TUTTI: Parallel CKKS Bootstrapping in under 3 ms on CPU

Robin Köstler[1] and Antonio Guimarães[2]

[1] University of Luxembourg, Luxembourg
[2] IMDEA Software Institute, Madrid, Spain

Abstract. This paper introduces TUTTI, a new CKKS bootstrapping procedure that outperforms previous low-latency solutions in multi-threaded latency for any number of messages and, most notably, has a lower latency than TFHE when bootstrapping a single one. Our bootstrapping builds upon SPRU (CK, TCHES 2027), improving it with new methods for exploiting parallelism and increasing the number of messages, as well as with a refined security analysis, of independent interest, of the block-binary secret keys of LMSS (AsiaCCS 2023). TUTTI bootstraps 4096 real-number messages in 46 to 83.4 ms on 192 threads to restore 1 to 9 multiplicative levels, outperforming SHIP (CHKS, Eurocrypt 2025) by a factor of up to 3 on the same machine. Unlike SHIP, TUTTI gains immediately from bootstrapping fewer messages, with a 256-message bootstrapping taking just 15 to 34 ms on 32 threads. For a single message, TUTTI takes 4.6 ms on 32 threads, whereas TFHE-rs needs 7.3 ms for a 4-bit bootstrapping on the same machine and thread count. We show that this advantage persists across different architectures: on an Apple M5 Pro laptop (ARM), TUTTI takes 2.4 ms on 14 threads, against 4.5 ms for TFHE-rs. Differences between CKKS and lightweight schemes such as TFHE remain on several fronts, from functionality to key sizes and security assumptions, but our work shows that multi-threaded latency is no longer one of them, closing a long-standing gap in the practical FHE literature.

Keywords: Fully homomorphic encryption · CKKS · Bootstrapping · Low latency · Sparse secrets

1 Introduction

Latency and throughput have long been the two axes that govern performance in fully homomorphic encryption (FHE). Both are set in practice by *bootstrapping* [Gen09], an operation that refreshes encryption noise and allows computations of unbounded depth. The idea of evaluating homomorphic operations in batches was first introduced by Smart and Vercauteren [SV10,SV14], who brought the concept of Single-Instruction-Multiple-Data (SIMD) [Fly66,Fly72] to FHE. Batching was a breakthrough in throughput, paving the way for FHE with only polylogarithmic overhead [GHS12b] and for the first homomorphic evaluations of AES on many blocks at once [GHS12c]. It came at the cost of latency, however, since the large parameters that SIMD schemes require made a bootstrapping take on the order of minutes for a thousand messages [HS15].

Brakerski and Vaikuntanathan [BV14] took the opposite direction and studied bootstrapping with polynomially bounded parameters. Their work was followed by a family of lightweight schemes built on the bootstrapping of Alperin-Sheriff and Peikert (AP) [AP14], such as FHEW [DM15] and TFHE [CGGI16,CGGI20], which today bootstrap a single message in just a few milliseconds [Zam26]. This created a fundamental divide in the practical FHE literature, and applications had to choose between low latency and high throughput, with few options in between. Several lines of research have since worked to close this divide. The AP bootstrapping framework was extended to batched settings with improved throughput [MS18,LW23a,LW23b,GPvL23,DKMS24,GP25,PDH+25]; scheme-switching techniques move data between schemes [BGGJ20,LHH+21,BCK+23]; and, more recently, a series of works [BCKS24,CK27,CHKS25,CS25,CJK26] improved the latency of the Cheon–Kim–Kim–Song (CKKS) scheme [CKKS17].

CKKS is a particularly promising scheme that excels in SIMD performance on fixed-point arithmetic and has gained significant traction thanks to its numerous applications to privacy-preserving machine learning [KSK+18,LLL+22,EGR25,MYJK25]. Its fixed-point encoding has also been shown to enable bootstrapping designs that vastly outperform the digit-extraction-based bootstrapping [GHS12a,HS15,CH18,GV23,GIKV23,MHWW24] of schemes such as BGV [BGV12] and BFV [Bra12,FV12], to the point that BFV ciphertexts are now bootstrapped faster through CKKS [KSS24,Kim25]. Still, conventional CKKS bootstrapping takes about nine seconds on a single core to bootstrap 2^{15} real messages [BCKS24], and it parallelizes poorly: on 32 cores, it still takes 1.09 s for 2^{15} messages, and 624 ms for 2^{14} [CHKS25]. This imposes significant limitations on latency-sensitive applications, such as private neural-network inference, where bootstrapping is either avoided through client interaction [JVC18,PZM+24] or accounts for most of the inference time [AB24,MYJK25].

To address this problem, Cheon, Hanrot, Kim and Stehlé introduced SHIP [CHKS25], a parallel bootstrapping for CKKS that requires a relatively low multiplicative depth and smaller parameters, thus enabling the bootstrapping of 4096 real messages in 215 ms on a 32-core CPU. Concurrently, Coron and Köstler introduced SPRU [CK27], a method that bootstraps significantly fewer messages in exchange for improved latency, refreshing a single one 5× faster than conventional bootstrapping. Both are based on techniques that embed $\mathbb{Z}_q$ into the complex roots of unity, which removes the usual polynomial approximation of the modular reduction function and, with it, most of the multiplicative depth of the bootstrapping circuit. Two main limitations persist in this line of work.

1. Schemes are still designed for a specific number of messages, and provide limited advantage away from it. For a polynomial ring of degree N and a sparse secret of Hamming weight h, SHIP performs $O(h \log(N/h))$ ring operations for any number of messages [CHKS25], and only its linear transform becomes lighter when fewer messages are bootstrapped. Concretely, we estimate that reducing the number of messages improves its latency by at most 15%. SPRU, in contrast, refreshes at most $N/(2h)$ messages per run, and its cost grows linearly with their number: it is overtaken by conventional bootstrapping from about 128 messages on [CK27].
2. Even for a single message, the fastest existing CPU implementations remain more than an order of magnitude slower than state-of-the-art parallel TFHE bootstrapping. On 32 threads of a 192-core machine, our implementation of SHIP takes at best 132 ms for 4096 messages (Table 6), and bootstrapping a single message instead saves it at most 15%, which still leaves it more than 15× above the 7.3 ms of a 4-bit TFHE-rs bootstrapping on the same machine and thread count [Zam22].

In this paper, we build upon SPRU to introduce TUTTI, a new bootstrapping procedure that lifts both limitations and achieves state-of-the-art multi-threaded latency for any number of messages. Figure 1 summarizes the latency of TUTTI for a varying number of messages and compares it with SHIP [CHKS25], PaCo [CS25] and TFHE-rs [Zam22].

1.1 Contributions and technical overview

Let us start with a high-level overview of SPRU, as introduced by Coron and Köstler [CK27]. In a ring $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N+1)$, a CKKS ciphertext $(c_0, c_1) \in \mathcal{R}_q^2$ encrypts a message m under a secret key s if $m \approx c_0 + c_1 \cdot s \bmod q$, up to a small error. For a binary s, each coefficient of m is thus a sum of coefficients of c_0 and c_1 (up to sign) selected by the bits of s, followed by a reduction modulo q. Conventional bootstrapping procedures evaluate that reduction with a high-degree polynomial approximation, which requires a large multiplicative depth. SPRU instead maps every value x of the decryption equation to the complex root of unity $e^{2\pi \mathrm{I} x/q}$, on which the reduction modulo q comes for free, since adding any multiple of q to x leaves

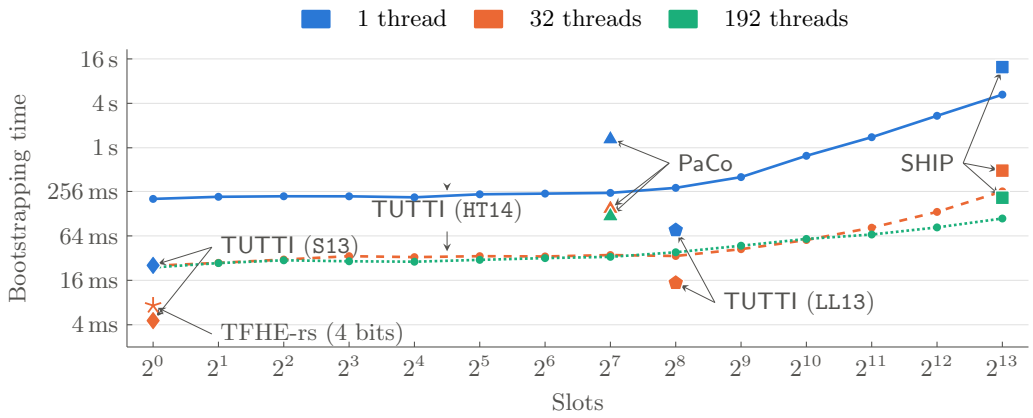


Fig. 1. Bootstrapping latency by the number of bootstrapped messages (fixed-point real-number slots). TUTTI on `S13` and `LL13` leaves 1 multiplicative level to the application, PaCo leaves 7, and TUTTI on `HT14` and SHIP on `HT14` leave 9. All have a similar precision. The full parameter sets are given in Tables 5 and 9.

the root unchanged (i.e., $e^{2\pi \mathrm{I}(x+kq)/q} = e^{2\pi \mathrm{I} x/q}$ for every integer k). In this way, it suffices to evaluate the decryption equation, without the reduction, in the exponent. Notice that sums turn into products, with one factor per nonzero bit of s, so SPRU switches to a sparse secret with only h nonzero bits [BTH22]. Further, following a block-binary structure [LMSS23], the N bits of the key are split into h blocks, each holding exactly one nonzero bit. Each factor is then a sum, over one block, of public roots weighted by the secret bits:

$$\underbrace{e^{2\pi \mathrm{I}\, m/q}}_{\text{root of the message}} = \underbrace{\prod_{b=1}^{h}}_{\text{depth } \log h} \Bigg(\sum_{j=1}^{N/h} \underbrace{s_{b,j}}_{\text{encrypted bit}} \cdot \underbrace{e^{2\pi \mathrm{I}\, c_{b,j}/q}}_{\text{public root}} \Bigg),$$

where the $c_{b,j}$ are coefficients of the ciphertext. SPRU evaluates this in three steps. First, it computes the public roots in the clear and multiplies them by the encrypted secret bits, which form the bootstrapping key; these plaintext–ciphertext products are cheap and independent. Second, it multiplies the h factors, at a cost of only $\log h$ multiplicative levels. Third, it takes the imaginary part, $\sin(2\pi m/q) \approx 2\pi m/q$ for a small message, which gives back m up to a known factor. All messages share the slots of one ciphertext, which is why one run of SPRU covers at most $N/(2h)$ of them.

A parallel version of SPRU. Recall that, in the equation above, the root of each message is a product of h *factors*, one per block of the secret: the factor of block b is the term in parentheses, that is, the public root selected by the secret bit of block b. The first step of SPRU computes these factors, and the second multiplies them. In SPRU, both steps take place *within* a single ciphertext: the first step writes the h factors of a message into h different slots, and the second multiplies them by rotating that ciphertext onto itself. The resulting root ends up copied in all h slots, so each message uses up h slots, and these copies cannot serve other messages (Figure 2, left). TUTTI keeps the three steps and changes only where the first step writes the factors: the factor of block b goes to the b-th of h ciphertexts, which we call accumulators. The second step then multiplies the accumulators slot by slot, without any rotation, and the third step is unchanged (Figure 2, right). Every slot then serves a different message: with h accumulators, a single run refreshes all $N/2$ slots, and in between, 2^k accumulators refresh 2^k times as many messages as SPRU, always at the same depth $\log h$. The only work that grows with the number of messages is the first step, which builds the

accumulators: a batch of plaintext–ciphertext products that do not depend on each other, and hence spread over any number of cores.

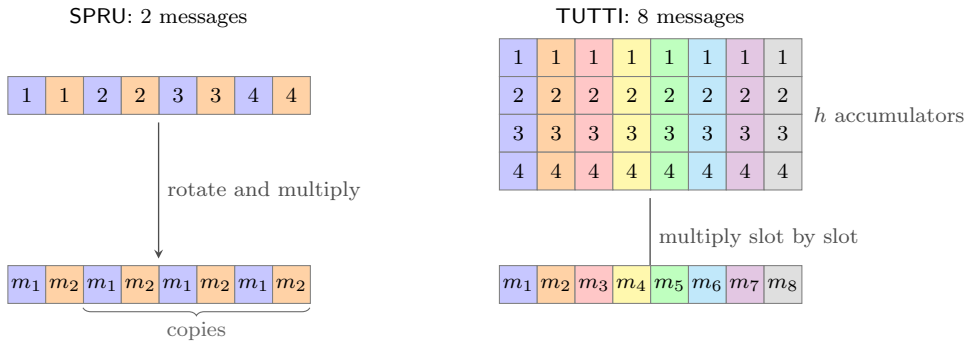


Fig. 2. Where the factors of each message live, for $h = 4$ blocks and 8 slots. A cell holds the factor of the block it is numbered with, and colours tell messages apart. SPRU (left) multiplies the factors within one ciphertext and keeps copies of each result; TUTTI (right) stores them in h accumulators and multiplies them slot by slot, so that every slot refreshes its own message.

Further optimizations. On top of this new layout, TUTTI shortens each of the three steps. First, the sparse secret is only needed during bootstrapping, so we can afford to restrict it further and take it in a subring, of the form $s(X^R)$: its blocks shrink by a factor R, and so does the number of products of the first step, while the key amounts to a secret of dimension N/R, which our security analysis takes into account (Section 4.2). Second, as in SHIP, the products of the first step are rescaled by the auxiliary prime used for key switching, so they consume no multiplicative level (Section 5.1). Third, as in SHIP, the linear transform that moves the messages between coefficients and slots (S2C) runs once on the incoming ciphertext, at the bottom of the modulus chain, instead of at the end of the bootstrapping, where it would act on a much larger modulus (Section 4.4). For a single message, we also remove most of the remaining rotations by computing more independent products in the first step, and add a second harmonic of the sine to the third step, which cancels its main approximation error; together, they bootstrap one message in 4.6 ms on 32 threads, at 6.5 bits of precision (Section 5.2). Finally, on machines with limited memory bandwidth, the full-slot bootstrapping can be split into several smaller ones that share a single key, almost 8 times smaller in our experiments, at some cost in latency on many cores (Section 4.5).

Refined security analysis of block-binary secrets. SPRU, SHIP and TUTTI all rely on sparse secrets: the fewer their nonzero bits h, the shorter the product of the second step, but the easier the secret becomes to attack. Their parameters therefore hinge on how sparse that secret may be for a given modulus, which in turn sets the ring degree N and, with it, the cost of every operation. Previous bootstrapping papers [LMSS23,CHKS25,CK27] estimated the combinatorial attacks on such secrets with a cost formula extrapolated from dense secrets, which, in the sparse regime, puts their cost at least 25 bits below that of the actual attacks (Section 3.2). We instead cost each attack on the exact block structure: lattice attacks with the recent ring-aware hybrids of Hou, Jiang and Ogilvie [HJO26], and combinatorial attacks with Odlyzko's meet-in-the-middle and May's Meet-LWE, with and without guessing the error [Odl90,May21,HKLS22,BLL+26] (Section 3). This gives, for each ring degree and Hamming weight, the largest base modulus that keeps 128 bits of security, including for the subring secrets of TUTTI. As a consequence, a single message can be bootstrapped at $N = 2^{13}$ instead of the $N = 2^{15}$ used by SPRU, and all our parameter sets push the sparse key to the limit that current attacks allow (Table 4). Applied to SHIP's secret, the same analysis keeps

its base moduli within these limits, provided the switch into the sparse secret happens at the base modulus.

Implementation and experiments. We implement TUTTI and SHIP in the same library, with the same thread control, and compare them on the same machine, with SHIP on its published chains and TUTTI on sets of the same ring degree and levels (Section 6). TUTTI trades total work for parallelism: on a single thread, SHIP is faster on two of the four parameter sets, but from 48 threads on, TUTTI is faster on every set, by up to 3× (155 against 467 ms on `HT14`), at a precision on par or better. Unlike SHIP, TUTTI also becomes faster as fewer messages are bootstrapped, down to 4.6 ms for a single one (Figure 1). This single-message latency is below that of TFHE: on the same 32 threads, a 4-bit bootstrapping of TFHE-rs [Zam22] takes 7.3 ms, and on the 14 threads of an Apple M5 Pro laptop, where an AArch64 port of TUTTI takes 2.4 ms, TFHE-rs takes 4.5 ms, so the advantage persists across architectures. The comparison with TFHE, however, needs some nuance. In particular, TFHE evaluates a lookup table on exact messages with bounded failure probability, whereas TUTTI refreshes a fixed-point value at a measured precision; Section 6 details the comparison. Our implementation and security estimates are open source.[3]

2 Preliminaries

Notation. For $r \in \mathbb{R}$, we write $\lfloor r \rceil$ for the nearest integer to r, with ties rounded up. We also write $[\![n]\!] := \{0, 1, \ldots, n-1\}$ and let I be a fixed square root of -1. All logarithms are in base 2. We represent $\mathbb{Z}_q$ by $\mathbb{Z} \cap (-q/2, q/2]$, and $z \bmod q$ denotes the centered reduction. For a power-of-two N, we let $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$ and $\mathcal{S} = \mathbb{R}[X]/(X^N + 1)$. We identify the elements of these rings with their coefficient vectors, written in plain type, and $\|a\|_\infty$ denotes the largest absolute value of a coefficient. Slot vectors are written in bold, and $\boldsymbol{v} \odot \boldsymbol{v}'$ denotes the slotwise product in $\mathbb{C}^{N/2}$. Whenever we count the slots, messages, values or coefficients that a bootstrapping refreshes, we count real numbers represented in fixed-point.

2.1 The CKKS scheme

CKKS [CKKS17] is a leveled homomorphic encryption scheme for approximate arithmetic. A ciphertext $\mathsf{ct} = (b, a) \in \mathcal{R}_{q_\ell}^2$ at level ℓ encrypts a plaintext $m \in \mathcal{R}$ under the secret key $\mathsf{sk} = (1, s)$ if

$$b + a \cdot s = m + e \pmod{q_\ell}$$

for a small error e. Decryption therefore recovers m only approximately. Unless stated otherwise, the secret s is dense and ternary, with coefficients uniform in $\{-1, 0, 1\}$. We write $\mathsf{Enc}_{\mathsf{pk}}$ for public-key encryption.

We use the RNS variant of the scheme [CHK+18b], in which $q_\ell = q_0\, p_1 \cdots p_\ell$ is a product of word-sized primes over the base prime q_0. An additional auxiliary prime P is used for key switching. The primes need not have the same bit length, so the number of levels measures the consumption of the modulus less precisely than the number of bits does.

Encoding. A plaintext holds a vector of $N/2$ complex numbers, called its slots. More precisely, $\mathcal{S}$ is identified with $\mathbb{C}^{N/2}$ through the map

$$\mathsf{DFT}(p) = \big(p(\zeta^{5^j})\big)_{j \in [\![N/2]\!]}, \qquad \zeta = \exp(\mathrm{I}\pi/N).$$

[3] https://github.com/robinkoestler/tutti

For a scaling factor Δ, a vector $\boldsymbol{z}$ is encoded as $\mathsf{Ecd}(\boldsymbol{z}) = \lfloor \Delta \cdot \mathsf{iDFT}(\boldsymbol{z}) \rceil$ and decoded by $\Delta^{-1} \cdot \mathsf{DFT}$ [CKKS17,CHK+18a]. Both maps are computed in RNS arithmetic at the current modulus [BEHZ16].

Shorter vectors are encoded by repetition. For a power-of-two $n \mid N/2$, sparse packing encodes $\boldsymbol{z} \in \mathbb{C}^n$ as Ecd of $N/(2n)$ concatenated copies of $\boldsymbol{z}$, which is a polynomial in $\mathbb{Z}[X^{N/(2n)}]/(X^N + 1)$. For $n = 1$ and a real z, this is the constant $\lfloor \Delta z \rceil$.

Operations. The operation Add adds two ciphertexts, and therefore their slots. The operation $\mathsf{ExtMult}(\mathsf{ct}, \mathsf{pt}) = (b \cdot \mathsf{pt},\ a \cdot \mathsf{pt}) \bmod q_\ell$ multiplies a ciphertext by a plaintext $\mathsf{pt} \in \mathcal{R}$, and Mult multiplies two ciphertexts and relinearizes the result. The product of two encodings carries an extra factor Δ, since

$$\mathsf{Ecd}(\boldsymbol{z}_1 \odot \boldsymbol{z}_2) \simeq \Delta^{-1} \cdot \mathsf{Ecd}(\boldsymbol{z}_1) \cdot \mathsf{Ecd}(\boldsymbol{z}_2).$$

A product is therefore followed by the rescaling $\mathsf{RS}(\mathsf{ct}) = \lfloor \mathsf{ct}/p_\ell \rceil \bmod q_{\ell-1}$, which consumes one level. We write $\mathsf{MultR} = \mathsf{RS} \circ \mathsf{Mult}$ and $\mathsf{ExtMultR} = \mathsf{RS} \circ \mathsf{ExtMult}$. Relinearization and the automorphisms of Section 2.2 change the secret key, and a key switch restores it.

