Slicing Bits and Cutting Costs in CDT Sampling

High-Order Masking of FrodoKEM's Gaussian Sampler, Revisited

Calvin Abou Haidar[1], Thomas Espitau[2], Clément Hoffmann[1] and Mehdi Tibouchi[1]

[1] NTT Social Informatics Laboratories, Tokyo, Japan
{calvin.haidar,clement.hoffmann,mehdi.tibouchi}@ntt.com
[2] PQShield
thomas.espitau@pqshield.com

Abstract. FrodoKEM, a key encapsulation mechanism based on the standard (unstructured) LWE assumption, is recommended as a conservative choice for post-quantum key exchange by agencies like BSI and ANSSI. As such, it has garnered substantial attention from an implementation security standpoint. In particular, several papers have looked into masking FrodoKEM, and, like for various other lattice-based cryptosystems, identified the Gaussian sampling operation as a major bottleneck. In FrodoKEM, it is carried out using a CDT-based sampler, which compares a uniformly random value in $[0, 1)$ (with fixed precision) to each of the probabilities in the cumulative distribution table of the target distribution.

Two recent works proposed masked gadgets at arbitrary order for that operation: one by Gérard and Guerreau (CASCADE 2026), which simply masks each of the comparisons with the table elements (using a ripple carry adder as the Boolean comparison circuit), and another by Eid et al. (TCHES 2026), that uses a novel approach based on a binary search tree, significantly reducing the number of masked comparisons for larger tables. Eid et al. also choose the Kogge–Stone adder as the basis for their comparison circuit.

In this paper, we provide an extensive analysis of the choices and trade-offs involved in masked CDT-based sampling, and propose several optimizations that further improve upon these previous works by a considerable margin. In particular, since FrodoKEM, like almost all the schemes that rely Gaussian sampling, generates hundreds of samples at a time, we find it beneficial to utilize approaches that lend themselves to a high degree of parallelization. This includes *bitslicing* (with which, importantly, the ripple carry adder has been proved optimal for comparisons), as well as techniques such as the use of algebraic normal form in masked multiplexers.

As a result, on an ARM Cortex-M4 STM32F4 target board, our companion implementation in pure C outperforms the hand-crafted assembly of Eid et al. (resp. Gérard–Guerreau) by a factor of around 2 (resp. 7 to 9) at masking order 1. The gap is even larger at higher order: we outperform the higher-order code of Eid et al. (in C with assembly for expensive gadgets) by a factor ranging from 5 at masking order 2 to 9 as masking order 8.

Keywords: Gaussian Sampling · Bitslicing · Masking · Lattice-based Cryptography

1 Introduction

Implementation security of lattice-based cryptography. As post-quantum cryptography gradually moves from standardization to deployment, implementation security has become a central concern. This is particularly true for lattice-based cryptography, whose efficiency

and versatility have made it one of the main paradigms for post-quantum encryption, key exchange, and signatures, but whose concrete realizations also introduce new implementation challenges. In addition to the familiar issues of leakage in, e.g., modular arithmetic, lattice-based schemes often rely on procedures that have no real counterpart in classical public-key cryptography, and whose secure implementation remains substantially less mature.

A notable example is *Gaussian sampling*. Discrete and rounded Gaussians arise throughout lattice-based cryptography: in trapdoor-based constructions, they are a fundamental algorithmic primitive, and in many concrete schemes they are used either directly or through closely related derived distributions. Unfortunately, Gaussian sampling is also well known to be a fragile operation from an implementation-security standpoint. A large body of work has shown that it can leak through timing, cache, power, or electro-magnetic side channels [BHLY16, PBY17, EFGT17, FKT+20, KA21, MWG+22, GMRR22, WGS22, MKK+23, ZLYW23, Pre23, GR24, LZY+25, SD25] as well as fault attacks [EFGT16, BBK16, BD23, BD25, HPT25], sometimes with devastating consequences for the security of the overall cryptosystem. This issue is not merely of academic interest: the complexity of implementing Gaussian samplers securely is one of the reasons why some practical lattice schemes like Dilithium [LDK+22] have preferred simpler noise distributions whenever possible.

For timing leakage, the situation is now comparatively well understood. Over the past several years, a substantial line of work has developed constant-time samplers and clarified how to avoid the obvious pitfalls of table accesses, data-dependent control flow, and variable-time arithmetic [MW17, BBE+19, ZSS20b, ZSS20a, HPRR20, DFW20]. Protecting against stronger side-channel attacks such as power analysis is, however, a significantly harder problem. In that setting, masking is the dominant generic countermeasure, but its cost is often driven by the number of non-linear operations needed to manipulate secret-shared values. This makes Gaussian sampling especially challenging: even when the target distribution itself is fixed and public, the standard constant-time sampling techniques translate rather poorly to masked form.

FrodoKEM and CDT-based Gaussian sampling. The present work focuses on the discrete Gaussian sampler used in FrodoKEM, a key encapsulation mechanism based on standard (unstructured) LWE. Because it avoids idealized algebraic structure, FrodoKEM is regarded as a conservative design point among post-quantum encryption schemes, and is often been highlighted as a natural conservative alternative to structured lattice schemes in contexts where, e.g., reducing key size is not the driving criterion and a stronger emphasis is placed on long-term, cautious security choices. As such, FrodoKEM is in the portfolio of post-quantum schemes recommended by agencies like BSI [BSI26], ANSSI [ANS26] and ENISA [EUC25].

In FrodoKEM, noise generation is performed through a *cumulative distribution table*-based (CDT-based) sampler. At a high level, this amounts to discrete inverse transform sampling: one stores a CDT for the target half-Gaussian, draws a uniform value with fixed precision, and outputs the unique interval in which that value falls. For the FrodoKEM parameters, the uniform input is represented on $\ell = 15$ bits, and the resulting half-Gaussian sample is then turned into a signed sample by applying a random sign bit. This process is simple and efficient, although it typically requires carrying out comparisons with each and every entry of the CDT to avoid secret-dependent memory accesses and achieve a constant-time implementation.

In order to further protect against multi-trace side-channel attacks, one would want to apply a masking countermeasure to this CDT sampling. Note, by the way, that even if, hypothetically, an alternate, more efficient approach to sampling the FrodoKEM Gaussian distribution was available, a FrodoKEM implementation *has* to use CDT sampling,

because the Fujisaki–Okamoto [FO13] transform it uses requires the same random coins to result in the same error vector. In a masked implementation, however, the repeated secure comparisons between the secret uniform input and the public CDT thresholds become costly. This comparison-heavy structure makes Gaussian sampling one of the main bottlenecks of masked FrodoKEM implementations.

Prior work on masked CDT sampling. Two recent works have substantially advanced the state of the art for masked FrodoKEM sampling. Gérard and Rossi [GR19] proposed an arbitrary-order masked implementation of the CDT sampler that follows the most direct approach: each comparison with a table element is protected in masked form, using a Boolean comparison circuit based on secure addition. This method was applied to the Gaussian sampler of FrodoKEM by Gérard and Guerreau [GG26] . Eid et al. [EGRZ26] then introduced a more sophisticated tree-based approach. Instead of comparing the secret input against all CDT entries, they organize the search as a binary tree and reduce the number of masked comparisons from linear in the table size to logarithmic. For larger tables, this is a substantial algorithmic improvement. Their implementation also relies on a different comparison strategy, using the Kogge–Stone adder as the core building block, in the style of [CGTV15], which the authors regard as another notable speed-up.

Nevertheless, important design choices in masked CDT sampling remain insufficiently understood. In particular, the best strategy depends not only on asymptotic comparison count, but also on how well the underlying circuits map to masked software implementations on word-oriented processors. This is where bitslicing becomes relevant. Because lattice schemes typically generate many samples in parallel, they offer exactly the kind of data-level parallelism that bitslicing can exploit. Concurrently to our work, [KLS+25] explored this direction by proposing a fully bitsliced implementation of the linear-search CDT approach. Their sampler relies on a different comparison strategy introduced in [KLRBG23], which interestingly incurs the exact same cost in non-linear operations as our underlying comparison gadget. As the current draft stresses, a simple ripple carry structure, when evaluated in bitsliced form across many independent instances, can outperform more elaborate parallel-prefix constructions in amortized masked cost.

Our contributions. In this work, we revisit high-order masked CDT sampling for FrodoKEM and provide a systematic analysis of the implementation trade-offs involved. Our starting point is the observation that, in the use case relevant to lattice-based cryptography, throughput matters at least as much as local circuit depth: practical implementations generate hundreds of Gaussian samples at a time, making the ability to exploit word-level parallelism crucial. This leads us to re-examine the basic design space of secure comparison and masked tree traversal in a way that takes into account possible improvements from *bitslicing*, following previous work by, e.g., Bronchain and Cassiers [BC22], who established that, assuming full parallelism, the ripple carry adder provides a significant performance improvement for masked comparisons at relevant bit-lengths.

Our first contribution is a detailed analysis of the secure-comparison strategies underlying masked CDT sampling. We compare ripple carry and parallel-prefix approaches in terms of non-linear complexity, depth, and software parallelization, and show that the appropriate choice is somewhat context-dependent. In particular, for fully bitsliced implementations of the kind most relevant to FrodoKEM, we find that, consistent with the observations of [BC22, ABC+23], the ripple carry approach is superior: although it has larger depth, it minimizes the number of masked non-linear operations and amortizes especially well over many parallel comparisons.

Our second contribution is to show that the tree-based approach of Eid et al. can itself be significantly improved when reformulated in a bitslicing-friendly way. The key issue is not only the number of comparisons, but also the cost of computing the successive

pivot values used to navigate the tree. We suggest a pivot-evaluation method based on an algebraic normal form (ANF) representation of those pivots in terms of the previous comparison bits. This reformulation makes it possible to separate the genuinely non-linear part of the computation from the multiplication by public word-sized constants, enabling extensive memoization and substantially reducing the cost of pivot generation in bitsliced form. The resulting savings are particularly pronounced because the relevant secret monomials are only one bit wide, whereas the CDT values themselves occupy full machine words.

