gao–Zheng scheme supports both arithmetic and Boolean operations for a vector of n -bit integers [GZ26]. The key idea of the GZ scheme is to use multiple complex slots to represent a single n -bit integer:

$$\mathbb{C}^{n/2} \cong \mathbb{R}[X]/\langle X^n - X + 2 \rangle \xleftarrow{\text{Trig}} \mathbb{Z}[X]/\langle X^n - X + 2, X - 2 \rangle \cong \mathbb{Z}_{2^n}$$

In this relation, the arrow in the middle stands for the triangle encoding, which will be defined shortly. In the following text, we will use the notation $\mathcal{Z} = \mathbb{Z}[X]/\langle X^n - X + 2 \rangle$ and $t = X - 2$. We will also use $\mathcal{Z}_{\mathbb{R}} = \mathbb{R}[X]/\langle X^n - X + 2 \rangle$ and $\mathcal{Z}_t = \mathbb{Z}[X]/\langle X^n - X + 2, X - 2 \rangle$.

For a message m from $\mathbb{Z}_{2^n}$, its *flattened* polynomial in $\mathcal{Z}_t$ is $[m]_t = \sum_{k=0}^{n-1} b_k X^k$, where b_k is the k -th bit of m . Intuitively, the symbol $[\cdot]_t$ can be understood as a modulo-by- t operation in the ring $\mathcal{Z}$.

To embed the flattened polynomial in the ring $\mathcal{Z}_{\mathbb{R}}$ so that the arithmetic properties are preserved, the *triangle encoding* scales $[m]_t$ by a CKKS-style scaling factor Δ and divides it by t , so that it has the form of $\Delta I + \frac{\Delta}{t}[m]_t + e \in \mathcal{Z}_{\mathbb{R}}$. The extra I , a polynomial with integer coefficients, represents the carries and overflows that accumulate during computation. The low-bit e stands for noise caused by encoding, homomorphic encryption and computation. The triangle encoding can decode correctly as long as $\|te\| < \Delta/2$.

The complete picture is as follows using $\mathcal{R}_{\mathbb{R}} = \mathbb{R}[X]/\langle X^N + 1 \rangle$:

$$\mathcal{R}_Q \xleftarrow{\text{Round}} \mathcal{R}_{\mathbb{R}} \cong \mathbb{C}^{N/2} = (\mathbb{C}^{n/2})^{N/n} \cong \mathcal{Z}_{\mathbb{R}}^{N/n} \xleftarrow{\text{Trig}} \mathcal{Z}_t^{N/n}$$

Therefore, the GZ scheme allows one RLWE ciphertext to encrypt a vector of n -bit integers of length N/n . This encoding supports arithmetic operations like addition/subtraction and multiplication, and is therefore called the arithmetic mode of the GZ scheme.

The Boolean mode for the GZ scheme is exactly the discrete CKKS for bits [DMPS24, BCKS24], which supports bitwise operations and bit shifting using CKKS multiplication and rotation.

Arithmetic refreshing. In the triangle encoding, both the overflow I and the noise e must be properly controlled as their sizes generally grow in homomorphic computation. The size of the overflow I is more crucial as it appears in cross terms in homomorphic multiplication. We elaborate on the process of A2A- I .

The whole process of A2A- I is (1) linear transformation $\mathcal{Z}$ -to- $\mathcal{R}$, (2) **Truncate** and **ModRaise**, (3) linear transformation $\mathcal{R}$ -to- $\mathcal{Z}$. Here these linear transformations are similar to **CoeffsToSlots** and **SlotsToCoeffs** in CKKS bootstrapping. After the linear transformation, the ΔI coefficient structure is moved to the ring $\mathcal{R}_Q$, and it can be truncated away by proper scaling. Then we need to execute **ModRaise** to regain the modulus and allow further computation, which introduces another much smaller I' , especially when combined with the sparse key encapsulation technique. We then move the coefficients in $\mathcal{R}$ back to $\mathcal{Z}_{\mathbb{R}}$, so that we maintain the original triangle encoding, with a much smaller overflow I' introduced by **ModRaise**.

3 Arithmetic-to-Boolean Conversion for $\mathbb{Z}_{2^n}$

In this section, we instantiate the homomorphic canonicalization operation by discussing the Arithmetic-to-Boolean conversion in the Gao–Zheng scheme [GZ26]

for power-of-two moduli 2^n . We present our new conversion method A2B_{CLA} for the Gao–Zheng scheme.

3.1 Modulo- t Reduction as Canonicalization of Redundant Binary Digits

Multiplication by t exposes a redundant Radix-2 digit vector. In the Gao–Zheng FHE scheme, an arithmetic-mode ciphertext is encrypted with the triangle encoding in $\mathcal{Z}_{\mathbb{R}} = \mathbb{R}[X]/\langle X^n - X + 2 \rangle$ in the form of $f_m = \Delta I + \frac{\Delta}{t}[m]_t + e$, where I is the polynomial representing overflow, and $[m]_t$ is the flattened polynomial. Rather than exploiting the triangle form of f_m to recover $[m]_t$ iteratively as in [GZ26], we instead work with a form that is multiplied by $t = X - 2$:

$$tf_m = \Delta(tI + [m]_t) + te := \Delta M + e' \in \mathcal{Z}_{\mathbb{R}}$$

tf_m encodes an integer polynomial $M = tI + [m]_t \in \mathcal{Z}$. Notice that $M \equiv [m]_t \pmod{t}$, so the canonicalization process can be viewed as recovering the binary-coefficient representative $[m]_t$ from the congruence class of $M \pmod{t}$.

From now on, when we use a polynomial in a vector context, we are referring to its coefficient vector $\mathbf{M}$.

For a vector $\mathbf{b} = (b_0, \dots, b_{n-1})$, define the radix-2 value map as $\text{val}_2(\mathbf{b}) := \sum_{i=0}^{n-1} b_i 2^i$, which is the integer represented by $\mathbf{b}$ in radix 2. By definition, $\text{val}_2(\mathbf{M}) = M(2)$, so the coefficient vector $\mathbf{M}$ can be regarded as a *redundant* radix-2 representation of $M(2)$. The term *redundant* means that its digits do not necessarily belong to $\{0, 1\}$, and we will use its binary representation as the *canonical form*. Now we show that $M(2)$ is congruent to m modulo 2^n . Hence $[m]_t$ is the last n bits in the binary representation of $M(2)$, which can be retrieved by *canonicalizing* $\mathbf{M}$ into a binary string and truncating higher-order bits. This is formalized in the following proposition.

Proposition 1 (Modulo- t reduction is binary canonicalization). *For $M = tI + [m]_t \in \mathcal{Z}$, we have the relation $[M(2)]_{2^n} = m$. In other words, the binary representation of m lies in the last n bits of the binary representation of $M(2)$, and can be recovered by canonicalizing $\mathbf{M}$ modulo 2^n .*

Proof. Since $M = tI + [m]_t$ in $\mathcal{Z}$, there exists $\alpha \in \mathbb{Z}[X]$ such that

$$M = tI + [m]_t + \alpha \cdot (X^n - X + 2)$$

Evaluating at $X = 2$ gives $M(2) = 0 \cdot I(2) + m + \alpha(2)2^n$, so $M(2) \equiv m \pmod{2^n}$. Since $m \in [0, 2^n)$, we have $[M(2)]_{2^n} = m$.

By Proposition 1, the remaining question is: given a redundant radix-2 digit vector $\mathbf{M}$, how can we efficiently find its binary representation? We do so in two steps: (1) reduce the redundant digits to the alphabet $\{0, 1, 2\}$; (2) use a carry-lookahead circuit to convert these digits into the canonical representation $[m]_t$. The second step can be accomplished using the LCtoC algorithm in [CPL26], and in the following subsection we will describe the first step in detail.

3.2 Reducing Redundant Digits to a Small Alphabet

In the first step, we want to reduce the redundant digit vector $\mathbf{M}$ to a digit vector $\mathbf{b} = (b_0, \dots, b_{n-1})$ with $b_i \in \{0, 1, 2\}$, while preserving its value modulo 2^n : $\text{val}_2(\mathbf{b}) \equiv M(2) \pmod{2^n}$.

We first add a public nonnegative offset vector that moves all digits into a nonnegative range. This allows us to decompose the vector into several binary strings using MV-FBT. We then compress them to two binary strings using a tree of 3-to-2 compressor circuits, and add the two binary strings coefficient-wise to obtain digits in $\{0, 1, 2\}$.

Moving the redundant digits into a nonnegative range. Suppose that the digits of $\mathbf{M}$ lie in $[-H, H]$ for a bound H , and let $\mathbf{h}$ be the binary representation of H . To make the digits nonnegative, we add the offset vector $\mathbf{r}_H := (H, H, \dots, H) + \mathbf{h}$. Note that $\text{val}_2(\mathbf{r}_H) \equiv 0 \pmod{2^n}$. We can obtain a new vector $\mathbf{M}^+ := \mathbf{r}_H + \mathbf{M}$ using homomorphic addition, which has nonnegative digits in the range $[0, 2H+1]$, while preserving $\text{val}_2(\mathbf{M}^+) \equiv \text{val}_2(\mathbf{M}) \pmod{2^n}$.

Concretely, after the preprocessing A2A-I on the input ciphertext, we have $\|I\| \leq 15$ except with failure probability $p_{\text{fail}} \approx 2^{-85}$ by Lemma 1. Then we obtain $\|tI\| \leq 4\|I\| \leq 60$ using [GZ26, Prop 4]. Therefore, we can set $H = 4K + 1 = 61$, and the digits of $\mathbf{M}^+$ lie in $[0, 123]$, which can be represented by $W = 7$ bits. W is a constant regardless of the bit-width n .

Use MV-FBT to extract the binary digits. Since the digits have a bounded range of $[0, 2H + 1]$, we use multi-value functional bootstrapping (MV-FBT) to efficiently reduce them to the small alphabet. We design an MV-FBT with $W := \lceil \log(2H + 2) \rceil$ LUTs to extract all bits of $\mathbf{M}^+$. The j -th LUT is defined as the j -th bit in the binary representation of the input digit x :

$$\text{LUT}_j(x) = \left\lfloor \left\lfloor \frac{x}{2^j} \right\rfloor \right\rfloor_2, \quad 0 \leq j < W$$

After applying the MV-FBT, we obtain W binary strings $\mathbf{b}^{(j)} := \text{LUT}_j(\mathbf{M}^+)$ with $\mathbf{M}^+ = \sum_{j=0}^{W-1} 2^j \mathbf{b}^{(j)}$.

Multiplication by 2^j can be implemented as vector rotation. Indeed, if we denote by $\text{rot}_j(\mathbf{v})$ the vector $\mathbf{v}$ rotated by j steps to the higher digits, we have $\text{val}_2(\text{rot}_j(\mathbf{v})) = \sum_{i=0}^{n-1} v_i 2^{i+j} = 2^j \text{val}_2(\mathbf{v})$. Therefore, we have the following relation:

$$\text{val}_2(\mathbf{M}^+) = \sum_{j=0}^{W-1} 2^j \text{val}_2(\mathbf{b}^{(j)}) = \sum_{j=0}^{W-1} \text{val}_2(\text{rot}_j(\mathbf{b}^{(j)}))$$

We use CKKS rotation to implement the vector rotation, and after rotating each bit string, we obtain a new vector $\mathbf{s} = \sum_{j=0}^{W-1} \text{rot}_j(\mathbf{b}^{(j)})$ with the same represented value. Since we only need to maintain the value modulo 2^n , we mask out the higher-order bits of $\mathbf{b}^{(j)}$ and maintain $\mathbf{s}$ as an n -slot vector.

Summation using 3-to-2 boolean compressors. Instead of directly summing the vectors $\mathbf{b}^{(j)}$, we exploit the fact that each of them is a binary string and use binary circuits to add them.

We design a 3-to-2 compressor circuit to iteratively reduce the number of binary strings. The functionality of the compressor circuit is the same as a full adder, while it is designed to contain fewer multiplications and be more efficient to implement using CKKS. The circuit takes three binary inputs $x, y, z \in \{0, 1\}$ and outputs two binary values, their sum s and carry c . The circuit is defined as $c := yz + (y + z - 2yz)x$ and $s := x + y + z - 2c$. One can verify that s and c are both binary values satisfying $s + 2c = x + y + z$. The circuit can be implemented in two multiplicative levels. In the concrete implementation, we need to incorporate proper masking for the input binary strings; the formula is given in Appendix B and has no additional level cost.