Key switching. A key switch turns an encryption under a secret $\bar{s}$ into an encryption of the same message under s. It uses a public switching key, made of encryptions of multiples of $\bar{s}$ under s, and adds a small error. We use two variants, detailed in Appendix C. The GHS switch [GHS12c] works at the modulus Pq_ℓ, extended by the auxiliary prime P: the ciphertext is multiplied with the switching key at that modulus, and a division by P, which we call ModDown, removes the factor P together with most of the switching error. In the RNS variant, the ciphertext is first split into dnum digits, each a group of primes of q_ℓ, and the key holds one sample per digit [CHK+18b]. The BV switch [BV11] needs no auxiliary prime: it splits the ciphertext into D digits in base 2^ω, its key holds D samples modulo q_ℓ, and it costs $2D$ multiplications in $\mathcal{R}_{q_\ell}$. For security, the difference that matters is the modulus of the published samples, Pq_ℓ for GHS against q_ℓ for BV. We use GHS for every switch under the dense key, and BV for the switch into the sparse key (Section 3).

2.2 Slot operators

The automorphism Ψ_r, defined by $\Psi_r(m)(X) = m(X^{5^r})$, rotates the slots of m by r positions to the left:

$$\mathsf{DFT}(\Psi_r(m))_j = \mathsf{DFT}(m)_{j+r \bmod N/2}.$$

The automorphism $\mathsf{conj}(m)(X) = m(X^{-1})$ conjugates the slots. Homomorphically, $\mathsf{Rot}(\mathsf{ct}, r)$ and $\mathsf{Conj}(\mathsf{ct})$ apply these maps to both components of ct and then key-switch. Neither consumes a level. The operator $\mathsf{Im2}(m) = -X^{N/2}(m - \mathsf{conj}(m))$ returns twice the imaginary part of each slot. We evaluate it as $\mathsf{ExtMult}(\mathsf{Conj}(\mathsf{ct}) - \mathsf{ct}, X^{N/2})$, without rescaling.

Trace and product. The next two operators combine the slots of a single ciphertext. For powers of two $d' \mid d \mid N/2$, the trace [AP13,CHK+18a] and the product operator [CK27] are the compositions

$$\mathsf{Tr}_{d \to d'} = (\mathsf{id} + \Psi_{d'}) \circ (\mathsf{id} + \Psi_{2d'}) \circ \cdots \circ (\mathsf{id} + \Psi_{d/2}),$$
$$\mathsf{Pr}_{d \to d'} = (\mathsf{id} \cdot \Psi_{d'}) \circ (\mathsf{id} \cdot \Psi_{2d'}) \circ \cdots \circ (\mathsf{id} \cdot \Psi_{d/2}),$$

where $(\mathsf{id} \cdot \Psi_r)(m) = m \cdot \Psi_r(m)$. Given the sparse packing of $\boldsymbol{z} \in \mathbb{C}^d$, they return the sparse packing of the d' partial sums and partial products

$$t_k = \sum_{i \in [\![d/d']\!]} z_{k+id'} \qquad \text{and} \qquad t_k = \prod_{i \in [\![d/d']\!]} z_{k+id'}, \qquad k \in [\![d']\!],$$

respectively. For example, $\mathsf{Pr}_{4\to 1} = (\mathsf{id} \cdot \Psi_1) \circ (\mathsf{id} \cdot \Psi_2)$ acts on four slots as

$$(z_0, z_1, z_2, z_3) \;\mapsto\; (z_0 z_2,\, z_1 z_3,\, z_0 z_2,\, z_1 z_3) \;\mapsto\; (p, p, p, p), \qquad p = z_0 z_1 z_2 z_3.$$

Notice that each layer writes every partial product into two slots, which Section 4.1 will exploit.

Homomorphically, $\mathsf{id} + \Psi_r$ is $\mathsf{Add}(\mathsf{ct}, \mathsf{Rot}(\mathsf{ct}, r))$ and $\mathsf{id} \cdot \Psi_r$ is $\mathsf{MultR}(\mathsf{ct}, \mathsf{Rot}(\mathsf{ct}, r))$. The trace therefore costs $\log(d/d')$ automorphisms and no level, while the product operator costs $\log(d/d')$ automorphisms and $\log(d/d')$ levels. When the factors of a product lie in distinct ciphertexts, no automorphism is needed, and $\bigodot_{i \in I} \mathsf{ct}_i$ denotes the binary tree of MultR, of depth $\lceil \log |I| \rceil$.

Slot-to-coefficient. Let $\boldsymbol{t} \in \mathbb{R}^n$ be sparsely packed as $v = \mathsf{Ecd}(\boldsymbol{t}) \in \mathbb{Z}[X^{N/(2n)}]/(X^N + 1)$. The linear map $\mathsf{S2C}_n$ moves the slot values into the coefficients:

$$\mathsf{S2C}_n(v) = \sum_{j \in [\![n]\!]} \lfloor \Delta \cdot t_j \rceil \cdot X^{jN/n} \in \mathbb{Z}[X^{N/n}]/(X^N + 1). \tag{1}$$

There are two standard ways to evaluate it. The first is a matrix–vector product by the diagonal method [HS14,CHK+18a]. With the baby-step giant-step technique, it takes $\mathcal{O}(\sqrt{n})$ rotations and $\mathcal{O}(n)$ plaintext products, in one level. The second is a homomorphic FFT, which decomposes the matrix into sparse factors, with $\mathcal{O}(\log n)$ operations but one level per factor [CHH18,CCS19]. The bootstraps of this paper are shallow, and their product takes only $\log h$ levels, so we do not spend several more levels on this map. We always use the matrix–vector product, as does SHIP, so $\mathsf{S2C}_n$ takes one level. We denote the inverse map by C2S. For real $\boldsymbol{t}$, refreshing n slots and refreshing the n coefficients of (1) are thus the same task. A complex vector of n slots amounts to $2n$ coefficients, its real and imaginary parts.

2.3 SPRU bootstrapping

Classical CKKS bootstrapping [CHK+18a] starts by raising the modulus of the ciphertext, which adds an unknown integer polynomial to the plaintext. To remove this polynomial, the bootstrapping needs an approximate reduction modulo 1. This reduction has to act on each coefficient, whereas homomorphic operations act on the slots, so the circuit moves the coefficients to the slots by C2S, evaluates the reduction slotwise, and returns to the coefficients by S2C.

SPRU [CK27] and SHIP [CHKS25] avoid both C2S and the reduction. They evaluate the decryption equation directly on the unit circle, using a sparse secret at q_0 that is reached by the sparse-secret encapsulation of Section 3. We recall SPRU in three steps: we write decryption as a matrix–vector product, map it to the unit circle, and pack the result into the slots.

Decryption matrix. Let $h \mid N$ be a power of two and $B = N/h$. The SPRU secret is block-binary (Section 3): $s \in \{0,1\}^N$ has exactly one 1 in each of the h blocks of length B, and $s_0 = 1$. Let n be a power of two with $1 \le n \le n_{\max} := B/2$, and let $\mathsf{ct} = (c_0, c_1)$

encrypt $m = \sum_{a \in [\![n]\!]} m_a X^{aN/n}$ modulo q. For $i \in [\![N]\!]$ and $a \in [\![n]\!]$, let $C_{i,a}$ be the coefficient of $X^{aN/n}$ in $c_1 \cdot X^i$, to which we add the coefficient of $X^{aN/n}$ in c_0 when $i = 0$. The matrix $\mathbf{C} = (C_{i,a}) \leftarrow \mathsf{DecMat}(\mathsf{ct})$ then satisfies, error included,

$$m_a = \sum_{i \in [\![N]\!]} s_i \, C_{i,a} \pmod q. \tag{2}$$

We fix $s_0 = 1$ so that c_0 can be merged into row $i = 0$. Decryption is then a sum over the h blocks, and the product in (3) below needs no additional factor for c_0. This reveals the first block of the secret, which we take into account in the security analysis (the anchored instance of Section 3.1).

Roots of unity. The map $x \mapsto \exp(2\pi \mathrm{I}\, x/q)$ turns the sum in (2) into a product. Since each block of s contains a single 1, it selects exactly one root:

$$\exp\left(2\pi \mathrm{I}\, \frac{m_a}{q}\right) = \prod_{b \in [\![h]\!]} \sum_{j \in [\![B]\!]} s_{bB+j} \cdot \exp\left(2\pi \mathrm{I}\, \frac{C_{bB+j,a}}{q}\right). \tag{3}$$

To recover m_a, we scale each root by $\delta = (q/(4\pi\Delta))^{1/h}$. Twice the imaginary part of the product is then $\frac{q}{2\pi\Delta} \sin(2\pi m_a/q) \simeq m_a/\Delta$ for $|m_a| \ll q$, and $\mathsf{Im2}$ extracts it. The quality of this approximation depends on the ratio $\gamma = q/\Delta$ of the modulus to the scale of the message: for $|m_a|/\Delta \leq 1$, the angle $2\pi m_a/q$ is at most $2\pi/\gamma$.

Packing. The slots must hold the candidate roots of (3). There are B of them for each of the n coefficients and h blocks, so nN values in total, and we spread them over $2n$ encodings of $N/2$ slots. Encoding $u \in [\![2n]\!]$ takes the sub-block of offsets $j(u,k) = u \cdot B/(2n) + k$, $k \in [\![B/(2n)]\!]$, of every block. The map $\iota(k,b,a) = k \cdot hn + b \cdot n + a$ is a bijection from $[\![B/(2n)]\!] \times [\![h]\!] \times [\![n]\!]$ onto $[\![N/2]\!]$, and slot $\iota(k,b,a)$ holds offset $j(u,k)$ of block b for coefficient a. The secret and the public packing share this layout:

$$\tilde{s}^u_{\iota(k,b,a)} = s_{bB+j(u,k)}, \qquad \rho^u_{\iota(k,b,a)} = \delta \cdot \exp\left(2\pi \mathrm{I}\, \frac{C_{bB+j(u,k),a}}{q}\right), \tag{4}$$

with $S_u = \mathsf{Ecd}(\tilde{\boldsymbol{s}}^u)$ and $E_u = \mathsf{Ecd}(\boldsymbol{\rho}^u)$. For example, for $N = 8$, $h = 2$, $B = 4$ and $n = 1$, the two secret packings are the following.

Slot $\iota(k,b,0)$	0	1	2	3
(k,b)	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
$\tilde{\boldsymbol{s}}^0$	s_0	s_4	s_1	s_5
$\tilde{\boldsymbol{s}}^1$	s_2	s_6	s_3	s_7

We first sum the slotwise products $\tilde{\boldsymbol{s}}^u \odot \boldsymbol{\rho}^u$ over u, and then over k with $\mathsf{Tr}_{N/2 \to hn}$. Only the offset of the single 1 of each block survives, so slot $bn + a$ now holds the scaled root that block b selects for coefficient a. In the example, the trace adds slots 0 and 2, which cover the first block, and slots 1 and 3, which cover the second. Next, $\mathsf{Pr}_{hn \to n}$ multiplies the h blocks, and $\mathsf{Im2}$ yields m_a/Δ, repeated $N/(2n)$ times. This is the input of $\mathsf{S2C}_n$. For $n = 1$, the input is the constant m_0/Δ in every slot, whose encoding is the constant polynomial m_0, so $\mathsf{S2C}_1$ is the identity.

Homomorphic evaluation. The bootstrapping key consists of the $2n$ ciphertexts $\mathsf{cs}_u \leftarrow \mathsf{Enc}_{\mathsf{pk}}(S_u)$, encrypted at a modulus high enough for the circuit (Algorithm 4). Algorithm 1 builds each E_u in the clear and evaluates the steps above, computing the slotwise products as $\mathsf{ExtMultR}(\mathsf{cs}_u, E_u)$. The plaintext–ciphertext products consume one level, which Section 5.1 moves to the auxiliary prime. The product operator consumes $\log h$ levels, the trace and $\mathsf{Im2}$ consume none, and $\mathsf{S2C}_n$ consumes one more, or none for $n = 1$, where it is the identity. The output encrypts $\sum_a m_a X^{aN/n}$ at the remaining modulus, which is larger than q.

Algorithm 1 SPRU bootstrapping [CK27]

Input: q, key $(\mathsf{cs}_u)_{u<2n}$, ciphertext ct, $\delta = (q/(4\pi\Delta))^{1/h}$
Output: Refreshed ciphertext ct'

$\mathbf{C} \leftarrow \mathsf{DecMat}(\mathsf{ct})$, $\mathsf{acc} \leftarrow 0$
for $u = 0$ to $2n - 1$ **do**
  for all $0 \le k < B/(2n)$, $0 \le b < h$, $0 \le a < n$ **do**
    $\rho^u_{\iota(k,b,a)} \leftarrow \delta \cdot \exp(2\pi\mathrm{I} \cdot C_{bB+j(u,k),a}/q)$
  $E_u \leftarrow \mathsf{Ecd}(\boldsymbol{\rho}^u)$
  $\mathsf{acc} \leftarrow \mathsf{Add}(\mathsf{acc}, \mathsf{ExtMultR}(\mathsf{cs}_u, E_u))$
return $\mathsf{ct}' \leftarrow \mathsf{S2C}_n\big(\mathsf{Im2}\big(\mathsf{Pr}_{hn\to n}(\mathsf{Tr}_{N/2\to hn}(\mathsf{acc}))\big)\big)$

The operator AccMul. A *single* AccMul turns one public packing $\boldsymbol{\rho}^u$ into its contribution to the accumulator of Algorithm 1. It encodes the packing by iDFT, applies an NTT to reach the RNS representation, computes the plaintext–ciphertext product with cs_u, and adds the result to the accumulator. Single AccMuls are independent of one another, so they are the units that we distribute over the cores. The *full* AccMul load is the set of all single AccMuls of one bootstrapping, which is $2n$ here and more in Section 4. Computing the roots that fill $\boldsymbol{\rho}^u$ is not part of AccMul (see Section 5.1).

2.4 SHIP bootstrapping

SHIP [CHKS25] evaluates the same unit-circle identity with the other side of the product encrypted. Let $\mathsf{ct} = (c_0, c_1)$ hold the messages in its coefficients m_t modulo q_0, under a sparse key s of weight h, and let $\omega = \exp(2\pi\mathrm{I}/q_0)$. Writing $(\cdot)_t$ for the coefficient of X^t, decryption gives

$$\omega^{m_t} = \omega^{(c_0)_t} \prod_{j\,:\,s_j \neq 0} \omega^{(s_j\, X^j c_1)_t}. \tag{5}$$

For each nonzero s_j, the factor over all t is a vector of roots of unity read off c_1, shifted by the position j and signed by s_j. SPRU builds such a factor by multiplying public roots with encrypted bits of the secret. SHIP instead starts from public vectors and rotates them by the encrypted position j, which is a *blind rotation*. Its secret is windowed ternary (Section 3): each nonzero coefficient lies in a public window, so only its position within the window has to stay encrypted.

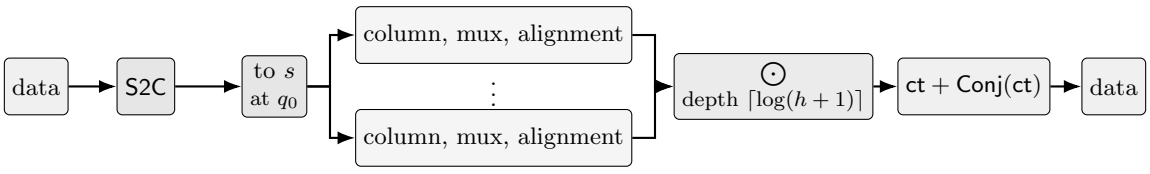


Fig. 3. SHIP bootstrapping. After S2C and the switch to the sparse key s at q_0, the h blind rotations run independently and return their factors at the top of the modulus chain, where the product tree and the conjugation follow.

Figure 3 shows the circuit. At the bottom of the chain, SHIP applies S2C and switches to the sparse key at q_0, as SPRU does in Section 2.3. Each of the h factors is then computed by a blind rotation that combines two methods [CHKS25, Section 4.3], with one step θ between them, and it starts with the column method, which needs public inputs.

- *Column.* The low part of the position, modulo θ, is resolved by 4θ plaintext–ciphertext products between public vectors and encrypted masks of the key. They use the auxiliary modulus and no key switch. Its cost grows linearly with θ. This is the order of their implementation; their Algorithm 4 exchanges the two parts of the position.
- *Multiplexer.* The high part is resolved one digit at a time. Section 4.3 of [CHKS25] uses binary digits, and their Section 5.1 generalizes this to a base ν. For each digit, an encrypted multiplexer selects among the corresponding rotations, with one key switch. The window of half-width w requires $\lceil \log_\nu \lceil 2w/\theta \rceil \rceil$ digits.
- *Alignment.* The start of the window is public, so an ordinary rotation moves the result there. In the implementation that we use, this takes one key switch.

The public term $\omega^{(c_0)_t}$, scaled by $\gamma/(4\pi\mathrm{I})$, and the h factors are then multiplied by $\bigodot$, of depth $\lceil \log(h+1) \rceil$, and $\mathsf{ct} + \mathsf{Conj}(\mathsf{ct})$ extracts twice the real part, which is again $\frac{\gamma}{2\pi} \sin(2\pi m_t/q_0)$.

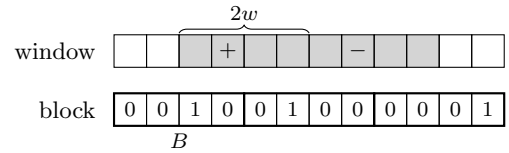
The published sets take $h = 31$, so the product has depth 5. Their experiments favour $\nu = 4$, and all four sets take it, with θ between 6 and 17 and a multiplexer range of $64 = 4^3$ [CHKS25, Table 2]. This gives 3 digits, hence four key switches in sequence per factor in the implementation that we use (Table 9). The h blind rotations are independent, but the chain of key switches within each of them bounds the latency even with unlimited cores. Only S2C depends on the number of slots. Table 7 in Section 6.2 compares a factor of SHIP with one of TUTTI.

3 Sparsely structured secret keys

SPRU and SHIP evaluate the decryption equation at the base prime q_0 homomorphically, and their cost grows with the Hamming weight h of the secret in that equation: SPRU multiplies h block roots at depth $\log h$, and SHIP performs h blind rotations. A sparse secret is weaker than a dense one at the same modulus, and the ciphertext chain needs a large modulus. We therefore use sparse-secret encapsulation [BTH22], which relies on two keys. The data lives under a dense ternary key, which must resist attacks at the full modulus PQ. The decryption equation that bootstrapping evaluates uses a sparse key, which is exposed only at q_0. A key switch at q_0 moves a ciphertext from the dense to the sparse key, and the bootstrapping key, an encryption of the sparse key under the dense one, moves it back. The dense key bounds $\log PQ$ (Table 1), and the sparse key bounds $\log q_0$ (Sections 3.1 and 3.2). Section 3.3 checks our parameter sets against both bounds. Our parameter sets (`S13`, `S13m2`, `LL13`, `LL14`, `HT14`, `HT15`) are listed in Table 5, and SHIP's (`LL13`, `LL14`, `HT14`, `HT15`) in Table 9.

Assumptions and cost model. Every published key except the switching key into the sparse key is an RLWE sample under the dense key. The switching key consists of RLWE samples under the sparse key, so it must live at a modulus the sparse key tolerates: the BV switch keeps it at q_0, whereas a GHS switch would place it at Pq_0 (Section 2.1). Our implementations of TUTTI and SHIP both use the BV switch. As every bootstrappable scheme does, we assume circular security, and we target IND-CPA security; IND-CPA^{D} [LM21] is out of scope, as for the schemes we compare with. The error is discrete Gaussian with standard deviation $\sigma = 3.2$, and costs are classical. A parameter set is accepted when the cheapest attack is estimated to cost at least 128 bits in word operations, the unit of the lattice estimator [APS15,APS26] at commit `53da598`; Section 3.2 converts the combinatorial costs into that unit.

Block-binary and windowed secrets. The block-binary secret of SPRU (Section 2.3) has exactly one 1 in each of the h blocks of length $B = N/h$. A windowed secret, as in SHIP [CHKS25], is ternary of weight h: its k-th nonzero coefficient lies in the window of half-width w centered at kB, and SHIP takes $w = \max(175,\, B/2)$.



Schematic, $w = B/2$: one 1 per block, one ± 1 per gray window around a block boundary.

Subring secrets. Some of our parameter sets restrict the block-binary key further, to the form $s(X) = s'(X^R)$ for a divisor R of B; Section 4.2 explains what this saves. The restriction leaves an RLWE problem of degree N/R:

Proposition 1. *Let N be a power of two and let $R \mid N$. Let $s' \in \mathbb{Z}[Y]/(Y^{N/R}+1)$ and set $s(X) = s'(X^R)$. For any modulus q and any distribution χ on $\mathbb{Z}$, the search and decision coefficient-form RLWE problems for one sample in dimension N with secret s and coefficient error χ are equivalent to the same problems for R samples in dimension N/R with secret s' and coefficient error χ.*

The equivalence is standard: ring switching [GHPS12] and subring secret encapsulation [MHWW25] rest on the same decomposition. We include the proof, a rearrangement of coefficients, for completeness.

