To extend or not to extend: Agile Masking Instructions for PQC

Markus Krausz[1], Georg Land[1], Florian Stolz[1], Dennis Naujoks[4*], Jan Richter-Brockmann[1], Tim Güneysu[1,2] and Lucie Kogelheide[3*]

[1] Ruhr-University, Bochum, Germany, firstname.lastname@rub.de,mail@georg.land
[2] DFKI GmbH, Bremen, Germany
[3] BWI GmbH, Bonn, Germany
[4] ETAS GmbH, Bochum, Germany

Abstract. Splitting up sensitive data into multiple shares – termed *masking* – has proven an effective countermeasure against various types of Side-Channel Analysis (SCA) on cryptographic implementations. However, in software this approach not only leads to dramatic performance overheads for non-linear operations, but also suffers from microarchitectural leakage, which is hard to avoid. Both problems can be addressed with one solution: masked hardware accelerators. In this context, Gao et *al.* [GGM+21] presented a RISC-V Instruction Set Extension (ISE) with masked Boolean and arithmetic instructions to accelerate masked software implementations of block ciphers. In this work, we demonstrate how this ISE can be applied to and extended for Post-Quantum Cryptography (PQC) components, forming a crypto-agile solution. We provide masked implementations based on three different ISE constellations for multiple highly relevant components, including Cumulative Distribution Table (CDT) sampling and polynomial rotation, which, to the best of our knowledge, have not been masked before. With the masked instructions, we measure speedups of more than one order of magnitude compared to sophisticated bitsliced implementations and even up to two orders of magnitude for non-bitsliced implementations. We assert the first-order security of our implementation with a practical evaluation.

Keywords: PQC, Fixed Weight Polynomial Sampling, SCA, Masking, RISC-V, BIKE, Frodo, HQC

1 Introduction

Nowadays, embedded devices are interconnected via a broad variety of communication channels. In case sensitive data is transmitted via these channels, the communication should be encrypted. Although the chosen algorithms are considered secure, *implementations* of cryptographic algorithms are generally vulnerable to side-channel attacks on any platform. For example, an adversary can exploit secret-dependent runtime differences of the algorithms caused by memory accesses, branching or instructions with data-dependent runtime. Moreover, having physical access to the target device allows the adversary to measure the device's power consumption directly or indirectly via its electromagnetic emission, thereby gaining information on secret key material. Especially power Side-Channel Analysis (SCA) recently became more and more accessible due to simple and inexpensive measurement setups [Inc].

*The respective work has been conducted at TÜV Informationstechnik GmbH.

Researchers from academia and industry proposed many approaches to counteract these attack vectors. In order to prevent timing attacks, the so-called constant-time programming paradigm [Por] emerged as a generic defense mechanism. The most promising countermeasure against power SCA is masking [PR13], splitting sensitive data into secret shares. Consequently, underlying operations have to be transformed to be able to perform these operations on shares.

A fact that is rarely expressed explicitly is that a correctly masked implementation is necessarily also secure against timing side-channels, because secret-dependent branches and memory accesses cannot be masked and thus must be secured using constant-time methods. Additionally, instructions with operand-dependent runtime can usually not be performed with masked values. Protection against timing attacks with constant-time implementations is often achieved by always executing with worst-case runtime and applying dummy memory accesses to all possible locations instead of only accessing the secret-dependent value. In order to achieve protection against power SCA, these extra operations need to be masked as well, introducing additional overhead.

Applying masking techniques to implementations of novel cryptographic schemes raises new challenges. This includes novel schemes designed to be secure against attacks mounted on quantum computers. The US-American National Institute of Standards and Technology (NIST) already selected four different Post-Quantum Cryptography (PQC) schemes for standardization, at least one of the Round 4 candidates is expected to be selected too, and 40 new schemes are currently examined in the call for additional signature schemes. On top of that, agencies from other countries with a strong economic infrastructure (e.g. the French ANSSI or German BSI) partially recommend other PQC schemes, South Korea even started a similar competition process as NIST. On the other hand, the cryptographic foundations of many schemes require more scrutiny, which can lead to drop-outs of promising candidates [CD23]. The set of relevant PQC schemes is therefore extensive and dynamic, and brings new issues regarding efficient and secure implementations.