Our third contribution is a collection of further optimizations for the complete masked sampler, including a low-depth treatment of the final sign application and an implementation strategy that is compatible with PINI composition throughout. Altogether, these techniques yield a masked Gaussian sampler that is both conceptually simpler and significantly more efficient than prior work.

Finally, we provide a full implementation and benchmarking study on an ARM Cortex-M4 microcontroller. The resulting code is, to the best of our knowledge, the fastest reported implementation of high-order masked CDT-based Gaussian sampling for FrodoKEM. Our pure-C implementation outperforms the first-order assembly implementation of Eid et al. by roughly a factor of 2, and the Gérard–Guerreau implementation by factors ranging from 7 to 9 depending on the parameter set. The advantage is even larger at higher masking orders, where our implementation improves over Eid et al.'s higher-order implementation (also in C but with assembly optimizations for the costlier individual gadgets) by factors ranging from about 5 at order 2 to about 9 at order 8. These gains are accompanied by a similarly substantial reduction in randomness consumption.

Organization of the paper. The remainder of the paper is organized as follows. In Section 2, we first recall the necessary background on masking, bitslicing, and the tree-based CDT-sampling method of Eid et al. In Section 3, we then analyze the design space of secure comparison circuits for masked software implementations, with particular attention to the interaction between circuit structure and bitsliced parallelism. Next, in Section 4, we present our optimized masked Gaussian sampler, including the ANF-based pivot computation and the remaining implementation refinements. Finally, in Section 5, we report benchmark results and compare our implementation against the current state of the art.

2 Background & State-of-the-art

2.1 Masking and probing security

2.1.1 Masking

Masking is a widely adopted countermeasure designed to protect sensitive data against side-channel analysis. It operates by splitting a secret variable x into d separate components x_i, referred to as *shares*, such that $\bigoplus_{i=0}^{d-1} x_i = x$. Consequently, any subset of up to $d-1$ shares is **statistically independent of** the secret x. Maintaining this property throughout the execution of a circuit for all sensitive intermediate values is mandatory for security. This requirement is formalized by the t-probing model [ISW03], which provides a security guarantee that an adversary observing up to $t = d-1$ intermediate variables learns no information about the secret. In this work, the variable d always denotes the number of shares.

In a masked implementation, standard functions are replaced by masked *gadgets* that process shared inputs and produce shared outputs. Linear operations, such as `NOT` and `XOR`, can be performed share-wise, resulting in a computational overhead that is linear in the number of shares d. Conversely, non-linear operations—such as an `AND` gate between

two shared values—typically incur a quadratic overhead **in** d and require the generation of fresh randomness. In practice, the cost of these non-linear operations dominates the total execution time, even at relatively low masking orders. Throughout this paper, we denote by SecOp the gadget equivalent of the operator Op (eg. SecAnd, SecXor).

In this work, we focus solely on Boolean masking. In practice, Gaussian sampling is mostly used for the implementation of lattice-based primitives, whose subsequent operations often require arithmetic masking[1]. To the best of our knowledge, the most efficient Boolean to Arithmetic (B2A) conversion gadgets rely on bitslicing [BC22] and are therefore highly compatible with our approach.

2.1.2 PINI vs SNI

To prove the security and composability of masking gadgets in the t-probing model, different composition frameworks co-exist. The most prominent are Non-Interference (NI) alongside Strong Non-Interference (SNI) [BBD+16], and the more recent Probe Isolating Non-Interference (PINI) [CS20].

The core feature of PINI is its stability under composition: a circuit composed of PINI gadgets is trivially PINI. Crucially, linear operations are inherently PINI, which guarantees that they can be executed without any randomness refresh. As detailed below, this composability comes at the cost of a slightly more expensive elementary gadget (eg. the multiplication), but this overhead is offset by simpler security proofs and the ability to deploy advanced implementation techniques like bitslicing globally.

Combining SNI proofs with bitslicing is notoriously challenging. The SNI notion relies on strict isolation of shares, typically enforced by inserting refresh gadgets to break dependencies. In a bitsliced implementation, where data is packed and processed in parallel, enforcing this isolation requires frequent refreshes—potentially at every non-linear gate—which destroys the performance gains of vectorization. Conversely, PINI is designed to be structurally compatible with such constraints. Since PINI gadgets compose trivially, they eliminate the need for inter-gadget refreshes. While the cost of a PINI SecAnd is roughly twice that of an SNI SecAnd (primarily due to increased randomness requirements), this local overhead is an upper bound. In a complete circuit, the removal of refresh layers outside non-linear operations results in a significant net performance gain.

2.2 Eid et al.'s tree-based CDT sampling

This subsection presents the main ideas of [EGRZ26].

In a non-masked implementation, Gaussian sampling is often performed through *inverse transform sampling* using a Cumulative Distribution Table (CDT). Concretely, this process consists of storing the Cumulative Distribution Function (CDF) of the target distribution in a table, sampling a uniform value, and comparing this uniform value with the stored table[2]. Implementing this technique in a masked way is relatively straightforward. In a masked settings, you have to compare the (masked) uniform value to every entry of the table. Then, your output is the number of successful comparisons. The required precision of this uniform is often computed using Rényi divergence arguments; in the case of FrodoKEM, this value is $\ell = 15$. Most implementations sample a half-Gaussian and then negate the result with probability $1/2$.

[1] Sharing a value x in shares x_i such that $x = \sum x_i \pmod q$, for a given q. This representation makes modular addition and multiplication with a constant linear, and greatly simplifies modular multiplication between secret values.

[2] If $U \sim \mathrm{Unif}([0,1])$, then $F^{-1}(U)$ has CDF F.

2.2.1 Tree optimization

When implementing the "naive" CDT evaluation in a masked way, the most expensive operation is the comparison.

The core idea of [EGRZ26] is to replace the linear scan of the table by a binary tree search. At each level of the tree, the pivot value—to which the uniform sample needs to be compared—is computed securely at a relatively low cost. This approach avoids comparing the sample against each of the k elements of the table, effectively reducing the number of comparisons required for the inverse sampling from $k-1$ to $\lceil \log_2(k-1) \rceil$.

To build the tree, the CDT must first be padded with unreachable values so that its size becomes one less than a power of 2. Concretely, this is done by filling the end of the table with the maximum probability value (2^p in practice, as the uniform sample is a ℓ-bit integer rather than a floating-point number in $[0,1]$). Each level of the tree then consists of one comparison that determines one bit of the output. To select the pivot value for the next comparison, the results of previous comparisons are used as control bits to secretly determine the path in the tree.

The first node value is the central element of the look-up table, and the result of the comparison b_0 determines the Most Significant Bit (MSB) of the output. Subsequently, the second pivot value is computed as $a \coloneqq c_0 \cdot b_0 \oplus c_1$, where c_0 and c_1 are constants chosen such that a equals the correct probability depending on the branch implicitly chosen by b_0. More generally, at the i-th level, the pivot value is computed via a selection function involving monomials of the previous comparison results $(b_j)_{j<i}$ and the CDT values at that level. This construction of the pivot value requires the computation of a ℓ-bit AND (eg. [EGRZ26, Algorithm 7, Line 7]). The final output is then a l-bit integer, where l is the tree's number of levels.

Figure 1 depicts that approach in the case of FrodoKEM-1344. The trees corresponding to the other FrodoKEM parameter sets can be found in [EGRZ26, Fig. 8]. The tree look-up outputs a half-Gaussian x on l bits, which is then converted to a signed Gaussian using two's complement arithmetic conditioned by a secret sign bit s. This is computed as:

$$((s,\ldots,s) \oplus x) + s = \begin{cases} x & \text{if } s = 0 \\ \neg x + 1 \equiv -x & \text{if } s = 1 \end{cases}$$

where $(s,\ldots,s)$ denotes the repetition of the bit s to match the word size of x. The final output is a $l+1$-bit integer.

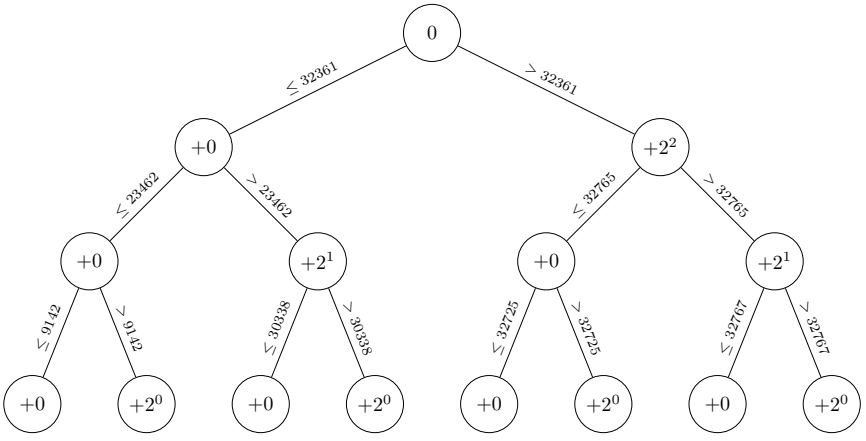

Figure 1: Tree structure for CDT sampling of FrodoKEM-1344's half-Gaussian.

2.3 Bitslicing

Bitslicing is a powerful software implementation technique that can be viewed as an extreme form of SWAR (SIMD Within A Register). Algorithms entirely composed as sequences of bitwise operations are prime candidates for bitslicing. By transposing the memory representation of the variables, bit-level operations can be executed in parallel across multiple independent data instances, fully leveraging the natural parallelism of a w-bit processor architecture (typically $w = 32$ or $w = 64$).