Proof. Write $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$ and $\mathcal{R}' = \mathbb{Z}[Y]/(Y^{N/R}+1)$, and $\mathcal{R}_q$, $\mathcal{R}'_q$ for their reductions modulo q. Coefficient-form RLWE is the variant of the search and decision problems of [LPR13, Definitions 3.2 and 3.3] with coefficient-wise independent errors; we do not assume an arbitrary error distribution in the canonical embedding.

The substitution $\phi\colon \mathcal{R}' \to \mathcal{R}$, $Y \mapsto X^R$, is a ring homomorphism, well defined because $(X^R)^{N/R} = X^N = -1$, and it reduces modulo q. For $a \in \mathcal{R}_q$ and $0 \le j < R$, let $a^{(j)} \in \mathcal{R}'_q$ collect the coefficients of a in the degrees congruent to j modulo R:

$$a^{(j)}(Y) = \sum_{k=0}^{N/R-1} a_{j+kR}\, Y^k, \qquad a = \sum_{j=0}^{R-1} X^j\, \phi\Big(a^{(j)}\Big). \tag{6}$$

The map $a \mapsto (a^{(0)}, \ldots, a^{(R-1)})$ only reorders coefficients. It is therefore a bijection $\mathcal{R}_q \to (\mathcal{R}'_q)^R$ of $\mathbb{Z}_q$-modules, it maps a uniform a to R independent uniform $a^{(j)}$, and it maps an error with independent coefficients from χ to R independent errors of the same kind.

Multiplication by $s = \phi(s')$ preserves the residue classes: $X^j\phi(a^{(j)}) \cdot \phi(s') = X^j\phi(a^{(j)} \cdot s')$, so the decomposition of $a \cdot s$ is $(a^{(j)} \cdot s')_{j<R}$. A sample $b = a \cdot s + e$ thus decomposes into the R samples $b^{(j)} = a^{(j)} \cdot s' + e^{(j)}$ of dimension N/R under s', and the inverse map rebuilds one sample under s from R such samples. The same bijection of $\mathcal{R}_q^2$ identifies the uniform distribution on pairs with R independent uniform pairs. An algorithm for the search or the decision problem therefore transfers in either direction, converting a recovered secret between s' and $\phi(s')$ in the search case, with k samples in dimension N corresponding to kR samples in dimension N/R. Success probabilities and advantages are identical, and the conversion costs $O(kN)$. For a random secret, independence is conditional on s', and the correspondence holds for any distribution of s', pushed forward to $s = \phi(s')$.

Table 1. Largest $\log q$ at which the dense key costs at least 128 bits.

N	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
Uniform ternary	106	214	430	868	1746
Hamming weight 512	—	212	428	862	—

Table 2. Largest $\log q_0$ at which the rotated primal hybrid [HJO26] costs at least 128 bits on the block-binary secret anchored by $s_0 = 1$, of dimension $N' = N - B$ and weight $h' = h - 1$. A key $s'(X^R)$ is checked at degree N/R.

N	h	N'	h'	$\log q_0$
2^{11}	64	2016	63	27
2^{12}	64	4032	63	51
2^{13}	64	8064	63	102
2^{14}	64	16128	63	209
2^{13}	32	7936	31	27
2^{14}	32	15872	31	59

In our setting, the coefficients of s' are $(s_0, s_R, s_{2R}, \ldots)$. Since $R \mid B$, the unique 1 in each block of s lies on the R-grid, and those grid positions form a contiguous block of length B/R in s'. Thus s' is a block-binary secret of degree N/R and weight h, and each published sample under s gives R samples under s'. The attacks below need fewer samples than one sample already provides (Section 3.1), so the sparse key is estimated at degree N/R.

Dense key. Table 1 gives the largest integer bit length $\log q$ at which the cheapest classical attack of the lattice estimator still costs at least 128 bits. The estimates at these caps lie between 128.0 and 129.1 bits, whether the attack gets the N LWE samples of one RLWE sample or unlimited samples, as from the many published keys. On LL13, TUTTI and SHIP both use a dense ternary key of Hamming weight 512, with as many entries -1 as $+1$ (Section 6.1). With the hybrid attacks included, it costs 128.3 bits at 212 and 127.7 at 213 for $N = 2^{13}$, and May's Meet-LWE [May21] costs over 2^{1000} on it at that degree.

3.1 Lattice attacks on the sparse key

Hybrid lattice attacks have long exploited the sparsity of such secrets [Alb17,SC19,CP19]. The most recent ones also exploit the ring structure of sparse-secret RLWE in power-of-two cyclotomic rings: Ogilvie's rotated primal hybrid and Hou and Jiang's enhanced hybrid decoding attack, merged into one paper [HJO26]. Table 2 applies both to SPRU's block-binary secret.

Since SPRU fixes $s_0 = 1$ (Section 2.3), the first block of the secret is known. An attacker therefore faces what we call the *anchored* instance: the remaining $h' = h - 1$ blocks, of total dimension $N' = N - B$, each still with exactly one 1. For each pair (N, h), Table 2 gives the largest integer bit length $\log q_0$ at which the rotated primal hybrid still costs at least 128 bits on this instance. The enhanced hybrid was more expensive on every row. Both attacks need fewer equations than a single RLWE sample provides: even on the row $(2^{11}, 64)$, which has the fewest, they use 765 of its $N' = 2016$. The additional samples that a subring secret gives (Proposition 1) are therefore of no use to them, and indeed, rerunning both attacks with these samples, for $R \leq 32$ and degrees up to 2^{16}, left the caps unchanged.

This modulus caps the sparse key, not the ciphertext chain, which stays under the dense key. At the same degree the sparse cap lies well below the dense one, 102 bits against 214

at $N = 2^{13}$ and $h = 64$, and halving the weight cuts it further, to 27 bits. The hybrid's meet-in-the-middle stage ignores the blocks (Appendix D.9).

SHIP's windowed secret has weight $h = 31$, the anchored weight h' of the rows with $h = 32$. SHIP fixes no coefficient, so its instance keeps the full dimension N instead of $N' = N - B$, and a cap of Table 2 can only be conservative for it. Its base primes lie within these caps: 25 bits on `LL13` against 27, 48 and 28 bits on `LL14` and `HT14` against 59, and 50 bits on `HT15`, whose degree 2^{15} is larger than that of the 59-bit row. SHIP's own lattice check used the GHS modulus $P \cdot q_0$, at 55, 100, and 105 bits [CHKS25], above these caps, so our implementation of SHIP also switches at q_0 by BV (Section 5.1).

3.2 Revisiting May's attack

A sparse key has few possible values, so it can also be attacked combinatorially, by searching for the secret directly instead of reducing a lattice. We cost three such attacks on the anchored instance of Section 3.1, whose search space has size $S = B^{h'}$, at the modulus of the same row of Table 2. Each of them reads one RLWE sample as N' uniform LWE equations modulo q_0 and does not use the ring. We describe each attack and then its cost on our rows, which Table 3 collects. Appendix D gives the details, and our script reproduces May's table for NTRU-HPS to within 1 to 5 bits. Our main finding comes last: the closed exponents that earlier bootstrapping papers derived from May's attack underestimate these costs by at least 25 bits. Costing the attacks instead admits a smaller ring degree for the same security, and with it a lower latency: a single message can be bootstrapped at $N = 2^{13}$ instead of $N = 2^{15}$.

Odlyzko's meet-in-the-middle. The unknowns are the h' remaining blocks, each with a single 1 at an unknown offset in $[\![B]\!]$. This attack [Odl90,HGSW03] splits the secret as $s = s_1 + s_2$, where s_1 holds the first $\lceil h'/2 \rceil$ blocks and s_2 the others, so that each part has about $B^{h'/2}$ candidates. It builds two lists, of $a \cdot s_1$ and of $b - a \cdot s_2$ over all candidates. Since $b = a \cdot s + e$ with a small error e, the correct pair agrees up to e, and sorting both lists on the top bits of a few coordinates finds it, so the attack does not have to guess the error. Its larger list has $T_{\mathrm{Odl}} = B^{\lceil h'/2 \rceil}$ elements, which is $S^{1/2}$ up to the rounding of an odd h'. This is the column T_{Odl} of Table 3, against which we compare the two other attacks.

May's Meet-LWE. May's attack [May21] makes these lists smaller with *representations*. It writes $s = s_1 + s_2$ in many ways, also with cancellations $0 = 1 + (-1)$ on zero coordinates, so that the secret has many representations as a pair, and finding any one of them suffices. The lists, that is, the candidate sets for s_1 and s_2, can therefore be filtered: a candidate is kept only if $a \cdot s_1$ takes a fixed value on a few coordinates of the LWE equation, which only a fraction of the representations satisfy. Each of s_1 and s_2 is found in the same way, by splitting it again, so the attack is a tree of lists with one level per split, and the number of levels is its *depth* (Appendix D.3). Fixing an exact value on these coordinates requires guessing the error there. For a Gaussian error, this guess costs more than for May's ternary one, and it grows as q_0 shrinks. As a result, Meet-LWE with error guessing beats Odlyzko's split on three rows only, by at most 8 bits, and we list it only in Table 10 of the appendix.

Guessing-free Meet-LWE. This variant [KM21,HKLS22,BLL+26] drops the guess. Its filter keeps a candidate when $a \cdot s_1$ lies in a box of the error's size on the selected coordinates, instead of taking an exact value, and two partial sums match when they agree up to that box. We instantiate it at depth 2 on the representations of Appendix D.3, with the cost model of [BLL+26, Section 3.3], which gives the column T_{free} of Table 3. On the rows with $h = 64$ a deeper tree may be cheaper, so these entries are upper bounds. A deeper tree would have to

Table 3. Rounded bit costs of combinatorial attacks (green) on the anchored block-binary secret at the moduli of Table 2: Odlyzko's split T_{Odl} and the depth-2 guessing-free Meet-LWE T_{free}, in list elements, and T_{free} in word operations, T_{ops}. The closed powers $T_{\mathrm{BBK}} = S^{0.28}$ [LMSS23] and $T_{\mathrm{SHIP}} = S^{0.4}$ [CHKS25] of $S = B^{h'}$ (red) are not attacks. Bold: the cheapest attack, in list elements and, for $N = 2^{13}$, $h = 32$, in word operations.

N	h	$\log q_0$	T_{Odl}	T_{free}	T_{ops}	T_{BBK}	T_{SHIP}
2^{11}	64	27	160	$\leq$ **158**	$\leq$ 165	88	126
2^{12}	64	51	192	$\leq$ **186**	$\leq$ 194	106	151
2^{13}	64	102	224	$\leq$ **216**	$\leq$ 224	123	176
2^{14}	64	209	256	$\leq$ **246**	$\leq$ 254	141	202
2^{13}	32	27	128	**125**	**132**	69	99
2^{14}	32	59	144	**139**	146	78	112

gain at least 30 bits to reach 128, whereas with error guessing, where Appendix D searches depths up to 4, the best tree never lies more than 8 bits below Odlyzko's split.

The row $N = 2^{13}$, $h = 32$. This is the only row within 10 bits of 128, so it is the only one where the unit of the cost matters. Its guessing-free top list has $L = 2^{125.1}$ elements. May counts list elements and omits the polynomial factors of sorting and hashing [May21, Section 2.3], whereas our criterion counts word operations. Hashing one element on about $\log L$ coordinates costs one word operation per coordinate, which adds $\log \log L \approx 7$ bits. In word operations, the row therefore costs 132 bits and Odlyzko's split 135, the figures of Table 4. Bi et al.'s hash repetitions and the memory of the lists only raise these costs (Appendix D.6). Our set `S13` uses this row at $\log q_0 = 24 < 27$, which leaves the list sizes, and with them these costs, unchanged.

Closed exponents underestimate the cost of these attacks. May's Meet-LWE [May21] is the reference attack when bootstrapping papers choose parameters for sparse secrets, and three of them quote it as a closed power of the key-space size. In the sparse regime, these powers lie at least 25 bits below the attacks instantiated above, so they are far too conservative. Lee, Min, Seo, and Song [LMSS23] estimate a block-binary key by $T = S^{0.28}$ with $S = (B+1)^h$, allowing a block to be entirely zero. SHIP quotes $T \approx (4w)^{0.4h}$ scalar products, the factor 4 accounting for ternary signs, and requires $(4w)^{0.4 \cdot 31} > 2^{115}$ for an estimated 128 bits at $N \geq 2^{13}$ [CHKS25]. Coron and Köstler use $T = S^{0.25}$ with $S = (N/h)^{h-1}$ [CK27, Section 4.5]. On $S = B^{h'}$, these closed powers claim 69 and 99 bits on the row $N = 2^{13}$, $h = 32$, where the cheapest attack costs 125 bits in list elements and 132 in word operations (Table 3). The reason is the regime: May's exponents 0.28 to 0.3 come from the dense ternary secrets of NTRU and BLISS [HPS98,DDLL13], of relative weight 0.3 to 0.667, and he claims the form $T \approx S^{1/4}$ for none of the smaller weights [May21]. Here $h/N \in \{2^{-9}, \ldots, 2^{-5}\}$, and the saving from representations shrinks relative to $\log S$ as $1/\log(N/h)$. This is what lets a single message be bootstrapped at $N = 2^{13}$, where the estimate of SPRU forced $N = 2^{15}$, and our parameter sets rely on the instantiated attacks (Section 3.3).

SHIP's windowed secret. The same attacks apply to SHIP's secret, where each window has $4w$ signed positions. Odlyzko's split over the windows is the cheapest of them on every published set, at no less than 142 bits at the base prime (Table 11 in Appendix D.8). These costs leave room to narrow the windows, down to the smallest half-width at least $N/(2h)$, at no less than 136 bits. On `LL13`, this takes the half-width from 175 to 133, and on the other sets the published half-width is already at this limit. Even on `LL13`, a narrower window hardly affects the latency of SHIP. The number of multiplexer digits stays 3, and with it the

Table 4. Security of the parameter sets of Table 5 against the caps of Tables 1 and 2; the last column gives the costs of Table 3 in word operations at the cap of each row. The cap of `LL13` is the one of its dense key of Hamming weight 512.

Set	N	R	$\log PQ$ (cap)	$(N/R, h)$	$\log q_0$ (cap)	Odlyzko / guessing-free
`S13`	2^{13}	1	208.4 (214)	$(2^{13}, 32)$	24 (27)	135 / 132
`S13m2`	2^{13}	1	213.6 (214)	$(2^{13}, 32)$	23 (27)	135 / 132
`LL13`	2^{13}	4	212.0 (212)	$(2^{11}, 64)$	24 (27)	167 / $\leq$ 165
`LL14`	2^{14}	4	430.0 (430)	$(2^{12}, 64)$	48 (51)	200 / $\leq$ 194
`HT14`	2^{14}	8	429.5 (430)	$(2^{11}, 64)$	27 (27)	167 / $\leq$ 165
`HT15`	2^{15}	8	769.0 (868)	$(2^{12}, 64)$	50 (51)	200 / $\leq$ 194

sequential key switches of a blind rotation, so at most the column step θ could shrink, from 6 to 5. We therefore keep the published windows.

Hybrid attacks. Hybrids of the guessing-free lists with lattice reduction gain up to 10 bits [HKLS22] and 18 bits [BLLW22,BLL+26] on unstructured homomorphic-encryption sets. Whether these gains carry over to block and window constraints, and combine with the ring isometries of [HJO26], is open. Like the bootstrapping papers we compare with, our estimates cover the attacks instantiated above. As for any set at the 128-bit line, such an improvement would call for smaller base primes or larger rings.

3.3 Security of our parameter sets

Table 4 checks each parameter set of Table 5 against the bounds of this section. The dense key is checked at $\log PQ$ against Table 1 at degree N. The sparse key has degree N/R (Proposition 1) and is checked at q_0 against the row $(N/R, h)$ of Tables 2 and 3. Every set uses a base prime at or below the cap of its row. No attack of this section becomes cheaper as the modulus decreases: the lattice attacks become more expensive, Odlyzko's split and the lists of Meet-LWE do not depend on q_0, and the error guess grows. The combinatorial costs at the cap therefore carry over to the smaller q_0.

4 The TUTTI bootstrapping

The original SPRU bootstrapping [CK27] (Algorithm 1 in Section 2.3) refreshes at most $n_{\max} = B/2$ coefficients. SPRU already refreshes further coefficients by monomial shifts [CK27, Algorithm 13]: a shift adjusts the decryption matrix $\mathsf{DecMat}(\mathsf{ct})$ of Equation (2), and the shifted bootstrappings share one key, run independently and are added, but every extra run takes cores from each bootstrapping.

TUTTI instead refreshes all coefficients within one run, so that every core works on a single bootstrapping. It exploits that the product operator of Section 2.2 wastes slots: each of its layers $\mathsf{id} \cdot \Psi_r$ multiplies the two halves of a period and writes the result into both. TUTTI fills one of the two copies with a further group of coefficients, layer by layer, up to all $N/2$ coefficients at the same depth $\log h$ (Section 4.1). We write TUTTI_n for our bootstrapping of n coefficients; for $n \leq n_{\max}$, it is SPRU with the optimizations of this section and of Section 5.

Compared with SPRU on $n_{\max}$ coefficients, TUTTI_n with $t = n/n_{\max}$ accumulators differs in two ways. Its AccMul load is t times larger, and a tree of $t - 1$ products of distinct accumulators replaces the first $\log t$ layers of the product operator at the same depth, which removes their rotations. With many cores, the main challenge is therefore to share the AccMul load. A subring secret shrinks this load (Section 4.2), and many cores share it (Section 4.3); S2C moves to the bottom of the chain (Section 4.4), and cosets trade cores for a smaller

key (Section 4.5). In the full-slot case the circuit takes the shape of SHIP's, a product of h ciphertexts without rotations; Section 6.2 compares the two.

4.1 From $2n_{\max}$ coefficients to all $N/2$ slots

We first show how TUTTI refreshes twice as many coefficients as SPRU by filling the slots that one layer of the product operator wastes, and then fill those of the further layers, up to all $N/2$ coefficients. Like SPRU, the construction of this section ends with $\mathsf{S2C}_n$; Section 4.4 moves it to the bottom of the chain.

Bootstrapping twice as many coefficients. SPRU refreshes $2n_{\max}$ coefficients by two shifted runs. Each run accumulates its own slot vector in Algorithm 1: $\boldsymbol{a} \in \mathbb{C}^{N/2}$ for the first group of $n_{\max}$ coefficients and $\boldsymbol{b}$ for the second. For $n = n_{\max}$ the trace $\mathsf{Tr}_{N/2\to hn}$ is the identity, since $hn = N/2$, so each run applies the product operator directly and returns $\boldsymbol{p} = \mathsf{Pr}_{N/2\to n_{\max}}(\boldsymbol{a})$ and $\boldsymbol{p}' = \mathsf{Pr}_{N/2\to n_{\max}}(\boldsymbol{b})$, respectively. The first layer of each product operator, $\mathsf{id} \cdot \Psi_{N/4}$, multiplies the two halves of its vector and writes every product twice, so each run wastes half of its slots in that layer.

TUTTI only changes the packing. Instead of accumulating $\boldsymbol{a}$ and $\boldsymbol{b}$ separately, the packings of the key and of the public roots place their entries interleaved in two accumulators,

$$\boldsymbol{v} = (a_0, b_0, a_1, b_1, \ldots, a_{N/4-1}, b_{N/4-1}), \quad \boldsymbol{w} = (a_{N/4}, b_{N/4}, \ldots, a_{N/2-1}, b_{N/2-1}).$$

This costs nothing, since it is a choice of packing. As slot $bn + a$ of $\boldsymbol{a}$ and $\boldsymbol{b}$ holds block b, $\boldsymbol{v}$ carries the blocks $b < h/2$ of both groups and $\boldsymbol{w}$ the others, which is the split by blocks that the generalization below uses.

A single Hadamard product $\boldsymbol{v} \odot \boldsymbol{w}$ now performs the first layer of both product operators at once. Its even slots hold the output of that layer on $\boldsymbol{a}$ and its odd slots its output on $\boldsymbol{b}$, so no slot is wasted. The remaining layers, $\mathsf{Pr}_{N/2\to 2n_{\max}}$, then give $N/(4n_{\max})$ copies of

$$\boldsymbol{o} = (p_0, p'_0, p_1, p'_1, \ldots, p_{n_{\max}-1}, p'_{n_{\max}-1}) \in \mathbb{C}^{2n_{\max}},$$

which holds the $2n_{\max}$ values of both runs in interleaved order, at the unchanged depth $\log h$. The packing also realizes any slot permutation for free, and one such permutation reorders $\boldsymbol{o}$ as $(p_0, p_1, \ldots, p_{n_{\max}-1}, p'_0, p'_1, \ldots, p'_{n_{\max}-1})$. This is a $2n_{\max}$-slot encoding, and $\mathsf{S2C}_{2n_{\max}}$ completes the bootstrapping of $2n_{\max}$ coefficients.