For example, as shown by Krausz et *al.* [KLRBG23], protecting fixed-weight polynomial samplers of common PQC schemes like BIKE, HQC, NTRU and many others with masking introduces a huge overhead with respect to the runtime and randomness requirements of the design. One reason for this huge overhead lies in the simultaneous protection against timing and power side channels.

Besides performance degradation, masked software implementations also have to tackle the issue of microarchitectural power leakage [MPW22], where a masked implementation can be secure from a source code perspective, but still exhibits side-channel leakage due to unforeseeable combinations of shares induced by microarchitectural behavior.

A powerful solution to tackle performance degradation and microarchitectural leakage at the same time, is to extend cores with dedicated accelerators for masking. In this context, Gao et *al.* [GGM+21] presented a RISC-V Instruction Set Extension (ISE) with masked Boolean and arithmetic instructions. With *masked instructions* we refer to instructions that operate on both shares (for first-order masking) of their masked operands simultaneously. Their low-level masked instructions mirror common general-purpose instructions, e.g., a Boolean AND. The simultaneous computation on both shares leads to acceleration of linear operations and with hardware logic for non-linear operations, the speed-up can be even more increased. Additionally, controlling the hardware allows for mitigation of any microarchitectural behavior that would lead to unexpected power leakage, such as hidden registers. However, the evaluation done by Gao et *al.* is limited to block ciphers.

In this work, we explore the application of masked instructions for PQC. To be able to demonstrate their versatility, we do not limit our evaluation to a single scheme. On the other hand, developing masked implementations for multiple complete PQC schemes is not feasible within the scope of this work. Therefore, we opt to evaluate masked instructions

Table 1: Masked implementations presented and evaluated in this work. A green check mark indicates an implementation that completely originates from this work. The orange check marks (with brackets) indicate implementations that are based on the public code by Krausz et *al.* [KLRBG23]. Dashes indicate combinations that are not applicable.

Application	Scheme	Algorithm	Representation	
			standard	bitsliced
Fixed-Weight Polynomial Sampling	each applicable to BIKE, HQC, NTRU, and many on-ramp signatures	Fisher-Yates	✔	(✔)
		Sorting	✔	(✔)
		Rejection	✔	(✔)
		Comparison	✔	(✔)
		I2C Conversion	✔	(✔)
Gaussian Sampling	FrodoKEM	CDT Sampling	✔	✔
	Haetae	CDT Sampling	✔	✔
	Hawk	CDT Sampling	—	✔
Polynomial Rotation	BIKE	Barrel Shifter	✔	—
Hashing	all PQC	Keccak	✔	—

based on several components that are used in a wide range of PQC schemes. We select three components that have been already targeted by side-channel attacks and are very costly to implement with masking without hardware acceleration. Besides the already mentioned fixed-weight sampling, this includes discrete Gaussian sampling based on Cumulative Distribution Tables (CDTs), and polynomial rotation by a secret amount. Furthermore, we implement a masked Keccak, which is used within SHA-3 and SHAKE, both of which are deployed in the majority of relevant PQC schemes.

We identify a small set of performance-critical core operations, relevant for most of the previously mentioned use cases. These operations are the conditional move (cmov) and integer comparisons (cmp) with Boolean shared operands. We extend the core by Gao et *al.* [GGM+21] with corresponding instructions and evaluate (1) the impact of the masked instructions from the original core alone, and (2) our additional instructions.