By homomorphically performing the circuit in a SIMD manner, we can reduce three binary strings $\mathbf{x}, \mathbf{y}, \mathbf{z}$ into two binary strings $\mathbf{s}$ and $\mathbf{c}$, while maintaining their represented value $\text{val}_2(\mathbf{x} + \mathbf{y} + \mathbf{z}) = \text{val}_2(\mathbf{s} + \text{rot}_1(\mathbf{c}))$. Again we mask out the highest bit of $\mathbf{c}$ to maintain n -slot vectors.

For the input binary strings $\mathbf{b}^{(j)}$, we build an $O(\log W)$ -level tree of 3-to-2 compressors to reduce them to two binary strings, which costs $O(\log W)$ multiplicative levels. For example, when $W = 7$, the 4-level tree has $7 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2$ binary strings in each level, respectively, which uses 8 multiplicative levels. The final two binary strings are then added together to obtain the final digit vector with digits in $\{0, 1, 2\}$.

Finally, reducing the vector with digit values in $\{0, 1, 2\}$ to the canonical binary digits can be solved by performing carry-lookahead on the digits. Under CKKS, this is optimized by [CPL26] using the LCtoC algorithm, and we adopt it to perform the final step of A2B_{CLA}.

3.3 Complexity and Level Analysis

In this section, we analyze the consumption of CKKS bootstrapping operations and multiplicative levels in the A2B_{CLA} algorithm. First, the preprocessing A2A- I consumes one CKKS bootstrapping; then the linear transformation $\mathcal{Z}$ -to- $\mathbb{C}$ takes 1 level; next, an MV-FBT with W LUTs over $\mathbb{Z}_{2^W}$ is applied to extract the binary strings; after that, the 3-to-2 compressor tree takes $O(\log W) = O(\log \log K)$ multiplicative levels; finally, the LCtoC algorithm takes 1 FBT to convert the symbols and $O(\log n)$ multiplicative levels, as described in [CPL26].

In total, A2B_{CLA} consumes 3 CKKS bootstrapping operations, and $O(\log \log K + \log n)$ multiplicative levels. Since K is a constant determined by the CKKS scheme, this can be seen as $O(1)$ CKKS bootstrapping and $O(\log n)$ multiplicative levels with respect to the plaintext bit-width n .

4 Extending Gao–Zheng and A2B_{CLA} for Prime Moduli

In this section, we show how to extend the Gao–Zheng scheme and the A2B_{CLA} conversion to support prime moduli. We also show that this conversion algorithm

can be used to implement homomorphic modular conversion and homomorphic division for a prime modulus.

4.1 Extending Gao–Zheng Scheme to Prime Moduli

While the Gao–Zheng scheme can already support arbitrary moduli through folding or the Montgomery form, as explored by the Cha–Park–Lee scheme [CPL26], these methods require a *reduction* operation consisting of multiple bootstrapping operations after *each* multiplication. Moreover, to obtain the bit representation from the Montgomery form, we need another Montgomery reduction before the conversion to bits.

To natively support a prime modulus q with bit-width n , we extend the ring in Gao–Zheng $\mathcal{Z} = \mathbb{Z}[X]/\langle X^n - X + 2 \rangle$ to a more general ring $\mathcal{Z}_{F_q} := \mathbb{Z}[X]/\langle F_q(X) \rangle$. Here $F_q(X)$ is called the *supporting polynomial* for q , which needs to satisfy the following properties: (1) F_q is a degree n monic polynomial with coefficients in $\{-1, 0, 1\}$; (2) $F_q(2) = q$; (3) F_q has n distinct non-real roots when n is even; F_q has $n - 1$ distinct non-real roots and one real root when n is odd. Property 1 gives a bound on $\|tI\| \leq 4\|I\|$ in $\mathcal{Z}_{F_q}$, which is used in A2B_{CLA}; Property 2 guarantees that $\mathcal{Z}_{F_q}/\langle X - 2 \rangle \cong \mathbb{F}_q$; Property 3 gives the isomorphism $\mathcal{Z}_{\mathbb{R}, F_q} := \mathbb{R}[X]/\langle F_q(X) \rangle \cong \mathbb{C}^{n/2}$ for even n , and $\mathcal{Z}_{\mathbb{R}, F_q} \cong \mathbb{R} \times \mathbb{C}^{(n-1)/2}$ for odd n , so we can efficiently encode the polynomial in complex slots. For any odd prime q , its supporting polynomial $F_q(X)$ always exists (see Appendix D for the proof). For a concrete instantiation, we perform a heuristic search over polynomials and select the one with the minimal multiplication noise growth based on experiments.

In $\mathcal{Z}_{F_q}$, the flattened polynomial $[m]_t$ for $m \in \mathbb{Z}_q$ is defined as $[m]_t = \sum_{i=0}^{n-1} b_i X^i$ where b_i is the i -th bit of m , using its representative in $[0, q)$. Note that this is an unsaturated mapping from m to a length- n binary string. We can similarly define the arithmetic triangle encoding $\Delta(I + [m]_t/t) + e$ in $\mathcal{Z}_{\mathbb{R}, F_q}$ to support addition and multiplication in $\mathbb{F}_q$. We can again define a homomorphic linear transformation between $\mathcal{Z}_{\mathbb{R}, F_q}$ and $\mathbb{C}^{n/2}$ (or $\mathbb{R} \times \mathbb{C}^{(n-1)/2}$ if n is odd) using the Vandermonde matrix of the roots of $F_q(X)$ in $\mathbb{C}$. Moreover, conversions like A2A, B2B and B2A can be performed using the same sequence of operations as in the Gao–Zheng scheme.

The main difference is the arithmetic-to-Boolean conversion. The A2B conversion and its batched version B-A2B in Gao–Zheng do not apply to the general triangle encoding because we cannot extract the bits one by one.⁷

4.2 A2B_{CLA} for Prime Modulus

To obtain arithmetic-to-Boolean conversion for a prime modulus q , a natural idea is to extend the A2B_{CLA} introduced in Section 3. However, the original A2B_{CLA} canonicalizes $M := tI + [m]_t \in \mathcal{Z}$ into a length- n bit vector, it cannot be interpreted as $[m]_t$, and we need to preserve all bits in $M \pmod{t}$:

⁷ For this reason, the arithmetic-to-Boolean conversion in the concurrent work of Shen, Li, Wang, and Lu [SLWL26] does not apply to a general modulus q .

Proposition 2 (Modulo- t binary canonicalization gives m modulo q).
For $M := tI + [m]_t \in \mathcal{Z}_{F_q}$, we have the relation $[M(2)]_q = m$. In other words, m is the remainder of $M(2)$ modulo q .

Proof. The proof is similar to Proposition 1 by replacing $X^n - X + 2$ with $F_q(X)$.

To homomorphically compute modulo-by- t operation, we still follow the template as in Section 3, but to maintain the value modulo q , we cannot discard the higher-order coefficients of $\mathbf{M}$, and the result will be a bit string $\mathbf{a}$ whose length is longer than n . We therefore are required to “fold” the higher-order coefficients into lower coefficients and produce a length- n vector $\mathbf{a}'$ with $\text{val}_2(\mathbf{a}') \in [0, 2^n)$. It will be further canonicalized into the bit string $[m]_t$ as $q < n$. They can be carried out during the A2B_{CLA} process with moderate-overhead additional steps.

Overflow Gathering. As the first step, we still perform the A2B_{CLA} algorithm described in Section 3, but we additionally gather the discarded overflows into a separate ciphertext. We regard the accumulated overflow A as a coefficient of X^n . For coefficient b_k at X^{n+k} , we regard it as $(2^k b_k)X^n$ and add $2^k b_k$ to the accumulated overflow A . This process does not consume additional multiplicative levels. After the A2B_{CLA} algorithm, we have a length- n bit vector $\mathbf{b}$ and a gathered overflow A . Since the represented value is preserved in each step, we have $M^+(2) = \text{val}_2(\mathbf{b}) + A \cdot 2^n$.

Overflow Wrap-around. We crucially find out that the binary representation $\mathbf{d}^{(0)}$ of $A_0 := [A \cdot 2^n]_q$ can be used to replace the term $A \cdot 2^n$ because the modulo- q value is preserved. To obtain $\mathbf{d}^{(0)}$ efficiently, we broadcast A in one slot to n slots using rotate-and-sum, then apply a SP-FBT with its i -th slot outputting the i -th bit of $[A \cdot 2^n]_q$ for input A . A is bounded due to the following observation:

Lemma 2. *There exists a constant $C_0 \geq 0$ such that $A \in [C_0, C_0 + 2K + 1]$.*

The proof of Lemma 2 is given in Appendix C.1. With this lemma, we can first subtract C_0 from A and obtain $A' = A - C_0$, then feed A' into the SP-FBT where the LUT at slot i outputs the i -th bit of $[(A' + C_0) \cdot 2^n]_q$. Then the LUT has a bit-width of $\log(2K + 2) = 5$ under the setting in Section 3.2, and outputs the binary representation $\mathbf{d}^{(0)}$.

Now we obtain $\mathbf{s}^{(0)} := \mathbf{b} + \mathbf{d}^{(0)}$. The represented value is preserved as $\text{val}_2(\mathbf{s}^{(0)}) = \text{val}_2(\mathbf{b}) + A_0 \equiv \text{val}_2(\mathbf{b}) + A \cdot 2^n = M(2) \bmod q$.

Final Canonicalization. Since the digits of $\mathbf{s}^{(0)}$ has a small alphabet $\{0, 1, 2\}$, it seems that applying LCtoC gives us $[m]_t$. However, LCtoC may produce an additional carry that should be wrapped around again. Moreover, the output n -bit binary string might be a representation of m or $m + q$ and is not unique.

Instead, we produce flag bits for all possibilities of the represented value (e.g. m or $m + q$), and use them to select the correct representation. In concrete, we find out that $\text{val}_2(\mathbf{s}^{(0)}) = \text{val}_2(\mathbf{b}) + A_0 < 2^n + q < 3q$, so we only need to distinguish between three cases: $\text{val}_2(\mathbf{s}^{(0)}) = m \in [0, q)$, $\text{val}_2(\mathbf{s}^{(0)}) = m + q \in [q, 2q)$ and

$\text{val}_2(\mathbf{s}^{(0)}) = m + 2q \in [2q, 3q)$. Then we need to subtract by q and $2q$ in the second and third case, respectively. The flag and subtracted value can be obtained by three carry-lookahead circuit in parallel, which is formulated by the following lemma (proof given in Appendix C.2).

Lemma 3. *Let $A_1 := A_0 + (2^n - q)$ and $A_2 := \max(0, A_0 + (2^n - 2q))$. For $i = 0, 1, 2$, consider the sum $s_i := \text{val}_2(\mathbf{b}) + A_i$ and decompose it as $s_i = x_i + c_i 2^n$ where $x_i \in [0, 2^n)$. We have*

$$x_0 + c_1(x_1 - x_0) + c_2(x_2 - x_1) = m \quad \text{and} \quad A_i \in [0, 2^n), c_i \in \{0, 1\}, \quad i = 0, 1, 2$$

We generate the binary representations of A_1 and A_2 , denoted as $\mathbf{d}^{(1)}$ and $\mathbf{d}^{(2)}$ respectively, together with $\mathbf{d}^{(0)}$ using a SP-MV-FBT. We want to obtain the bit string $\mathbf{s}^{(i)}$ and c_i such that $\text{val}_2(\mathbf{s}^{(i)}) + c_i 2^n = s_i$.

This is done by $\mathbf{b} + \mathbf{d}^{(i)}$ in parallel using carry-lookahead that produces $\mathbf{s}^{(i)}$ and the overflow bit c_i . The range in the lemma guarantees that $\mathbf{d}^{(i)}$ can be represented by n bits, and the overflow bit c_i is either 0 or 1. At last, we perform a MUX using $\mathbf{s}^{(0)} + c_1(\mathbf{s}^{(1)} - \mathbf{s}^{(0)}) + c_2(\mathbf{s}^{(2)} - \mathbf{s}^{(1)})$ to obtain $[m]_q$, which finishes the A2B_{CLA} conversion under modulus q .

The three carry-lookahead circuit needs special symbols and we can use a single multi-value symbol-LUT to produce them by packing the four input binary strings ($\mathbf{d}^{(0)}$, $\mathbf{d}^{(1)}$, $\mathbf{d}^{(2)}$, $\mathbf{b}$) as the input. Then the three carry-lookahead circuit can be performed in parallel, consuming one MV-FBT and $\log(n) + 1$ CKKS levels.

Especially, we show that when q is close to 2^n , A_2 and c_2 are always 0, and $\mathbf{d}^{(2)}$ and $\mathbf{s}^{(2)}$ are not needed. This removes one output of the SP-MV-FBT, one input width, and therefore one level, of the packed symbol-LUT, and one use of the carry-lookahead circuit. The following lemma gives the concrete formulation (see Appendix C.3 for its proof).