This approach requires a high level of data-level parallelism, ie. multiple independent data elements that need to be processed through the exact same sequence of instructions. This is rarely a limitation in the implementation of lattice-based cryptography, which inherently processes polynomials of large degrees with identical coefficient-level operations. The primary limitation of bitslicing is the computational overhead of the memory transposition itself (packing and unpacking the state), which can be achieved with a complexity of $\mathcal{O}(w \log w)$ using the algorithm detailed in [War13][3].

For a set of w independent k-bit values $x_0, \ldots, x_{w-1}$, where $x_i[j]$ represents the j-th bit of x_i, the canonical memory representation of these values (assuming $w \geq k$) is the following:

$$\begin{pmatrix} x_0[0] & \cdots & x_0[k-1] & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ x_{w-1}[0] & \cdots & x_{w-1}[k-1] & 0 & \cdots & 0 \end{pmatrix}$$

We obtain the bitsliced representation by transposing this matrix and removing the padded zero columns:

$$\begin{pmatrix} x_0[0] & \cdots & x_{w-1}[0] \\ \vdots & & \vdots \\ x_0[k-1] & \cdots & x_{w-1}[k-1] \end{pmatrix}$$

Assuming each x_i requires the same sequence of computations, we can now perform bitwise logical gates (eg. AND, XOR) directly on the rows of this bitsliced representation. Each instruction simultaneously processes the j-th bit of w different elements, maximizing the ALU utilization.

To provide a concrete estimation of the performance gains offered by bitslicing, let us compare a bitsliced naive Ripple Carry Adder (RCA) with a non-bitsliced Kogge-Stone adder for integers of size ℓ. The Kogge-Stone adder computes carry values in parallel using a tree network of *generate* and *propagate* signals (more details on adders can be found in Section 3).

This tree structure minimizes the logical depth to $\lceil \log_2 \ell \rceil$. However, each step of the non-bitsliced Kogge-Stone adder requires a full w-bit vector operation (which translates to one non-linear SecAnd gate in a masked implementation) that only computes a single ℓ-bit addition. The total unamortized cost of the adder is therefore $\approx \log_2 \ell$ calls to a w-bit SecAnd.

On the other hand, the naive Ripple Carry Adder computes each carry sequentially. While the multiplicative depth of the circuit is $\mathcal{O}(k)$ (resulting in higher latency for the batch), each step strictly requires only a 1-bit SecAnd. When bitsliced, these k sequential SecAnd operations compute w additions simultaneously. The amortized cost per addition is therefore k/w SecAnd operations.

For $\ell = 15$ (as in the FrodoKEM Gaussian sampler), the Kogge-Stone adder has an amortized cost of roughly 4 SecAnd per addition. In stark contrast, the bitsliced Ripple

[3]The naive transposition algorithm runs with a complexity of $\mathcal{O}(w^2)$. Leveraging this $\mathcal{O}(w \log w)$ algorithm is standard practice in efficient implementations, as seen in [BC22, SS16].

Carry Adder achieves a cost of $15/32 \approx 1/2$ SecAnd per addition. This highlights how bitslicing provides massive throughput improvements for algorithms that tolerate higher local latencies.

Cost metric. The non-linear building block of all masked gadgets is the SecAnd operation. In implementations proven secure under the PINI composability framework, randomness consumption is almost entirely concentrated in these gates. Consequently, even at low masking orders, the overall performance is dominated by the number of SecAnd evaluations. Bitsliced implementations allow us to fully exploit the natural word-level parallelism of 32-bit microcontroller architectures. Throughout this work, we therefore evaluate computational complexity using an *amortized* SecAnd metric: one SecAnd_{32} operating on 32-bit words and thirty-two independent SecAnd_1 operations both count as a single amortized SecAnd. When relevant, we additionally report the multiplicative depth of the resulting circuits.

3 Comparison of strategies from different adders

Secure comparison between two unsigned n-bit integers $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$ can be reduced to subtraction using two's complement arithmetic:

$$a - b \equiv a + \neg b + 1 \pmod{2^n}.$$

Let $(c_i)_{0 \le i \le n}$ denote the carries generated during the addition $a + \neg b + c_0$. Setting $c_0 = 1$ yields the true difference $a - b + 2^n$, so the final carry encodes a non-strict comparison:

$$c_n = 1 \iff a \ge b.$$

Setting $c_0 = 0$ instead computes $a - b - 1 + 2^n$, so the final carry encodes a strict comparison:

$$c_n = 1 \iff a > b.$$

Hence, secure comparison reduces to computing the carry propagation associated with this subtraction.

In masked implementations, execution cost is dominated by secure non-linear operations (SecAnd). When both operands are masked, each non-linear gate requires a call to a secure AND gadget; when one operand is public, the corresponding gate is linear and incurs no masking cost. Comparison strategies therefore differ mainly according to

- the total number of secure non-linear operations,
- the multiplicative depth of the circuit,
- and the degree of achievable software parallelization.

3.1 Ripple carry comparison

Ripple carry propagates carries sequentially:

$$c_{i+1} = \mathrm{maj}(a_i, \neg b_i, c_i).$$

As demonstrated in [BC22], a carry bit can be securely computed with exactly one non-linear operation, resulting in n dependent steps. The total non-linear complexity is $n\mathsf{SecAnd}_1$, with multiplicative depth n.

On a w-bit processor, bitslicing evaluates w independent comparisons simultaneously by packing one bit from each instance into a single machine word. This yields an amortized cost of

$$\frac{n}{w}\,\mathsf{SecAnd}_w.$$

Ripple carry thus minimizes the number of non-linear operations but exhibits maximal latency, making it best suited for throughput-oriented settings where depth is not a concern.

3.2 Parallel-prefix comparison framework

To reduce the $O(n)$ latency of ripple carry, we turn to parallel-prefix networks that exploit the associativity of carry propagation. Prefix adders decompose each bit position into a *generate* signal g_i and a *propagate* signal p_i:

$$g_i = a_i \wedge \neg b_i, \qquad p_i = a_i \oplus \neg b_i.$$

By definition, these two signals are mutually exclusive (i.e., $g_i \wedge p_i = 0$). The propagate signal is linear. Computing g_i requires one SecAnd when both operands are masked; when one operand is public, g_i is also linear and the generation step is free. This distinction accounts for the difference between the *masked–masked* and *masked–public* cost models used throughout.

Carry propagation is then performed through the associative prefix operator

$$(g, p) \circ (\widehat{g}, \widehat{p}) = \big(g \vee (p \wedge \widehat{g}),\ p \wedge \widehat{p}\big).$$

Its masked evaluation requires three SecAnd operations, which can be seen by expanding the OR into XOR form:

$$\begin{aligned} g' &= g \oplus (p \wedge \widehat{g}), && (1\ \mathsf{SecAnd}),\\ p' &= p \wedge \widehat{p}, && (1\ \mathsf{SecAnd}). \end{aligned}$$

The total cost of any parallel-prefix comparison therefore comprises

1. a *generation layer*: n independent SecAnd (masked–masked) or zero (masked–public),
2. a *prefix network*: $3N$ SecAnd, where N is the number of prefix nodes in the chosen topology.

The multiplicative depth is $1 + D$ in the masked–masked case and D in the masked–public case, where D is the depth of the prefix network.

3.2.1 Kogge–Stone comparison

The Kogge–Stone [KS73] topology propagates carries across spans 2^i for $i = 0, \ldots, \log_2(n) - 1$. At level i, all positions $k \geq 2^i$ perform one prefix evaluation, giving $n - 2^i$ operations at that level. The total number of prefix nodes is

$$N_{\mathrm{KS}} = \sum_{i=0}^{\log_2(n)-1} (n - 2^i) = n \log_2(n) - (n - 1).$$

The prefix network depth is $D = \log_2(n)$. Including signal generation, the total non-linear complexity is

$$\begin{cases} n + 2\big(n \log_2(n) - (n-1)\big) & \text{masked–masked},\\ 2\big(n \log_2(n) - (n-1)\big) & \text{masked–public}, \end{cases}$$

with multiplicative depth

$$\begin{cases} 1 + \log_2(n) & \text{masked–masked,} \\ \log_2(n) & \text{masked–public.} \end{cases}$$

Because almost all bit positions remain active at every level, the computation densely occupies processor registers, limiting opportunities for interleaving independent comparisons via SWAR packing.

3.2.2 Brent–Kung comparison

The Brent–Kung [BK82] topology evaluates prefixes using a balanced reduction tree. A full Brent–Kung network contains $2n - 2 - \log_2(n)$ prefix nodes, arranged into a forward reduction tree ($n - 1$ nodes) and an inverse broadcast tree ($n - 1 - \log_2(n)$ nodes). Since secure comparison only requires the final carry c_n, the inverse tree is unnecessary. The

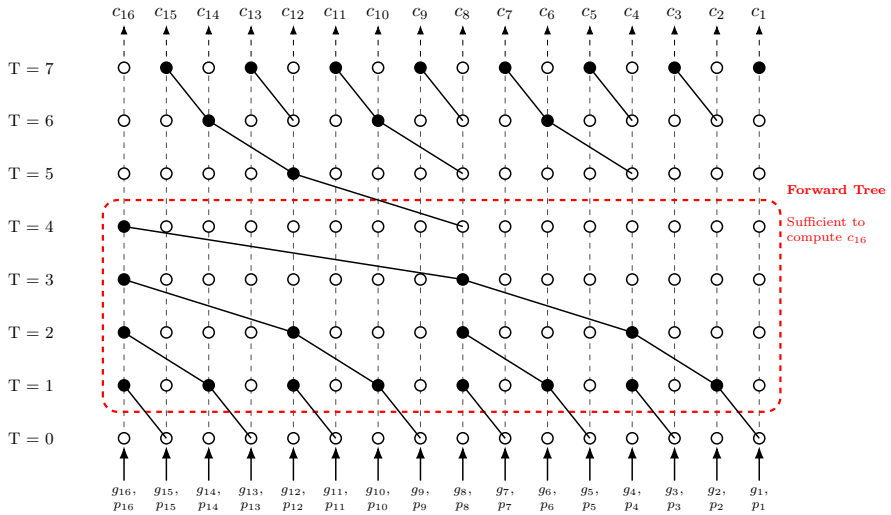

Figure 2: Forward Brent–Kung reduction tree. The highlighted region corresponds to the minimal computation required to obtain the final carry; the inverse tree (not shown) is unnecessary for comparison.