Generalization to more coefficients. The case above is $t = 2$, with the two accumulators $\boldsymbol{v}$ and $\boldsymbol{w}$. For $1 \le k \le \log h$ and $t = 2^k$, the same construction uses t accumulators $\boldsymbol{v}^{(g)}$, and $\boldsymbol{v}^{(g)}$ carries only the $h_t = h/t$ blocks $\{gh_t, \ldots, (g+1)h_t - 1\}$, for all $n = t \cdot n_{\max}$ coefficients. The Hadamard product $\bigodot_{g\in[\![t]\!]} \boldsymbol{v}^{(g)}$ multiplies across the t groups of blocks, and $\mathsf{Pr}_{N/2\to n}$ multiplies the h_t blocks within a group. At $k = \log h$ one has $n = N/2$, the product operator is the identity, and a product tree of depth $\log h$ replaces it, with no rotations. These vectors already fill every slot, so the trace after AccMul is the identity. Im2 and S2C are unchanged from ordinary SPRU. Like SHIP, $\mathsf{TUTTI}_{N/2}$ refreshes real values, so a complex vector of $N/2$ slots takes two refreshes (Section 2.2). The proposition records the slot contents, including the AccMul encodings; it uses the sequential slot order, which differs from the interleaved order of the example by a slot permutation.

Proposition 2 (2^k**-vector product).** *Let $1 \le k \le \log h$, $t = 2^k$, $h_t = h/t$, and $n = t \cdot n_{max}$. Let* ct *encrypt m under the block-binary secret s, with coefficients m_a of $X^{aN/n}$ for $a \in [\![n]\!]$,*

and let $\mathbf{C} \leftarrow \mathsf{DecMat}(\mathsf{ct}) \in \mathbb{Z}_q^{N\times n}$*, so that* $m_a = \sum_{i\in[\![N]\!]} s_i\, C_{i,a} \pmod q$ *as in* (2)*. For* $g \in [\![t]\!]$ *and* $u \in [\![B]\!]$*, define slot vectors in* $\mathbb{C}^{N/2}$ *by*

$$\tilde{s}^{u,(g)}_{rn+a} = s_{(gh_t+r)B+u}, \qquad \rho^{u,(g)}_{rn+a} = \delta \cdot \exp\Big(2\pi\mathrm{I}\, C_{(gh_t+r)B+u,\,a}/q\Big) \tag{7}$$

for $r \in [\![h_t]\!]$ *and* $a \in [\![n]\!]$*, and set*

$$\boldsymbol{v}^{(g)} = \sum_{u\in[\![B]\!]} \tilde{\boldsymbol{s}}^{u,(g)} \odot \boldsymbol{\rho}^{u,(g)}. \tag{8}$$

Then every slot $rn + a$ *of*

$$\mathsf{Pr}_{N/2\to n}\Big(\bigodot_{g\in[\![t]\!]} \boldsymbol{v}^{(g)}\Big) \tag{9}$$

holds

$$\delta^h \cdot \exp\big(2\pi\mathrm{I}\, m_a/q\big), \tag{10}$$

and the multiplicative depth of (9) *is* $k + \log h_t = \log h$*.*

Proof. Fix a slot $rn+a$ and a group g, and let $b = gh_t+r$. The secret has a single 1 in block b, at offset $j^\star_b$, so exactly one summand of (8) survives in that slot: $v^{(g)}_{rn+a} = \delta\cdot\exp(2\pi\mathrm{I}\, C_{bB+j^\star_b,\,a}/q)$, the root that block b selects for coefficient a. The Hadamard product multiplies these roots over $g \in [\![t]\!]$, at fixed r. The operator $\mathsf{Pr}_{N/2\to n}$ multiplies the slots congruent modulo n, that is, over $r \in [\![h_t]\!]$ at fixed a, and writes the result to each of them. Every block $b = gh_t + r \in [\![h]\!]$ thus contributes exactly once, and by (2) the product is

$$\begin{aligned}\prod_{b\in[\![h]\!]} \delta\cdot\exp\Big(2\pi\mathrm{I}\,\frac{C_{bB+j^\star_b,\,a}}{q}\Big) &= \delta^h\cdot\exp\Big(\frac{2\pi\mathrm{I}}{q}\sum_{i\in[\![N]\!]} s_i C_{i,a}\Big)\\ &= \delta^h\cdot\exp\Big(2\pi\mathrm{I}\,\frac{m_a}{q}\Big).\end{aligned}$$

For the depth, the Hadamard product of $t = 2^k$ vectors costs k levels, and $\mathsf{Pr}_{N/2\to n}$ costs $\log h_t$ more, since $(N/2)/n = h_t$.

At $k = \log h$ one has $h_t = 1$ and $n = N/2$, so $\mathsf{Pr}_{N/2\to n}$ is the identity and (9) is $\bigodot_{g\in[\![h]\!]} \boldsymbol{v}^{(g)}$. For $n < N/2$ the output holds the n values h_t times, the sparse packing of Section 2.1. The proposition holds for an arbitrary m, since DecMat reads only the coefficients of $X^{aN/n}$; the cosets of Section 4.5 rely on this. Homomorphically, $\boldsymbol{v}^{(g)}$ is the accumulator of the encodings $S^{(g)}_u = \mathsf{Ecd}(\tilde{\boldsymbol{s}}^{u,(g)})$, encrypted in the key, and $E^{(g)}_u = \mathsf{Ecd}(\boldsymbol{\rho}^{u,(g)})$, built in the clear; Algorithm 2 records TUTTI_n, and Algorithm 5 its key generation.

4.2 Dimension reduction

Algorithm 2 builds the vectors $\boldsymbol{v}^{(g)}$ from $t\cdot B$ single AccMuls. The load parallelizes, as low latency requires, but the absolute work is still large: for $t = h$, it takes N single AccMuls. We therefore restrict the block-binary secret to the shape $s(X) = s'(X^R)$ for a divisor $R \mid B$, independently of the packing of Section 4.1. The unique 1 in each block then lies on a multiple of R, that is, $s_i = 0$ whenever $R \nmid i$. Algorithms 5 and 2 let u run only over the grid $0, R, 2R, \ldots, B-R$, so the key $(\mathsf{cs}^{(g)}_u)_{g\in[\![t]\!],\,u\in R\cdot[\![B/R]\!]}$ has $t\cdot B/R$ ciphertexts instead of $t\cdot B$. The product depth stays $\log h$, and the Hadamard product, the operator $\mathsf{Pr}_{N/2\to n}$ and $\mathsf{S2C}_n \circ \mathsf{Im2}$ are unchanged. The AccMul load $t\cdot B/R = t\cdot(N/R)/h$ thus depends on the degree N/R that security requires, not on the ring degree N.

Algorithm 2 TUTTI_n bootstrapping

Input: q, key $(\mathsf{cs}_u^{(g)})$, ciphertext ct encrypting n coefficients
Input: $\delta = (q/(4\pi\Delta))^{1/h}$, and $t, h_t, n_{\max}, n$ as above
Output: Refreshed ciphertext ct'

$\mathbf{C} \leftarrow \mathsf{DecMat}(\mathsf{ct})$
for $g = 0$ to $t-1$ **do**
  $\mathsf{acc}^{(g)} \leftarrow 0$
  for $u = 0$ to $B-1$ **do**
    for all $r \in [\![h_t]\!]$, $a \in [\![n]\!]$ **do**
      $\rho_{rn+a}^{u,(g)} \leftarrow \delta \cdot \exp(2\pi\mathrm{I} \cdot C_{(gh_t+r)B+u,\,a}/q)$
    $E_u^{(g)} \leftarrow \mathsf{Ecd}(\boldsymbol{\rho}^{u,(g)})$
    $\mathsf{acc}^{(g)} \leftarrow \mathsf{Add}(\mathsf{acc}^{(g)}, \mathsf{ExtMult}(\mathsf{cs}_u^{(g)}, E_u^{(g)}))$
  $\mathsf{acc}^{(g)} \leftarrow \mathsf{RS}(\mathsf{acc}^{(g)})$ ▷ one division, by P (Section 5.1)
$\mathsf{acc} \leftarrow \bigodot_{g \in [\![t]\!]} \mathsf{acc}^{(g)}$
$\mathsf{acc} \leftarrow \mathsf{Pr}_{N/2 \to n}(\mathsf{acc})$
return $\mathsf{ct}' \leftarrow \mathsf{S2C}_n(\mathsf{Im2}(\mathsf{acc}))$

The constraint has a price in security. By Proposition 1, it leaves a block-binary key of degree N/R, which Section 3 estimates at the row $(N/R, h)$ of Table 2. For instance, for $N = 2^{14}$ and $h = 64$, the unconstrained key allows $\log q_0 \leq 209$, whereas $R = 8$, of degree $N/R = 2^{11}$, allows $\log q_0 \leq 27$. The largest usable R is the one whose row still admits the base prime, and Section 6.1 gives this choice for each of our parameter sets.

Secrets in the subring $\mathbb{Z}[X^R]/(X^N + 1)$ are not new. Ring switching moves ciphertexts into such a subring [GHPS12], and subring secret encapsulation switches to a dense secret of the subring before bootstrapping [MHWW25]. Here the subring is applied to the sparse key alone, and its purpose is the smaller AccMul load.

4.3 Parallelization

After dimension reduction, Algorithm 2 runs $t \cdot B/R$ independent single AccMuls (Section 2.3), which may be assigned to distinct cores. The encoding is arithmetic-bound and takes the larger share of a single AccMul on one core; the plaintext–ciphertext product takes most of the rest and reads its key ciphertext once, without reuse. The full-slot key has $t \cdot B/R$ ciphertexts at the top modulus, several gigabytes for our full-slot sets (Section 6.1). On a machine with limited memory bandwidth, the key traffic, rather than the arithmetic, then bounds how far more cores help, and Section 4.5 shrinks the key for such machines. Otherwise, the arithmetic of AccMul keeps scaling, so on many cores it stops being the largest stage, and $\mathsf{S2C}$ and the product operator set the latency (Section 6.2).

4.4 Placement of $\mathsf{S2C}$

Without its final $\mathsf{S2C}_n$, Algorithm 2 maps the coefficients of its input to the slots of its output. As in SHIP, we therefore apply $\mathsf{S2C}$ to the input instead of the output, and TUTTI_n stops after $\mathsf{Im2}$. The bootstrapping then becomes the identity on slots, and the transform becomes cheaper: an opening $\mathsf{S2C}_{N/2}$ runs once, at the two lowest primes, whereas a finishing one runs on every prime of the output modulus. We evaluate the opening $\mathsf{S2C}$ by baby-step giant-step, as in SHIP [HS14,CHKS25]; the baby steps are independent, so the transform parallelizes across the diagonals. The transform is the half-width one of [Seu25], $\mathbf{U}'\mathsf{ct}$ followed by the addition of its conjugate, whose result has slots $2\,\mathrm{Re}(\mathbf{U}'\boldsymbol{z})$. A fresh encryption has real slots, but a computed ciphertext does not, since its noise also has an imaginary part. A complex $\mathbf{U}'$ would fold that imaginary part into the refreshed values, so we use $\mathrm{Re}(\mathbf{U}')$ instead. The

Algorithm 3 TUTTI_n with h_t cosets

Input: Ciphertext ct with $N/2$ real slots; shared TUTTI_n key; masks $\boldsymbol{\mu}^{(j)}$
Output: Refreshed ciphertext ct', with the same slot order

1: $\mathsf{ct} \leftarrow \mathsf{S2C}_{N/2}(\pi(\mathsf{ct}))$ ▷ one merged linear map at the two lowest primes
2: **for** $j = 0$ to $h_t - 1$ **do** ▷ parallel cosets, shared key
3: $\quad \mathsf{ct}_j \leftarrow \mathsf{ExtMult}(\mathsf{ct}, X^{-2j})$
4: $\quad \mathsf{ct}_j \leftarrow \mathsf{TUTTI}_n(\mathsf{ct}_j)$ through $\mathsf{Im2}$
5: $\quad \mathsf{ct}_j \leftarrow \mathsf{ExtMultR}(\mathsf{ct}_j, \boldsymbol{\mu}^{(j)})$
6: **return** $\mathsf{ct}' \leftarrow \sum_{j \in [\![h_t]\!]} \mathsf{ct}_j$

result on real slots is unchanged, the imaginary part cancels in the addition, and no operation is added.

Speed-ups of a finishing $\mathsf{S2C}$ for medium numbers of slots [Seu25,Zhe26] rely on the repeated layout of SPRU's output, which an opening $\mathsf{S2C}$ does not see, and they would also have to make up for the larger modulus. A finishing $\mathsf{S2C}$ at output level ℓ works modulo q_ℓ, with $\ell + 1$ primes, whereas an opening one works at level 1, with two. A plaintext product costs one multiplication per prime, so $\ell + 1$ against 2. A key switch, with one digit per prime, extends each of its $\ell + 1$ digits to the $\ell + 2$ primes of Pq_ℓ and multiplies it with the key there, so it costs about $(\ell + 1)(\ell + 2)$ operations on single primes, against $2 \cdot 3 = 6$. A finishing transform is thus about $(\ell + 1)/2$ times more expensive in its plaintext products and $(\ell + 1)(\ell + 2)/6$ times in its key switches, and a speed-up of a finishing $\mathsf{S2C}$ would have to exceed these factors to compete with an opening one.

4.5 Sharing keys across cosets

On a machine whose memory bandwidth limits the plaintext–ciphertext products (Section 4.3), the monomial shifts of SPRU can shrink the key. For $1 \le k < \log h$, $t = 2^k$, $n = t \cdot n_{\max}$ and $h_t = h/t$, one TUTTI_n refreshes the coefficients of $X^{aN/n}$ with a key of $t \cdot B/R$ ciphertexts, h_t times smaller than the full-slot key. The h_t cosets $\mathsf{ct} \cdot X^{-2j}$, $j \in [\![h_t]\!]$, share that key and refresh the coefficients of $X^{2\ell}$ with $\ell \equiv j \pmod{h_t}$, hence all $N/2$ real coefficients, in independent pipelines. Algorithm 3 shows the idea, and one $\mathsf{S2C}_{N/2}$ at the bottom serves all cosets. The output of a coset holds its n values h_t times (Proposition 2), so a 0–1 mask $\boldsymbol{\mu}^{(j)}$ keeps the copy in the slot block $\{jn, \dots, jn + n - 1\}$, and the masked outputs are added, at the cost of one level. Slot $jn + a$ then holds the coefficient of index

$$\pi(jn + a) = j + a\, h_t.$$

To return the slots in their original order, the transform is preceded by the slot permutation that moves slot s to slot $\pi(s)$, which merges into its diagonals at no cost.

The cosets interpolate between SPRU, whose h monomial shifts are the cosets of $k = 0$, and $\mathsf{TUTTI}_{N/2}$, the single coset of $k = \log h$.

Advantages and drawbacks. Cosets divide the key traffic of AccMul by h_t, since one pass over the key serves every coset, which helps where memory bandwidth limits AccMul; the key's memory footprint shrinks by h_t as well. In exchange, the mask costs one level, cores spent on cosets are lost inside each TUTTI_n, and every coset repeats its product operator: h_t trees of $t - 1$ multiplications and $\log h_t$ rotations, against one tree of $h - 1$ multiplications. On our machine, this pays off only on few threads (Section 6.2): with many threads, AccMul stays arithmetic-bound and the repeated product operators dominate.

5 Optimizations

This section describes the remaining optimizations of our implementation. Section 5.1 covers those that apply to all our parameter sets (Table 5): they reduce the number of exponentiations, set the digit size of the sparse-secret encapsulation, and save a prime of the modulus chain. Section 5.2 covers the single-slot sets `S13` and `S13m2`, for which we remove sequential automorphisms and improve the approximation of the sine.

5.1 Optimizations for all parameter sets

One exponentiation per coefficient. Algorithm 1 builds $2n$ encodings E_u with $N/2$ roots each, so a direct implementation computes nN roots, and Algorithm 2 builds $t \cdot B/R$ encodings. Most of these exponentiations are redundant. Each root is $\delta \cdot \exp(2\pi \mathrm{I}\, C_{i,a}/q)$ for an entry of $\mathsf{DecMat}(\mathsf{ct})$. Since $c_1 \cdot X^i$ is a negacyclic shift of c_1, every entry is, up to sign, one of the N coefficients of c_1, with a coefficient of c_0 added in row $i = 0$. A change of sign conjugates the root, so we compute the cosine and the sine once per distinct value and apply the sign to the sine. This reduces the number of exponentiations from nN to $N + n$, or to $2N$ with the second harmonic of Section 5.2.

Digit size in the sparse-secret encapsulation. We switch to the sparse key at q_0 with a BV decomposition [BV11] (Section 3) into $D = \lceil \log q_0/\omega \rceil$ digits in base 2^ω (Appendix C). The choice of ω balances noise against running time. Smaller digits give a smaller switching error (Equation (14) in Appendix C), but the switch costs $2D$ multiplications and $2D - 1$ additions in $\mathcal{R}_{q_0}$. For the full-slot sets, this cost is negligible compared with the rest of the bootstrapping, since $\mathcal{R}_{q_0}$ is a ring of a single prime, so we take $\omega = 1$. The single-slot sets bootstrap in a few milliseconds, so the switch is no longer negligible. On `S13`, we use $\omega = 3$, which gives $D = \lceil 24/3 \rceil = 8$, that is, 16 multiplications and 15 additions. This gains 0.6 to 1 bit of precision over $\omega = 4$, because the other error terms of the single-slot circuit are small (Section 5.2).

Plaintext–ciphertext products at the auxiliary modulus. As written, Algorithm 1 spends a prime of the chain on the rescaling after its plaintext–ciphertext products. Like SHIP, which handles these products in its auxiliary modulus, we use the auxiliary prime P for this rescaling instead. The only products at the top of the bootstrapping are the products $\mathsf{ExtMult}(\mathsf{cs}_u^{(g)}, E_u^{(g)})$ of AccMul. We supply the key ciphertexts at the top modulus extended by P, and we encode $E_u^{(g)}$ at a scale of about P. The products are summed modulo PQ, and one division by P, the ModDown that ends every key switch, rescales the sum (Algorithm 2). After AccMul, the ciphertext is therefore at the top of the ordinary modulus chain. In Algorithm 2, and in Algorithm 7 of Appendix B, the rescaling after AccMul denotes this division by P.

5.2 Optimizations for the single-slot sets

Fewer sequential steps. On `S13` and `S13m2`, the trace $\mathsf{Tr}_{N/2 \to h}$ of Algorithm 1 adds up the B offsets of each block with $\log(N/(2h))$ sequential automorphisms at the top level, 7 on `S13`. These automorphisms run one after the other, whereas AccMuls are independent, so we replace the trace by more AccMuls. The key now holds one ciphertext cs_j per offset $j \in [\![B]\!]$, which we call a *key row*. Slot ℓ of cs_j encrypts s_{bB+j} with $b = \ell \bmod h$, and slot ℓ of E_j holds the matching root $\delta \cdot \exp(2\pi \mathrm{I}\, C_{bB+j,0}/q)$. As each block of s has a single 1, the sum $\sum_j \mathsf{ExtMult}(\mathsf{cs}_j, E_j)$ holds in slot ℓ the root selected by block b, ready for $\mathsf{Pr}_{h \to 1}$ and Im2.

In this layout, each single AccMul is cheap. The encoding E_j has period h in the slots, hence only $2h$ nonzero coefficients, so its encoding and its RNS representation take transforms of $2h$ instead of N points, 128 times fewer on S13. The drawback is memory traffic rather than memory size: every bootstrapping reads the whole key, which is B rows or 128 MiB on S13, and at a latency of a few milliseconds this means tens of GB per second. The idea is the same as in Section 4.1, applied to a different operator. There, we split the blocks over several accumulators to save rotations of the product operator; here, we split the offsets over several key rows to save the rotations of the trace.

A key row can also hold several offsets, which reduces this traffic and interpolates between this circuit and Algorithm 1. For a power of two $G \mid B$, let slot ℓ in group $g = \lfloor \ell/h \rfloor \bmod G$ take offset $g + Gj$ of its block. This leaves B/G rows, and $\mathsf{Tr}_{Gh\to h}$, with $\log G$ sequential automorphisms, adds the groups before the product. The case $G = 1$ is the circuit above, and $G = B/2$ is Algorithm 1. A machine with more memory bandwidth per core favours a smaller G, since the extra rows are then cheap to read. S13 and S13m2 use $G = 2$, with one automorphism and a 64 MiB key on S13. Our implementation of TUTTI_n for $2 \le n \le n_{\max}$ (Section 4) uses the same layout with $G = 1$: one accumulator of B/R rows, in which slot ℓ serves block $\lfloor \ell/n \rfloor \bmod h$ of coefficient $\ell \bmod n$, followed by $\mathsf{Pr}_{hn\to n}$.

A second harmonic. The bootstrapping relies on the approximation $\sin\varphi \approx \varphi$, whose error is significant at the small ratio $\gamma = 2^{4.5}$ of S13. For inputs $m_0/\Delta \in [-1, 1]$, the angle $\varphi = 2\pi m_0/q$ satisfies $|\varphi| \le 2\pi/\gamma$ (Section 2.3). The output $\frac{q}{2\pi\Delta}\sin\varphi$ of Im2 then differs from $m_0/\Delta = \frac{q}{2\pi\Delta}\varphi$ by about $\frac{q}{2\pi\Delta}\cdot\varphi^3/6$, which is largest at the ends of the range. A second harmonic of the sine cancels this term. Indeed,

$$\alpha \sin\varphi + \beta \sin 2\varphi = (\alpha + 2\beta)\,\varphi - (\alpha + 8\beta)\,\frac{\varphi^3}{6} + \cdots,$$

so the weights $\alpha = 4/3$ and $\beta = -1/6$ cancel the cubic term and keep the slope $\alpha + 2\beta = 1$. We therefore return

$$\frac{q}{2\pi\Delta}\Big(\frac{4}{3}\sin\varphi - \frac{1}{6}\sin 2\varphi\Big) = \frac{q}{2\pi\Delta}\Big(\varphi - \frac{\varphi^5}{30} + \cdots\Big),$$

and since the slope at $\varphi = 0$ is still 1, an error in the input reaches the output at the same size. This is the five-point central difference [BF11] applied to $e^{\mathrm{I}\varphi}$, and the two-term sine series of Jutla and Manohar [JM22].