The general purpose nature of the masked instructions evaluated in this work makes them useful for any cryptographic primitive or scheme and leads to great crypto-agility: Instead of accelerating only one or a small subset of the relevant PQC schemes, they are able to significantly improve the performance of a large quantity of quantum-secure, asymmetric cryptography on top of symmetric cryptography.

Contribution. To summarize the contributions of this work, we provide masked software implementations for a wide set (see Table 1) of algorithms used in PQC in multiple variants, to evaluate the impact of masked instructions as proposed by Gao et al. [GGM+21] beyond their application for symmetric cryptography. To allow a fair and in-depth comparison we develop or adapt existing bitsliced implementations, besides our non-bitsliced variants. The latter profit significantly more from masked instructions. Nevertheless, we are able to reach speed-up factors over an order of magnitude. To the best of our knowledge, we provide the first masked software implementation for CDT samplers and syndrome rotation in BIKE.

Moreover, we identify core operations (cmov, cmp) required in multiple critical PQC components and extend the RISC-V core presented in [GGM+21] with masked instructions for these operations. Furthermore, we discover and fix a flaw in the randomness generation implemented by Gao et *al.*, which caused power leakage. Underlining the necessity of a masked syndrome rotation, we confirm the power SCA attack on BIKE presented in very recent work [CARG23] with our own measurements. We validate the side-channel security

of our implementations by performing t-test evaluations on our additional instructions and an exemplary sampling algorithm. Finally, we perform extensive benchmarks with multiple parameter sets and Instruction Set Architecture (ISA) constellations to evaluate the performance impact of masked instructions on PQC. To facilitate adaption and future research, our source code for software and hardware will be publicly available.

Related Work. Gao et *al.* [GGM+21] present an ISE for RISC-V featuring power side-channel secure (first masking order) instructions operating on shared values. The ISE consists of instructions for Boolean and arithmetic operations in the Boolean and arithmetic masking domain, conversions between the two domains and arithmetic operations in finite fields. They implement the masked instructions in the SCARV CPU, a 5-stage pipelined microprocessor, and evaluate it with common block ciphers. Marshall et *al.* [MP21] investigate how higher masking orders can be realized, utilizing RISC-V's vector ISE. Cheng et *al.* [CP23] also work with an ISE to address power side-channel leakage, but instead of accelerating instructions by computing over multiple shares, they only focus on the microarchitectural leakage aspect.

Accelerators for masked Kyber and Saber implementations have been presented by Fritzmann et *al.* [FBR+22]. In contrast to our work, they provide masked accelerators for higher-level functionalities. Nevertheless, they include, e.g., a masked adder for Boolean shared values.

2 Applications

In this work, we focus on three different PQC components: fixed-weight polynomial sampling, CDT sampling, and syndrome rotation. All three components have in common that unprotected implementations are susceptible to SCA as shown in prior work and demonstrated in this section. The next commonality is that masked software implementations for these algorithms unfortunately suffer from extraordinary performance overheads. Fortunately, they can all significantly be accelerated by the same small set of core operations. Additionally, we examine Keccak, which is used in the majority of PQC schemes.

2.1 Fixed-Weight Polynomial Sampling

A wide range of PQC schemes requires fixed-weight polynomial samplers to randomly generate a binary or ternary polynomial with a fixed length N and a fixed number of non-zero coefficients (weight) W from a uniform distribution.

Algorithms. The known algorithms for this problem can be divided into three categories. The index rejection method and its bounded variant [ND22] sample the non-zero coefficients with rejection sampling. Another approach aims at generating random bitstrings, this direction is followed by the Comparison method [KLRBG23]. Lastly, two algorithms start with fixed polynomials with the correct length N and weight W and then apply shuffling with a Fisher-Yates variant [Sen21] or sorting [Ber22]. Notably, these algorithms generate polynomials in one of two possible representations: the index representation stores indices of the non-zero coefficients whereas the coefficient representation stores the plain coefficients. The index representation allows for a denser representation of polynomials with a low W/N ratio. Ultimately, the implementation of the operations on the polynomial following the sampling process determine which representation is required. For most operations, the coefficient representation is required. A masked conversion from index to coefficient representation and vice versa is straightforward but costly.