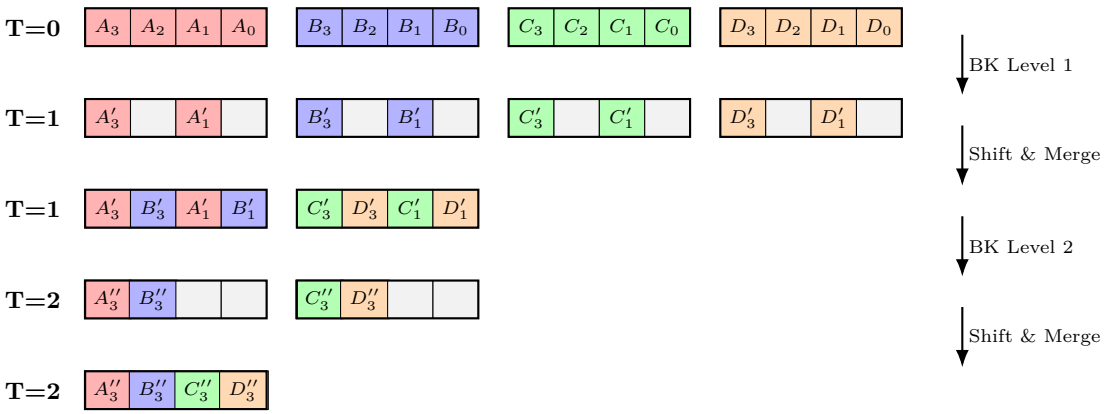

Figure 3: SWAR packing exploiting Brent–Kung sparsity.

Table 1: Comparison of secure comparison strategies (masked–masked case).

Method	Total SecAnd_1	Amortized SecAnd	Depth	Parallelization
Ripple Carry	n	$\approx n/w$	n	High (bitsliced)
Kogge–Stone	$n + 2(n\log_2 n - n + 1)$	$\approx 2n\log_2 n \,/\, w$	$1 + \log_2 n$	Low
Brent–Kung	$3n - 3$	$\approx 3n/w$	$1 + \log_2 n$	High (SWAR)

forward reduction tree alone computes c_n using exactly $n - 1$ prefix evaluations, as illustrated in Figure 2.

The propagation cost therefore equals $2(n-1)$ SecAnd_1. Including signal generation yields total complexity

$$\begin{cases} 3n-3 & \mathsf{SecAnd}_1 \text{ masked–masked,} \\ 2(n-1) & \mathsf{SecAnd}_1 \text{ masked–public,} \end{cases}$$

with multiplicative depth

$$\begin{cases} 1+\log_2(n) & \mathsf{SecAnd}_1 \text{ masked–masked,} \\ \log_2(n) & \mathsf{SecAnd}_1 \text{ masked–public.} \end{cases}$$

Unlike Kogge–Stone, the number of active nodes halves at each level $(n/2, n/4, \ldots, 1)$, producing a sparse computation pattern. This sparsity enables efficient software parallelization through SWAR packing: at each level, active signals from multiple independent comparisons (or from distinct bit positions) can be packed side by side within a single machine word, as illustrated in Figure 3.

3.3 Comparison of strategies

Secure comparison exhibits a fundamental trade-off between non-linear complexity and circuit depth. Table 1 summarizes the design space on a w-bit architecture.

To make the trade-off concrete, consider $n = 32$ on a $w = 32$-bit platform in the masked–masked setting. Ripple carry requires 32 SecAnd at depth 32. Brent–Kung requires 125 SecAnd at depth 6, a $4\times$ increase in operation count for a $5\times$ reduction in depth. Kogge–Stone requires 419 SecAnd at the same depth 6; its higher node count offers no depth advantage over Brent–Kung but substantially limits SWAR packing opportunities.

3.3.1 Impact of bitslicing overhead

Bitslicing evaluates w independent bit operations in parallel within processor registers. However, transforming data into bitsliced representation requires a bit-matrix transposition. Efficient software transposition algorithms typically incur $O(w \log w)$ word operations, with comparable cost for reversing the layout.

Let C denote the total number of SecAnd_1 operations in the comparison circuit. The effective cost per instance under bitslicing is then

$$\frac{C}{w} + O(w \log w).$$

It is negligible when many comparisons are executed on already bitsliced data, but may dominate execution time when comparisons are sparse or frequently interact with standard data representations.

3.3.2 Use-case dependent strategy selection

The optimal comparison strategy depends strongly on the execution context.

Fully bitsliced implementations. When operands are already in bitsliced form and many comparisons are evaluated, ripple carry is typically optimal: it minimizes the total number of secure non-linear operations, and its depth penalty is irrelevant because w instances are processed in bulk.

Masked software without global bitslicing. When bitslicing would introduce frequent and costly data transpositions, Brent–Kung provides the best compromise. Its logarithmic depth limits latency while its sparse structure enables efficient SWAR parallelization without global data reorganization.

Hardware or latency-critical environments. Kogge–Stone achieves minimal logical depth with a larger number of prefix evaluations. It is mainly advantageous when massive hardware parallelism is available and all intermediate carries must be computed (e.g., for a full adder rather than a comparator).

Remark 1 (On alternative parallel-prefix adders). Several alternative topologies exist, including Sklansky, Ladner–Fischer, and Han–Carlson constructions [Skl60, HC87, LF80]. These explore different trade-offs between prefix node count, fanout, and circuit depth. In particular, Ladner–Fischer networks interpolate between dense topologies such as Kogge–Stone and sparse ones such as Brent–Kung. Logic reformulations such as the Ling adder [Lin81] further reduce constant factors in hardware critical paths by modifying carry equations. However, in masked software implementations, performance is primarily determined by the count and utilization of secure non-linear operations rather than by gate delay or fanout (which reflects physical wire loading and is irrelevant in software). Dense prefix networks limit opportunities for SWAR packing, while logic reformulations do not significantly reduce the number of masked AND operations.

4 Masked Gaussian Sampler

Most of the performance improvements over [EGRZ26] stem from a higher degree of parallelization. In this section, we first introduce an optimized method for computing the pivot value used in tree-based Gaussian sampling (Section 4.1). A direct implementation of the pivot computation described by Eid et al. would account for approximately 21% (resp. 46%) of the total cost for the FrodoKEM-1344 (resp. FrodoKEM-640 / 976) parameter sets; our reformulation eliminates this bottleneck. We then describe a low-depth sign evaluation that reduces multiplicative depth from $O(l_{\text{out}})$ to 1 (Section 4.2). Finally, we detail the complete bitsliced masked Gaussian sampler and argue that it preserves the Probe-Isolating Non-Interference (PINI) property of its building blocks (Section 4.3).

4.1 Optimized pivot value computation

The core principle of tree-based sampling is to dynamically compute a pivot value at each depth level, allowing the algorithm to perform a single comparison per level rather than exhaustively scanning the entire CDT. In this section, we introduce a highly efficient method for evaluating these pivots based on the public CDT constants and the secret comparison bits derived from previous levels.

Review of the state-of-the-art approach. Eid et al. evaluate the pivot using a nested factorization of the comparison results. For instance, rather than computing a standard multiplexer $(b \wedge c_0) \oplus (\neg b \wedge c_1)$, they use the equivalent form $(c_0 \oplus c_1) \wedge b \oplus c_1$, which seemingly halves the number of SecAnd gates. However, because the secret comparison bits are multiplied at each level by different layer-specific masked constants, no intermediate product can be memoized across levels. Evaluating the pivot at level i therefore requires $2^{i-1} - 1$ invocations of a SecAnd gate operating on full n-bit masked values. Summing over all l levels of the tree yields a total amortized cost of

$$\frac{n\,(2^l - l - 1)}{w}\ \mathsf{SecAnd}$$

on a w-bit bitsliced architecture.

ANF-based pivot evaluation. We propose to shift the granularity of the multiplexer so as to fully exploit the bitsliced representation. Rather than relying on a nested evaluation involving n-bit masked multiplications, we express the pivot value directly in Algebraic Normal Form (ANF) over the 1-bit comparison results. At any given level, the pivot is written as a XOR-sum of public n-bit constants (derived from the CDT), each conditioned by a 1-bit *path monomial* formed from previous comparison bits (eg. b_0, b_0b_1, $b_0b_1b_2$, ...). This formulation provides two key advantages:

- **Free scalar multiplications.** Multiplying a 1-bit masked monomial by a public n-bit constant is a linear operation in a bitsliced context: the single-bit mask is conditionally XORed into the active slices of the constant. This consumes zero SecAnd gates.
- **Perfect memoization.** Unlike the nested approach, the control monomials depend exclusively on the secret comparison bits b_k and not on the level-specific constants. A monomial such as b_2b_1 computed at Level 3 is kept in memory and directly reused to form $b_2b_1b_0$ at Level 4 via a single additional SecAnd.