The circuit above changes in four places.

- *Public roots.* Each coefficient $C_{i,0}$ also gives the root of the doubled angle, $\exp(4\pi\mathrm{I}\,C_{i,0}/q)$. The matrix DecMat(ct) itself is unchanged.
- *Layout.* The even and the odd slots carry the first and the second harmonic of block $\lfloor \ell/2 \rfloor \bmod h$, which doubles every slot period above. Since the product multiplies h factors, the key rows carry the h-th roots of the weights, $(4/3)^{1/h}$ and $(1/6)^{1/h}e^{\mathrm{I}\pi/h}$, where $e^{\mathrm{I}\pi/h}$ supplies the sign.
- *Trace and product.* The partial trace becomes $\mathsf{Tr}_{2Gh\to 2h}$ and the product $\mathsf{Pr}_{2h\to 2}$, which leaves $\frac{4}{3}\delta^h e^{\mathrm{I}\varphi}$ and $-\frac{1}{6}\delta^h e^{2\mathrm{I}\varphi}$ in neighbouring slots.
- *Final sum.* A rotation by one slot adds these two values before Im2.

The second harmonic costs one sequential automorphism, but no level and no key row. Together with the 3-bit BV digits of Section 5.1, it raises the precision on S13 from 5.21 bits with Algorithm 1 to 6.53 bits (Table 6).

The technique would equally apply to TUTTI_n with $n \le N/(4h)$, whose slots hold at least $2h$ copies of each value, h of them spare for the second harmonic. We did not implement this variant.

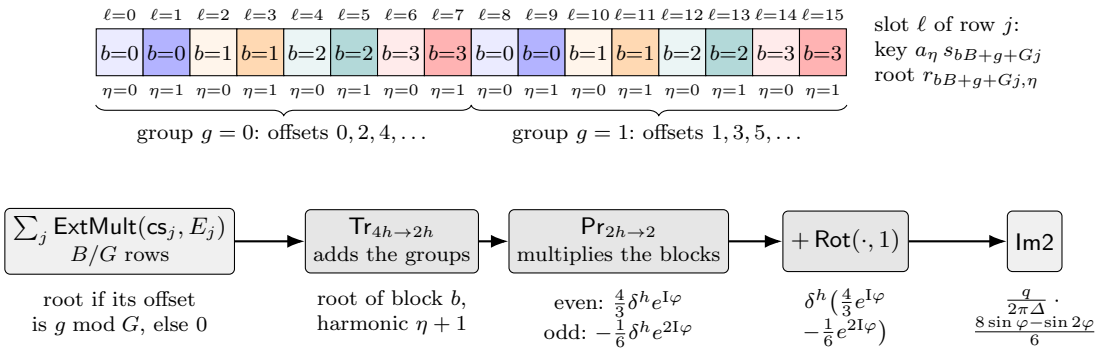


Fig. 4. Single-slot bootstrapping with the second harmonic and $G = 2$ groups, drawn for $h = 4$. Top: one period of $2Gh = 16$ slots; the colour is the block b, light and dark the first and the second harmonic, and the braces the groups. Bottom: the steps after AccMul, with the slot contents after each step. With $G = 1$, the partial trace is dropped.

The complete circuit. Figure 4 shows the single-slot circuit with G groups and the second harmonic, as used by `S13` with $G = 2$: one period of the slots and the steps after AccMul. The pseudocode is in Appendix B. For a single slot, C_i denotes the constant coefficient of $c_1 \cdot X^i$, to which that of c_0 is added for $i = 0$, so that $m_0 = \sum_i s_i C_i \bmod q$ as in (2). Slot $\ell \in [\![N/2]\!]$ of key row j serves harmonic $\eta(\ell) + 1$ of the secret coefficient with index $i_j(\ell)$, where

$$\eta(\ell) = \ell \bmod 2, \qquad i_j(\ell) = bB + g + Gj,$$

with the block $b = \lfloor \ell/2 \rfloor \bmod h$ and the group $g = \lfloor \ell/(2h) \rfloor \bmod G$ of slot ℓ, so every encoding has period $2Gh$ in the slots. In the figure, a_η is the amplitude of harmonic $\eta + 1$, that is $a_0 = (4/3)^{1/h}$ or $a_1 = (1/6)^{1/h} e^{\mathrm{I}\pi/h}$, and $r_{i,\eta} = \delta \cdot \exp(2\pi \mathrm{I}\,(\eta + 1)\, C_i/q)$ is the corresponding root. Without the second harmonic, $\eta(\ell) = 0$, $b = \ell \bmod h$, $g = \lfloor \ell/h \rfloor \bmod G$, the amplitude is 1, the trace and the product become $\mathsf{Tr}_{Gh \to h}$ and $\mathsf{Pr}_{h \to 1}$, and the final rotation is dropped.

6 Experimental Results

We implemented TUTTI, PaCo and SHIP in the `vfhe` library [CCC+26], so that all three share the same arithmetic and the same thread control. We run all experiments on an `r8i.metal-96xl` instance on AWS (2x Intel Xeon 6975P-C processors), at 1, 8, 32, 48, 192 and 384 threads. Latency is the mean over at least three runs after a warm-up. Memory is the peak resident set of the process, keys included. Precision follows the metric of SHIP [CHKS25]: for real inputs $z_i = 2i/n - 1 \in [-1, 1]$ and the real parts z'_i of the output slots, it is the median over runs of $-\log \max_i |z'_i - z_i|$. The worst slot is stricter than an average over the slots, and the inputs include the end of the range, -1. For a single message, a run is 4096 bootstraps of uniform $z \in [-1, 1]$, with the maximum taken over them, and the median is over three runs. The output of TUTTI is at the scale from which one multiplication returns to the input scale, so that bootstrapped ciphertexts can be multiplied without further rescaling.

6.1 Parameter selection

Table 5 lists the parameter sets of our experiments. `S13` and `S13m2` are the single-slot sets described below; all others refresh all $N/2$ real slots with $\mathsf{TUTTI}_{N/2}$. `LL13`, `LL14`, `HT14` and `HT15` are compared with the SHIP sets of the same name [CHKS25]; `LL13` is a low-precision full-slot set with one multiplicative level. On TUTTI's `LL13`, $R = 4$ needs $h = 64$, which spreads the budget over ten primes of about 21 bits. At such primes the rounding noise of

rescaling and ModDown dominates, and its variance grows with $\|s\|^2$ of the dense key. Both TUTTI and SHIP therefore use, on LL13, a dense ternary key of Hamming weight 512 instead of a uniform one ($\|s\|^2 \approx 2N/3$): its noise is 1.7 bits lower, for a budget of 212 instead of 214 bits (Table 1), and it raises the precision on LL13 from 3.9 to 4.9 bits for TUTTI and, in our implementation, from 4.7 to 5.0 bits for SHIP. The ratio γ trades the error of the sine, which falls as γ^{-2}, against the noise, which the output carries multiplied by $\gamma/2\pi$; with the lower noise, TUTTI on LL13 takes $\gamma = 12$. We run SHIP on its published chains, although each exceeds the dense-key cap of Table 1 at its degree, by 4.5 to 14 bits, which puts it at 125.5 to 126.3 bits of security in our model (column Sec. of Table 6).

The modulus chain is listed from the bottom: the base prime q_0, one prime consumed by the opening S2C, the multiplicative levels left to the application, and $\log h$ primes consumed by the product of the h AccMul outputs. A single auxiliary prime P serves both the ModDown after AccMul and every GHS key switch (rotations of S2C, automorphisms of the product and of Im2, relinearization), with one digit per prime of the current level. The sparse-secret encapsulation is the BV key switch of Section 5.1, at q_0 only, with $D = \lceil \log q_0/\omega \rceil$ digits in base 2^ω. The sparse key has degree N/R (Proposition 1), so each pair $(N/R, h)$ with $\log q_0$ is a row in Table 2. In S13, S13m2, LL13 and HT14, every prime, including P, lies below 2^{30}, so our implementation stores each residue in a 32-bit word, which is faster than the 64-bit words of LL14 and HT15. A key ciphertext holds $2N$ words per prime of the chain, P included, so the bootstrapping key takes 1.25, 10, 4.5 and 36 GiB on LL13, LL14, HT14 and HT15, and 64 and 72 MiB on S13 and S13m2, whose rows have a smaller period (Section 5.2).

How a set is chosen. Two caps determine a set: the dense-key budget of Table 1 bounds $\log PQ$ at degree N, and the row $(N/R, h)$ of Table 2 bounds $\log q_0$. All our full-slot sets take $h = 64$ and the largest R whose row admits the base prime, which is $R = 8$ for the 27-bit prime of HT14 and the 50-bit prime of HT15, and $R = 4$ for LL13 and LL14. The levels left to the application are those of the SHIP set of the same name, and the remaining budget goes to the other primes. The S2C prime only scales the encoded transform, since γ fixes the input scale q_0/γ, so it is the first to shrink when the budget is tight, to 18 bits on HT14. The Boot primes take what is left, and they matter: on HT14, raising two of them from 25 to 26 bits raises the precision from 4.73 to 5.15 bits at the same latency.

The ratio γ is set last. The error of the sine alone limits the precision to 7.3 bits at $\gamma = 2^5$ and to 19.3 bits at $\gamma = 2^{11}$. The high-precision sets LL14 and HT15 therefore take $\gamma = 2^{11}$, and their precision is then limited by the noise. On LL13, every bit of the budget counts, so the primes are requested at fractional widths: each width b takes the largest unused prime $p \equiv 1 \bmod 2N$ below 2^b, for a total of 211.99 bits. This set also scales its output by a constant s, 1.035 at $\gamma = 12$, chosen so that the error of $s \cdot \frac{\gamma}{2\pi} \sin(2\pi v/\gamma)$ equioscillates over $v \in [-1, 1]$. The constant is folded into the encodings of AccMul, so it costs no operation, and it raises the floor set by the sine from 4.5 to 6.4 bits.

Single-slot sets. A single message needs no S2C, which saves a prime, so it can stay at a smaller degree than a full-slot refresh. S13 and S13m2 use $N = 2^{13}$ and $h = 32$. The attacks of Section 3 allow this: the row $(2^{13}, 32)$ of Table 2 caps the base prime at 27 bits, and Table 3 costs the combinatorial attacks there, whereas the estimate $T = S^{0.25}$ in SPRU [CK27, Section 4.5] forced $N = 2^{15}$ even for $n \leq 128$ slots. These values are close to the limit. A Hamming weight near 16 makes the cap on q_0 too small for the computation, and $N = 2^{12}$ with $h = 64$ no longer fits the bootstrapping into the 106-bit budget of that degree. Keeping $N = 2^{13}$ with $h = 64$ would be possible, but the product would then have depth 6 and 63 ciphertext multiplications instead of 31.

Table 5. Parameter sets of TUTTI. R is the subring reduction of the sparse key, dnum the number of digits of a GHS key switch at the top level, D (base 2^ω) the number of digits of the BV key switch at q_0, and $\gamma = q_0/\Delta_{\text{in}}$ the ratio of Section 2.3 for the base prime and the scale of the bootstrapped message. `LL13` uses a dense key of Hamming weight 512; its primes are the largest primes $p \equiv 1 \bmod 2N$ below 2^b for fractional widths b, here rounded to bits.

Set	Slots	N	h	R	$\log PQ$	dnum	(prime bits) × (# primes)					D (ω)	γ
							Base	S2C	Mult	Boot	Aux		
`S13`	1	2^{13}	32	1	208.4	7	24×1	—	20×1	27×5	30×1	8 (3)	$2^{4.5}$
`S13m2`	1	2^{13}	32	1	213.6	8	23×1	—	20×2	25×5	27×1	8 (3)	$2^{3.7}$
`LL13`	$N/2$	2^{13}	64	4	212.0	9	24×1	17×1	20×1	$19, 20, 21, 22 \times 3$	25×1	24 (1)	$2^{3.6}$
`LL14`	$N/2$	2^{14}	64	4	430.0	9	48×1	37×1	37×1	43×6	50×1	48 (1)	2^{11}
`HT14`	$N/2$	2^{14}	64	8	429.5	17	27×1	18×1	23×9	$25 \times 4, 26 \times 2$	30×1	27 (1)	2^{5}
`HT15`	$N/2$	2^{15}	64	8	769.0	17	50×1	39×1	39×9	46×6	53×1	50 (1)	2^{11}

Both sets use the circuit of Section 5.2, with $G = 2$ groups and the second harmonic. `S13` spends its budget on larger Boot primes and a larger P, for $\log PQ = 208.4$ out of 214 bits: Boot primes of 27 instead of 25 bits alone gain one bit. `S13m2` fits a second multiplicative level in the same ring instead, at $\log PQ = 213.6$ bits, with the same precision of 6.53 bits and 0.5 ms more on 32 threads.

6.2 Main results

Table 6 compares TUTTI with SHIP on the parameter sets of Table 5 and their SHIP counterparts [CHKS25], and lists the single-slot set `S13`. Our implementation of SHIP also uses the BV key switch at q_0 (Section 5.1). We note that conventional bootstrapping parallelizes poorly: SHIP reports a latency on 32 cores 2.5× below the smallest it observed with HEaaN, on 48 cores, and we refer to [CHKS25] for that comparison.

Precision. The precision in Table 6 is measured, not derived, but the construction of Section 4 explains why we expect TUTTI to match the precision of SPRU. TUTTI changes where the factors of a message are stored, not how each factor is computed. In both schemes, a factor is a sum of B/R products of a key ciphertext with a public root, which in SPRU the trace completes, and the h factors of a message are multiplied at depth $\log h$. TUTTI$_n$ computes more of these products because it refreshes more messages, but they go to different slots, so each slot still receives a sum of B/R products. It also replaces rotations of the product operator by products of distinct accumulators, which removes key switches. The precision of a set therefore depends on its primes and its ratio γ, chosen as in Section 6.1, as it does for SPRU.

TUTTI and SHIP. Both schemes evaluate the decryption equation $m_a = \sum_i s_i C_{i,a}$ on the unit circle with a sparse key at q_0, and both end in a product of h factors. They differ in which side of this matrix–vector product is encrypted. TUTTI computes $\mathbf{C} \cdot s$: public roots built from $\mathbf{C}$ multiply encryptions of the secret's block indicators, as plaintext–ciphertext products. SHIP computes, in effect, $\mathbf{S} \cdot c_1$, with $\mathbf{S}$ the negacyclic matrix of s, so that $c_1 \cdot s = \mathbf{S} \cdot c_1$: an encryption of each secret position rotates the public roots blindly, so the secret acts as an encrypted permutation. Both share everything around that product: S2C, the encapsulation, the product of the h factors, and the final conjugation. The comparison is therefore between AccMul and blind rotation, and Table 7 lists what follows.

TUTTI trades total work for parallelism and targets latency on many cores; neither scheme dominates. Single-threaded, SHIP is faster on `LL14` and `HT15`. On `LL13` and `HT14`, TUTTI

Table 6. Bootstrapping latency of TUTTI and SHIP, in ms, by number of threads. Lev. is the number of multiplicative levels left to the application after bootstrapping. Bold marks the faster scheme on sets with both, and the lowest latency of the row on the single-slot sets. For SHIP, precision and parameters are the ones reported in the original paper [CHKS25], except on its LL13, which, like TUTTI's LL13, runs with a dense key of Hamming weight 512 (Section 6.1); its precision is that of our implementation.

Set	Scheme	Slots	Lev.	Sec.	Latency (ms) at T threads						Prec.	Mem.
				(bits)	1	8	32	48	192	384	(bits)	(GB)
S13	TUTTI	1	1	≥ 128	25.5	5.97	**4.55**	4.68	6.27	6.28	6.53	0.3
S13m2	TUTTI	1	2	≥ 128	30.1	6.89	5.04	**4.92**	6.93	5.88	6.53	0.3
LL13	TUTTI	4096	1	≥ 128	**1345**	**214**	**72.5**	**47.2**	**45.8**	**49.6**	4.89	2.6
LL13	SHIP	4096	1	125.5	2942	412	132	135	84.0	91.0	4.96	6.8
LL14	TUTTI	8192	1	≥ 128	5424	774	260	**183.5**	**113**	**115**	17.0	17.5
LL14	SHIP	8192	1	126.3	**3717**	**511**	**186**	183.5	163	169	16.9	22.0
HT14	TUTTI	8192	9	≥ 128	**5251**	**789**	**257**	**155**	**110**	**113**	5.16	9.5
HT14	SHIP	8192	9	125.7	12393	1743	491	467	211	202	4.55	29.5
HT15	TUTTI	16384	9	≥ 128	23107	3143	1152	**839**	**424**	**427**	18.44	66.1
HT15	SHIP	16384	9	125.9	**22120**	**2897**	**959**	919	651	690	18.1	103.5

Table 7. How TUTTI and SHIP produce one of the h factors, for the full-slot sets of Table 5. S2C, the encapsulation, the product of the factors, and the final conjugation are common to both.

	SHIP	TUTTI
Matrix–vector product	encrypted secret, public roots rotated: $\mathbf{S} \cdot c_1$	public roots, encrypted indicators: $\mathbf{C} \cdot s$
Position within θ	column step: 4θ independent ExtMults	—
Remaining position	3 base-4 digits and one alignment	B/R independent ExtMults
Key switches in sequence	4, each over up to 4 keys	none
Unit of parallelism	h branches	$t \cdot B/R$ single AccMul s
Fewer slots	only S2C gets lighter	AccMul load and key shrink with t

is faster at every thread count. From 48 threads on, TUTTI is faster on LL14, HT14 and HT15, on LL14 at 48 threads only by 0.02%, a tie. On LL13, where both schemes refresh 4096 messages at about the same precision, TUTTI is 1.8 to 2.9× faster and needs 2.6× less memory. Both schemes stream keys from memory: the AccMul load reads its key once, and each blind-rotation branch reads rotation keys through a sequential chain of digits. SHIP's blind rotation key-switches at the largest non-auxiliary modulus, so its larger sets need several auxiliary primes to control that error, four on HT15. TUTTI's AccMul phase runs once and in parallel, so one auxiliary prime suffices, and HT15 stays 99 bits below the 868-bit cap, room for more than two more multiplicative levels than SHIP. Concurrent work builds on SHIP and reduces its latency further [OLG+26]; it appeared too recently for a comparison.

Where the time goes. Figure 5 splits the latency by stage. AccMul dominates on one thread, with 76% of the time on LL13, and it scales well, by 46× on 192 threads. S2C stops scaling at about 32 threads, and the product operator gains little beyond it, so on 192 threads the three stages are of similar size, on HT14 34, 39 and 34 ms. Blind rotation scales as well as AccMul on LL13, by 49×, but starts 2.7× higher, and it takes most of SHIP's time at every thread count.

Thread allocation. A budget of T threads means three things in our implementation. The library, and with it S2C, may use T threads. AccMul uses at most one pinned worker per physical core, so at most 192, because workers on SMT siblings were slower; the single-slot

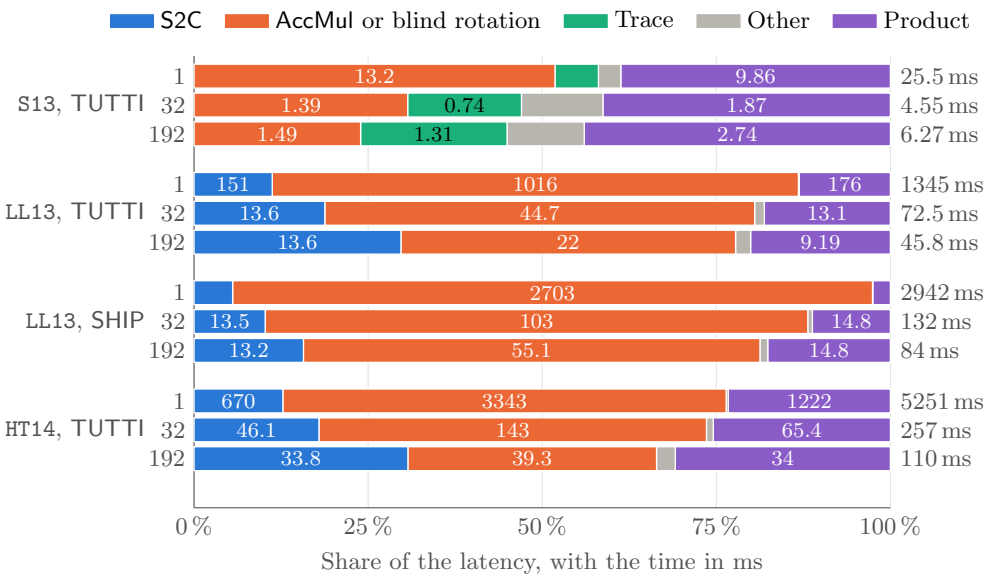


Fig. 5. Share of each stage in the latency, with its time in ms where the segment is wide enough, by number of threads, and the total on the right. Other is the switch into the sparse key together with Im2, or with the conjugation on SHIP.

sets use at most 32. The trace, the product and Im2 run on a pool of at most 96 workers, one socket's physical cores, because all 192 cores were slower on the product. Going from 192 to 384 threads therefore only changes S2C, and it does not help. On S13, the latency is lowest at 32 threads: a key switch of its short chain has at most about 56 parallel tasks, so further threads only add synchronization, and the trace and the product get slower from 32 to 192 threads, from 0.74 to 1.31 and from 1.87 to 2.74 ms. The machine has 6 NUMA nodes. We interleave the memory of the process over them, each AccMul worker allocates its buffers on its own node, and we move the key so that each worker's data sit on its node. This also opts the process out of automatic NUMA balancing, whose page faults cost about 60 ms of AccMul per bootstrap on the 36 GiB key of HT15.