Applications. Fixed-weight polynomial sampling is used in BIKE, HQC, McEliece, NTRU, Streamlined NTRU Prime, and NTRU LPRrime, well-known from NIST's standardization process, and also SMAUG [CCHY23], which was recently submitted to the Korean Post-Quantum Cryptography Competition. Moreover, several schemes for NIST's signature on-ramp call feature fixed-weight-sampling procedures (CROSS, EagleSign, Raccoon, ALTEQ) or random shuffling functions (FuLeeca, LESS). A recent paper [BCMP24] about sampling random permutations in PQC presents further applications. While requirements and parameters slightly differ between the schemes, the sampling problem at its heart is the same. Timing side-channel attacks on the samplers of BIKE and HQC have already been demonstrated [KAA21, GHJ+22, Sen21] and without any dedicated protection, similar attacks could be performed based on information gained from power side-channel attacks.

Masking. A recent work [KLRBG23] compares the performance of all known fixed-weight polynomial sampling algorithms as masked software implementations. The benchmarks show that the performance of the algorithms clearly varies depending on the parameters N and W, and the preference for different algorithms depends on the individual use case. However, all reported runtimes are far from satisfying. The situation is worst for BIKE and HQC, where the sampling is required during decapsulation, which imposes additional requirements. The most efficient algorithms cannot be applied here for a side-channel secure implementation, because they inherently leak the secret seed for the Pseudorandom Number Generator (PRNG) via their timing behavior (cf. [GHJ+22]). For these cases, the most efficient algorithms require millions to hundreds of millions of clock cycles on a Cortex-M4, depending on the polynomial representation form. An efficient approach for a masked implementation has been presented earlier [CGTZ23], but as [KLRBG23] already points out, it is likely susceptible to a horizontal SCA [BCPZ16].

The most performance-critical operation for almost all sampling algorithms is either a masked conditional move or a masked integer comparison. In Section 3.2, we explain why exactly these operations are so crucial.

2.2 Sampling from a Discrete Gaussian Distribution

As mentioned above, discrete Gaussian sampling is essential for many lattice-based cryptographic schemes [DDLL13, BCD+16, GPV08]. Hence, efficient and secure implementations of these sampling algorithms are important for reliable modern cryptography. In order to discuss and highlight crucial parts, we denote the discrete Gaussian distribution with variance σ^2 as $\mathcal{N}_{\mathbb{Z}}(\sigma^2)$. Then, $\mathcal{N}^+_{\mathbb{Z}}(\sigma^2)$ is defined to have the same distribution as $\mathcal{N}_{\mathbb{Z}}(\sigma^2)$ for all samples in $\mathbb{N}$, half the probability for sampling zero and probability zero for sampling negative values.

Algorithm. There are multiple well-studied algorithms to perform sampling from discrete Gaussian distribution [DN12, BCG+14, DDLL13]. The most common approach is CDT sampling [Pei10], where a uniform random bit string is compared with the entries in a pre-computed Cumulative Distribution Table. This CDT approximates $\mathcal{N}^+_{\mathbb{Z}}(\sigma^2)$ for a given precision and σ^2. For a random bit string r, the output sample is the index i in the table T, for which $T_i < r \geq T_{i+1}$. The optimal way to achieve this in terms of memory access and number of comparisons is an adapted binary search that takes into account the distribution itself. Unfortunately, this opens a (cache-)timing side channel, because the memory access pattern would depend on the sampled value. Thus, to avoid branches and secret-depending memory accesses, the random bit string must be compared against each entry in the table. Then, the comparison result bits are accumulated to obtain the sample, which is mapped to $\mathcal{N}_{\mathbb{Z}}(\sigma^2)$ by conditional negation with another random bit.