As a result, the only non-linear operations required are those that build new monomials of degree ≥ 2. At level i (for $i \geq 3$), the newly introduced comparison bit must be multiplied with each of the $2^{i-2} - 1$ previously computed non-trivial monomials, yielding $2^{i-2} - 1$ single-bit SecAnd operations. Summing over all levels gives a total of

$$\sum_{i=3}^{l} \left(2^{i-2} - 1\right) = 2^{l-1} - l$$

single-bit SecAnd gates. By eliminating the dependence on the word size n, our ANF-based pivot evaluation reduces the total amortized cost to

$$\frac{2^{l-1} - l}{w}\ \mathsf{SecAnd},$$

a strictly superior throughput compared to the $n\,(2^l - l - 1)/w$ of the state of the art.

To illustrate the derivation of the public constants concretely, consider the pivot at Level 3 in Algorithm 1. At this depth, the pivot is selected among four CDT-derived constants (denoted $P_{00}, P_{01}, P_{10}, P_{11}$) according to the previous secret comparison bits b_2 and b_1. The naive selection formula is

$$\mathsf{pv} = P_{00}\,(\neg b_2\,\neg b_1) \oplus P_{01}\,(\neg b_2\,b_1) \oplus P_{10}\,(b_2\,\neg b_1) \oplus P_{11}\,(b_2\,b_1).$$

Table 2: Precomputed ANF differential constants for the FrodoKEM-1344 parameter set.

Constant	C_0	C_1	C_2	C_3	C_4	C_5	C_6
Value	32361	23462	9307	9142	23651	21812	21790

Table 3: Precomputed ANF differential constants for the FrodoKEM-640 and FrodoKEM-976 parameter sets.

	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7
FrodoKEM-640	32525	25843	6925	13363	19418	19866	19852	4643
FrodoKEM-976	32731	28571	4196	15915	16853	17394	17395	5638
	C_8	C_9	C_{10}	C_{11}	C_{12}	C_{13}	C_{14}	
FrodoKEM-640	28050	24584	16960	24646	16907	19724	19783	
FrodoKEM-976	27134	28554	19087	28557	19080	19558	19553	

Expanding the negations as $\neg b_i = 1 \oplus b_i$ and collecting terms by their positive monomials yields the flattened form

$$\mathsf{pv} = P_{00} \oplus (P_{00} \oplus P_{10})\, b_2 \oplus (P_{00} \oplus P_{01})\, b_1 \oplus (P_{00} \oplus P_{01} \oplus P_{10} \oplus P_{11})\, (b_2\, b_1).$$

Identifying the coefficients gives the differential constants used in our algorithm: $C_3 = P_{00}$, $C_4 = P_{00} \oplus P_{10}$, $C_5 = P_{00} \oplus P_{01}$, and $C_6 = P_{00} \oplus P_{01} \oplus P_{10} \oplus P_{11}$. We provide a standalone Python script in our public repository that automates this expansion and generates the full set of constants C_i from any given CDT. The resulting precomputed constants for our target parameter sets are given in Tables 2 and 3.

4.2 Low-depth sign evaluation

The second optimization targets the sign application step. In implementations sampling from a half-Gaussian distribution, the final step applies a random sign bit s to the sampled positive value X. In two's complement representation, this requires conditionally computing the negation $-X = \neg X + 1$.

The standard approach, as used by Eid et al., conditionally adds 1 to the bitwise negation of X using a ripple carry adder. Since the output X is typically small (eg. $l_{\text{out}} = 4$ bits for the distributions used by FrodoKEM), this requires $l_{\text{out}} - 1$ sequential SecAnd gates to propagate the carry, yielding multiplicative depth $l_{\text{out}} - 1$.

We reduce the multiplicative depth to exactly 1 by reformulating the sign evaluation as a masked multiplexer. Define the differential $D = X \oplus (-X)$. The signed output is then

$$Y = X \oplus s \cdot D,$$

since $Y = X$ when $s = 0$ and $Y = X \oplus X \oplus (-X) = -X$ when $s = 1$.

The key observation is that D can be computed at no non-linear cost by leveraging the ANF representation from Section 4.1. Each bit of the positive value X depends on a single comparison bit, so X is already expressed as a linear combination of the path monomials M_j conditioned by public constants. The negative value $-X$ admits an identical ANF representation using the two's complement of those same public constants. Consequently, the differential $D = X \oplus (-X)$ is obtained entirely with XOR operations over the already computed monomials, requiring no additional SecAnd.

Alg. 1 :: Masked FrodoKEM-1344

Output : Boolean sharing $\mathbf{out}^k$ of x following the error distribution of FrodoKEM-1344, where $k = 4$.

/* Initialization */
$\ell \leftarrow 15$
$\boldsymbol{r}^\ell \leftarrow \mathsf{SecUnif}^d(\ell)$
$\mathbf{signs}^1 \leftarrow \mathsf{SecUnif}^d(1)$
$\mathbf{out}^k \leftarrow 0$
/* Level 1: pivot is the constant C_0 */
$\mathbf{pv}^\ell \leftarrow \mathsf{Mask}(C_0)$
$\boldsymbol{b}_\mathbf{2}^1 \leftarrow \mathsf{SecLeq}(\boldsymbol{r}^\ell, \mathbf{pv}^\ell)$
$\mathbf{out}^k[2] \leftarrow \neg \boldsymbol{b}_\mathbf{2}^1$
/* Level 2: pivot depends on b_2 */
$\mathbf{pv}^\ell \leftarrow C_1 \oplus C_2 \cdot \boldsymbol{b}_\mathbf{2}^1$
$\boldsymbol{b}_\mathbf{1}^1 \leftarrow \mathsf{SecLeq}(\boldsymbol{r}^\ell, \mathbf{pv}^\ell)$
$\mathbf{out}^k[1] \leftarrow \neg \boldsymbol{b}_\mathbf{1}^1$
/* Level 3: build monomial $b_2 b_1$, evaluate ANF pivot */
$\boldsymbol{b_2 \cdot b_1^1} \leftarrow \mathsf{SecAnd}(\boldsymbol{b}_\mathbf{2}^1, \boldsymbol{b}_\mathbf{1}^1)$
$\mathbf{pv}^\ell \leftarrow C_3 \oplus C_4 \cdot \boldsymbol{b}_\mathbf{2}^1 \oplus C_5 \cdot \boldsymbol{b}_\mathbf{1}^1 \oplus C_6 \cdot \boldsymbol{b_2 \cdot b_1^1}$
$\boldsymbol{b}_\mathbf{0}^1 \leftarrow \mathsf{SecLeq}(\boldsymbol{r}^\ell, \mathbf{pv}^\ell)$
$\mathbf{out}^k[0] \leftarrow \neg \boldsymbol{b}_\mathbf{0}^1$
/* Sign application */
$\mathbf{out}^k \leftarrow \mathsf{SecAdd}\big(\mathbf{out}^k \oplus (2^\ell - 1) \cdot \mathbf{signs}^1,\ \mathbf{signs}^1\big)$
return $\mathbf{out}^k$

Furthermore, in two's complement arithmetic, the least significant bit (LSB) of any integer x is invariant under negation ($x_0 = (-x)_0$). Computing $-x = \neg x + 1$ first inverts the LSB, but the subsequent addition of 1 inevitably restores its original value. As a result, the LSB of the differential is always zero ($D_0 = 0$), making the evaluation of $s \cdot D_0$ entirely free.

The final signed output requires computing $s \cdot D_i$ for each of the remaining $l_{\text{out}} - 1$ bits. These multiplications are completely independent, yielding exactly $l_{\text{out}} - 1$ single-bit SecAnd gates at multiplicative depth 1. This matches the gate count of the RCA but eliminates the sequential carry dependency.

Remark 2. In a fully bitsliced software implementation, circuit depth does not affect throughput; the RCA and the parallel multiplexer have identical amortized cost. The depth reduction is primarily beneficial in hardware or Multi Party Computation (MPC) contexts where circuit latency is the dominant bottleneck. In our bitsliced software implementation, we therefore retain the simpler RCA formulation for the sign step (cf. Algorithms 1 and 2).

4.3 Complete masked Gaussian sampling gadget

We now describe the full procedure for our bitsliced masked Gaussian sampler and establish its security in the t-probing model. Complete pseudocode is provided as Algorithms 1 and 2.

Our implementation relies exclusively on established PINI gadgets: the SecAnd gate from [CS20, Algorithm 2], the SecUnmask operation from [ABC+23, Algorithm 3], and a variant of the SecLeq comparison from [ABC+23, Algorithm 4]. Our SecLeq gadget

Alg. 2 :: Masked FrodoKEM-976

Output : Boolean sharing $\mathbf{out}^k$ of x following the error distribution of FrodoKEM-976, where $k = 5$.