Varying the number of messages. Figure 1, in the introduction, shows the latency of TUTTI$_n$ on HT14 for n from 1 to $N/2$, together with SHIP on all slots and PaCo at 128 slots. Dimension reduction cuts the AccMul load, so for $n \leq 2^8$ the latency barely changes with n at any thread count: most of the time goes to the product operator, which is sequential and independent of n. Above 2^8, the AccMul load grows linearly in n and the S2C load as $\sqrt{n}$, and parallelization absorbs most of that growth. SHIP, in contrast, gains at most 15% from bootstrapping fewer messages, since only its S2C becomes lighter. Figure 6 adds the precision and the memory. Precision rises as n falls because the maximum runs over fewer slots, and memory falls with the key.

Repeated bootstrapping. We also ran three cycles of a bootstrap followed by $x \mapsto 2x^2 - 1$, which maps $[-1, 1]$ to itself. With the real diagonals of Section 4.4, each bootstrap keeps its precision against its own input: 4.88, 4.85 and 4.89 bits on LL13.

PaCo. PaCo [CS25] targets a medium number of slots: one instance bootstraps $C \leq N/(4h)$ coefficients, and its parallel variant covers all $N/2$ slots with $N/(2C)$ instances side by side, which favours throughput over latency. We run it within the modulus budget of HT14, with $N = 2^{14}$, $h = 32$ and $C = N/(4h) = 128$: its circuit takes 8 levels (2 for its partial C2S, 5 for

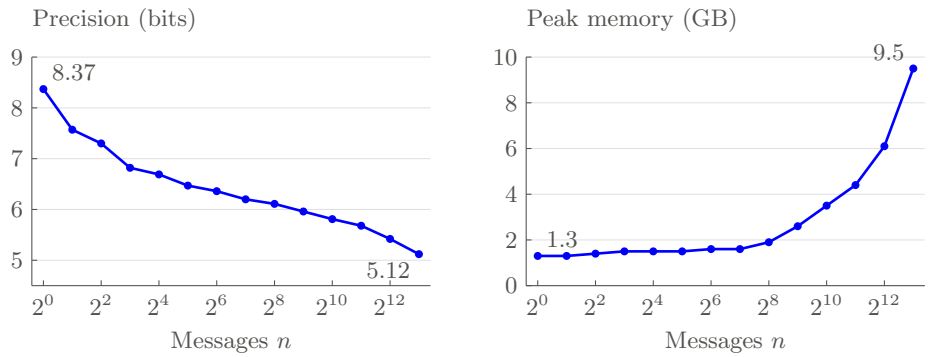


Fig. 6. Precision and peak memory of TUTTI$_n$ on `HT14`, for the runs of Figure 1, with uniform random messages.

the product and 1 for its final transform) and leaves 7 to the application, against 9 for TUTTI. One instance takes 1307, 149 and 118 ms on 1, 32 and 192 threads, at 6.1 bits of precision, whereas TUTTI refreshes the same 128 slots in 245, 35 and 33 ms (Figure 1); covering all $N/2$ slots with $2h = 64$ instances takes PaCo 7.1 s on 192 threads, against 110 ms for TUTTI. Our choice of ring favours PaCo: its original parameters use $N = 2^{15}$ and 2^{16} with $h = 64$ [CS25], but an instance that bootstraps only $N/(4h)$ coefficients is faster in the smaller ring.

TFHE-rs. We ran TFHE-rs 1.8.1 [Zam22] on our machine with the parameters of Zama's benchmark [Zam26]: a keyswitch followed by a programmable bootstrapping (PBS) on 4-bit messages, with failure probability 2^{-128}. The PBS is the *"multi-bit"* variant based on [ZYL+18], which processes 4 secret coefficients per step and so exposes parallelism within one bootstrapping. Its fastest configuration takes 7.3 ms, on 32 threads. On the same number of threads, single-slot TUTTI takes 4.55 ms, 1.6× less, and TUTTI$_n$ takes 25 to 35 ms for up to 256 slots on `HT14` and 15 ms for 256 slots on `LL13`. Zama reports 4.57 ms on a different machine, with two AMD EPYC 9R45 processors (192 cores) [Zam26], which single-slot TUTTI matches on ours. The nominal precisions are similar: 6.5 bits for single-slot TUTTI, and 5.1 to 8.4 bits for TUTTI$_n$ on `HT14`, rising as n falls since the maximum runs over fewer slots. The guarantees differ, however: the 4 bits of TFHE-rs are exact up to a failure probability of 2^{-128}, whereas our precision is a median worst-slot error, for which we did not measure a failure probability. TFHE-rs also evaluates a lookup table during bootstrapping [DM15,BR15,BGGJ19], and the two schemes also differ in key sizes and security assumptions [BV14]. We therefore see single-slot TUTTI and TFHE as complementary: at matching precision, an application may use TUTTI for fixed-point multiplications and switch to TFHE [BGGJ20,LHH+21] to evaluate arbitrary functions.

On a laptop. We also ran single-slot TUTTI and TFHE-rs on a MacBook Pro with an Apple M5 Pro (15 cores), with an AArch64 implementation of TUTTI whose outputs are bit-identical to those of the implementation above, so that its precision is unchanged. On one thread, TUTTI takes 12.0 ms on `S13`, and TFHE-rs 1.8.1, on the same 4-bit parameters, 13.1 ms with its single-threaded PBS. On 14 threads, the fastest configuration of both, TUTTI takes 2.4 ms and TFHE-rs 4.5 ms with its multi-bit PBS and a parallel keyswitch, 1.9× more. For TUTTI these are medians over three processes of 500 bootstraps each.

Cosets. We implemented the coset variant of Section 4.5 with $h_t = 8$ on `HT15`, at the same precision. It shrinks the bootstrapping key by almost h_t, from 36 to 4.75 GiB, since its mask

Table 8. Cosets on `HT15` with $h_t = 8$, against full-slot TUTTI: latency in ms at T threads, precision in bits, key in GiB and peak memory in GB.

	1	8	32	48	192	384	Prec.	Key	Mem.
Full slots	23107	**3143**	**1152**	**839**	**424**	**427**	18.44	36	66.1
Cosets	**21779**	3483	1205	992	972	971	18.63	4.75	48.9

needs one more prime, but is faster than full-slot TUTTI only on few threads, by 6% on one thread, where its AccMul, reading each key row once for all cosets, is 23% faster with the same arithmetic. Table 8 gives the full comparison. From 48 threads on, cosets lose: every coset repeats its product operator, and the cores spent on cosets are lost inside each TUTTI_n.

Acknowledgements

We thank Jean-Sébastien Coron and Tim Seuré for helpful discussions. Large language models assisted in writing and editing the text and in developing the implementation and the security-estimation scripts; the authors verified all results, proofs and claims. Robin Köstler was supported by the ERC Advanced Grant Cloudmap no. 787390. Antonio Guimarães was supported by grants JDC2024-055789-I and CEX2024-001471-M, funded by MICIU/AEI/10.13039/501100011033 and ESF+, and by the ERC Proof of Concept grant VERIFHE (GA: 101287502).

References

AB24. Wei Ao and Vishnu Naresh Boddeti. AutoFHE: Automated adaption of CNNs for efficient evaluation over FHE. In *33rd USENIX Security Symposium, USENIX Security 2024*, pages 2173–2190. USENIX Association, 2024.

Alb17. Martin R. Albrecht. On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. In *Advances in Cryptology - EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 103–129. Springer, 2017.

AP13. Jacob Alperin-Sheriff and Chris Peikert. Practical bootstrapping in quasilinear time. In *Advances in Cryptology - CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2013.

AP14. Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In *Advances in Cryptology - CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 297–314. Springer, 2014.

APS15. Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

APS26. Martin R. Albrecht, Rachel Player, and Sam Scott. Lattice estimator. https://github.com/malb/lattice-estimator, 2026. Software.

BCK+23. Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, Jai Hyun Park, and Damien Stehlé. HERMES: Efficient ring packing using MLWE ciphertexts and application to transciphering. In *Advances in Cryptology - CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 37–69. Springer, 2023.

BCKS24. Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, and Damien Stehlé. Bootstrapping bits with CKKS. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology - EUROCRYPT 2024, Part II*, volume 14652 of *Lecture Notes in Computer Science*, pages 94–123. Springer, 2024.

BEHZ16. Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. A full RNS variant of FV-like somewhat homomorphic encryption schemes. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 423–442. Springer, 2016.

BF11. Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Brooks/Cole, 9th edition, 2011.

BGGJ19. Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. Simulating homomorphic evaluation of deep learning predictions. In *Cyber Security Cryptography and Machine Learning - CSCML 2019*, Lecture Notes in Computer Science, pages 212–230. Springer, 2019.

BGGJ20. Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. CHIMERA: Combining ring-LWE-based fully homomorphic encryption schemes. *Journal of Mathematical Cryptology*, 14(1):316–338, 2020.

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science, ITCS 2012*, pages 309–325. ACM, 2012.

BLL+26. Lei Bi, Yijian Liu, Xianhui Lu, Junjie Luo, and Kunpeng Wang. An improved hybrid dual attack on LWE with sparse secrets and its application to FHE. Cryptology ePrint Archive, Paper 2026/1060, 2026.

BLLW22. Lei Bi, Xianhui Lu, Junjie Luo, and Kunpeng Wang. Hybrid dual and meet-LWE attack. In Khoa Nguyen, Guomin Yang, Fuchun Guo, and Willy Susilo, editors, *Information Security and Privacy - 27th Australasian Conference, ACISP 2022, Wollongong, NSW, Australia, November 28–30, 2022, Proceedings*, volume 13494 of *Lecture Notes in Computer Science*, pages 168–188. Springer, 2022. https://eprint.iacr.org/2022/1330.

BR15. Jean-François Biasse and Luis Ruiz. FHEW with efficient multibit bootstrapping. In *Progress in Cryptology - LATINCRYPT 2015*, volume 9230 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2015.

Bra12. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Advances in Cryptology - CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 868–886. Springer, 2012.

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, *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*, volume 13269 of *Lecture Notes in Computer Science*, pages 521–541. Springer, 2022.

BV11. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22–25, 2011*, pages 97–106. IEEE Computer Society, 2011.

BV14. Zvika Brakerski and Vinod Vaikuntanathan. Lattice-based FHE as secure as PKE. In *Innovations in Theoretical Computer Science, ITCS 2014*, pages 1–12. ACM, 2014.

CCC+26. Ignacio Cascudo, Anamaria Costache, Daniele Cozzo, Adrien Dubois, Dario Fiore, Antonio Guimarães, Robin Köstler, Michał Osadnik, Alin-Petru Roșu, and Eduardo Soria-Vazquez. The `vfhe` Library. https://github.com/vfhe/vfhe, 2026. Apache-2.0 license.

CCS19. Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 34–54. Springer, 2019. https://eprint.iacr.org/2018/1043.pdf.

CGGI16. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2016.

CGGI20. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

CH18. Hao Chen and Kyoohyung Han. Homomorphic lower digits removal and improved FHE bootstrapping. In *Advances in Cryptology - EUROCRYPT 2018, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 315–337. Springer, 2018.

CHH18. Jung Hee Cheon, Kyoohyung Han, and Minki Hhan. Faster homomorphic discrete fourier transforms and improved FHE bootstrapping. Cryptology ePrint Archive, Paper 2018/1073, 2018.

CHK+18a. 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, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 360–384. Springer, 2018.

CHK+18b. Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. A full RNS variant of approximate homomorphic encryption. In Carlos Cid and Michael J. Jacobson Jr., editors, *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*, volume 11349 of *Lecture Notes in Computer Science*, pages 347–368. Springer, 2018.

CHKS25. Jung Hee Cheon, Guillaume Hanrot, Jongmin Kim, and Damien Stehlé. SHIP: A shallow and highly parallelizable CKKS bootstrapping algorithm. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4–8, 2025, Proceedings, Part III*, volume 15603 of *Lecture Notes in Computer Science*, pages 398–428. Springer, 2025. https://eprint.iacr.org/2025/784.

CJK26. Jung Hee Cheon, Daehyun Jang, and Jaehee Kang. SWIFT: Shallow and SIMD-aware CKKS functional bootstrapping for low-latency. Cryptology ePrint Archive, Paper 2026/1163, 2026.

CK27. Jean-Sébastien Coron and Robin Köstler. Low-latency bootstrapping for CKKS using roots of unity. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2027, 2027. To appear. First version: Cryptology ePrint Archive, Paper 2025/651, April 2025. https://eprint.iacr.org/2025/651.

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, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

CP19. Benjamin R. Curtis and Rachel Player. On the feasibility and impact of standardising sparse-secret LWE parameter sets for homomorphic encryption. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC 2019*, pages 1–10. ACM, 2019.

CS25. Jean-Sébastien Coron and Tim Seuré. PaCo: Bootstrapping for CKKS via partial CoeffToSlot. In Goichiro Hanaoka and Bo-Yin Yang, editors, *Advances in Cryptology - ASIACRYPT 2025 - 31st International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, VIC, Australia, December 8–12, 2025, Proceedings, Part VII*, volume 16251 of *Lecture Notes in Computer Science*, pages 36–67. Springer, 2025. https://eprint.iacr.org/2025/886.

DDLL13. Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2013.

DKMS24. Gabrielle De Micheli, Duhyeong Kim, Daniele Micciancio, and Adam Suhl. Faster amortized FHEW bootstrapping using ring automorphisms. In *Public-Key Cryptography - PKC 2024, Part IV*, volume 14604 of *Lecture Notes in Computer Science*, pages 322–353. Springer, 2024.

DM15. Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640. Springer, 2015.

EGR25. Austin Ebel, Karthik Garimella, and Brandon Reagen. Orion: A fully homomorphic encryption framework for deep learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2025, Volume 2*, pages 734–749. ACM, 2025.

Fly66. Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

Fly72. Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

FV12. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012.

Gen09. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009*, pages 169–178. ACM, 2009.

GHPS12. Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. Ring switching in BGV-style homomorphic encryption. In *Security and Cryptography for Networks - SCN 2012*, volume 7485 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2012.

GHS12a. Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping in fully homomorphic encryption. In *Public Key Cryptography - PKC 2012*, volume 7293 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

GHS12b. Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *Advances in Cryptology - EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012.

GHS12c. Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 850–867. Springer, 2012.

GIKV23. Robin Geelen, Ilia Iliashenko, Jiayi Kang, and Frederik Vercauteren. On polynomial functions modulo p^e and faster bootstrapping for homomorphic encryption. In *Advances in Cryptology - EUROCRYPT 2023, Part III*, volume 14006 of *Lecture Notes in Computer Science*, pages 257–286. Springer, 2023.

GP25. Antonio Guimarães and Hilder V. L. Pereira. Fast amortized bootstrapping with small keys and polynomial noise overhead. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025*, pages 2967–2981. ACM, 2025.

GPvL23. Antonio Guimarães, Hilder V. L. Pereira, and Barry van Leeuwen. Amortized bootstrapping revisited: Simpler, asymptotically-faster, implemented. In *Advances in Cryptology - ASIACRYPT 2023, Part VI*, volume 14443 of *Lecture Notes in Computer Science*, pages 3–35. Springer, 2023.

GV23. Robin Geelen and Frederik Vercauteren. Bootstrapping for BGV and BFV revisited. *Journal of Cryptology*, 36(2):12, 2023.

HGSW03. Nick Howgrave-Graham, Joseph H. Silverman, and William Whyte. A meet-in-the-middle attack on an NTRU private key. Technical Report 004, NTRU Cryptosystems, 2003. Version 2.

HJO26. Jianhua Hou, Haodong Jiang, and Tabitha Ogilvie. Careful with the ring! concrete hardness gaps between LWE and MLWE. In Nadia Heninger and Mike Rosulek, editors, *Advances in Cryptology - CRYPTO 2026 - 46th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17–20, 2026, Proceedings, Part III*, volume 16802 of *Lecture Notes in Computer Science*, pages 462–497. Springer, 2026. Merges ePrint 2026/279 and 2026/366. https://eprint.iacr.org/2026/279.

HKLS22. Minki Hhan, Jiseung Kim, Changmin Lee, and Yongha Son. A hybrid of lattice-reduction and meet-LWE via near-collision on Babai's plane. Cryptology ePrint Archive, Paper 2022/1473, 2022. Revised October 2025. https://eprint.iacr.org/2022/1473.

HPS98. Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21–25, 1998, Proceedings*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998.

HS14. Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014.

HS15. Shai Halevi and Victor Shoup. Bootstrapping for HElib. In *Advances in Cryptology - EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 641–670. Springer, 2015.

JM22. Charanjit S. Jutla and Nathan Manohar. Sine series approximation of the mod function for bootstrapping of approximate HE. In *Advances in Cryptology - EUROCRYPT 2022, Part II*, volume 13276 of *Lecture Notes in Computer Science*, pages 491–520. Springer, 2022.

JVC18. Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium, USENIX Security 2018*, pages 1651–1669. USENIX Association, 2018.

Kim25. Jaehyung Kim. Bootstrapping GBFV with CKKS. Cryptology ePrint Archive, Paper 2025/888, 2025.

KM21. Elena Kirshanova and Alexander May. How to find ternary LWE keys using locality sensitive hashing. In Maura B. Paterson, editor, *Cryptography and Coding - 18th IMA International Conference, IMACC 2021, Virtual Event, December 14–15, 2021, Proceedings*, volume 13129 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2021. https://eprint.iacr.org/2021/1255.

KSK+18. Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC Medical Genomics*, 11(Suppl 4):83, 2018.

KSS24. Jaehyung Kim, Jinyeong Seo, and Yongsoo Song. Simpler and faster BFV bootstrapping for arbitrary plaintext modulus from CKKS. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, CCS 2024*, pages 2535–2546. ACM, 2024.

LHH+21. Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *42nd IEEE Symposium on Security and Privacy, SP 2021*, pages 1057–1073. IEEE, 2021.

LLL+22. Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. Low-complexity deep convolutional neural networks on fully homomorphic encryption using multiplexed parallel convolutions. In *Proceedings of the 39th International Conference on Machine Learning, ICML 2022*, volume 162 of *Proceedings of Machine Learning Research*, pages 12403–12422. PMLR, 2022.

LM21. Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. In *Advances in Cryptology - EUROCRYPT 2021*, volume 12696 of *Lecture Notes in Computer Science*, pages 648–677. Springer, 2021.

LMSS23. Changmin Lee, Seonhong Min, Jinyeong Seo, and Yongsoo Song. Faster TFHE bootstrapping with block binary keys. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ACM ASIACCS, 2023. https://eprint.iacr.org/2023/958.

LPR13. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM*, 60(6):43:1–43:35, 2013. Preliminary version in EUROCRYPT 2010. Full version: https://eprint.iacr.org/2012/230.

LW23a. Feng-Hao Liu and Han Wang. Batch bootstrapping I: A new framework for SIMD bootstrapping in polynomial modulus. In *Advances in Cryptology - EUROCRYPT 2023, Part III*, volume 14006 of *Lecture Notes in Computer Science*, pages 321–352. Springer, 2023.

LW23b. Feng-Hao Liu and Han Wang. Batch bootstrapping II: Bootstrapping in polynomial modulus only requires $\tilde{O}(1)$ FHE multiplications in amortization. In *Advances in Cryptology - EUROCRYPT 2023, Part III*, volume 14006 of *Lecture Notes in Computer Science*, pages 353–384. Springer, 2023.

May21. Alexander May. How to meet ternary LWE keys. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, Virtual Event, August 16–20, 2021, Proceedings, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 701–731. Springer, 2021.

MHWW24. Shihe Ma, Tairong Huang, Anyu Wang, and Xiaoyun Wang. Accelerating BGV bootstrapping for large p using null polynomials over $\mathbb{Z}_{p^e}$. In *Advances in Cryptology - EUROCRYPT 2024, Part II*, volume 14652 of *Lecture Notes in Computer Science*, pages 403–432. Springer, 2024.

MHWW25. Shihe Ma, Tairong Huang, Anyu Wang, and Xiaoyun Wang. Practical dense-key bootstrapping with subring secret encapsulation. In *Advances in Cryptology - ASIACRYPT 2025*, 2025. https://eprint.iacr.org/2025/1594.

MS18. Daniele Micciancio and Jessica Sorrell. Ring packing and amortized FHEW bootstrapping. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPIcs*, pages 100:1–100:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

MYJK25. Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. THOR: Secure transformer inference with homomorphic encryption. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025*, pages 3765–3779. ACM, 2025.