Applications. The most notable application of CDT sampling is FrodoKEM, where it is used for sampling error terms during key generation, encapsulation and decapsulation. In particular, the procedure is performed deterministically during decapsulation in the re-encryption step, because of the Fujisaki-Okamoto transform. Obtaining (side-channel) information about these errors may result in a recovery of the shared secret or the private key. Notably, a practical attack on FrodoKEM was shown recently [MKK+23]. Thus, the sampling operation requires masking to comprehensively secure FrodoKEM.

Another important example is Hawk [DPPvW22], a PQC signature scheme based on the newly proposed lattice isomorphism problem. In contrast to prior work, it does not require floating-point arithmetic which would prohibit a masked implementation. Furthermore, the authors claim that the only missing part to enable a fully masked implementation is sampling discrete Gaussian variates from a CDT.

Finally, the recently presented scheme Haetae [CCD+23], which is similar to Dilithium, samples the signature nonce $\mathbf{y}$ from a Gaussian distribution. Again, CDT sampling is deployed for that purpose. For a high-assurance version of its deterministic variant, this sampler must be protected accordingly using masking countermeasures.

Squirrels is another scheme in NIST's signature on-ramp, that features CDT sampling.

Masking. The basic operation within the sampler is the comparison of a *secret bit string* with a *public constant*. A standard software approach to achieve this is bitslicing the inputs and then using the optimized comparison as described in [KLRBG23]. However, bitslicing transformations also cause an overhead and may lead to additional microarchitectural leakage. A non-bitsliced implementation with hardened instructions may yield better overall results. For a masked comparison, masked subtraction and shift operations as provided by [GGM+21] are sufficient. However, a dedicated masked greater-than comparison instruction could further improve the performance while inducing only a miniscule area overhead in the core.

2.3 BIKE Syndrome Rotation

Our third example is related to the power side-channel resistance of BIKE, which is one of the three remaining Key Encapsulation Mechanism (KEM) candidates in NIST's PQC standardization process. In contrast to the numerous lattice-based schemes, BIKE's security foundation relies on linear error-correcting codes. Just recently, Cheriere et *al.* [CARG23] demonstrated a power side-channel attack on the optimized Cortex-M4 microcontroller implementation presented at CHES 2021 by Chen, Chou, and Krausz [CCK21]. Independently, we developed the same attack for the portable software implementation of BIKE [DGK] (see Appendix A).

The decapsulation in BIKE is conducted by the iterative Black-Grey-Flip (BGF) decoder [DGK20] which rotates the syndrome of the underlying linear error-correcting code by the secret indices of the private key. To avoid timing side-channel attacks, a secure algorithm always loads the rotated and unrotated polynomial from the memory (cf. Algorithm 4). The decision which of those two are stored – and therefore represent the correct result – is made based on a mask m which either is set to `0xFFFFFFFF` or `0x00000000`. Due to the significant difference between the two valid values for m, they can be easily distinguished in the power consumption of a device executing the algorithm. Therefore, an attacker can learn major parts of all secret indices of the private key, since the mask m is directly connected to the bits representing the indices.

Masking. The aforementioned procedure is required to achieve a constant-time implementation and masking all relevant parts is straightforward – apart from introducing a huge implementation overhead. More precisely, as described above, the crucial operation

that generates the leakage is the *conditional move* controlled by the mask m. To this end, first-order masking of the syndrome would double the implementation's memory footprint and thus double the cost for loads and stores for this already memory instruction intense operation. A masked implementation should therefore focus on improving the performance of a conditional move instruction.

2.4 Hashing

Hashing, or related tasks like deterministic randomness expansion as a Pseudo-Random Function (PRF) or an Extendable-output Function (XOF), are part of nearly all PQC schemes. For the NIST standardization efforts, it is required to use NIST-standardized algorithms for these tasks, and the overwhelming majority of submissions deploy SHA-3 and SHAKE variations. The basic permutation function behind these is Keccak.