Lemma 4. *When $q \geq \left(1 - \frac{1}{5K+4}\right) 2^n$, we have $A_2 = 0$.*

Notice that both prime moduli p and u in the ECDSA signing protocol satisfy the condition in Lemma 4, and therefore this optimization is applied to our homomorphic ECDSA signing.

Remark on nonnegative redundant digits. Notice that under modulus q , the offset vector $\mathbf{r}_H := (H, H, \dots, H) + \mathbf{h}'$ for $\mathbf{h}'$ being the binary representation of $[H(1 - 2^n)]_q$, which is different from the construction in Section 3.2. Nonetheless, the bound for the digits of $\mathbf{M}^+$ remains the same.

4.3 Complexity and Level Analysis

In this section, we analyze the number of FBTs and multiplicative levels consumed in the A2B_{CLA} conversion for modulus q . The overall level consumption of A2B_{CLA} is presented in Table 2. First, the original A2B_{CLA} algorithm consumes 3

Table 2. The level consumption of A2B_{CLA} for $\mathbb{Z}_{2^n}$ and $\mathbb{F}_q$. The additional operations or bootstrapping steps needed by A2B_{CLA} for $\mathbb{F}_q$ are marked with $\dagger$. This table should be read from left to right and top to bottom to understand the level consumption of each operation. After each C2R, the **ModRaise** operation restores the available level to L , and this is represented as a new line in this table.

Before BTS				Z2C 1	C2R 2
A2A-I	R2C 3	C2Z 1		Z2C(t) 1	Mask* 1
MV-LUT [‡]	R2C 3	MV-LUT 16	Contraction 8		C2R 2
LCtoC [‡]	R2C 3	Symbol LUT 11	Carry-lookahead* $\log(n) + 1$		C2R [†] 2
δ-LUT [†]	R2C [†] 3	SP-MV-LUT [†] 14			C2R [†] 2
LCtoC [†]	R2C [†] 3	Symbol LUT [†] 12 or 13	Carry-lookahead [†] * $\log(n) + 1$	MUX [†] 1	

* **Mask** is designed to resolve the “breaking-into-halves” effect [GZ26] and gather each word into consecutive slots. The original format is restored after the propagation, which we can fuse into the final step of the last carry-lookahead.

[‡] The overflow A is gathered during these two processes.

CKKS bootstrapping operations (including the bootstrapping for **A2A-I**), and $O(\log \log K + \log n)$ multiplicative levels; Then, in the overflow wrap-around step, we apply SP-FBT to get the binary representation of $A^{(i)}$, which is one SP-MV-FBT and refreshes the level of the ciphertext; After that, the LCtoC algorithm is applied to $(\mathbf{d}'(0), \mathbf{d}^{(1)}, \mathbf{d}^{(2)}, \mathbf{b})$ to get $\mathbf{s}_i$, which takes 1 MV-FBT over $\mathbb{Z}_{2^4}$ to convert the symbols, and $O(\log n)$ multiplicative levels; Finally, we use the overflow bits c_1 and c_2 to select the final result $\mathbf{s}$, which takes 1 multiplicative level.

In total, the A2B_{CLA} conversion under modulus q consumes 5 CKKS bootstrapping operations, and $O(\log \log K + \log n)$ multiplicative levels. Similar to the previous modulo- 2^n scenario, this can also be seen as $O(1)$ CKKS bootstrapping and $O(\log n)$ multiplicative levels with respect to the plaintext bit-width n .

4.4 Homomorphic Modular Conversion Using A2B_{CLA}

Our A2B_{CLA} algorithm can also be used to convert ciphertexts between different moduli. The core idea of the conversion comes from the fact that the inputs of B2A do not need to be in binary form: if slots have a larger digit range, the output arithmetic mode ciphertext will still be an encryption of $\frac{[m]_t}{t}$, plus an additional overflow I . This is exactly the triangle encoding, and we only need one **A2A-I** to truncate I if it is too large.

In general, if we want to convert a ciphertext from a modulus p to a modulus q with the same or larger bit-width, we can first convert the ciphertext to Boolean mode using A2B_{CLA} under $\mathbb{F}_p$, then pad the Boolean string to the bit-width of q , and use B2A under $\mathbb{F}_q$ to convert it back to arithmetic mode modulo q . The p to u conversion in the ECDSA signing protocol is a concrete example. When the bit-width of q is smaller than that of p , an additional “overflow wrapping” turns the high-order bits into the low-order bits. More specifically, if p has bit-width n and q has bit-width m , then a bit b_i in the high-order slot $i \in [m, n)$ should be wrapped into the value $[b_i \cdot 2^i]_q$. We can get its binary representation by preparing the binary string of $[2^i]_q$ in plaintext at discrete CKKS encoding and multiplying it by the slot value b_i . Then we add these $n - m$ strings and the original low-order slots together, which will give a string with digits in $[0, n - m + 1)$. Then after B2A under $\mathbb{F}_q$ we can get the final result.

We note that such conversion implies homomorphic ciphertext-plaintext division for a prime modulus q : for an encrypted value m modulo p , we can first convert m to modulus q , which gives $[m]_q$. Then, we can convert $[m]_q$ back to modulo p , and compute $(m - [m]_q)q^{-1} \pmod{p}$, which is exactly $\lfloor m/q \rfloor$.

5 One-Round Threshold ECDSA Using A2B_{CLA}

In this section, we demonstrate the application of A2B_{CLA} by constructing a one-round threshold ECDSA scheme.

The deterministic ECDSA signing process over secp256k1 consists of the following steps: (1) from the message hash h and the signing key sk_{SIG} , derive the nonce $k = \text{HMAC-DRBG}(h, \text{sk}_{\text{SIG}})$ by a hash-based PRF; (2) compute the curve point $R = [k]G$ on $E(\mathbb{F}_p)$, where G is the base point; (3) compute $r = R_x \bmod u$, where R_x is the x -coordinate of R and u is the order of G ; (4) compute $s = k^{-1}(h + r \cdot \text{sk}_{\text{SIG}}) \bmod u$; and (5) output the signature (r, s) .

The above process is a mixed-mode, mixed-modulus workload. Step (1) is Boolean, steps (2) and (4) are arithmetic over two different prime moduli (p for the curve and u for the scalar field), step (3) requires a transformation from one modulus to another, and step (5) must return to bits to perform noise cleaning for threshold decryption. Therefore, the underlying FHE scheme must support Boolean operations and arithmetic operations over different fields $\mathbb{F}_p$ and $\mathbb{F}_u$, as well as efficient conversion between them. We achieve this with our new A2B_{CLA} algorithm, which works for elliptic-curve moduli.

Fig. 5 provides the concrete algorithm of the proposed ECDSA protocol, and Fig. 2 traces the data flow and the mode changes through the protocol. In the rest of this section, we outline how the ECDSA signing protocol is evaluated under our FHE scheme.

5.1 Supporting Polynomials for secp256k1

For the secp256k1 curve, we select the supporting polynomial $F_p(X)$ for modulus p and $F_u(X)$ for modulus u , with the criterion that their multiplication noise

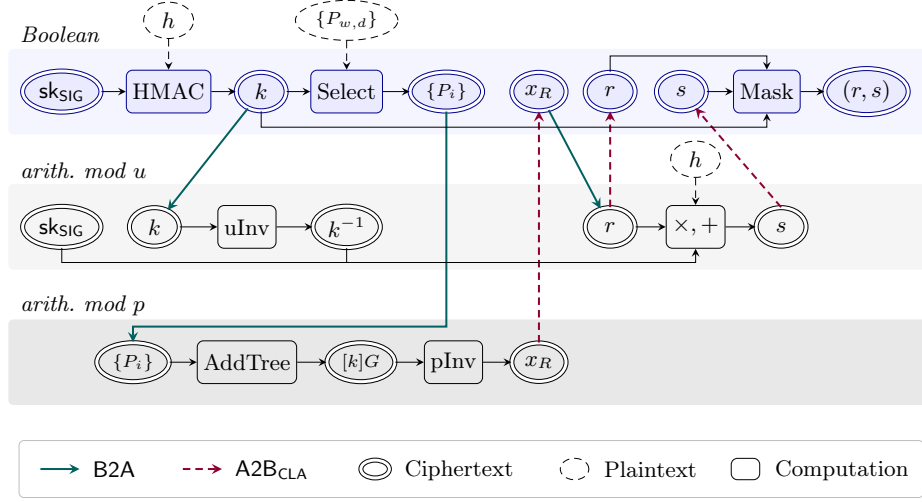


Fig. 2. Dataflow of one-round threshold ECDSA signature protocol. The rows correspond to the three ciphertext modes: Boolean mode, arithmetic mode modulo u , and arithmetic mode modulo p . The arrows indicate data flow, and the colored arrows between different rows indicate a conversion using B2A or A2B_{CLA}.

growth is minimal among the instantiations we searched. The concrete definitions are as follows:

$$\begin{aligned}
 F_p(X) &= X^{256} - X^{32} - X^{10} + X^5 + X^4 - X + 1 \\
 F_u(X) &= X^{256} + \sum_{i \in A} X^i - \sum_{j \in B} X^j \\
 A &= \{0, 6, 8, 14, 22, 28, 30, 31, 36, 39, 45, 46, 49, 52, 70, \\
 &\quad 77, 79, 83, 86, 99, 104, 116, 117, 119, 121, 126, 127\} \\
 B &= \{17, 19, 34, 41, 54, 62, 66, 88, 92, 94, 96, 101, 106, 109, 112, 123, 129\}
 \end{aligned}$$

5.2 Hash Function Evaluation

In step (1) of ECDSA, we evaluate the deterministic nonce-generation procedure HMAC-DRBG on the message hash h and the signing key sk_{SIG} . We instantiate it with HMAC-SHA3-256, since SHA3-256 is relatively friendly to evaluate in discrete CKKS and is recognized in FIPS-186-5 as a valid hash function for ECDSA [Nat23]. We pack the Keccak states into one ciphertext in Boolean mode, where each bit of the state is encrypted in a separate slot of the ciphertext, and at most 16 states can be packed in parallel.

As the generated nonce k is required to lie in $[1, u - 1]$, nonce generation includes a rejection sampling process. Under homomorphic evaluation, we generate a fixed number $cand$ of candidates, compute their validity flags, and obviously

$\text{KeyGen}_{\text{ThSIG}}(1^\lambda, 1^\eta, 1^\theta)$
1 : $(\text{sk}_{\text{SIG}}, \text{pk}_{\text{SIG}}) \leftarrow \text{KeyGen}_{\text{SIG}}(1^\lambda)$ 2 : $(\text{pp}_{\text{ThFHE}}, \text{pk}_{\text{ThFHE}}, \text{evk}_{\text{ThFHE}}, \{\text{sk}_{\text{ThFHE}}^{(i)}\}_{i \in [\eta]}) \leftarrow \text{KeyGen}_{\text{ThFHE}}(1^\lambda, 1^\eta, 1^\theta)$ 3 : $\llbracket \text{sk}_{\text{SIG}} \rrbracket = \text{Enc}_{\text{ThFHE}}(\text{sk}_{\text{SIG}}, \text{pk}_{\text{ThFHE}})$ 4 : $\text{pp} \leftarrow (\text{pp}_{\text{ThFHE}}, \text{pk}_{\text{ThFHE}}, \text{evk}_{\text{ThFHE}}, \llbracket \text{sk}_{\text{SIG}} \rrbracket)$ 5 : return $(\text{pp}, \text{pk}_{\text{SIG}}, \{\text{sk}_{\text{ThFHE}}^{(i)}\}_{i \in [\eta]})$
$\text{PartSign}_{\text{ThSIG}}(\text{sk}_{\text{ThFHE}}^{(i)}, \mathcal{M})$
1 : $h \leftarrow \mathcal{H}(\mathcal{M})$ 2 : $\llbracket k \rrbracket \leftarrow \text{HMAC-DRBG}(h, \llbracket \text{sk}_{\text{SIG}} \rrbracket)$ // Homomorphically compute HMAC 3 : $\llbracket R \rrbracket \leftarrow \llbracket k \rrbracket G$ // Scalar multiplication in $E(\mathbb{F}_p)$ 4 : $\llbracket x_R \rrbracket \leftarrow \llbracket R \rrbracket$ // Affine recovery from projective coordinates 5 : $\llbracket r \rrbracket \leftarrow \llbracket x_R \rrbracket \pmod{u}$ // Modular reduction from $\mathbb{F}_p$ to $\mathbb{F}_u$ 6 : $\llbracket s \rrbracket \leftarrow \llbracket k \rrbracket^{-1}(h + \llbracket r \rrbracket \llbracket \text{sk}_{\text{SIG}} \rrbracket)$ // In $\mathbb{F}_u$ 7 : $\llbracket (r, s) \rrbracket \leftarrow \text{Mask}(\llbracket r \rrbracket, \llbracket s \rrbracket, \llbracket k \rrbracket)$ // Mask, bit conversion and bit cleaning 8 : $\text{part}_{\text{ThSIG}}^{(i)}(\mathcal{M}) \leftarrow \text{PartDec}_{\text{ThFHE}}(\llbracket (r, s, \text{flag}) \rrbracket, \text{sk}_{\text{ThFHE}}^{(i)})$ 9 : return $\text{part}_{\text{ThSIG}}^{(i)}(\mathcal{M})$
$\text{Combine}_{\text{ThSIG}}(\{\text{part}_{\text{ThSIG}}^{(i)}(\mathcal{M})\}_{i \in S, S =\theta})$
1 : $\sigma \leftarrow \text{FinDec}_{\text{ThFHE}}(\{\text{part}_{\text{ThSIG}}^{(i)}(\mathcal{M})\}_{i \in S, S =\theta})$ 2 : return σ

Fig. 3. Our one-round threshold ECDSA signature protocol. The steps in blue are homomorphic computations.

select the first valid candidate. We choose $\text{cand} = 2$, which gives a negligible failure probability.