Odl90. Andrew M. Odlyzko. The rise and fall of knapsack cryptosystems. In Carl Pomerance, editor, *Cryptology and Computational Number Theory*, volume 42 of *Proceedings of Symposia in Applied Mathematics*, pages 75–88. American Mathematical Society, 1990.

OLG+26. Rending Ouyang, Jian Liu, Peimin Gao, Pan Xiao, Heng Zhang, Man Ho Au, Cong Zhang, Kui Ren, and Chun Chen. CKKS bootstrapping in less than 10ms. Cryptology ePrint Archive, Paper 2026/2119, 2026.

PDH+25. Thales B. Paiva, Gabrielle De Micheli, Syed Mahbub Hafiz, Marcos A. Simplício Jr., and Bahattin Yildiz. Faster amortized bootstrapping using the incomplete NTT for free. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):520–543, 2025.

PZM+24. Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. BOLT: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy, SP 2024*, pages 4753–4771. IEEE, 2024.

SC19. Yongha Son and Jung Hee Cheon. Revisiting the hybrid attack on sparse secret LWE and application to HE parameters. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC 2019*, pages 11–20. ACM, 2019.

Seu25. Tim Seuré. SCORE: A SlotToCoeff optimization for real-vector encryption in CKKS. Cryptology ePrint Archive, Paper 2025/1983, 2025.

SV10. Nigel P. Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography - PKC 2010*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010.

SV14. Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Designs, Codes and Cryptography*, 71(1):57–81, 2014.

Zam22. Zama. TFHE-rs: A pure Rust implementation of the TFHE scheme for boolean and integer arithmetics over encrypted data. https://github.com/zama-ai/tfhe-rs, 2022.

Zam26. Zama. TFHE-rs benchmarks: Programmable bootstrapping on CPU. https://docs.zama.org/tfhe-rs/get-started/benchmarks/cpu/cpu-programmable-bootstrapping, 2026. Accessed October 2026.

Zhe26. Xiaopeng Zheng. Faster CoeffToSlot and SlotToCoeff for sparsely packed ciphertexts with application to CKKS bootstrapping. Cryptology ePrint Archive, Paper 2026/1023, 2026.

ZYL+18. Tanping Zhou, Xiaoyuan Yang, Longfei Liu, Wei Zhang, and Ningbo Li. Faster bootstrapping with multiple addends. *IEEE Access*, 6:49868–49876, 2018.

A Details on SPRU, TUTTI and SHIP

SPRU key generation. Algorithm 4 generates the bootstrapping key of Algorithm 1. For each sub-block u, it packs the secret coefficients that (4) selects and encrypts their encoding under the public key.

Algorithm 4 SPRU key generation [CK27]

Input: Block-binary s, weight h, and $n \leq B/2$
Output: Key $(\mathsf{cs}_u)_{0 \leq u < 2n}$

for $u = 0$ to $2n - 1$ **do**
  for all $0 \leq k < B/(2n)$, $0 \leq b < h$, $0 \leq a < n$ **do**
    $\tilde{s}^u_{\iota(k,b,a)} \leftarrow s_{bB+j(u,k)}$
  $S_u \leftarrow \mathsf{Ecd}(\tilde{\boldsymbol{s}}^u)$
  $\mathsf{cs}_u \leftarrow \mathsf{Enc}_{\mathsf{pk}}(S_u)$
return $(\mathsf{cs}_u)_{0 \leq u < 2n}$

TUTTI key generation. Algorithm 5 generates the key of Algorithm 2: for each group g and offset u, it packs the secret coefficients that (7) selects and encrypts their encoding under the public key.

Algorithm 5 TUTTI_n key generation

Input: Block-binary s, weight h, and $1 \leq k \leq \log h$
Input: $t = 2^k$, $h_t = h/t$, $n_{\max} = B/2$, $n = t \cdot n_{\max}$
Output: Key $(\mathsf{cs}^{(g)}_u)_{g \in [\![t]\!],\, u \in [\![B]\!]}$

for $g = 0$ to $t - 1$ **do**
  for $u = 0$ to $B - 1$ **do**
    for all $r \in [\![h_t]\!]$, $a \in [\![n]\!]$ **do**
      $\tilde{s}^{u,(g)}_{rn+a} \leftarrow s_{(gh_t+r)B+u}$
    $S^{(g)}_u \leftarrow \mathsf{Ecd}(\tilde{\boldsymbol{s}}^{u,(g)})$
    $\mathsf{cs}^{(g)}_u \leftarrow \mathsf{Enc}_{\mathsf{pk}}(S^{(g)}_u)$
return $(\mathsf{cs}^{(g)}_u)_{g \in [\![t]\!],\, u \in [\![B]\!]}$

SHIP parameter sets. Table 9 lists the SHIP sets of Table 6, from [CHKS25, Table 2], as our benchmark builds them. The chain is listed from the bottom, as in Table 5: the base prime, the S2C prime, the data levels, five primes for the product of depth 5, and the auxiliary primes. Every set has $h = 31$, multiplexer base $\nu = 4$, and the published half-width $w = \max(175, \lfloor N/(2h) \rfloor)$. Every set exceeds the dense-key cap of Table 1 at its degree, by 4.5 to 14 bits: 216.5 bits against the 212 of the dense key of Hamming weight 512 that `LL13` runs with, 437.0 and 438.9 against 430, and 882.0 against 868. We keep the published chains, which favours SHIP in the comparison.

B Single-slot bootstrapping with fewer sequential steps

Algorithms 6 and 7 give the single-slot circuit of Section 5.2, with G groups and the second harmonic, in the notation of that section.

Table 9. The SHIP parameter sets of our comparison [CHKS25, Table 2]. A prime of b bits is a prime $p < 2^b$ with $p \equiv 1 \bmod 2N$ chosen by the prime generator of our implementation, and $\log PQ$ is computed on the actual primes; on LL13 the S2C and Mult primes have 19.5 and 19.3 bits. dnum is the number of digits of a GHS key switch at the top level, θ the column step, and $\lceil \log_4 \lceil 2w/\theta \rceil \rceil$ the number of multiplexer digits; $\gamma = q_0/\Delta_{\text{in}}$ as in Table 5.

Set	N	$\log PQ$	dnum	(prime bits) × (# primes)					θ	w	digits	γ
				Base	S2C	Mult	Boot	Aux				
LL13	2^{13}	216.5	8	25×1	20×1	20×1	25×5	28×1	6	175	3	2^5
LL14	2^{14}	437.0	4	48×1	37×1	37×1	43×5	50×2	9	264	3	2^{11}
HT14	2^{14}	438.9	8	28×1	23×1	23×9	25×5	29×2	9	264	3	2^5
HT15	2^{15}	882.0	4	50×1	39×1	39×9	46×5	53×4	17	528	3	2^{11}

Algorithm 6 Single-slot key generation with G groups and the second harmonic

Input: Block-binary s, weight h, $B = N/h$, a power of two $G \mid B$
Output: Key $(\mathsf{cs}_j)_{0 \le j < B/G}$

$a_0 \leftarrow (4/3)^{1/h}$, $a_1 \leftarrow (1/6)^{1/h} e^{\mathrm{I}\pi/h}$ ▷ h-th roots of the weights $\frac{4}{3}$ and $-\frac{1}{6}$
for $j = 0$ to $B/G - 1$ **do**
  for $\ell = 0$ to $N/2 - 1$ **do**
    $\tilde{s}^j_\ell \leftarrow a_{\eta(\ell)} \cdot s_{i_j(\ell)}$
  $S_j \leftarrow \mathsf{Ecd}(\tilde{\boldsymbol{s}}^j)$
  $\mathsf{cs}_j \leftarrow \mathsf{Enc}_{\mathsf{pk}}(S_j)$ ▷ at the top modulus times P (Section 5.1)
return $(\mathsf{cs}_j)_{0 \le j < B/G}$

Algorithm 7 Single-slot bootstrapping with G groups and the second harmonic

Input: q, key $(\mathsf{cs}_j)_{j < B/G}$, ciphertext ct under the sparse key, $\delta = (q/(4\pi\Delta))^{1/h}$
Output: Refreshed ciphertext ct′

for $i = 0$ to $N - 1$ **do** ▷ $2N$ exponentiations
  $r_{i,0} \leftarrow \delta \cdot \exp(2\pi\mathrm{I}\, C_i/q)$, $r_{i,1} \leftarrow \delta \cdot \exp(4\pi\mathrm{I}\, C_i/q)$
acc $\leftarrow 0$
for $j = 0$ to $B/G - 1$ **do** ▷ independent rows
  for $\ell = 0$ to $N/2 - 1$ **do**
    $\rho^j_\ell \leftarrow r_{i_j(\ell),\, \eta(\ell)}$
  $E_j \leftarrow \mathsf{Ecd}(\boldsymbol{\rho}^j)$ ▷ $4Gh$ coefficients, transforms of $4Gh$ points
  acc $\leftarrow \mathsf{Add}(\mathsf{acc}, \mathsf{ExtMultR}(\mathsf{cs}_j, E_j))$
acc $\leftarrow \mathsf{Tr}_{2Gh \to 2h}(\mathsf{acc})$ ▷ adds the groups; $\log G$ automorphisms
acc $\leftarrow \mathsf{Pr}_{2h \to 2}(\mathsf{acc})$ ▷ multiplies the blocks; $\log h$ automorphisms and levels
acc $\leftarrow \mathsf{Add}(\mathsf{acc}, \mathsf{Rot}(\mathsf{acc}, 1))$ ▷ adds the two harmonics
return ct′ $\leftarrow \mathsf{Im2}(\mathsf{acc})$

C BV and GHS key switching

This appendix recalls the two key switches compared in Section 3 and counts their cost. Throughout, $\mathsf{ct} = (c_0, c_1) \in \mathcal{R}^2_{q_\ell}$ encrypts a message under $\overline{\mathsf{sk}} = (1, \bar{s})$, so $c_0 + c_1 \cdot \bar{s} = m + e \pmod{q_\ell}$, and the output should encrypt the same message under $\mathsf{sk} = (1, s)$. In the sparse-secret encapsulation, $\bar{s}$ is the dense key, s the sparse key, and $q_\ell = q_0$. When an automorphism produced (c_0, c_1), the permutation of coefficients is not part of the counts below, which cover the switch itself.

C.1 The BV decomposition

The decomposition of Brakerski and Vaikuntanathan [BV11] works entirely modulo q_ℓ. It writes each coefficient of c_1, taken in $\{0, \dots, q_\ell - 1\}$, in base $W = 2^\omega$ with

$$D = \lceil \log q_\ell / \omega \rceil$$

digits. This produces digit polynomials $c_{1,0}, \dots, c_{1,D-1}$ with coefficients in $\{0, \dots, W-1\}$ such that

$$c_1 = \sum_{i=0}^{D-1} c_{1,i}\, W^i \pmod{q_\ell}. \tag{11}$$

The key consists of the samples $\mathsf{swk}_i = (b_i, a_i) \in \mathcal{R}_{q_\ell}^2$ under sk of the multiples $W^i \bar{s}$,

$$b_i + a_i \cdot s = W^i \bar{s} + e_i \pmod{q_\ell}, \tag{12}$$

with e_i the error of the i-th sample. The switched ciphertext is

$$(c_0', c_1') = (c_0, 0) + \sum_{i=0}^{D-1} c_{1,i} \cdot \mathsf{swk}_i \pmod{q_\ell}. \tag{13}$$

Substituting (12) and then (11) gives

$$\begin{aligned} c_0' + c_1' \cdot s &= c_0 + \sum_{i=0}^{D-1} c_{1,i}(W^i \bar{s} + e_i) \\ &= c_0 + c_1 \cdot \bar{s} + \sum_{i=0}^{D-1} c_{1,i} e_i \pmod{q_\ell}, \end{aligned}$$

so the message is preserved and $\sum_i c_{1,i} e_i$ is the switching error. A coefficient of a ring product collects at most N terms, and the digits are at most $W - 1$, so

$$\Big\| \sum_{i=0}^{D-1} c_{1,i} e_i \Big\|_\infty \le D \cdot N \cdot (W-1) \cdot \max_i \|e_i\|_\infty. \tag{14}$$

Narrower digits lower this bound and add further pairs of multiplications. The published key is the tuple $(\mathsf{swk}_i)_{i<D}$, all of it in $\mathcal{R}_{q_\ell}^2$.

Algorithm 8 BV key switch with ω-bit digits

Input: $\mathsf{ct} = (c_0, c_1) \in \mathcal{R}_{q_\ell}^2$ under $\overline{\mathsf{sk}} = (1, \bar{s})$, digit width ω, keys swk_i as in (12)
Output: A ciphertext under $\mathsf{sk} = (1, s)$
1: $W \leftarrow 2^\omega$, $D \leftarrow \lceil \log q_\ell / \omega \rceil$
2: Decompose $c_1 = \sum_{i=0}^{D-1} c_{1,i} W^i$ with coefficients of $c_{1,i}$ in $\{0, \dots, W-1\}$
3: $(c_0', c_1') \leftarrow (c_0, 0)$
4: **for** $i = 0$ to $D - 1$ **do**
5: $\quad (c_0', c_1') \leftarrow (c_0', c_1') + c_{1,i} \cdot \mathsf{swk}_i \pmod{q_\ell}$
6: **return** (c_0', c_1')

C.2 Operation count

Equation (13) multiplies each of the D digits by both polynomials of swk_i, which is $2D$ multiplications in $\mathcal{R}_{q_\ell}$. The first output component is c_0 plus D products, which takes D additions, and the second is the sum of D products, which takes $D - 1$. The switch therefore performs

$$2D \text{ multiplications and } 2D - 1 \text{ additions in } \mathcal{R}_{q_\ell}.$$

Line 2 of Algorithm 8 only reads the base-W digits of the N coefficients of c_1. A digit width $\omega \ge \log q_\ell$ gives $D = 1$: the single digit is c_1 itself, and (14) multiplies the key error by a polynomial of coefficient size q_ℓ. Every useful decomposition therefore has $D \ge 2$.

C.3 The GHS switch

Gentry, Halevi, and Smart [GHS12c] avoid the digit sum by enlarging the modulus by an auxiliary prime P. The key is one sample $\mathsf{swk} = (b_{\mathsf{swk}}, a_{\mathsf{swk}}) \in \mathcal{R}^2_{Pq_\ell}$ under sk of $P \cdot \bar{s}$,

$$b_{\mathsf{swk}} + a_{\mathsf{swk}} \cdot s = P \cdot \bar{s} + e_{\mathsf{swk}} \pmod{Pq_\ell}.$$

Lift c_1 to $\mathcal{R}_{Pq_\ell}$ by its representative in $\{0, \dots, q_\ell - 1\}$ and set

$$(d_0, d_1) = c_1 \cdot \mathsf{swk} \pmod{Pq_\ell}, \qquad (c_0', c_1') = (c_0, 0) + \lfloor P^{-1}(d_0, d_1) \rceil \pmod{q_\ell}.$$

Then $d_0 + d_1 \cdot s = c_1 \cdot P \cdot \bar{s} + c_1 \cdot e_{\mathsf{swk}} \pmod{Pq_\ell}$, and the division by P recovers $c_1 \cdot \bar{s}$ with the scaled key error $c_1 \cdot e_{\mathsf{swk}}/P$ and a rounding error of at most $1/2$ per coefficient. With $P \geq q_\ell$ this error is small without any digits. The switch costs two multiplications modulo Pq_ℓ, about four of the size of one modulo q_ℓ when $\log P = \log q_\ell$, against the $2D$ of the BV switch.

The price is the modulus of the published sample: one sample under s modulo Pq_ℓ, against D samples modulo q_ℓ. For the sparse key at q_0, any auxiliary prime lifts that sample above the cap of Table 2 once q_0 sits at it, to at least 54 bits for a 27-bit q_0 with $\log P = \log q_0$. This is why the encapsulation uses the decomposition, in TUTTI as in our implementation of SHIP.

D Combinatorial attacks on block-binary and windowed secrets

This appendix details the combinatorial estimates of Section 3.2. Sections D.1 to D.5 cost Odlyzko's split T_{Odl} (Table 3) and May's Meet-LWE with Gaussian error guessing, T_{ML} (Table 10), on the anchored block-binary secret of SPRU, following May's paper [May21]. Odlyzko's meet-in-the-middle is his Section 3, applied to a product of blocks; Meet-LWE is his Algorithm 3, with the representation counts of his Section 7 rewritten for a binary secret and the list-cost recurrence of his Section 7.4. Section D.6 records the tree behind the guessing-free columns T_{free} and T_{ops}, Section D.7 checks that digits ± 2 (his Section 8) leave the rounded costs unchanged, Section D.8 treats the windowed secret of SHIP, and Section D.9 lists what the estimate leaves out.

The procedure prices the attack without building the lists. Which term dominates is read off his list-merging recurrence, and the polynomial factors of sorting, hashing, and the number of lists are omitted, as in his tables (his Section 2.3). Where his Lemma 1 replaces multinomial coefficients by binary entropy, the numbers below use the multinomials themselves,

$$\binom{n}{c_1, \dots, c_k} = \frac{n!}{c_1! \cdots c_k! \, (n - c_1 - \cdots - c_k)!},$$

evaluated as $\log(n!) = \ln \Gamma(n+1)/\ln 2$; this is the exact count behind his non-asymptotic tables.

D.1 The anchored instance

A block-binary secret lies in $\{0,1\}^N$ and has exactly one coefficient equal to 1 in each of h blocks of length $B = N/h$; SPRU fixes $s_0 = 1$, so the first block is known. Deleting its B columns leaves a secret of dimension $N' = N - B$ and weight $h' = h - 1$, uniform on the $S = B^{h'}$ placements of the remaining ones.

One RLWE sample is read in the coefficient embedding. May's Definition 1 asks for a uniform matrix, and his Section 2.1 notes that Ring-LWE samples are dependent and that

the attack does not use the dependence. We follow that choice: the N' equations are charged as uniform LWE samples modulo q, where q is the base prime q_0 of the row of Table 2.

The error is the discrete Gaussian of standard deviation 3.2 (Section 3), whereas May's secrets and errors are ternary (his Definition 1). Odlyzko's collision only needs the two sides of a genuine pair to differ by a short vector (his Section 3.1), which holds for this Gaussian at every modulus in the table; the guessing cost inside Meet-LWE depends on the alphabet and is computed in Section D.4.

Two rows of Table 3 serve as running examples: $N = 2^{13}$, $h = 32$, $\log q = 27$, with $B = 256$, $N' = 7936$, and $h' = 31$; and $N = 2^{11}$, $h = 64$, $\log q = 27$, with $B = 32$, $N' = 2016$, and $h' = 63$, the row of the sparse key of `LL13` and `HT14`.

D.2 Odlyzko's meet-in-the-middle on the blocks

May's Algorithm 1 splits a ternary secret by coordinates, with lists of size $\sqrt{S}$ after a permutation of the columns (his Section 3.2). The block-binary secret is a product of h' independent symbols from an alphabet of size B, so the same meet-in-the-middle splits the blocks instead, and no permutation is needed (Algorithm 9).

Algorithm 9 Odlyzko's meet-in-the-middle on the blocks, following Algorithm 1 of [May21]

Input: Matrix $\mathbf{A}$ with N' columns, target $\mathbf{b}$, block length B, weight h'
Output: A block-binary $\mathbf{s}$ of weight h' with $\mathbf{As} - \mathbf{b}$ short

$a \leftarrow \lceil h'/2 \rceil$. Fix a partition of the h' blocks into a left group of size a and a right group of size $h' - a$.
for all placements $\mathbf{s}_L$ of a single 1 in each left block **do**
    Store $\big(\mathbf{s}_L,\ \ell(\mathbf{A}_L\mathbf{s}_L)\big)$.
for all placements $\mathbf{s}_R$ of a single 1 in each right block **do**
    Store $\big(\mathbf{s}_R,\ \ell(\mathbf{b} - \mathbf{A}_R\mathbf{s}_R)\big)$.
for all pairs whose hash labels agree **do**
    $\mathbf{s} \leftarrow \mathbf{s}_L + \mathbf{s}_R$
    if $\mathbf{s}$ has one 1 in every block and $\mathbf{As} - \mathbf{b}$ is short **then**
        return $\mathbf{s}$

The hash ℓ is May's Equation (2): the most significant bit of each coordinate, with both labels stored on a residue where a short error can flip the bit. A coordinate of a genuine pair fails to collide only when it lies within $|e_i|$ of 0 or of $q/2$. At $\log q \geq 27$ and standard deviation 3.2, that fraction of coordinates is negligible beside the polynomial factors his Section 3 already drops. The algorithm enumerates secret halves only, and the larger list has size B^a, so time and memory are

$$T_{\mathrm{Odl}} = \lceil h'/2 \rceil \cdot \log B$$

bits. For an odd weight the larger half has one block more; fixing $s_0 = 1$ turns an even h into $h' = h - 1$ and leaves T_{Odl} unchanged, since $\lceil (h-1)/2 \rceil = h/2$. On the first running example, $a = 16$ and $T_{\mathrm{Odl}} = 16 \cdot \log 256 = 128$; the coordinate split of his Section 3.2, which ignores the blocks, costs 147.0 bits there.