/* Initialization */
$\ell \leftarrow 15$
$\boldsymbol{r}^\ell \leftarrow \mathsf{SecUnif}^d(\ell)$
$\mathbf{signs}^1 \leftarrow \mathsf{SecUnif}^d(1)$
$\mathbf{out}^k \leftarrow 0$
/* Level 1: pivot is the constant C_0 */
$\mathbf{pv}^\ell \leftarrow \mathsf{Mask}(C_0)$
$\boldsymbol{b_3^1} \leftarrow \mathsf{SecLeq}(\boldsymbol{r}^\ell, \mathbf{pv}^\ell)$
$\mathbf{out}^k[3] \leftarrow \neg \boldsymbol{b_3^1}$
/* Level 2: pivot depends on b_3 */
$\mathbf{pv}^\ell \leftarrow C_1 \oplus C_2 \cdot \boldsymbol{b_3^1}$
$\boldsymbol{b_2^1} \leftarrow \mathsf{SecLeq}(\boldsymbol{r}^\ell, \mathbf{pv}^\ell)$
$\mathbf{out}^k[2] \leftarrow \neg \boldsymbol{b_2^1}$
/* Level 3: build monomial $b_3 b_2$, evaluate ANF pivot */
$\boldsymbol{b_3 \cdot b_2^1} \leftarrow \mathsf{SecAnd}(\boldsymbol{b_3^1}, \boldsymbol{b_2^1})$
$\mathbf{pv}^\ell \leftarrow C_3 \oplus C_4 \cdot \boldsymbol{b_3^1} \oplus C_5 \cdot \boldsymbol{b_2^1} \oplus C_6 \cdot \boldsymbol{b_3 \cdot b_2^1}$
$\boldsymbol{b_1^1} \leftarrow \mathsf{SecLeq}(\boldsymbol{r}^\ell, \mathbf{pv}^\ell)$
$\mathbf{out}^k[1] \leftarrow \neg \boldsymbol{b_1^1}$
/* Level 4: build new monomials, evaluate ANF pivot */
$\boldsymbol{b_3 \cdot b_1^1} \leftarrow \mathsf{SecAnd}(\boldsymbol{b_3^1}, \boldsymbol{b_1^1})$
$\boldsymbol{b_2 \cdot b_1^1} \leftarrow \mathsf{SecAnd}(\boldsymbol{b_2^1}, \boldsymbol{b_1^1})$
$\boldsymbol{b_3 \cdot b_2 \cdot b_1^1} \leftarrow \mathsf{SecAnd}(\boldsymbol{b_3 \cdot b_2^1}, \boldsymbol{b_1^1})$ /* reuses $b_3 b_2$ */
$\mathbf{pv}^\ell \leftarrow C_7 \oplus C_8 \cdot \boldsymbol{b_3^1} \oplus C_9 \cdot \boldsymbol{b_2^1} \oplus C_{10} \cdot \boldsymbol{b_1^1}$
$\quad \oplus C_{11} \cdot \boldsymbol{b_3 \cdot b_2^1} \oplus C_{12} \cdot \boldsymbol{b_3 \cdot b_1^1} \oplus C_{13} \cdot \boldsymbol{b_2 \cdot b_1^1} \oplus C_{14} \cdot \boldsymbol{b_3 \cdot b_2 \cdot b_1^1}$
$\boldsymbol{b_0^1} \leftarrow \mathsf{SecLeq}(\boldsymbol{r}^\ell, \mathbf{pv}^\ell)$
$\mathbf{out}^k[0] \leftarrow \neg \boldsymbol{b_0^1}$
/* Sign application */
$\mathbf{out}^k \leftarrow \mathsf{SecAdd}\big(\mathbf{out}^k \oplus (2^\ell - 1) \cdot \mathbf{signs}^1,\ \mathbf{signs}^1\big)$
return $\mathbf{out}^k$

takes two secret-shared values x and y as inputs and computes the final carry c_n of the addition $x + \neg y$ (with initial carry $c_0 = 0$). As shown in Section 3, this carry satisfies $c_n = 1 \iff x > y$; negating the output—which is free in Boolean masking (flip one share)—therefore yields the less-than-or-equal predicate $x \leq y$. Unlike [ABC+23], we retain the output in shared form rather than unmasking it, which does not affect the PINI property of the gadget.

As detailed in Section 4.1, the primary structural divergence from the sampler of Eid et al. [EGRZ26] lies in the pivot computation. To decouple the 1-bit non-linear operations from the n-bit linear scalings, we evaluate the pivot at each level using a flat ANF polynomial over the comparison bits. The generation of monomials via 1-bit SecAnd gates is detailed step by step in the algorithms below.

The layer-specific precomputed differential constants are denoted $C_0, C_1, \ldots, C_{14}$. Their values, derived from the CDT by fully expanding and flattening the ANF as described in Section 4.1, are listed in Tables 2 and 3. The algorithm for FrodoKEM-640 follows the same structure as FrodoKEM-976, differing only in the values of the public constants.

The sign application uses a conditional two's complement negation over q-bit words: $\mathsf{out} \leftarrow (\mathsf{out} \oplus (2^q - 1) \cdot s) + s$, where $q = 16$ is the Frodo modulus bit-width and s is the 1-bit sign share. When $s = 0$ the output is unchanged; when $s = 1$ this evaluates $\neg\mathsf{out} + 1 = -\mathsf{out}$ in 16-bit two's complement, embedding the k-bit sampled value into the ring $\mathbb{Z}_{2^q}$.

4.3.1 PINI security

Our gadget is constructed exclusively by composing PINI-secure building blocks (SecAnd, SecUnmask, SecLeq). Since the PINI property is preserved under arbitrary composition with shared-wire connections [CS20, Theorem 1][4], the complete sampler inherits the PINI property and is therefore secure in the t-probing model for any number of shares $d \geq t + 1$.

4.4 Gate-level cost analysis

To properly compare the inherent efficiency of the different sampling strategies independently of their software implementation artifacts, we present a theoretical gate-level analysis focusing on the amortized non-linear cost (SecAnd gates). We consider the evaluation of a CDT of n elements with k-bit precision, followed by an ℓ-bit sign application, where $\ell = \lceil \log_2 n \rceil$ corresponds to the bit-length of the output. The tree-based approach evaluates exactly ℓ comparisons and one pivot computation, whereas the linear approach evaluates n comparisons. Note that the base randomness cost (excluding the masking overhead) is strictly identical across all methods. Furthermore, while we have also evaluated the linear gate count, which slightly favors the tree-based approach, linear operations represent a negligible fraction of the total execution time compared to non-linear operations and are therefore omitted from this breakdown.

The exact cost formulas for the underlying operations (comparisons, pivot computation, and sign application) depend heavily on the chosen architecture and whether bitslicing is applied, and is detailed in Section 3.3 for the comparisons, Section 4.1 for the pivot and Section 4.2 for the sign application.

Remark 3. Krausz et al. [KLS+25] do not compute their comparisons exactly as described in our framework; they instead rely on an optimized comparison gadget from [KLRBG23]. However, in bitsliced form, their gadget exhibits exactly the same multiplicative depth (k) and amortized cost (k SecAnd and $3k$ SecXor).

Finally, it is crucial to note that the SecAnd gates used in our bitsliced implementation are PINI-secure. As detailed in Section 2.1, PINI multipliers inherently require twice the operations and the same amount of randomness than the standard ISW multipliers used by non-bitsliced approaches like [EGRZ26]. However, standard implementations must routinely interleave expensive refresh gadgets throughout their circuit to prevent leakage and preserve security. These refresh operations are completely omitted from our gate-level analysis.

Table 4 evaluates the non-linear cost concretely for the FrodoKEM-1344 parameter set. In this table, we also include a BK variant for a non bitsliced implementation to highlight the efficacy of this new adder in non-bitsliced settings.

5 Implementation and benchmarks

We evaluate our masked Gaussian sampler on an ARM Cortex-M4 (STM32F415) microcontroller and compare against the implementations of Eid et al. [EGRZ26], Gérard and Guerreau [GG26] , and Krausz et al. [KLS+25] . We first describe the benchmarking

[4]Composability holds because PINI simulators for each gadget require only the output shares and the specific input shares indexed by the probe positions; no coordination between gadgets is needed.

Table 4: Concrete amortized SecAnd gate count per sample for FrodoKEM-1344 ($n = 7, \ell = 3, w = 32$). For standard implementations, values assume evaluation over full 32-bit registers. The SWAR BK projection assumes two 16-bit parallel instances.

Sampler	Comparisons	Pivot	Sign Add	Total SecAnd
[GG26]	$7 \times 290^*$	–	290	**2320**
[EGRZ26]	$3 \times 290^*$	128	290	**1288**
[EGRZ26] (BK)	3×22.5	32	22.5	**122**
[KLS+25]	105/32	–	2/32	**3.34**
This work	45/32	1/32	2/32	**1.50**

* The Kogge-Stone comparison could pack 16-bit words side-by-side into a single 32-bit register, thereby dividing the amortized cost by two. However, neither [GG26] nor [EGRZ26] employ this strategy.

methodology (Section 5.1), then present first-order results (Section 5.2), higher-order scaling (Section 5.3), and a per-gadget cost breakdown (Section 5.4). Our implementation is publicly available (Section 5.5).

5.1 Benchmarking methodology

The source code of Eid et al. [EGRZ26] is not publicly available; all comparisons are therefore conducted against their published cycle counts rather than re-measured on identical hardware. Similarly, Krausz et al. [KLS+25] report total cycle counts for a full batch of N samples, omitting the cost of randomness generation. To enable a meaningful comparison on a per-sample basis, we divide their reported batch cycles by N to obtain the amortized base cost per sample. This comprehensive framework makes the comparison inherently conservative: the first-order implementation of [EGRZ26] is written entirely in hand-optimized assembly , and [KLS+25] utilizes inline assembly for critical inner loops , whereas our implementation is written in pure C throughout.

A second methodological difference concerns randomness generation. Eid et al. and Krausz et al. exclude the cost of randomness from their reported cycle counts, whereas randomness generation dominates the cost of our masked gadgets. We therefore include it in all our measurements. Randomness is produced by the built-in TRNG of the STM32F415 microcontroller, which outputs 32 bits every 108 cycles ($C_{\mathrm{TRNG}} = 54$ cycles per 16-bit word). To obtain comparable figures, we adjust the cycle counts of [EGRZ26], [GG26], and [KLS+25] by adding the exact TRNG cost for the randomness they consume.