Often, these tasks process or generate secret data that needs to be protected against power SCA. Apart from that, hashing, PRFs, and XOFs have a major influence on the overall performance of the execution of PQC schemes. This is especially the case for embedded devices, as shown impressively by the "pqm4" benchmarks [KRSS19], where hashing takes up to 75 % of the clock cycles for relevant procedures. For masked implementations, it is to be expected that this percentage even increases for some schemes (e.g., Haetae), as Keccak involves many non-linear operations.

Algorithm. Keccak-f, the underlying function for all SHA-3 and SHAKE variations, consists of 24 permutation rounds, each of which performs the five steps θ, ρ, π, χ, and ι on the 1 600-bit state. The overall permutation is utilized within a so-called sponge construction, which allows the design of different functionalities, such as hash functions, where an arbitrary-length input generates a fixed-length output, or XOFs, where an arbitrary-length input is transformed into an arbitrary-length output. Consequently, the only difference between the single SHA-3 and SHAKE variations is

- a domain separation byte, and
- the size of the state's part that the input is mixed into and that the output will be taken from.

Applications. Keccak is used (1) in all post-quantum schemes already selected by NIST for standardization, (2) the fourth-round KEM candidates, and (3) almost all schemes submitted to the additional signature call. Consequently, the importance of secure and efficient implementations cannot be overstated. Moreover, as highlighted above, many schemes have Keccak as a main contributor to their latency. For instance, the signature generation in the reference implementation of Haetae reportedly spends around 60 % with Keccak [CCD+23].

Masking. The first three steps of each permutation round (θ, ρ, π) are linear in the Boolean sharing domain, and the ι step is an affine function. Thus, these steps can be masked trivially, either by applying the steps share-wise for linear functions, or only to the first share for ι. The crux of masking Keccak is the χ step, which is a quadratic mapping that requires special care.

3 Software Implementation

We develop efficient C code for all applications to evaluate the hardware accelerators. Table 1 provides an overview of the algorithms implemented in this work. We implement

four different fixed-weight polynomial sampling algorithms and one conversion algorithm, adapt existing bitsliced variants, and develop secure CDT samplers for FrodoKEM, Haetae and Hawk, as well as a rotation function for BIKE. Additionally, we develop a masked implementation of Keccak. Our code works for arbitrary masking orders, however, the masked instructions on the core only support first-order masking.

Bitslicing. For highspeed symmetric cryptographic software implementations, bitslicing [Bih97] has been an important technique for many years. The general idea is to reach better resource utilization by using the full register width although operating on values bounded by fewer bits – in particular when operating on single bits. Therefore, data has to be transformed to the bitsliced representation. This operation corresponds to a matrix transposition. For example, given a register width of 32 bit, in a bitsliced implementation 32 10-bit values are not stored in 32 registers, instead the i-th bit of each value is aggregated in one register. Therefore, the values are stored in ten registers in total. Operations can then be done on 32 values in parallel, always using the full register width. In theory, a speedup of a factor equal to the register width can be achieved for operations on single-digit values, whenever parallelization is possible. However, transformations from and to the bitsliced representation and sometimes padding introduce overhead.

Bitslicing is also a very efficient technique to optimize *masked* software implementations because it can reduce the number of costly non-linear operations significantly [BC22, KLRBG23]. However, for hardware accelerators that execute a masked instruction in a single or only few cycles, the overhead introduced by bitslicing can be unprofitable in some cases. To allow a fair comparison, we provide standard (non-bitsliced) and bitsliced implementations for all algorithms if applicable.

Masked Instructions. We implement wrapper functions with inline assembly utilizing the masked instructions. Via preprocessor directives, we are able to determine whether our code is compiled (1) without any masked instructions, (2) with the Boolean and arithmetic, masked instructions developed by Gao et *al.* [GGM+21], or (3) additionally with the instructions we deem useful for PQC.