HMAC-DRBG needs eight HMAC evaluations to generate two candidates: four for initialization, one per candidate, and two between consecutive candidates [Por13]. One HMAC evaluation invokes two SHA3-256 hash function, each with two blocks of input. We use SIMD and pack the first input block of both hash function into one ciphertext, then evaluates them in parallel, resulting in three Keccak permutation calls for each HMAC. We further reuse the two prefix states across HMAC calls with the same key, which saves another four permutations, resulting in 20 packed permutation calls.

Each Keccak permutation consists of 24 rounds [Nat15]. We carefully design the homomorphic evaluation and implement each round in nine multiplicative levels and three rounds can be refreshed using two B2B. Therefore, each Keccak permutation uses 16 B2B. With 20 packed permutation calls, we have 320 B2B operations. We need some additional comparison and masking for the validity flag, which consumes 2 B2B.

5.3 Scalar Multiplication on the Elliptic Curve

In the next step, we need to compute the scalar multiplication $R = [k]G$ for a fixed public base point G on the elliptic curve $E(\mathbb{F}_p)$. To evaluate this homomorphically, we first use the classical fixed-base window method [BGMW93, LL94]: we pack the bits of k in groups of 4, and write it as $k = \sum_{w=0}^{63} 16^w d_w$, where $d_w \in [0, 15]$ is the w -th digit of k in base 16. We precompute the $64 \cdot 16$ point table $P_{w,d} = [d16^w]G$ in clear, and pack them into 8 plaintexts in Boolean mode. The lookup for each point is a four-level multiplexer tree whose leaves are public table entries $P_{w,0}, \dots, P_{w,15}$ and whose selectors are the encrypted digit bits d_w . The selection of all 64 windows can be performed in parallel with the SIMD packing. After the final 64 points are selected, they are converted to arithmetic mode modulo p using one B2A conversion.

For the next step, we need to compute the sum of the selected points $P_w := P_{w,d_w}$ for $w = 0, \dots, 63$. Since point addition under the standard affine coordinates (x, y) requires a modular inverse modulo p , which is expensive in the FHE setting, we instead use homogeneous projective coordinates⁸ $(X : Y : Z)$ to represent a point (XZ^{-1}, YZ^{-1}) . Then no modular inverse is needed in the addition process, and each point is stored as three 256-bit words modulo p . The point addition is then performed by a multiplication tree with depth $\log(64) = 6$. Since the points are packed in one ciphertext, point addition on each level is performed in parallel, and we design a dedicated SIMD-style homomorphic evaluation for the point addition (see Appendix E for the detailed implementation). After the addition tree finishes, we are left with the final point $R = [k]G$ in homogeneous projective coordinates $(X_R : Y_R : Z_R)$, and the final step is to compute the affine x -coordinate $x_R = X_R Z_R^{-1}$ modulo p , which we will describe in the next subsection.

5.4 Modular Inverse Computation

For the ECDSA signature, two modular inverse computations are needed: $Z_R^{-1} \bmod p$ for the affine recovery discussed above, and $k^{-1} \bmod u$ for the computation of the scalar s . Both are computed by Fermat's Little Theorem (Z_R^{p-2} and k^{u-2} , respectively) using a fixed multiplication chain [Smi17]. The p -inverse chain contains 270 ciphertext multiplications and costs 270 levels, while the u -inverse chain contains 290 ciphertext multiplications and costs 290 levels. A2A is performed after each multiplication to refresh the overflow I and error e of the ciphertext.

5.5 The Conversion to Bits and Cleaning

Finally, we need to compute $s = k^{-1}(h + r \cdot \text{sk}_{\text{SIG}})$ under $\mathbb{F}_u$. Since we have the ciphertext for k^{-1} , r and sk_{SIG} in arithmetic mode modulo u , this is a simple arithmetic computation.

⁸ We use homogeneous projective coordinates instead of Jacobian coordinates, since only the former gives a uniform point addition formula even when the input is the infinity point O , which is more efficient to implement in FHE.

Table 3. Parameters for all experiments

RingDim	Scaling Factor	MulDepth [†]	Modulus	SK Dist
N	$\log \Delta$	L	$\log QP$	$(h ; \tilde{h})$
2^{16}	43	29	1741	(Uniform Ternary ; 32)

[†] Refers to the CKKS multiplicative depth, not the available multiplicative depth for arithmetic mode in our scheme.

Then we convert the results r and s back to Boolean mode using A2B_{CLA} for modulus u . With the Boolean mode ciphertext, we can evaluate the validity predicate $\text{flag} = [1 \leq k < u] \wedge [r \neq 0] \wedge [s \neq 0]$, and use it to mask the output (r, s) , converting it to $(0, 0)$ if the signature is invalid. Then r and s are packed into a single Boolean ciphertext. To enable noise flooding, we apply the noise cleaning polynomial $h : x \mapsto 3x^2 - 2x^3$ that consumes two CKKS levels [CKK20].

5.6 Signature Failure Probability

A signing attempt fails if one of the following happens: $k \notin [1, u - 1]$, $r = 0$, or $s = 0$. Since the nonce k is pseudorandom, its value can be modeled as uniformly distributed in $[0, 2^{256})$. Then with $cand = 2$, the failure probability of k is $p_1 = (1 - \frac{u-1}{2^{256}})^2 \approx 2^{-255.3}$. Then by the uniformity of k , the value of r is uniformly chosen among the coordinates of nonidentity curve points, where the probability of $r = 0$ is $p_2 = \frac{2}{u-1} \approx 2^{-255.0}$; by similar analysis on s , we have the probability of $s = 0$ is $p_3 \leq \frac{4}{u-1} \approx 2^{-254.0}$. Thus the total failure probability of the signature is bounded by $p_1 + p_2 + p_3 \leq 2^{-253}$, which is negligible.

6 Experiments

6.1 Implementation and Parameters

Our implementation is based on the code from [GZ26], using OpenFHE [BAB⁺22] v1.5.1. It uses the HEXL [BKS⁺21] backend for CPU experiments and the FIDESlib [ADOVLG⁺25] backend for GPU experiments. Our CPU experiments were conducted in a single-threaded setting on a Linux machine with an AMD RYZEN AI MAX+ 395 CPU and 128GB RAM, running Debian 13. The compiler used is clang 19. Our GPU experiments were conducted on a Linux machine with 110GB host memory and an NVIDIA RTX Pro 6000 with 96GB VRAM, running Ubuntu 22.04.

Most parameters are summarized in Table 3. Compared with [GZ26], one difference is that we select $L = 29$ to support MV-FBT with LUT over $\mathbb{Z}_{2^w}$, and to accommodate the additional level consumption caused by the 3-to-2 contraction circuit and the carry-lookahead circuit in L_CtoC. Note that [GZ26] selected $L = 20$. While we could in principle select a smaller L with more bootstrapping operations, we use the current form for easier exposition.

Table 4. Detailed performance numbers for A2B_{CLA} . n is the bit-width of the modulus. The first table reports single-threaded CPU times, and the second reports GPU times. The first number in each cell is the total time of the operation, and the second number is the amortized time per slot. We reuse the A2A-I data across different moduli with the same bit-width.

Mod	#Slot	A2A-I	A2B _{CLA} (w/o A2A-I)	A2B _{CLA}	Noise
2^{64}	1024	6.972s / 6.809ms	33.69s / 32.90ms	40.66s / 39.71ms	-12.67
2^{128}	512	7.788s / 15.21ms	35.02s / 68.40ms	42.81s / 83.61ms	-13.90
2^{256}	256	9.299s / 36.32ms	35.19s / 137.5ms	44.49s / 173.8ms	-14.05
p	256	9.299s / 36.32ms	63.40s / 247.7ms	72.70s / 284.0ms	-13.67
u	256	9.299s / 36.32ms	63.05s / 246.3ms	72.35s / 282.6ms	-13.51
p_{25519}	256	9.299s / 36.32ms	63.16s / 246.7ms	72.46s / 283.0ms	-13.13

Mod	#Slot	A2A-I	A2B _{CLA} (w/o A2A-I)	A2B _{CLA}	Noise
2^{64}	1024	45.67ms / 44.60us	231.4ms / 226.0us	277.1ms / 270.6us	-13.51
2^{128}	512	52.33ms / 102.2us	237.7ms / 464.3us	290.0ms / 566.4us	-14.71
2^{256}	256	60.03ms / 234.5us	244.9ms / 956.6us	304.9ms / 1.191ms	-13.15
p	256	60.03ms / 234.5us	426.1ms / 1.664ms	486.1ms / 1.899ms	-11.90
u	256	60.03ms / 234.5us	428.0ms / 1.672ms	488.0ms / 1.906ms	-13.52
p_{25519}	256	60.03ms / 234.5us	431.6ms / 1.686ms	491.6ms / 1.920ms	-12.93

Another difference is that we use a uniform ternary main secret key with an encapsulated sparse secret key of Hamming weight $\tilde{h} = 32$. By using the scaling factor $\log_2 \Delta \approx 43$, $\text{dnum}=3$, and 9 auxiliary primes (50-bit) to construct P , we obtain $\log QP \approx 1741$, which maintains 128-bit security according to [BCC⁺24] and [BTH22].

6.2 Performance of A2B_{CLA}

We report the latency and noise (in $\log_2(\|e\|/\Delta)$) for our A2B_{CLA} algorithm in Table 4. For modulus 2^n , the execution latency does not change significantly with respect to the bit-width n , and only a slight increase is observed when n increases from 64 to 256. This is because the cost of the MV-LUT, symbol LUT and 3-to-2 contraction circuit is independent of the bit-width n ; the only increase in latency comes from the additional level consumption in LCtoC, which needs $O(\log n)$ rounds. For the moduli p and u that are not powers of 2, the latency is nearly doubled with the modulus 2^n case, due to the computation of the two additional LUTs and LCtoC.

Table 5. Comparison of the cost of one multiplication followed by conversion to a bit representation. We use single-threaded CPU timings and normalize other methods using their comparisons with Gao–Zheng [GZ26].

Method	Mod	#Slot	Mult + A2B (Lat./Amort.)
TFHE-rs [Zam22]	p	1	569s (GPU 16.0s)
	u	1	840s (GPU 21.1s)
[CPL26]	2^{64}	1024	35.6s / 35ms
	2^{256}	256	66.5s / 260ms
	p_{25519}	128	216.3s / 1.69s
	p	128	≈ 239 s / 1.9s
	u	128	≈ 518 s / 4.1s
[GZ26]	2^{64}	1024	80.5s / 79ms
	2^{256}	256	327s / 1.3s
	u	128	≈ 76 min / 35s
[GZ26] (Batched)	2^{64}	16384	128s / 7.8ms
	2^{256}	16384	823s / 50ms
[SLWL26]	2^{64}	1024	31.0s / 30.2ms
	2^{64}	2048	37.6s / 18.3ms
	2^{256}	256	49.2s / 192ms
	2^{256}	512	57.6s / 112ms
[LMWW26]	2^{64}	1024	28.0s / 27.3ms
	2^{256}	256	32.0s / 125ms
	p_{25519}	256	> 269 s / > 1.04 s
Ours	2^{64}	1024	40.7s / 39.7ms
	2^{256}	256	44.5s / 174ms
	p	256	72.7s / 284ms
	u	256	72.4s / 283ms
	p_{25519}	256	72.5s / 283ms

6.3 Comparison with Other Schemes

We compare the cost of one multiplication and of converting the result to a bit representation, and the results are presented in Table 5.