Finally, a third methodological consideration concerns the overhead of data representation conversions (i.e., packing into and unpacking from a bitsliced state), which we explicitly exclude from our benchmarks. This choice is grounded in the operational integration of masked sampling within post-quantum cryptography. On the input side, when randomness is directly drawn from a hardware TRNG, the resulting random words can be natively interpreted as a bitsliced stream, completely bypassing the need for an initial data transposition. On the output side, the Boolean shares of the generated Gaussian sample must subsequently undergo a Boolean-to-Arithmetic (B2A) conversion to be used in the wider scheme. Because the state-of-the-art masked B2A gadget [BC22] inherently requires bitsliced inputs, a sampler relying on a standard word representation would merely defer the bitslicing penalty to its output layer. Consequently, incorporating representation transformation overheads into the benchmarks would, at worst, balance out between the

Table 5: Comparison of execution time (in clock cycles) for a first-order masked Gaussian sample across FrodoKEM parameter sets. For Eid et al. [EGRZ26], [GG26], and [KLS+25], the reported cycles are adjusted by adding the cost of generating the required randomness. $C_R = 54$ is the cost of generating 16 bits of randomness using our TRNG. Ratios (in parentheses) are relative to this work.

Sampler	FrodoKEM-640	FrodoKEM-976	FrodoKEM-1344
[GG26]	$5\,590 + 244\,C_R$ (**9.35**×)	$4\,780 + 206\,C_R$ (**7.92**×)	$3\,160 + 130\,C_R$ (**7.26**×)
[EGRZ26]	$800 + 57\,C_R$ (**1.93**×)	$800 + 57\,C_R$ (**1.93**×)	$557 + 40\,C_R$ (**1.94**×)
[KLS+25]	$1\,870 + 21.4\,C_R$ (**1.51**×)	$1\,617 + 21.4\,C_R$ (**1.38**×)	$1\,024 + 11.5\,C_R$ (**1.17**×)
This work	**2 008**	**2 008**	**1 403**

Table 6: Comparison of the required amount of randomness (number of 16-bit random values generated) for a first-order masked Gaussian sample across FrodoKEM parameter sets. Ratios (in parentheses) are relative to this work.

Sampler	FrodoKEM-640	FrodoKEM-976	FrodoKEM-1344
Gérard–Guerreau [GG26]	244 (**39.04**×)	206 (**32.96**×)	130 (**25.68**×)
Eid et al. [EGRZ26]	57 (**9.12**×)	57 (**9.12**×)	40 (**7.90**×)
Krausz et al. [KLS+25]	21.43 (**3.43**×)	21.43 (**3.43**×)	11.47 (**2.27**×)
This work	**6.25**	**6.25**	**5.06**

two paradigms, and at best, further emphasize the performance advantage of our bitsliced approach.

5.2 First-order comparison

Table 5 reports execution times for a single first-order masked Gaussian sample, the only setting for which [EGRZ26] provides a fully assembly-optimized implementation. Our sampler achieves a consistent speedup over all existing works , outperforming the concurrent bitsliced approach of [KLS+25] by a factor of 1.17× to 1.51× across all three FrodoKEM parameter sets .

Table 6 further compares the randomness requirements of each approach. The reduction remains substantial: our sampler consumes up to 3.4× fewer random values than the concurrent bitsliced design of [KLS+25], reflecting our more compact evaluation strategy that drastically minimizes the number of non-linear operations.

5.3 Higher-order scaling

Table 7 extends the comparison to masking orders 2 through 8, reporting both execution time (including randomness generation) and randomness consumption. For conciseness, only the results of [EGRZ26] are included as a baseline. Because Krausz et al. [KLS+25] do not report execution times for higher masking orders, their implementation is omitted from this table. However, our analysis in Section 4.4 highlights that our sampler uses twice as few non-linear gates. As the cost of these gates is quadratic in the number of shares, in both randomness consumption and computations, the trend when going at higher-order will be in our favor (exactly in the same way as the comparison with [EGRZ26]).

Table 7: Execution time (in clock cycles) and randomness requirements (number of 16-bit random values generated) of our masked Gaussian sampler for higher masking orders across the different FrodoKEM parameter sets.

Parameter	Masking order	2	4	6	8
FrodoKEM-640 / 976	Cycles [EGRZ26]	15 450 (**5.13**×)	43 960 (**7.29**×)	87 884 (**8.57**×)	147 147 (**9.29**×)
	Cycles (This work)	**3 009**	**6 031**	**10 257**	**15 840**
	Rands [EGRZ26]	171 (**10.86**×)	570 (**12.00**×)	1197 (**12.44**×)	2052 (**12.67**×)
	Rands (This work)	**15.8**	**47.5**	**96.2**	**162.0**
FrodoKEM-1344	Cycles [EGRZ26]	10 850 (**5.14**×)	30 833 (**7.31**×)	61 609 (**8.54**×)	103 123 (**9.22**×)
	Cycles (This work)	**2 110**	**4 218**	**7 213**	**11 190**
	Rands [EGRZ26]	120 (**9.85**×)	400 (**11.23**×)	840 (**11.78**×)	1440 (**12.08**×)
	Rands (This work)	**12.2**	**35.6**	**71.3**	**119.2**

Table 8: Detailed execution time breakdown (in clock cycles) of our masked Gaussian sampler for masking orders 1, 3, 5, and 7. For each order, timings are reported for the FrodoKEM-640 and FrodoKEM-1344 parameter sets.

Operation	Order 1		Order 3		Order 5		Order 7	
	640	**1344**	**640**	**1344**	**640**	**1344**	**640**	**1344**
Total	2 008	1 403	4 330	3 043	7 951	5 594	12 797	9 034
SecLeq	1 286	1 017	3 061	2 348	5 929	4 498	9 912	7 485
Pivot computation	486	178	796	281	1 245	423	1 721	569
Add	104	79	227	172	423	320	693	522
Other	132	129	246	242	354	353	471	458

The speedup grows with the masking order: from approximately 5× at order 2 to over 9× at order 8. This widening gap reflects the quadratic scaling of SecAnd cost in the number of shares d: because our approach requires fundamentally fewer non-linear operations, the savings compound as d increases. Randomness consumption follows the same trend, with our implementation requiring over 12× fewer random values at order 8.

5.4 Per-gadget cost breakdown

Table 8 provides a detailed breakdown of the execution time spent in each individual gadget. The secure comparison (SecLeq) dominates overall cost at every masking order, accounting for over 64% of the total at first order and growing to over 77% at order 7. The pivot computation constitutes the second largest contribution for FrodoKEM-640 / 976 (where the tree has four levels), while remaining modest for FrodoKEM-1344 (three levels). The final addition incurs a non-negligible but secondary cost. At the elementary level, execution time is almost entirely determined by the number of SecAnd evaluations, a dominance that increases with the number of shares.

5.5 Availability and reproducibility

The full implementation is publicly available, together with build scripts targeting both an emulator and STM32F3,/ STM32F4 boards, and can be found here:

https://github.com/hoffmanncl/frodo_cdt_sampler

To the best of our knowledge, no open-source cycle-accurate emulator exists for this architecture; precise benchmark reproduction therefore requires execution on physical hardware. Assembly optimization of the performance-critical SecAnd gadget, which would yield additional speedups orthogonal to the algorithmic improvements presented here, is left as future work.

5.6 Leakage assessment

To ensure that our implementation does not exhibit trivial leaks, we performed a first order Test Leakage Assessment (TVLA) on our C implementation. In particular, we run it on the FrodoKEM-1344 parameter set in order to mimick the TVLA of [EGRZ26]. As in [EGRZ26], we perform a "fixed vs random" TVLA, where two sets are generated:

- the fixed set, where the input randomness of the sampler is fixed,
- the random set, where the input randomness of the sampler is freshly sampled at each run.

For both sets, the masking randomness is freshly sampled at each run, using the on-board TRNG. Following the methodology of [SM15], measurements for each set were interleaved, i.e. we performed alternatively one run for the fixed set followed by one run for the random set.

A Chipwhisperer Pro CW-1200 was used for the measurements, using an STM32F415 target board. Capture was triggered by the Device Under Test (DUT) by setting a GPIO pin high right before sampling. Since the Chipwhisperer Pro drived the DUT's clock signal, no additional preprocessing was needed to achieve trace alignement.

Welch's t-test was computed over 200k traces (100k fixed and 100k random), using the Python library SCALib [CB23]. Contrary to [EGRZ26], we do not set the leakage threshold to 4.5 and instead adopt the methodology of [DZD+17] since our measurements span a much longer timeframe, which could cause measurements to cross the 4.5 threshold even on non-leaky devices. This is due to the fact that bitslicing relies on high throughput compensating high latency. Our threshold value is set as $\mathsf{CDF}^{-1}_{N(0,1)}(1-\alpha_{\mathrm{TH}}/2)$, where $\mathsf{CDF}^{-1}_{N(0,1)}$ is the inverse of the CDF of a standard normal, α_{TH} is set as the conservative $1-(1-0.01)^{1/n}$, for n the number of samples (See [DZD+17, Table 1]). For our $n = 38600$, we get a threshold of 4.75 [5]. We present the results of our t-test in Figure 4, where it can be observed that none of the values cross the above threshold. The code used for generating this tests is public and can be found alongside our implementation

[5] However, we notice that our TVLA still comply to the 4.5 threshold requirement.

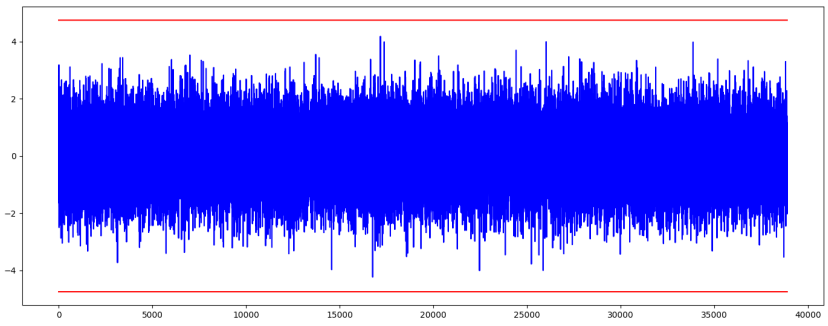

Figure 4: fixed vs random TVLA results using 200k traces for our masked FrodoKEM-1344 sampler. The 4.75 threshold (see Section 5.6) is delimited by the red lines.