From the masked instructions by Gao et *al.*, only a subset is used in our code, all of them operate on Boolean shares: mask, unmask, not, and, or, xor, slli, srli, add and sub. The rest of the instructions – e.g., instructions to switch from Boolean to arithmetic masking domain and vice versa – do not apply. Without masked instructions, all masked operations are realized by secure gadgets presented by [BC22, KLRBG23], which rely on unmasked instructions. We implement some inner loops with inline assembly to have more control over the memory and register usage. This allows us to use the benefit of reduced register pressure, which comes with the masked instructions leading to fewer memory loads and stores. Nevertheless, the parametrizability of our high-level C code (arbitrary masking order, parameters N and W and different ISEs) leads to some unnecessary memory accesses, which could be avoided with implementations optimized towards a specific parameter set.

3.1 Additional Masked Instructions

As already mentioned before, during our work we identified a small set of repeatedly occurring operations. For a branch-less, secret-dependent selection of different values, timing side-channel secure implementations rely on dedicated select or conditional move instructions, if available on the architecture. A corresponding *masked* cmov can be applied at performance critical locations for multiple fixed-weight polynomial sampling algorithms, all CDT samplers and is the most relevant operation for a polynomial rotation. Two other common general purpose instructions are integer comparison. Many sampling algorithms

repeatedly check if a random value exceeds a threshold (cmpgt), or if two values collide (cmpeq).

We therefore investigate, if dedicated masked instructions for these operations are worthwhile. They clearly do not provide an advantage for the block ciphers implemented by Gao et *al.*, but are potentially useful for asymmetric cryptography beyond the applications presented in this paper.

To allow a fair evaluation of the benefit of the additional instructions, we examine in detail how the operations can alternatively be computed with only the masked instructions by Gao et *al.* and without any masked instructions.

cmov. Given a mask c of zeros, if the condition is false and a mask of ones if the condition is true, we can implement a conditional move of b to a with only three Boolean operations. We compute $tmp := b \oplus a$, followed by $tmp := tmp \wedge c$ and $a := a \oplus tmp$ leading to $a := a \oplus b \oplus a = b$, if the condition bits are set and $a := a \oplus 0 = a$, if not. All of these sub-operations must be side-channel secure and can be realized with their respective masked instructions by Gao et *al.* if available.

cmpeq. For the base ISA, we use the operation sequence depicted in Algorithm 1, to express a cmpeq. As can be seen there, the performance will be dominated by the five secure or operations. In contrast to this, we use a dedicated operation sequence whenever masked, Boolean instructions are available. In this case, we compute the first operand minus the second one and vice versa, combining the two results with a secure or, which is shown in Algorithm 2. Then, the most significant bit is set, if the operands are *not* equal. To obtain the single-bit result, we shift this value right by the offset 31 using the `b_srli` instruction.

Algorithm 1 cmpeq with base ISA. All Boolean operations are performed with secure gadgets.

Input: Boolean-shared values a, b
Output: Boolean-shared res, 1 if a and b are equal
res $:= a \oplus b$
res $:=$ res $\vee (res \gg 16)$
res $:=$ res $\vee (res \gg 8)$
res $:=$ res $\vee (res \gg 4)$
res $:=$ res $\vee (res \gg 2)$
return (res $\vee (res \gg 1)) \oplus 1$

Algorithm 2 cmpeq with ISE by Gao et *al.* All arithmetic operations are performed with the Boolean-masked instructions.