The performance data of [CPL26] are borrowed from [GZ26] as they were measured in the same environment. We further estimate the performance of multiplication and conversion to bit representation for $\mathbb{F}_p$ and $\mathbb{F}_u$ in [CPL26], using their implementation of `baseREDC` and `MontREDC`, respectively, by multiplying the bootstrapping count (18 and 22) with their bootstrapping latency. Note that to obtain a bit representation from the Montgomery form, another `MontREDC` is required and the additional bootstrapping count is 17. The comparison indicates that our native support for large prime moduli gives at most $14.5\times$ speedup over Montgomery-based approaches when comparing the amortized latency. Our performance for large bit-width power-of-two moduli is also $1.5\times$ better.

The performance data of [GZ26] are borrowed from their paper, in both the single-ciphertext setting and the batched setting. In [GZ26] the multiplication is essentially one key-switching and only the data for A2B is used. We further estimate the cost of modular multiplication and conversion to bit representation using the Montgomery form, which requires 7 arithmetic-to-Boolean conversions for $\mathbb{Z}_{2^{512}}$. Compared with them for power-of-two moduli, we obtain a $2 \sim 18.5\times$ better latency for the single-ciphertext setting, but we still have $0.2 \sim 0.28\times$ worse latency in the batched setting; for large prime moduli using Montgomery-based approach, our latency can be $147\times$ better.

The performance data of [SLWL26] and [LMWW26] are borrowed from their papers, normalized according to their comparison with [GZ26], and we add our performance data on A2A-I as it is not contained in their benchmark. The performance of p_{25519} for [LMWW26] is estimated using their performance data for 2^{256} and their intermediate carry combination process, as in Section 1.3.

We also carry out experiments using TFHE-rs 1.8.1 [Zam22] for both $\mathbb{F}_p$ and $\mathbb{F}_u$ by adapting the TFHE-rs secp256k1 implementation.⁹ We run it both in single-threaded CPU setting with AVX512 and GPU setting in the same way as ours. Arithmetic over $\mathbb{F}_p$ is carried out through Mersenne-style folding. Arithmetic over $\mathbb{F}_u$ requires Barrett reduction. Compared with them, our overall latency is significantly better, and our amortized latency can be $\sim 3000\times$ and $\sim 19000\times$ better for CPU and GPU, respectively.

For dBfV [PZZ26] and REFHE [BFG⁺26], the concrete performance comparison could not be carried out due to the lack of performance reports and implementations at the time of writing, especially for moduli p and u at a bit-width of 256 and the need for a generic modulo-by-lattice method.

6.4 Homomorphic ECDSA

We report the GPU latency for homomorphic ECDSA signing in Table 6. Using the ratio in other CPU/GPU benchmarks, we estimate that the required CPU time is roughly 2.5 hours. The GPU execution uses 89GB GPU memory, which mainly includes 376 evaluation keys (26.22 GB) and 8510 plaintexts (54.54 GB).

7 Conclusion

We extend the Gao-Zheng [GZ26] scheme to support large prime moduli, and we introduce a homomorphic canonicalization operation that can be used to implement arithmetic-to-Boolean conversion for power-of-two moduli and prime moduli. We additionally find modular reduction and ciphertext-plaintext division algorithms for a prime dividend. Our bootstrapping count already does not change with the bit-width of the plaintext modulus, but it is always interesting to see whether it can be further reduced. Another interesting problem is whether such an $O(1)$ count could be achieved for scalar schemes.

⁹ <https://github.com/RyanWei0224/tfhe-ecdsa-bench>

Table 6. Detailed GPU performance breakdown of homomorphic ECDSA signing on curve secp256k1. The superscripts (p) and (u) indicate that the operation is performed in $\mathbb{F}_p$ and $\mathbb{F}_u$, respectively.

Step	Technical Step	FHE components	Time (s)
Nonce Gen	HMAC / SHA3	$322 \times \text{B2B}$, $\text{B2A}^{(u)}$	19.577
Elliptic Curve	kG in $(X : Y : Z)$	$12 \times \text{A2A}^{(p)}$, $2 \times \text{B2B}$, $\text{B2A}^{(p)}$	1.415
	Z^{-1}	$271 \times \text{A2A}^{(p)}$	13.978
	$r_x = XZ^{-1}$	$\text{A2A}^{(p)}$	0.029
Signature Gen	$r = r_x \pmod{u}$	$\text{A2B}_{\text{CLA}}^{(p)}$, B2B , $\text{B2A}^{(u)}$	0.460
	k^{-1}	$291 \times \text{A2A}^{(u)}$	19.540
	Compute s	$\text{A2A}^{(u)}$	0.074
Prep Thres Dec	r, s bits	$2 \times \text{A2B}_{\text{CLA}}^{(u)}$, $2 \times \text{B2B}$	0.905
	validity bit	-	0.011
Total			55.989

As an application of our scheme, we demonstrate one-round threshold ECDSA signing. While its performance is not entirely satisfactory, it is a new addition to the broad landscape of FHE applications, and we are optimistic that its performance could be further enhanced.

Acknowledgments. The authors used ChatGPT 5.6 Sol and DeepSeek v4.1 Flash in agent mode for code generation and debugging. We additionally employ them for grammar and spell checking. We also use them for the proof of existence of supporting polynomial for arbitrary prime moduli. The authors have checked the generated code and take full responsibility for the content of this paper and the produced results. We thank Seonhong Min and Dongwon Lee for helpful discussions. We thank Yangyu Chen for providing CPU machines for the experiments.

References

- ADOVLG⁺25. Carlos Agulló-Domingo, Óscar Vera-López, Seyda Guzelhan, Lohit Daksha, Aymane El Jerari, Kaustubh Shivdikar, Rashmi Agrawal, David Kaeli, Ajay Joshi, and José L. Abellán. FIDESlib: A Fully-Fledged Open-Source FHE Library for Efficient CKKS on GPUs. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Ghent, Belgium, 2025. IEEE. Poster paper.
- AJL⁺12. Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 483–501. Springer, Berlin, Heidelberg, April 2012.

- AKP25. Andreea Alexandru, Andrey Kim, and Yuriy Polyakov. General functional bootstrapping using CKKS. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part III*, volume 16002 of *LNCS*, pages 304–337. Springer, Cham, August 2025.
- AKPZ26. Andreea Alexandru, Andrey Kim, Yuriy Polyakov, and Hongren Zheng. Sparse hermite interpolation method for discrete-CKKS functional bootstrapping. Cryptology ePrint Archive, Paper 2026/1026, 2026.
- ASY22. Shweta Agrawal, Damien Stehlé, and Anshu Yadav. Round-optimal lattice-based threshold signatures, revisited. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *ICALP 2022*, volume 229 of *LIPICs*, pages 8:1–8:20. Schloss Dagstuhl, July 2022.
- BAB⁺22. Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R. V., Kurt Rohloff, Jonathan Saylor, Dmitriy Sponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Report 2022/915, 2022.
- BCC⁺24. Jean-Philippe Bossuat, Rosario Cammarota, Ilaria Chillotti, Benjamin R. Curtis, Wei Dai, Huijing Gong, Erin Hales, Duhyeong Kim, Bryan Kumara, Changmin Lee, Xianhui Lu, Carsten Maple, Alberto Pedrouzo-Ulloa, Rachel Player, Yuriy Polyakov, Luis Antonio Ruiz Lopez, Yongsoo Song, and Donggeon Yhee. Security guidelines for implementing homomorphic encryption. *CiC*, 1(4):26, 2024.
- BCKS24. Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, and Damien Stehlé. Bootstrapping bits with CKKS. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part II*, volume 14652 of *LNCS*, pages 94–123. Springer, Cham, May 2024.
- BFG⁺26. Zvika Brakerski, Offir Friedman, Daniel Golan, Alon Gurny, Dolev Mutzari, and Ohad Sheinfeld. REFHE: Fully homomorphic ALU. In Joan Daemen and Emmanuel Thomé, editors, *EUROCRYPT 2026, Part IV*, volume 16545 of *LNCS*, pages 183–213. Springer, Cham, May 2026.
- BGG⁺18. Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 565–596. Springer, Cham, August 2018.
- BGMW93. Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David Bruce Wilson. Fast exponentiation with precomputation (extended abstract). In Rainer A. Rueppel, editor, *EUROCRYPT’92*, volume 658 of *LNCS*, pages 200–207. Springer, Berlin, Heidelberg, May 1993.
- BK25. Dan Boneh and Jaehyung Kim. Homomorphic encryption for large integers from nested residue number systems. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part III*, volume 16002 of *LNCS*, pages 338–370. Springer, Cham, August 2025.

- BK26. Dan Boneh and Jaehyung Kim. A universal blinder: One-round blind signatures from FHE. Cryptology ePrint Archive, Paper 2026/574, 2026.
- BKS⁺21. Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, Vinodh Gopal, et al. Intel HEXL (release 1.2). <https://github.com/intel/hexl>, 2021.
- BKSS24. Youngjin Bae, Jaehyung Kim, Damien Stehlé, and Elias Suvanto. Bootstrapping small integers with CKKS. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part I*, volume 15484 of *LNCS*, pages 330–360. Springer, Singapore, December 2024.
- BTH22. Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 2022*, volume 13269 of *LNCS*, pages 521–541. Springer, Cham, June 2022.
- CGG⁺20. Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1769–1787. ACM Press, November 2020.
- CHK⁺18. Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 360–384. Springer, Cham, April / May 2018.
- CHM⁺25. Hyunho Cha, Intak Hwang, Seonhong Min, Jinyeong Seo, and Yongsoo Song. MatriGear: Accelerating authenticated matrix triple generation with scalable prime fields via optimized HE packing. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *2025 IEEE Symposium on Security and Privacy*, pages 2453–2471. IEEE Computer Society Press, May 2025.
- CKK20. Jung Hee Cheon, Dongwoo Kim, and Duhyeon Kim. Efficient homomorphic comparison methods with optimal complexity. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 221–256. Springer, Cham, December 2020.
- CKKS17. Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 409–437. Springer, Cham, December 2017.
- CLPX18. Hao Chen, Kim Laine, Rachel Player, and Yuhou Xia. High-precision arithmetic in homomorphic encryption. In Nigel P. Smart, editor, *CT-RSA 2018*, volume 10808 of *LNCS*, pages 116–136. Springer, Cham, April 2018.
- CPL26. Gyeongwon Cha, Dongjin Park, and Joon-Woo Lee. Improved radix-based approximate homomorphic encryption for large integers via lightweight bootstrapped digit carry. In Joan Daemen and Emmanuel Thomé, editors, *EUROCRYPT 2026, Part IV*, volume 16545 of *LNCS*, pages 243–273. Springer, Cham, May 2026.
- Det14. Peter Dettman. Re-write secp256k1_fe_sqrt and secp256k1_fe_inv to use a sliding window over blocks of

- 1s. <https://github.com/bitcoin-core/secp256k1/commit/f8ccd9befdb22824ef9a845a90e3db57c1307c11>, 2014. libsecp256k1 source code, commit `f8ccd9b`.
- DMPS24. Nir Drucker, Guy Moshkovich, Tomer Pelleg, and Hayim Shaul. BLEACH: Cleaning errors in discrete computations over CKKS. *Journal of Cryptology*, 37(1):3, January 2024.
- EY24. Ehsan Ebrahimi and Anshu Yadav. Strongly secure universal thresholdizer. In Kai-Min Chung and Yu Sasaki, editors, *ASIACRYPT 2024, Part III*, volume 15486 of *LNCS*, pages 207–239. Springer, Singapore, December 2024.
- Gen09. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.
- GGH⁺13. Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.
- GV25. Robin Geelen and Frederik Vercauteren. Fully homomorphic encryption for cyclotomic prime moduli. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part III*, volume 15603 of *LNCS*, pages 366–397. Springer, Cham, May 2025.
- GZ26. Mingyu Gao and Hongren Zheng. Fhe for simd arithmetic logic units with amortized $o(1)$ bootstrapping per ciphertext. In *Advances in Cryptology – CRYPTO 2026: 46th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17–20, 2026, Proceedings, Part II*, page 299–330, Berlin, Heidelberg, 2026. Springer-Verlag.
- JL26. Daehyun Jang and Junho Lee. Arithmetic for large-characteristic finite fields in CKKS. *Cryptology ePrint Archive*, Paper 2026/1958, 2026.
- Kim25. Jaehyung Kim. Efficient homomorphic integer computer from CKKS. *IACR TCHES*, 2025(4):873–898, 2025.
- Kim26. Jaehyung Kim. Faster logical operations from discrete CKKS. *Cryptology ePrint Archive*, Paper 2026/732, 2026.
- KLN⁺26. Taeseong Kim, Jonghoo Lee, Taeyeong Noh, Jung Hee Cheon, and Guillaume Hanrot. Latency-aware, high-throughput homomorphic AES evaluation with CKKS. *Cryptology ePrint Archive*, Paper 2026/1209, 2026.
- KN25. Jaehyung Kim and Taeyeong Noh. Modular reduction in CKKS. *IACR Communications in Cryptology*, 2(2), 2025.
- Lin17. Yehuda Lindell. Fast secure two-party ECDSA signing. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 613–644. Springer, Cham, August 2017.
- LL94. Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In Yvo Desmedt, editor, *CRYPTO’94*, volume 839 of *LNCS*, pages 95–107. Springer, Berlin, Heidelberg, August 1994.
- LLZD25. Yingjie Lyu, Zengpeng Li, Hong-Sheng Zhou, and Xudong Deng. Threshold ECDSA in two rounds. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *ACM CCS 2025*, pages 2937–2950. ACM Press, October 2025.
- LMWW26. Jiaming Liu, Shihe Ma, Anyu Wang, and Xiaoyun Wang. High-precision homomorphic ALU over arbitrary moduli with $o(1)$ bootstrapping. *Cryptology ePrint Archive*, Paper 2026/1869, 2026.