D.3 Representation 1 for a binary secret

Meet-LWE writes the secret as a sum $\mathbf{s} = \mathbf{s}_1 + \mathbf{s}_2$ in many ways and merges lists of the summands (May's Algorithm 3 and Figure 2). A binary vector has, up to order, one writing of each one, $1 = 1 + 0 = 0 + 1$, and one of each zero, $0 = 0 + 0$: this is his Rep-0 (Section 6). Rep-1 (his Section 7) also writes

$$0 = 1 + (-1) = (-1) + 1,$$

and each such pair is an extra representation. The summands become ternary, so the enumerated set grows, but each secret is hit more often, so the merge keeps a smaller fraction of it; the list size at a level is the size of the set divided by the number of representations.

May's Section 7.1 starts from a balanced ternary secret, whose two signs each contribute a binomial coefficient, so the coefficients appear squared. A block-binary secret has no -1: the first level splits only the h' ones, and the cancellations are the only source of minus signs. From the second level on, the parent is an arbitrary ternary vector with p entries $+1$ and m entries -1, and his Section 7.2, with the rounding he allows for concrete parameters (his Section 2.1), applies to each sign separately. The block length B enters only T_{Odl} and the search space $S = B^{h'}$ against which Section 3.2 measures exponents; the representation counts depend on h' and N' alone, as for the ambient set his Section 7 enumerates.

Algorithm 10 Representation counts for a binary secret of weight h' (May, Section 7, unbalanced)

Input: Dimension N', weight h', and level parameters $\varepsilon^{(1)}, \dots, \varepsilon^{(k)} \geq 0$
Output: Search-space sizes $S^{(j)}$ and representation counts $R^{(j)}$ for $j = 1, \dots, k$, or rejection

$a \leftarrow \lceil h'/2 \rceil$, $p \leftarrow a + \varepsilon^{(1)}$, $m \leftarrow \varepsilon^{(1)}$
if $2\varepsilon^{(1)} > N' - h'$ **then return** rejection
$S^{(1)} \leftarrow \binom{N'}{p,\, m}$
$R^{(1)} \leftarrow \binom{h'}{a}\binom{N'-h'}{\varepsilon^{(1)},\, \varepsilon^{(1)}}$
for $j = 2$ to k **do**
    $z \leftarrow N' - p - m$
    if $2\varepsilon^{(j)} > z$ **then return** rejection
    $p' \leftarrow \lceil p/2 \rceil + \varepsilon^{(j)}$, $m' \leftarrow \lceil m/2 \rceil + \varepsilon^{(j)}$
    $R^{(j)} \leftarrow \binom{p}{\lceil p/2 \rceil}\binom{m}{\lceil m/2 \rceil}\binom{z}{\varepsilon^{(j)},\, \varepsilon^{(j)}}$
    $S^{(j)} \leftarrow \binom{N'}{p',\, m'}$
    $p \leftarrow p'$, $m \leftarrow m'$
return $(S^{(j)}, R^{(j)})_{j=1}^{k}$

At the first level, $\mathbf{s}_1$ has $a + \varepsilon^{(1)}$ entries $+1$ and $\varepsilon^{(1)}$ entries -1; the factor $\binom{h'}{a}$ chooses which of the ones lie in $\mathbf{s}_1$, and the multinomial in $N' - h'$ chooses the zero coordinates that carry the two kinds of cancellation. When the parent is balanced and both sign counts are even, the later levels reduce to the squared binomial of his Section 7.2.

On the first running example, one level with $\varepsilon^{(1)} = 4$ gives $a = 16$, so $\mathbf{s}_1$ has 20 entries $+1$ and 4 entries -1, and

$$\log S^{(1)} = 245.19, \qquad \log \binom{31}{16} = 28.16, \qquad \log \binom{7905}{4,\, 4} = 94.41,$$

so $\log R^{(1)} = 122.58$ and the level-1 list $S^{(1)}/R^{(1)}$ has 122.61 bits. With $\varepsilon^{(1)} = 0$, May's Rep-0 on this binary vector, the list has 134.8 bits, larger than both the list for $\varepsilon^{(1)} = 4$ and $T_{\mathrm{Odl}} = 128$.

D.4 The cost of one tree

Algorithm 10 fixes the shape of the lists. May's Algorithm 3 then builds them as a binary tree (his Figure 4) and guesses part of the error so that one representation of the true secret survives an exact filter.

His Section 5 takes that filter to be $r = \lfloor \log_q R^{(1)} \rfloor$ whole coordinates, because the range q^r must be at most the number of representations. Inner levels match exactly on a nonincreasing sequence of coordinate counts $r^{(j)} = \lfloor \log_q R^{(j)} \rfloor$ (his Sections 7.2 and 7.3). Only the top merge is approximate, by the hash ℓ, on the remaining $N' - r$ coordinates (his Section 7.4). On the

moduli of Table 2 this coordinate count is often zero at an inner level: $R^{(j)} < q$ leaves that level with no filter, and the list stays at size $S^{(j)}$.

The costs in Table 10 use a filter whose range is the representation count itself. Any hash of at most $\log R^{(j)}$ bits has that range, as does reduction modulo a divisor of q of the same bit length. May's argument in Section 5 still applies: the range is at most $R^{(j)}$, so some representation is expected to hit the target. Inner levels consume a nonincreasing number of those bits, which is his requirement that each level fix a subset of the bits fixed above it. A tree whose representation counts increase is rejected. The error is guessed only at the top, and only on whole coordinates. If $\log R^{(1)}$ is positive, the number of coordinates is the smallest r with $r \cdot \log q \geq \log R^{(1)}$,

$$r = \left\lceil \log R^{(1)} / \log q \right\rceil .$$

Bits of the last coordinate beyond $\log R^{(1)}$ are left unused, so the filter range stays equal to $R^{(1)}$. Each of the r coordinates is nevertheless drawn from the full guessing alphabet.

That alphabet is where the Gaussian error enters. May's Section 5.2 guesses a ternary error and charges $3^{r/2}$, the meet-in-the-middle over $r/2$ coordinates on each side of his Algorithm 3. Here each guessed coordinate lies in

$$\mathcal{E} = \{-8, \ldots, 8\}, \qquad |\mathcal{E}| = 17,$$

about ± 2.5 standard deviations. The guessing cost is $|\mathcal{E}|^{r/2}$, that is $T_g = (r/2)\log|\mathcal{E}|$ bits. Under the discrete Gaussian proportional to $\exp(-t^2/(2 \cdot 3.2^2))$ on the integers, a coordinate falls outside $\mathcal{E}$ with probability 0.77%. Over the $r \leq 7$ coordinates that occur on these rows, redrawing such a guess costs under a tenth of a bit and does not move a cost rounded to the nearest bit.

The list-merging recurrence is his Section 7.4, with $q^{r^{(j)} - r^{(j+1)}}$ replaced by the bit budget $R^{(j)}/R^{(j+1)}$. The leaves are an ordinary meet-in-the-middle of size $\sqrt{S^{(k)}}$, as in his depth-d tree. Match-and-filter (his Section 7.3) drops sums whose weight is wrong. The time of a level is therefore the larger of the stored list and the number of pairs that survive the filter; in bits, the level cost is $\max(\log L^{(j)},\ 2\log L^{(j+1)} - \log(R^{(j)}/R^{(j+1)}))$, which is how Algorithm 11 is evaluated.

Algorithm 11 Bit cost of one representation tree (May, Section 7.4, bit-granular filter)

Input: Dimension N', modulus q, alphabet $\mathcal{E}$, and counts $(S^{(j)}, R^{(j)})_{j=1}^{k}$ from Algorithm 10
Output: Bit cost $T = T_\ell + T_g$, or rejection

if $R^{(1)} \geq \cdots \geq R^{(k)} > 0$ does not hold **then return** rejection
$L^{(j)} \leftarrow S^{(j)}/R^{(j)}$ for $j = 1, \ldots, k$. Reject if any $L^{(j)} < 1$.
$L^{(k+1)} \leftarrow \sqrt{S^{(k)}}$ and $R^{(k+1)} \leftarrow 1$
$T^{(k+1)} \leftarrow L^{(k+1)}$
for $j = k$ down to 1 **do**
    $T^{(j)} \leftarrow \max\Big(L^{(j)},\ \big(L^{(j+1)}\big)^2 \cdot R^{(j+1)}/R^{(j)}\Big)$
$r \leftarrow \left\lceil \log R^{(1)} / \log q \right\rceil$
$T^{(0)} \leftarrow \max\Big(L^{(1)},\ \big(L^{(1)}\big)^2 / 2^{N'-r}\Big)$
$T_\ell \leftarrow \log\big(\max_{0 \leq j \leq k+1} T^{(j)}\big)$ and $T_g \leftarrow (r/2)\log|\mathcal{E}|$
return $T_\ell + T_g$

The same recurrence, fed with May's balanced ternary counts and with his coordinate filter $r = \lfloor \log_q R \rfloor$, is what reproduces his NTRU-HPS tables to within 1–5 bits.

On the first running example the tree has one level, $\log L^{(1)} = 122.61$, and the leaves have the same size because $\log R^{(1)}$ accounts for the whole gap between $S^{(1)}$ and the meet-in-the-middle. Here $r = \lceil 122.58/27 \rceil = 5$, so

$$T_g = \log 17^{5/2} = 10.22 \qquad \text{and} \qquad T = 122.61 + 10.22 = 132.83.$$

Table 10. Trees attaining the Meet-LWE cost $T_{\mathrm{ML}} = T_\ell + T_g$ with Gaussian error guessing, against Odlyzko's split T_{Odl} of Table 3. T_ℓ and T_g are in bits before rounding; T_{ML} and T_{Odl} are rounded to the nearest bit. The depth is May's tree depth. On every row the parameters below the first level are zero, and depth 4 is never chosen.

N	h	$\log q_0$	depth	ε	r	T_ℓ	T_g	T_{ML}	T_{Odl}
2^{11}	64	27	3	$(6, 0)$	7	152.81	14.31	167	160
2^{12}	64	51	3	$(8, 0)$	5	179.95	10.22	190	192
2^{13}	64	102	3	$(8, 0)$	3	211.85	6.13	218	224
2^{14}	64	209	3	$(8, 0)$	2	243.80	4.09	248	256
2^{13}	32	27	2	(4)	5	122.61	10.22	133	128
2^{14}	32	59	2	(4)	3	138.61	6.13	145	144

The approximate top merge costs $T^{(0)} = L^{(1)}$: a hash of length $N' - r$ eliminates the false pairs, and writing the list is the expensive step. Rounded, this is 133 bits, above $T_{\mathrm{Odl}} = 128$, so with error guessing Odlyzko's split is cheaper on this row; the guessing-free tree (Section D.6) brings it to 125 list elements.

The second running example is a depth-3 tree in May's numbering: two representation levels, with $(\varepsilon^{(1)}, \varepsilon^{(2)}) = (6, 0)$, and a meet-in-the-middle underneath. Algorithm 10 returns

$$\log S^{(1)} = 324.27, \quad \log R^{(1)} = 171.81, \quad \log L^{(1)} = 152.46,$$
$$\log S^{(2)} = 181.99, \quad \log R^{(2)} = 39.36, \quad \log L^{(2)} = 142.63,$$

and the leaves have size $\frac{1}{2} \log S^{(2)} = 91.00$ bits. The inner join has range $R^{(2)}$ and costs $\max(142.63,\ 181.99 - 39.36) = 142.63$. The join of the two level-2 lists has range $R^{(1)}/R^{(2)}$ and costs

$$\max\big(152.46,\ 2 \cdot 142.63 - (171.81 - 39.36)\big) = 152.81.$$

That join is the most expensive level, and with $r = \lceil 171.81/27 \rceil = 7$ one has $T_g = \log 17^{7/2} = 14.31$, so the tree costs 167.12 bits, which rounds to the 167 of Table 10, above Odlyzko's 160. May's own coordinate counts $r^{(1)} = \lfloor 171.81/27 \rfloor = 6$ and $r^{(2)} = \lfloor 39.36/27 \rfloor = 1$ give 167.3 bits after re-optimizing ε. The bit-granular filter matters when an inner level receives no whole coordinate: on the row $N = 2^{13}$, $h = 64$, $\log q = 102$, May's rule gives $r^{(2)} = \lfloor 43.13/102 \rfloor = 0$, and the cost rises from 218.0 to 230.7 bits, above Odlyzko's 224. The table reports the bit-granular filter.

List construction is independent of q, while a smaller modulus increases r and hence T_g; this is the correction of May's ternary guessing term for a Gaussian error. With error guessing, it is why Meet-LWE loses to Odlyzko's split (Algorithm 9) on the rows with $\log q \in \{27, 59\}$; the guessing-free lists (Section D.6) drop T_g and are the cheaper attack on those rows.

D.5 The search

A tree is a depth d in May's sense, that is $k = d - 1$ representation levels and a final meet-in-the-middle, together with a tuple $(\varepsilon^{(1)}, \ldots, \varepsilon^{(k)})$. His Section 7.5 found depth 5 no better than depth 4 on ternary secrets of much larger relative weight, so the search covers depths 2, 3, and 4, with $\varepsilon^{(1)} \in \{0, \ldots, 40\}$, $\varepsilon^{(2)} \le \min(\varepsilon^{(1)}, 24)$, and $\varepsilon^{(3)} \le \min(\varepsilon^{(2)}, 12)$. Tuples rejected by Algorithm 10 or Algorithm 11 are discarded, and the Meet-LWE cost of the instance is the minimum of T over the survivors.

The winning first-level counts are 4, 6, and 8, so the cap at 40 does not bind. With error guessing, the cost of May's original attack is the cheaper of T_{ML} and T_{Odl}; the guessing-free cost is the separate column T_{free} of Table 3.

D.6 The guessing-free tree

Kirshanova and May remove the error guess by locality-sensitive hashing [KM21]; the success-probability analysis of that attempt is flawed [HKLS22, Appendix A]. The column T_{free} of Table 3 instead replaces the exact match by a box, as in [HKLS22], with the cost model of [BLL+26, Section 3.3], on the representation counts of Algorithm 10 at depth 2. On the first running example the tree uses $\varepsilon = 3$, seven coordinates in boxes of half-width 3072 and one of half-width 4096. The top list has $2^{125.1}$ elements and is the bottleneck; the leaves have size 2^{113}, with $2^{11.8}$ hash repetitions. The column T_{ops} adds $\log(\log L)$ bits for hashing each element of the largest list L on about $\log L$ coordinates. With the ten hash repetitions per expected hit that Bi et al. charge [BLL+26], where May charges one, the row $N = 2^{13}$, $h = 32$ costs 128 list elements and 135 word operations instead of 125 and 132. Its top list stores 2^{125} elements and Odlyzko's 2^{128}, far above the memory of a lattice attack at the 128-bit line, so a memory-aware cost would put both well above 128.

D.7 Digits ±2

May's Rep-2 (his Section 8) enlarges the digit set by ±2. Appendix A.1 of his paper introduces three parameters, collected in his Table 11. The parameter ε_{10} counts extra pairs $(1, -1)$ on zeros, which is ε from Rep-1. The parameter ε_{21} counts pairs that realize a parent ± 1 by $(2, -1)$ or $(-1, 2)$. The parameter ε_{20} counts pairs $(2, -2)$ on zeros. His formulas square a multinomial because the parent secret has both signs. The block-binary parent has only the sign $+1$, so the square becomes one multinomial on the h' ones. With $a = \lceil h'/2 \rceil$ and $b = \lfloor h'/2 \rfloor$, the ones are partitioned into ε_{21} coordinates written $(2, -1)$, ε_{21} coordinates written $(-1, 2)$, $a - \varepsilon_{21}$ coordinates written $(1, 0)$, and $b - \varepsilon_{21}$ coordinates written $(0, 1)$. The zeros are partitioned into ε_{20} coordinates of each of $(2, -2)$ and $(-2, 2)$, and ε_{10} coordinates of each of $(1, -1)$ and $(-1, 1)$. The first summand then has

$$\begin{aligned} n_{2+} &= \varepsilon_{21} + \varepsilon_{20}, & n_{2-} &= \varepsilon_{20}, \\ n_{1+} &= a - \varepsilon_{21} + \varepsilon_{10}, & n_{1-} &= \varepsilon_{21} + \varepsilon_{10} \end{aligned}$$

entries of $+2$, -2, $+1$, and -1 respectively, and

$$\begin{aligned} S^{(1)} &= \binom{N'}{n_{2+},\, n_{2-},\, n_{1+},\, n_{1-}}, \\ R^{(1)} &= \binom{h'}{\varepsilon_{21},\, \varepsilon_{21},\, a - \varepsilon_{21},\, b - \varepsilon_{21}} \binom{N' - h'}{\varepsilon_{20},\, \varepsilon_{20},\, \varepsilon_{10},\, \varepsilon_{10}}. \end{aligned}$$

These are the level-1 formulas of his Appendix A.1 for a one-signed parent. The tree is then priced by Algorithm 11.

Deeper levels in his Appendix A.2 keep the digits ±2. The search here folds each $+2$ into a $+1$ and each -2 into a -1, and continues with Algorithm 10. The folded tail is at least as large as a tail that keeps ±2. The first level is searched for $\varepsilon_{10} \in \{0, \ldots, 27\}$, $\varepsilon_{21} \in \{0, \ldots, 7\}$, and $\varepsilon_{20} \in \{0, \ldots, 5\}$. The tail is either empty, a single further parameter in $\{0, \ldots, 15\}$, or a pair $(\varepsilon', \varepsilon'')$ with $\varepsilon' \le 11$ and $\varepsilon'' \le \min(\varepsilon', 8)$, under the same monotonicity constraint as Algorithm 11. On the six rows of Table 10 this search stays within 0.08 bits of Representation 1. The rounded column T_{ML} of Table 10 does not change. On the two rows with $h = 32$ the minimum uses $\varepsilon_{21} = \varepsilon_{20} = 0$, so the digits ±2 are absent and the two trees coincide.

Table 11. May's attack on SHIP's windowed secret ($h = 31$, dimension N). One window is fixed in our estimate; SHIP fixes none. $\log q_0$ is the base prime of the set. SHIP's lattice check used $P \cdot q_0$, not this prime. $w = \max(175, \lfloor N/(2h) \rfloor)$ is the published half-width, and $w^\star$ is the smallest integer at least $N/(2h)$ whose cost is still at least 128 bits. Green columns are Odlyzko's split, which is cheaper than Meet-LWE on every row. Bold entries are that cost. Costs are rounded to the nearest bit.

Set	N	$\log q_0$	w	May	$w^\star$	May*
LL13	2^{13}	25	175	**142**	133	**136**
LL14	2^{14}	48	264	**151**	265	**151**
HT14	2^{14}	28	264	**151**	265	**151**
HT15	2^{15}	50	528	**166**	529	**166**

D.8 May's attack on the windowed secret of SHIP

Table 11 applies the same attacks to SHIP's windowed secret, with Hamming weight $h = 31$ and dimension N. SHIP fixes no coefficient. Our estimate fixes one window, which only removes search space, so the costs are conservative for SHIP's secret. The modulus is the base prime q_0 of the BV switch: 25, 48, 28, and 50 bits for `LL13`, `LL14`, `HT14`, and `HT15` [CHKS25]. Odlyzko splits the remaining windows, each with $4w$ signed positions. On every set the best tree uses no extra cancellation, so the bold cost is the window split. That cost is at least 142 bits at the published half-width $w = \max\big(175, \lfloor N/(2h) \rfloor\big)$, and it stays at least 128 bits down to the smallest integer half-width at least $N/(2h)$. For `LL14` and `HT14`, $\lfloor N/(2h) \rfloor = 264$ sits just below $N/(2h)$, so that minimum is 265. On every set the smaller window leaves the number $\lceil \log_4 \lceil 2w/\theta \rceil \rceil = 3$ of multiplexer digits unchanged (Table 9), and with it the sequential key switches of a blind rotation, so our implementation of SHIP keeps the published windows.

D.9 What is left out

Three parts of May's paper are outside the estimate. His Section 9 combines the same representations with the collision search of van Oorschot and Wiener, which removes the lists at the price of a higher time exponent; the estimate uses the list version of his Section 7.4, whose memory is the largest list, T_ℓ on the rows above. His Section 10 discusses a hybrid of the representation tree with lattice reduction: the tree keeps the vectors that satisfy the LWE equation on the filtered coordinates, and the hybrid projection discards those coordinates. He leaves open whether the combination improves on either attack alone; for unstructured secrets the primal and dual hybrids of Section 3.2 have since answered it [HKLS22,BLLW22,BLL$^+$26]. Finally, the estimate omits the ring structure he declines to use (his Section 2.1).

The lattice estimate of Section 3.1 leaves out the blocks in turn. A block with a single 1 takes B values, $\log B$ bits, whereas B free coordinates of density $1/B$ carry $B \cdot H(1/B) \approx \log(e \cdot B)$ bits, with H the binary entropy. Guessing whole blocks in the hybrid's meet-in-the-middle stage would thus save about $\log e \approx 1.44$ bits of entropy per guessed block; we have not computed its effect on the cost of the attack.

The polynomial factors omitted throughout are those he omits: sorting and hashing the lists, the duplicate labels of ℓ near the two boundaries, the match-and-filter arithmetic, and the number of lists in a tree of depth at most 4.