References

[ABC+23] Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Tobias Schneider, Markus Schönauer, François-Xavier Standaert, and Christine van Vredendaal. Protecting Dilithium against leakage revisited sensitivity analysis and improved implementations. *IACR TCHES*, 2023(4):58–79, 2023.

[ANS26] Règles et recommendations concernant le choix et le dimensionnement des mécanismes cryptographiques v3.00. Technical Report PG-083, Agence nationale pour la sécurité des systèmes d'information, March 2026. https://messervices.cyber.gouv.fr/documents-guides/anssi-guide-mecanismes-crypto-3.00.pdf.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 116–129. ACM Press, October 2016.

[BBE+19] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Mélissa Rossi, and Mehdi Tibouchi. GALACTICS: Gaussian sampling for lattice-based constant- time implementation of cryptographic signatures, revisited. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2147–2164. ACM Press, November 2019.

[BBK16] Nina Bindel, Johannes Buchmann, and Juliane Krämer. Lattice-based signature schemes and their sensitivity to fault attacks. Cryptology ePrint Archive, Report 2016/415, 2016.

[BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based KEMs. *IACR TCHES*, 2022(4):553–588, 2022.

[BD23] Sven Bauer and Fabrizio De Santis. A differential fault attack against deterministic falcon signatures. Cryptology ePrint Archive, Report 2023/422, 2023.

[BD25] Sven Bauer and Fabrizio De Santis. Forging dilithium and falcon signatures by single fault injection. Cryptology ePrint Archive, Report 2025/2029, 2025.

[BHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, gauss, and reload - A cache attack on the BLISS lattice-based signature scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 323–345. Springer, Berlin, Heidelberg, August 2016.

[BK82] Brent and Kung. A regular layout for parallel adders. *IEEE transactions on Computers*, 100(3):260–264, 1982.

[BSI26] Cryptographic mechanisms: Recommendations and key lengths. Technical Report TR-02102-1, Budesamt für Sicherheit in der Informationstechnik, January 2026. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf.

[CB23] Gaëtan Cassiers and Olivier Bronchain. Scalib: A side-channel analysis library. *Journal of Open Source Software*, 8(86):5196, 2023.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to Boolean masking with logarithmic complexity. In Gregor Leander, editor, *FSE 2015*, volume 9054 of *LNCS*, pages 130–149. Springer, Berlin, Heidelberg, March 2015.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Transactions on Information Forensics and Security*, 15:2542–2555, 2020.

[DFW20] Yusong Du, Baoying Fan, and Baodian Wei. Arbitrary-centered discrete gaussian sampling over the integers. In Joseph K. Liu and Hui Cui, editors, *ACISP 20*, volume 12248 of *LNCS*, pages 391–407. Springer, Cham, November / December 2020.

[DZD+17] A Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. In *International Conference on Smart Card Research and Advanced Applications*, pages 105–122. Springer, 2017.

[EFGT16] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Loop-abort faults on lattice-based Fiat-Shamir and hash-and-sign signatures. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 140–158. Springer, Cham, August 2016.

[EFGT17] Thomas Espitau, Pierre-Alain Fouque, Benoît Gérard, and Mehdi Tibouchi. Side-channel attacks on BLISS lattice-based signatures: Exploiting branch tracing against strongSwan and electromagnetic emanations in microcontrollers. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1857–1874. ACM Press, October / November 2017.

[EGRZ26] Elie Eid, Aurélien Greuet, Nathan Reboud, and Rina Zeitoun. Efficient high-order masking of FrodoKEMs CDT-based Gaussian sampler. *IACR TCHES*, 2026(3), 2026.

[EUC25] Eucc agreed cryptographic mechanisms v2. Technical report, European Union Agency for Cybersecurity, May 2025. https://certification.enisa.europa.eu/publications/eucc-guidelines-cryptography_en.

[FKT+20] Pierre-Alain Fouque, Paul Kirchner, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Key recovery from Gram-Schmidt norm leakage in hash-and-sign signatures over NTRU lattices. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 34–63. Springer, Cham, May 2020.

[FO13] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. *Journal of Cryptology*, 26(1):80–101, January 2013.

[GG26] François Gérard and Morgane Guerreau. High-order and Cortex-M4 first-order implementations of masked FrodoKEM. In *CASCADE 2026*, LNCS. Springer, 2026. to appear.

[GMRR22] Morgane Guerreau, Ange Martinelli, Thomas Ricosset, and Mélissa Rossi. The hidden parallelepiped is back again: Power analysis attacks on falcon. *IACR TCHES*, 2022(3):141–164, 2022.

[GR19] François Gérard and Mélissa Rossi. An efficient and provable masked implementation of qTESLA. In Sonia Belaïd and Tim Güneysu, editors, *CARDIS 2019*, volume 11833 of *LNCS*, pages 74–91. Springer, 2019.

[GR24] Morgane Guerreau and Mélissa Rossi. A not so discrete sampler: Power analysis attacks on HAWK signature scheme. *IACR TCHES*, 2024(4):156–178, 2024.

[HC87] T. Han and D. A. Carlson. Fast area-efficient vlsi adders. *Proceedings of the 8th IEEE Symposium on Computer Arithmetic*, pages 49–56, 1987.

[HPRR20] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. Isochronous gaussian sampling: From inception to implementation. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 53–71. Springer, Cham, 2020.

[HPT25] Calvin Abou Haidar, Quentin Payet, and Mehdi Tibouchi. Crowhammer: Full key recovery attack on falcon with a single rowhammer bit flip. In *CRYPTO 2025, Part V*, LNCS, pages 103–135. Springer, Cham, August 2025.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Berlin, Heidelberg, August 2003.

[KA21] Emre Karabulut and Aydin Aysu. FALCON down: Breaking FALCON post-quantum signature scheme through side-channel attacks. In *DAC 2021*, pages 691–696. IEEE, 2021.

[KLRBG23] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. A holistic approach towards side-channel secure fixed-weight polynomial sampling. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part II*, volume 13941 of *LNCS*, pages 94–124. Springer, Cham, May 2023.

[KLS+25] Markus Krausz, Georg Land, Florian Stolz, Jan Richter-Brockmann, and Tim Güneysu. To extend or not to extend: Agile masking instructions for PQC. In *CANS 25*, LNCS, pages 70–92. Springer, Singapore, December 2025.

[KS73] Peter M Kogge and Harold S Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE transactions on computers*, 100(8):786–793, 1973.

[LDK+22] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

[Lin81] H. Ling. High-speed binary adder. *IBM Journal of Research and Development*, 25(3):156–166, 1981.

[LZY+25] Xiuhan Lin, Shiduo Zhang, Yang Yu, Weijia Wang, Qidi You, Ximing Xu, and Xiaoyun Wang. Thorough power analysis on falcon gaussian samplers and practical countermeasure. In Tibor Jager and Jiaxin Pan, editors, *PKC 2025, Part I*, volume 15674 of *LNCS*, pages 229–258. Springer, Cham, May 2025.

[MKK+23] Soundes Marzougui, Ievgen Kabin, Juliane Krämer, Thomas Aulbach, and Jean-Pierre Seifert. On the feasibility of single-trace attacks on the gaussian sampler using a CDT. In Elif Bilge Kavun and Michael Pehl, editors, *COSADE 2023*, volume 13979 of *LNCS*, pages 149–169. Springer, Cham, April 2023.

[MW17] Daniele Micciancio and Michael Walter. Gaussian sampling over the integers: Efficient, generic, constant-time. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 455–485. Springer, Cham, August 2017.

[MWG+22] Soundes Marzougui, Nils Wisiol, Patrick Gersch, Juliane Krämer, and Jean-Pierre Seifert. Machine-learning side-channel attacks on the GALACTICS constant-time implementation of BLISS. In *ARES 2022*, pages 34:1–34:11. ACM, 2022.

[PBY17] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. To BLISS-B or not to be: Attacking strongSwan's implementation of post-quantum signatures. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1843–1855. ACM Press, October / November 2017.

[Pre23] Thomas Prest. A key-recovery attack against mitaka in the t-probing model. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 205–220. Springer, Cham, May 2023.

[SD25] Vladimir Sarde and Nicolas Debande. Improvement of side-channel attacks on mitaka. Cryptology ePrint Archive, Report 2025/873, 2025.

[Skl60] Jack Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(2):226–231, 1960.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Tim Güneysu and Helena Handschuh, editors, *CHES 2015*, volume 9293 of *LNCS*, pages 495–513. Springer, Berlin, Heidelberg, September 2015.

[SS16] Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 180–194. Springer, Cham, August 2016.

[War13] Henry S. Warren, Jr. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2013.

[WGS22] Nils Wisiol, Patrick Gersch, and Jean-Pierre Seifert. Cycle-accurate power side-channel analysis using the ChipWhisperer: A case study on Gaussian sampling. In Ileana Buhan and Tobias Schneider, editors, *CARDIS 2022*, volume 13820 of *LNCS*, pages 205–224. Springer, 2022.

[ZLYW23] Shiduo Zhang, Xiuhan Lin, Yang Yu, and Weijia Wang. Improved power analysis attacks on falcon. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part IV*, volume 14007 of *LNCS*, pages 565–595. Springer, Cham, April 2023.

[ZSS20a] Raymond K. Zhao, Ron Steinfeld, and Amin Sakzad. COSAC: COmpact and scalable arbitrary-centered discrete gaussian sampling over integers. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 284–303. Springer, Cham, 2020.

[ZSS20b] Raymond K. Zhao, Ron Steinfeld, and Amin Sakzad. FACCT: FAst, Compact, and Constant-Time discrete Gaussian sampler over integers. *IEEE Trans. Computers*, 69(1):126–137, 2020.