- LN18. Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1837–1854. ACM Press, October 2018.
- LPR10. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Berlin, Heidelberg, May / June 2010.
- MHP⁺25. Seonhong Min, Guillaume Hanrot, Jai Hyun Park, Alain Passetlègue, and Damien Stehlé. Distributed key generation for efficient threshold-CKKS. Cryptology ePrint Archive, Paper 2025/2057, 2025.
- Nat15. National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical Report FIPS PUB 202, National Institute of Standards and Technology, 2015.
- Nat23. National Institute of Standards and Technology. Digital signature standard (dss). Federal Information Processing Standards Publication (FIPS) NIST FIPS 186-5, U.S. Department of Commerce, Washington, D.C., February 2023.
- NHY⁺25. Chao Niu, Zhicong Huang, Zhaomin Yang, Yi Chen, Liang Kong, Cheng Hong, and Tao Wei. XBOOT: Free-XOR gates for CKKS with applications to transcribing. *IACR TCHES*, 2025(4):118–144, 2025.
- NP26. Ariel Nof and Luke Parker. Trout++: Robust asynchronous two-round ECDSA for arbitrary thresholds. Cryptology ePrint Archive, Paper 2026/1455, 2026.
- Por13. Thomas Pornin. Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). RFC 6979, 2013.
- PPS26. Jai Hyun Park, Alain Passetlègue, and Damien Stehlé. THED: Threshold dilithium from FHE. Cryptology ePrint Archive, Paper 2026/638, 2026.
- PZZ26. Chris Peikert, Doron Zarchy, and Guy Zyskind. High-precision exact fhe made simple, general, and fast. In Nadia Heninger and Mike Rosulek, editors, *Advances in Cryptology – CRYPTO 2026*, pages 525–557, Cham, 2026. Springer Nature Switzerland.
- RCB16. Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 403–428. Springer, Berlin, Heidelberg, May 2016.
- Rov26. Lorenzo Rovida. A flexible and polynomial framework for integer arithmetic in CKKS. Cryptology ePrint Archive, Paper 2026/450, 2026.
- SLWL26. Xuan Shen, Zhihao Li, Ruida Wang, and Xianhui Lu. Arithmetic-to-boolean conversion in ALU with o(1) bootstrapping via overflow cancellation. Cryptology ePrint Archive, Paper 2026/1840, 2026.
- Smi17. Brian Smith. The most efficient known addition chains for field element & scalar inversion for the most popular & most unpopular elliptic curves. <https://briansmith.org/ecc-inversion-addition-chains-01>, 2017. Section on secp256k1 scalar inversion.
- SS26a. Tim Seuré and Elias Suvanto. Faster homomorphic evaluation of SHA3. Poster at FHE.org Conference 2026, March 2026.

- SS26b. Tim Seuré and Elias Suvanto. Finite-field arithmetic in CKKS. Cryptology ePrint Archive, Paper 2026/1102, 2026.
- Zam22. Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. <https://github.com/zama-ai/tfhe-rs>.

A Additional Preliminaries

A.1 Deterministic ECDSA

Let $E(\mathbb{F}_p)$ be an elliptic curve defined by some equation over a finite field $\mathbb{F}_p$ with prime p . For points P and Q on the elliptic curve, there are two operations: point addition $(P + Q)$ and point doubling $2P$. Therefore, for a scalar $k \in \mathbb{Z}$, the scalar multiplication kP can be defined using point addition and doubling. In our implementation, the scalar multiplication operation uses the fixed-base window method, which requires the *bit* representation of the scalar k ; we also use homogeneous projective coordinates $(X : Y : Z)$ for a point (x, y) with $x = X/Z$ and $y = Y/Z$, so that point arithmetic proceeds without a modular inverse at each step. To recover from homogeneous projective coordinates to the usual affine coordinates (x, y) , we need the modular inverse operation Z^{-1} in $\mathbb{F}_p$. We also assume there is a base point $G = (x_G, y_G)$ for E such that the group generated by G is cyclic of prime order u .

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a widely recognized and deployed signature scheme. We use its deterministic variant, since our setting is distributed. Let $\mathcal{H}$ be the SHA3 hash function. The deterministic ECDSA works as follows (we omit the description of range checks here for brevity).

Definition 4 (Deterministic ECDSA). *The deterministic ECDSA scheme is a tuple of PPT algorithms $(\text{KeyGen}_{\text{SIG}}, \text{Sign}_{\text{SIG}}, \text{Verify}_{\text{SIG}})$ with the following functionalities:*

- $\text{KeyGen}_{\text{SIG}}(1^\lambda)$ takes the security parameter λ and outputs a random $\text{sk}_{\text{SIG}} \in \mathbb{F}_u$ as the private key and $\text{pk}_{\text{SIG}} = \text{sk}_{\text{SIG}} \cdot G$ as the public key.
- $\text{Sign}_{\text{SIG}}(\text{sk}_{\text{SIG}}, \mathcal{M})$ takes the private key sk_{SIG} and message $\mathcal{M}$, generates the nonce $k = \text{HMAC-DRBG}(\mathcal{H}(\mathcal{M}), \text{sk}_{\text{SIG}})$ based on $\mathcal{H}$, and computes $R = [k]G = (r_x, r_y)$. It then computes $r = r_x \pmod{u}$ and $s = k^{-1}(\mathcal{H}(\mathcal{M}) + r \cdot \text{sk}_{\text{SIG}}) \in \mathbb{F}_u$, and outputs $\sigma = (r, s)$ as the signature.
- $\text{Verify}_{\text{SIG}}(\mathcal{M}, (r, s))$ takes the message $\mathcal{M}$ and the signature (r, s) , calculates $(v_x, v_y) = V = s^{-1}\mathcal{H}(\mathcal{M}) \cdot G + s^{-1}r \cdot \text{pk}_{\text{SIG}}$ and outputs 1 if $v_x = r \pmod{u}$, and 0 otherwise.

From the perspective of computation, we remark that the elliptic curve point arithmetic happens in $\mathbb{F}_p$, but the calculation of s happens in $\mathbb{F}_u$. We also have a modular reduction operation from $r_x \in \mathbb{F}_p$ to $r \in \mathbb{F}_u$. The recovery of R from the projective coordinates requires computing $Z^{-1} \in \mathbb{F}_p$, and the computation of k^{-1} happens in $\mathbb{F}_u$. We use Fermat’s Little Theorem to compute them as $Z^{-1} \equiv Z^{p-2} \pmod{p}$ and $k^{-1} \equiv k^{u-2} \pmod{u}$.

A.2 Threshold FHE

Threshold FHE is an extension of FHE where the FHE secret key sk_{FHE} is distributed to η parties. By setting the threshold parameter θ , at least θ of the parties are required to collaboratively decrypt and recover the message. This approach can be used to distribute trust across multiple operators and prevent any operator from becoming a single point of failure. We use $\llbracket m \rrbracket$ for a message m to represent the homomorphic encryption of m . A general definition of threshold FHE is as follows.

Definition 5 (Threshold FHE). *A threshold FHE scheme is a tuple of PPT algorithms $(\text{KeyGen}_{\text{ThFHE}}, \text{Enc}_{\text{ThFHE}}, \text{Eval}_{\text{ThFHE}}, \text{PartDec}_{\text{ThFHE}}, \text{FinDec}_{\text{ThFHE}})$ with the following functionalities:*

- $\text{KeyGen}_{\text{ThFHE}}(1^\lambda, 1^\eta, 1^\theta)$ takes the security parameter λ , a number of parties η and a threshold θ and outputs the public parameters pp (e.g., the FHE scheme and encoding methods used), a public key pk_{ThFHE} , an evaluation key $\text{evk}_{\text{ThFHE}}$, and a partial secret key $\text{sk}_{\text{ThFHE}}^{(i)}$ for each party i with $i \in [\eta]$. All algorithms below implicitly take pp as input.
- $\text{Enc}_{\text{ThFHE}}(\mathcal{M}, \text{pk}_{\text{ThFHE}})$ takes the message $\mathcal{M}$ and the public key pk_{ThFHE} . It encodes and encrypts $\mathcal{M}$ using the public key and outputs a ciphertext $\llbracket \mathcal{M} \rrbracket$.
- $\text{Eval}_{\text{ThFHE}}(\{\llbracket \mathcal{M} \rrbracket\}, \mathcal{C}, \text{evk}_{\text{ThFHE}})$ takes the ciphertexts $\{\llbracket \mathcal{M} \rrbracket\}$ and the evaluation circuit $\mathcal{C}$, evaluates the circuit using the evaluation key, and outputs the evaluated ciphertexts $\{\llbracket \mathcal{M}' \rrbracket\}$.
- $\text{PartDec}_{\text{ThFHE}}(\llbracket \mathcal{M} \rrbracket, \text{sk}_{\text{ThFHE}}^{(i)})$ for party i takes the ciphertext $\llbracket \mathcal{M} \rrbracket$ and its secret key $\text{sk}_{\text{ThFHE}}^{(i)}$, decrypts the ciphertext using the partial secret key and outputs the partial result $\text{part}_{\text{ThFHE}}^{(i)}(\mathcal{M})$.
- $\text{FinDec}_{\text{ThFHE}}(\{\text{part}_{\text{ThFHE}}^{(i)}(\mathcal{M})\}_{i \in S, |S|=\theta})$ takes θ different partial results $\text{part}_{\text{ThFHE}}^{(i)}(\mathcal{M})$ and recovers $\mathcal{M}$.

We focus on threshold CKKS in this paper. For the key generation process, we can either rely on a trusted key dealer, or use the recently proposed distributed key generation (DKG) protocol for CKKS to generate a main secret key and an encapsulated sparse key [MHP⁺25]. After the key generation process, the threshold CKKS homomorphic evaluation process is almost the same as regular CKKS evaluation. Note that it also satisfies the encapsulated sparse key requirement of the Gao–Zheng FHE scheme.

In the partial decryption process, the i -th party holding secret key $\text{sk}_{\text{ThFHE}}^{(i)}$ can decrypt $\llbracket m \rrbracket$ using its secret share to obtain $\text{part}_{\text{ThFHE}}^{(i)}(m)$ and broadcast the result to all other parties. If partial decryption results from at least θ parties are received, the result can be recovered using $\text{FinDec}_{\text{ThFHE}}$. The decryption process only takes *one round* of communication.

To prevent the partial decryption results from revealing the evaluation noise and then exposing the secret key, a large amount of fresh noise is added to the partial decryption result. This technique is called *noise flooding* [Gen09].

We employ the partial decryption process and the noise flooding procedure as specified in THED [PPS26].

For security notions related to multi-party computation, we focus on semi-honest security in this paper. We can achieve malicious security by adding a zero-knowledge proof of the honest execution of the partial decryption process for each party, as discussed in THED [PPS26].

A.3 Threshold Signature from Threshold FHE

A threshold signature can be constructed using threshold FHE to achieve a one-round signing protocol [AJL⁺12, BGG⁺18, ASY22]. Intuitively, the signing key is homomorphically encrypted. Each party then homomorphically performs the signing process and obtains an encrypted signature. To recover the signature, each signer partially decrypts the result and broadcasts it to the other signers. If at least θ partial decryption results are collected, the signature is recovered. We summarize the threshold signature scheme with a one-round signing protocol as follows:

Definition 6 (Threshold Signature with a One-Round Signing Protocol). *A threshold signature scheme with a one-round signing protocol is a tuple of PPT algorithms $(\text{KeyGen}_{\text{ThSIG}}, \text{PartSign}_{\text{ThSIG}}, \text{Combine}_{\text{ThSIG}}, \text{Verify}_{\text{SIG}})$ with the following specifications:*

- $\text{KeyGen}_{\text{ThSIG}}(1^\lambda, 1^\eta, 1^\theta)$ takes the security parameter λ , a number of parties η and a threshold θ and outputs the public parameters pp , a public key pk_{SIG} , and a partial signing key $\text{sk}_{\text{ThSIG}}^{(i)}$ for each party i with $i \in [\eta]$.
- $\text{PartSign}_{\text{ThSIG}}(\text{sk}_{\text{ThSIG}}^{(i)}, \mathcal{M})$ takes a partial signing key $\text{sk}_{\text{ThSIG}}^{(i)}$ and a message $\mathcal{M}$, and outputs $\text{part}_{\text{ThSIG}}^{(i)}(\mathcal{M})$.
- $\text{Combine}_{\text{ThSIG}}(\{\text{part}_{\text{ThSIG}}^{(i)}(\mathcal{M})\}_{i \in S, |S|=\theta})$ takes θ partial signing results $\{\text{part}_{\text{ThSIG}}^{(i)}(\mathcal{M})\}_{i \in S, |S|=\theta}$, and outputs the signature σ for $\mathcal{M}$.
- $\text{Verify}_{\text{SIG}}(\mathcal{M}, \sigma)$ is the same as defined in Definition 4.

B Masked 3-to-2 Compressor

Let m_x, m_y, m_z be the masks for the input x, y, z , with x from the bit string with the largest number of bits masked out; we can replace m_y and m_z by m_x in the second term of the expression for c , and we still consume only two multiplicative levels.

$$\begin{aligned} c &:= ym_yzm_z + (y + z - 2yz)xm_x \\ s &:= xm_x + ym_y + zm_z - 2c \end{aligned}$$

C Proofs

Before proving the lemmas, we first give the expressions for M and M^+ . Notice that the polynomial $tI + [m]_t$ has degree at most n , where the coefficient of X^n is I_{n-1} . Since $M = tI + [m]_t$ under the ring $\mathcal{Z}_{F_q} = \mathbb{Z}[X]/\langle F_q(X) \rangle$, and $F_q(X)$ is monic, M can be expressed as

$$M = tI + [m]_t - I_{n-1}F_q(X) \quad (1)$$

Therefore, evaluating at $X = 2$ gives $M(2) = m - I_{n-1}q$. Similarly, for the polynomial M^+ , we have

$$M^+(2) = M(2) + \text{val}_2(\mathbf{r}_H) = m - I_{n-1}q + H(2^n - 1) + [H(1 - 2^n)]_q \quad (2)$$

C.1 Proof of Lemma 2

Proof. Since $M^+(2) = \text{val}_2(\mathbf{b}) + A \cdot 2^n$, and $\mathbf{b}$ is a length- n binary string, we have $A = \lfloor \frac{M^+(2)}{2^n} \rfloor$. Let $C = H(2^n - 1) + [H(1 - 2^n)]_q$. By equation 2, we have

$$C - K \cdot q \leq M^+(2) \leq C + K \cdot q + (2^n - 1)$$

Therefore, if we let $C_0 = \lfloor \frac{C - K \cdot q}{2^n} \rfloor$, we have

$$C_0 \cdot 2^n \leq M^+(2) \leq (C_0 \cdot 2^n + 2^n - 1) + 2Kq + (2^n - 1) < (C_0 + 2K + 2) \cdot 2^n$$

which gives $C_0 \leq A \leq C_0 + 2K + 1$. Moreover, $C \geq H(2^n - 1) \geq (4K + 1)q > Kq$, hence $C_0 \geq 0$.

C.2 Proof of Lemma 3

Proof. First, we have $A_0 \in [0, q)$, so $A_1 = A_0 + (2^n - q) \in [2^n - q, 2^n)$ and $A_0 + (2^n - 2q) < A_1 < 2^n$, therefore we have $A_2 \in [0, 2^n)$ as well.

Notice that $\text{val}_2(\mathbf{b}) \in [0, 2^n)$, thus $s_i = \text{val}_2(\mathbf{b}) + A_i \in [0, 2 \cdot 2^n)$ for $i = 0, 1, 2$. Therefore we get $c_i \in \{0, 1\}$ for $i = 0, 1, 2$.

Denote the sum by $r = x_0 + c_1(x_1 - x_0) + c_2(x_2 - x_1)$. We prove the equation by considering the possible ranges of $s_0 = \text{val}_2(\mathbf{b}) + A_0$.

(1) When $s_0 \in [0, q)$, $m = s_0$, and we have $s_1 = s_0 + (2^n - q) < 2^n$, and $s_2 < s_1 < 2^n$. Thus $c_1 = c_2 = 0$, and $r = x_0 = s_0 = m$;

(2) When $s_0 \in [q, 2q)$, $m = s_0 - q$, and we have $s_1 = s_0 + (2^n - q) \geq 2^n$, and $s_2 = \max(\text{val}_2(\mathbf{b}), s_0 + (2^n - 2q)) < 2^n$. Thus $c_1 = 1$ and $c_2 = 0$, and $r = x_1 = s_1 - 2^n = s_0 - q = m$;

(3) When $s_0 \in [2q, 3q)$, $m = s_0 - 2q$, and we have $s_1 = s_0 + (2^n - q) \geq 2^n$, and $s_2 \geq s_0 + (2^n - 2q) \geq 2^n$. Thus $c_1 = c_2 = 1$, and then $r = x_2 = s_2 - 2^n = s_0 - 2q = m$.

Therefore we have proved $r = m$ in all cases, finishing the proof of the lemma.

C.3 Proof of Lemma 4

Proof. Since $M^+(2) = \text{val}_2(\mathbf{b}) + A \cdot 2^n$, by equation 2 we have

$$A \cdot 2^n \leq M^+(2) \leq Kq + (H+1)(2^n - 1) + (q-1) < (K+H+2) \cdot 2^n$$

which gives $A \leq K+H+1 = 5K+2$. Then we have

$$A_0 = [A \cdot 2^n]_q \leq A \cdot 2^n - Aq \leq (5K+2) \cdot (2^n - q)$$

since $q \geq \left(1 - \frac{1}{5K+4}\right) 2^n$, we have

$$A_0 + (2^n - 2q) \leq (5K+3)2^n - (5K+4)q \leq 0$$

Therefore we have $A_2 = \max(0, A_0 + (2^n - 2q)) = 0$, which proves the lemma.

D Existence of Supporting Polynomials for Odd Prime Moduli

We construct supporting polynomials for every odd prime, with the degree and root signature required in Section 4.

Definition 7 (Radix-4 polynomial). Set $H_1(X) = 1$. For an odd integer $m > 1$, write

$$\frac{m-1}{2} = \sum_{j=0}^{k-1} u_j 4^j, \quad u_j \in \{0, 1, 2, 3\}, \quad u_{k-1} \neq 0,$$

and define

$$P_0 = 0, \quad P_1 = X^2 - X, \quad P_2 = X^2, \quad P_3 = X^2 + X,$$

$$H_m(X) = 1 + \sum_{j=0}^{k-1} X^{2^j} P_{u_j}(X).$$

Thus $H_m(2) = m$, since $P_u(2) = 2u$.

Theorem 1. Let $q \in (2^{n-1}, 2^n)$ be an odd prime. Then

$$F_q(X) = \begin{cases} H_q(X), & n \text{ even}, \\ X^n - H_{2^n-q}(X), & n \text{ odd} \end{cases}$$

is a supporting polynomial for q : it is monic of degree n , has coefficients in $\{-1, 0, 1\}$, satisfies $F_q(2) = q$, and has n distinct roots, exactly $n \bmod 2$ of which are real.

We establish the real-root counts and the bounds needed to exclude repeated roots in three lemmas.

Lemma 5 (Roots of the radix-4 polynomial). *For every positive odd integer m , $H_m(x) > 0$ for $x \in \mathbb{R}$, and every complex root z of H_m satisfies $\operatorname{Re} z < 3/2$.*

Proof. The case $m = 1$ is immediate. For $m > 1$, write $\deg H_m = 2k$. If $|x| \geq 1$, every $x^{2j} P_{u_j}(x)$ is nonnegative. For $t = |x| < 1$,

$$H_m(x) \geq 1 - (1-t) \sum_{j=0}^{k-1} t^{2j+1} = 1 - \frac{t(1-t^{2k})}{1+t} > \frac{1}{2}.$$

This proves positivity.

Suppose $H_m(z) = 0$ and $\operatorname{Re} z \geq 3/2$. Since H_m is monic with coefficients of modulus at most one, $R = |z| < 2$: otherwise $R^{2k} \leq \sum_{i=0}^{2k-1} R^i < R^{2k}$. By conjugation, write $z = Re^{i\theta}$ with

$$\frac{3}{2} \leq R < 2, \quad R \cos \theta \geq \frac{3}{2}, \quad 0 \leq \theta < \frac{\pi}{4}.$$

Reversing the paired coefficients gives

$$\frac{H_m(z)}{z^{2k}} = 1 + \frac{o_0}{z} + \sum_{j \geq 1} B_j, \quad B_j = e_j z^{-2j} + o_j z^{-2j-1},$$

where $o_0 \in \{-1, 0, 1\}$, $e_j \in \{0, 1\}$, and $|o_j| \leq e_j$. The constant term is represented by $(e_k, o_k) = (1, 0)$; $e_j = o_j = 0$ for $j > k$. Consequently, for $J \geq 1$,

$$\left| \sum_{j \geq J} B_j \right| \leq T_J(R) := \sum_{\ell \geq 2J} R^{-\ell} = \frac{1}{R^{2J-1}(R-1)}. \quad (3)$$

Also,

$$\operatorname{Re} B_j = R^{-2j-1} (e_j R \cos(2j\theta) + o_j \cos((2j+1)\theta)).$$

If $0 \leq \theta \leq \pi/8$, then

$$0 \leq \cos 3\theta \leq \cos 2\theta, \quad \cos 4\theta \geq 0, \quad -\frac{1}{2} < \cos 5\theta \leq \cos 4\theta.$$

Hence $\operatorname{Re} B_1 \geq 0$ and $\operatorname{Re} B_2 \geq -1/(2R^5)$, so

$$\operatorname{Re} \frac{H_m(z)}{z^{2k}} \geq 1 - R^{-1} - \frac{1}{2R^5} - T_3(R) \geq \frac{1}{3} - \frac{16}{243} - \frac{64}{243} = \frac{1}{243} > 0.$$

If $\pi/8 \leq \theta < \pi/4$, put $t = \cos \theta \geq 3/4$. For $\theta \leq \pi/6$, $\operatorname{Re} B_1 \geq 0$ as above. For $\theta \geq \pi/6$, $R^2 \geq 3$ and

$$0 \leq -\cos 3\theta = t(3-4t^2) \leq 3t(1-t) \leq R^2(1-t),$$

so $\operatorname{Re} B_1 \geq -(1-t)/R$. In both cases,

$$\operatorname{Re}(1 + o_0/z + B_1) \geq 1 - t/R - (1-t)/R = 1 - R^{-1}.$$

Moreover, $R \geq 3/(2 \cos(\pi/8)) > \varphi = (1 + \sqrt{5})/2$; indeed,

$$\varphi^2 \cos^2 \frac{\pi}{8} = \frac{(3 + \sqrt{5})(2 + \sqrt{2})}{8} < \frac{(3 + 9/4)(2 + 10/7)}{8} = \frac{9}{4}.$$

It follows that

$$\operatorname{Re} \frac{H_m(z)}{z^{2k}} \geq 1 - R^{-1} - T_2(R) = \frac{R^2(R-1)^2 - 1}{R^3(R-1)} > 0.$$

Both cases contradict $H_m(z) = 0$.

Lemma 6 (Real root of the odd-degree construction). *Let $n = 2k + 1 \geq 3$ and let δ be odd with $1 \leq \delta < 2^{n-1}$. Then $F = X^n - H_\delta$ has exactly one real root, and it is simple.*

Proof. Allowing leading zero pairs, write

$$H_\delta = 1 + \sum_{j=0}^{k-1} (e_j X^{2j+2} + o_j X^{2j+1}), \quad e_j \in \{0, 1\}, \quad |o_j| \leq e_j.$$

Lemma 5 gives $F(x) < 0$ for $x \leq 0$. For $0 < x < 1$,

$$H_\delta(x) \geq 1 - (1 - x) \sum_{j=0}^{k-1} x^{2j+1} = \frac{1 + x^n}{1 + x} > x^n.$$

Thus F has no real root below 1. Set $S_j = \sum_{\ell=j}^{k-1} (e_\ell + o_\ell)$, with $S_k = 0$. Synthetic division gives

$$F = (X - 1)Q - S_0, \quad Q = \sum_{i=0}^{2k} b_i X^i, \quad \begin{cases} b_{2j} = 1 - S_j, & 0 \leq j \leq k, \\ b_{2j+1} = 1 - S_{j+1} - e_j, & 0 \leq j < k. \end{cases}$$

Since $S_j \geq S_{j+1}$ are integers and $e_j \in \{0, 1\}$, the descending coefficient sequence of Q has at most one sign change, from positive to negative. Hence some $r \geq 0$ satisfies $b_i(i - r) \geq 0$ for all i . If $S_0 = 0$, all $b_i \geq 0$ and $b_{2k} = 1$, so 1 is the unique real root and $F'(1) = Q(1) > 0$. If $S_0 > 0$, then for $x > 1$,

$$\left(\frac{F(x)}{(x-1)x^r} \right)' = \sum_i b_i(i-r)x^{i-r-1} + S_0 \frac{((x-1)x^r)'}{((x-1)x^r)^2} > 0.$$

This quotient tends to $-\infty$ as $x \downarrow 1$ and is positive for sufficiently large x . Its unique zero, and hence that of F , is simple.

Lemma 7 (Nonreal roots of the odd-degree construction). *Under the hypotheses of Lemma 6, every nonreal root z of $F = X^n - H_\delta$ satisfies $\operatorname{Re} z < 3/2$.*

Proof. Suppose $F(z) = 0$ with $z = a + ib = Re^{i\theta}$, $a \geq 3/2$, and $b > 0$. Conjugation covers $b < 0$. Since F is monic with coefficients of modulus at most one, $R < 2$, as $R \geq 2$ would imply $R^n \leq \sum_{i=0}^{n-1} R^i < R^n$. The inequality $(\delta - 1)/2 < 2 \cdot 4^{k-1}$ makes the highest base-4 digit zero or one. Reversing the coefficient pairs therefore gives

$$z = r(z) := \frac{H_\delta(z)}{z^{2k}} = E(1 - z^{-1}) + \sum_{j \geq 1} B_j, \quad B_j = e_j z^{-2j} + o_j z^{-2j-1}, \quad (4)$$

where $E, e_j \in \{0, 1\}$, $|o_j| \leq e_j$, $(e_k, o_k) = (1, 0)$, and $e_j = o_j = 0$ for $j > k$. In particular,

$$\left| \sum_{j \geq 1} B_j \right| \leq \sum_{\ell \geq 2} R^{-\ell} = \frac{1}{R(R-1)}.$$

If $E = 0$, this gives $R \leq 4/3 < 3/2 \leq R$, a contradiction. Thus $E = 1$. Taking real parts in (4) yields

$$a(1 + R^{-2}) \leq 1 + \frac{1}{R(R-1)}, \quad (R-1)(R^2 + 1) \leq 2.$$

If $R^2 \geq 5/2$, the latter inequality would give $R \leq 11/7$ and $R^2 \leq 121/49 < 5/2$. Consequently,

$$R^2 < \frac{5}{2}, \quad b^2 = R^2 - a^2 < \frac{1}{4}, \quad 0 < \theta < \tan \theta = \frac{b}{a} < \frac{1}{3}.$$

For $\ell \geq 1$, define

$$h_\ell = -\frac{\operatorname{Im} z^{-\ell}}{b} = \frac{\sin(\ell\theta)}{R^{\ell+1} \sin \theta}.$$

For $1 \leq j \leq 4$, $(2j+1)\theta < 3 < \pi$. Since $\sin x/x$ decreases on $(0, \pi)$,

$$0 < \sin((2j+1)\theta) \leq \frac{2j+1}{2j} \sin(2j\theta) \leq R \sin(2j\theta).$$

Thus $h_{2j} \geq |h_{2j+1}|$ and

$$\frac{\operatorname{Im} B_j}{b} = -e_j h_{2j} - o_j h_{2j+1} \leq 0 \quad (1 \leq j \leq 4).$$

Factoring $z^\ell - \bar{z}^\ell$ gives

$$|h_\ell| = \frac{|z^\ell - \bar{z}^\ell|}{|z - \bar{z}| R^{2\ell}} \leq \frac{\ell}{R^{\ell+1}}.$$

Therefore

$$\frac{\operatorname{Im} r(z)}{b} \leq R^{-2} + \sum_{\ell \geq 10} \frac{\ell}{R^{\ell+1}} \leq \frac{4}{9} + 24 \left(\frac{2}{3} \right)^{10} = \frac{16940}{19683} < 1,$$

contradicting $z = r(z)$.

Proof (of Theorem 1). For even $n = 2k$, $4^{k-1} \leq (q-1)/2 < 2 \cdot 4^{k-1}$, so H_q is monic of degree n . For odd $n = 2k+1$, $\delta = 2^n - q$ is odd with $1 \leq \delta < 4^k$, so $\deg H_\delta \leq n-1$ and $X^n - H_\delta$ is monic of degree n . In both cases the coefficients lie in $\{-1, 0, 1\}$ and $F_q(2) = q$. Lemmas 5 and 6 give the required real-root count and simplicity of every real root.

If F_q had a repeated root, Gauss's lemma would give a nonconstant monic $G \in \mathbb{Z}[X]$ with $G^2 \mid F_q$. No root of G is real; Lemmas 5 and 7 place every root α of G in $\text{Re } \alpha < 3/2$. Hence

$$|2 - \alpha|^2 - |1 - \alpha|^2 = 3 - 2 \text{Re } \alpha > 0, \quad |G(2)| > |G(1)| \geq 1.$$

But $G(2)^2 \mid F_q(2) = q$ forces $|G(2)| = 1$, a contradiction. Thus all roots of F_q are distinct, completing the proof.

E Parallel Point Addition in Homogeneous Coordinates

We describe the packed addition tree used in Section 5.3. All coordinate arithmetic is over $\mathbb{F}_p$, where $p = 2^{256} - 2^{32} - 977$. We use homogeneous coordinates on the secp256k1 curve

$$E : Y^2 Z = X^3 + 7Z^3, \quad (X : Y : Z) \mapsto (X/Z, Y/Z) \quad (Z \neq 0), \quad \mathcal{O} = (0 : 1 : 0).$$

Triples differing by a common nonzero factor represent the same point. Each coordinate occupies one logical arithmetic word representing a 256-bit field element; word indices below are not individual CKKS slot indices. We write $C[i]$ for the field value represented at word i . The selected window points $P_w = [d_w 16^w]G$ enter the tree as

$$C_0[4w : 4w + 3] = (X_w, Y_w, Z_w, 0), \quad 0 \leq w < 64, \quad (5)$$

where $C[a : b]$ denotes the inclusive range of logical words $a, \dots, b$. For $d_w = 0$, the selected triple is $(0, 1, 0)$; otherwise it is $(x(P_w), y(P_w), 1)$. Thus all 64 points, including zero windows, are valid curve points packed into one arithmetic ciphertext with 256 logical words.

Complete addition with two multiplication layers. We specialize the complete addition law of Renes, Costello, and Batina [RCB16, Algorithm 7] to the curve constant 7. For $P_1 = (X_1 : Y_1 : Z_1)$ and $P_2 = (X_2 : Y_2 : Z_2)$, compute

$$\begin{aligned} t_0 &= X_1 X_2, & t_1 &= Y_1 Y_2, & t_2 &= Z_1 Z_2, \\ q_3 &= (X_1 + Y_1)(X_2 + Y_2), \\ q_4 &= (Y_1 + Z_1)(Y_2 + Z_2), \\ q_5 &= (X_1 + Z_1)(X_2 + Z_2). \end{aligned} \quad (6)$$

These six products are independent. Linear combinations give

$$\begin{aligned} a &= q_3 - t_0 - t_1, & b &= t_1 - 21t_2, & c &= t_1 + 21t_2, \\ d &= q_4 - t_1 - t_2, & e &= 21(q_5 - t_0 - t_2), & f &= 3t_0. \end{aligned} \quad (7)$$

The second multiplication layer computes the six products in

$$X_3 = ab - de, \quad Y_3 = bc + ef, \quad Z_3 = cd + fa. \quad (8)$$

Thus each point addition uses twelve field products arranged in two layers. The output represents $P_1 + P_2$. Completeness applies to all pairs in $E(\mathbb{F}_p)$, whose order is the odd prime u , including doubling, inverse pairs, and infinity. In particular, substituting $P_1 = \mathcal{O}$ gives $(X_3, Y_3, Z_3) = Y_2(X_2, Y_2, Z_2)$. The factor Y_2 is nonzero: a finite point with $Y_2 = 0$ would have order two. Hence zero windows require no conditional selection during point addition.

Packed schedule. Let $s \in \{4, 8, 16, 32, 64, 128\}$ be the spacing between input point records at one tree stage. Pair j , for $0 \leq j < 256/(2s)$, occupies the block starting at $v = 2sj$: its points are stored at $v + (0, 1, 2)$ and $v + s + (0, 1, 2)$. The following linear maps act independently on every such block and set unspecified output words to zero:

- $\text{Pack}_s^{(h)}$, for $h \in \{0, 1\}$, reads the triple (X, Y, Z) at $v + hs + (0, 1, 2)$ and writes $(X, Y, Z, X + Y, Y + Z, X + Z)$ at $v + (0, \dots, 5)$.
- Cycle_s reads $(t_0, t_1, t_2, q_3, q_4, q_5)$ at $v + (0, \dots, 5)$ and writes (a, b, c, d, e, f, a) at $v + (0, \dots, 6)$, using (7).
- Out_s reads $(v_0, \dots, v_5)$ at $v + (0, \dots, 5)$ and writes $(v_0 - v_3, v_1 + v_4, v_2 + v_5)$ at $v + (0, 1, 2)$.

All three maps use public plaintext masks, rotations, and additions. Write $\rho_1(C)[i] = C[(i + 1) \bmod 256]$ for a rotation by one logical word, and $\odot$ for a short packed ciphertext multiplication, whose encoding correction is included in the following arithmetic refresh. After the first multiplication, Cycle_s , and refresh, the intended field values satisfy

$$U[v : v+7] = (a, b, c, d, e, f, a, 0), \quad (U \odot \rho_1(U))[v : v+7] = (ab, bc, cd, de, ef, fa, 0, 0).$$

The zero padding prevents products between distinct blocks, including at the cyclic boundary. Thus one packed multiplication evaluates an entire multiplication layer for all active point pairs.

Here each $\text{A2A}^{(p)}$ is the arithmetic refresh for a product ciphertext, applied after the indicated linear map. It includes the product-encoding correction and restores the arithmetic encoding while preserving the computed field values. The logical arithmetic capacity remains 256 words at every stage.

Correctness and cost. At the start of stage ℓ , the record at $4 \cdot 2^\ell j$ represents

$$S_j^{(\ell)} = \sum_{w=2^\ell j}^{2^{\ell(j+1)-1}} P_w.$$

Equations (6)–(8) and the specified routing show that the stage replaces each pair by $S_{2j}^{(\ell)} + S_{2j+1}^{(\ell)} = S_j^{(\ell+1)}$, stored at $2sj = 4 \cdot 2^{\ell+1} j$. Induction therefore gives $S_0^{(6)} =$

Algorithm 1 Packed homogeneous addition tree

Require: Arithmetic ciphertext C_0 with the layout in (5).

Ensure: Arithmetic ciphertext containing coordinates of $\sum_{w=0}^{63} P_w$ in words 0, 1, 2; all other words are zero.

```
1:  $C \leftarrow C_0$ 
2: for  $\ell = 0, \dots, 5$  do
3:    $s \leftarrow 4 \cdot 2^\ell$ 
4:    $L \leftarrow \text{Pack}_s^{(0)}(C); R \leftarrow \text{Pack}_s^{(1)}(C)$ 
5:    $T \leftarrow L \odot R$ 
6:    $U \leftarrow \text{A2A}^{(p)}(\text{Cycle}_s(T))$ 
7:    $V \leftarrow U \odot \rho_1(U)$ 
8:    $C \leftarrow \text{A2A}^{(p)}(\text{Out}_s(V))$ 
9: end for
10: return  $C$ 
```

$\sum_w [d_w 16^w]G = [k]G$. Each stage uses two packed ciphertext multiplications and two arithmetic refreshes. The six stages perform 63 point additions using 12 packed multiplications and 12 arithmetic refreshes, with no field inversion inside the tree. These counts exclude Boolean table selection, its refreshes, the initial B2A, and the final affine conversion. They count multiplication layers and refresh calls, not total CKKS levels: the routing and linear transformations also consume levels. For a valid nonce $1 \leq k < u$, the final point is finite, and ECDSA recovers its affine x -coordinate as $X_R Z_R^{-1} \bmod p$.