Input: Boolean-shared values a, b
Output: Boolean-shared res, 1 if a and b are equal
$\text{tmp}_0 := a - b$
$\text{tmp}_1 := b - a$
res $:= \text{tmp}_0 \vee \text{tmp}_1$
return (res $\gg 31) \oplus 1$

cmpgt. We apply a similar handling for the cmpgt operation. For the base ISA, we adapt the optimized comparison from [KLRBG23] to the standard representation. However, this is very slow and usually yields impractical results, which is why a bitsliced implementation is preferable when no ISE is available. For the evaluation case in which a dedicated masked cmpgt instruction is not available, but masked subtraction and shift instructions are, we can express the comparison efficiently with a subtraction followed by a shift to the right to get the most significant bit. This is only correct when both operands can be represented with at most 31 bit, or in other words, when the most significant bit of both operands is zero. Due to the possible parameter ranges for N and W, this is always the case in our applications.

3.2 Masked Fixed-Weight Polynomial Sampling

In addition to implementing non-bitsliced variants of the sampling algorithms, we adapted the bitsliced implementations from [KLRBG23] so that they can be compiled with masked instructions to enable a fair comparison. In the following, we briefly summarize the algorithms presented in [KLRBG23] and discuss their key instructions.

The side-channel secure Fisher-Yates method has a quadratic runtime in the weight W. In the inner loop, a masked cmov based on the result of a masked cmpeq is executed. Nevertheless, for the standard implementation, we use bitslicing for another single loop, because it requires a 48-bit transformation from the Boolean to the arithmetic domain and vice versa. Although the core by Gao et *al.* has dedicated instructions for these conversions, they only work for 32-bit operands. In contrast, the gadget implemented by Bronchain and Cassiers [BC22] allows (with minor modifications) efficient bitsliced conversions for this size, and thus the costs are amortized.

Constant-time sorting can be done with sorting networks. The bitsliced sorting algorithm in [KLRBG23] is based on Batchers's Bitonic mergesort, which is easy to parallelize. For the standard implementation, we opted for djb-sort [BCLv17] instead. Both sorting algorithms have an asymptotic runtime of $\mathcal{O}(N \log^2(N))$, where N in our case is the polynomial length, but djb-sort is capable of reaching a slightly lower constant factor. The fundamental operation in a sorting network is a cmpgt followed by a conditional swap (i.e., two cmov). Although the values to be swapped during sorting cover the full 32-bit registers, only the upper 30 bits consist of the random values that are compared. By shifting the values to the right before the cmpgt, we can use the optimized operation sequence based on subtraction.

Rejection sampling requires a cmpgt to verify whether a sampled bitstring represents a valid index below N. Additionally, a loop over the already sampled indices checks for collisions using cmpeq. We did not implement and evaluate the bounded rejection sampling method, because the Fisher-Yates approach is strictly superior as already pointed out by Krausz et *al.* [KLRBG23].

Comparison sampling sets each coefficient bit individually with probability $p = W/N$. This is implemented by comparing a random l-bit string with the constant $\lfloor 2^l p \rceil$ and setting the coefficient to one if the random value is below the threshold. This obviously heavily relies on a cmpgt operation.

Both the Fisher-Yates shuffle and rejection sampling generate polynomials in the index representations. In most cases, however, the coefficient representation is required. The conversion is straightforward: we start from a polynomial in coefficient representation where all coefficients are set to zero. Then, a public counter i runs from 0 to $N-1$ and in each iteration, we check whether i is equal to each coefficient in the list of indices. Depending on this cmpeq result, we cmov a one into the position i in the result polynomial. Thus, the conversion performs exactly $N \cdot W$ cmpeq and cmov.

3.3 Masked CDT Sampler

To the best of our knowledge, we are the first to report masked CDT samplers. The most important masked operations are integer greater-than comparison and bit accumulation. For our bitsliced variant, we implement the optimized comparison with a public constant t from [KLRBG23], which only uses $\lceil \log_2 t \rceil$ secure and operations. Bit accumulation can be performed with half adders, adapting the masked implementation from Bronchain and Cassiers [BC22]. This requires up to $\lceil \log_2 |T| \rceil$ secure and operations, where $|T|$ is the number of entries in the CDT.

Our non-bitsliced implementation varies depending on the masked instructions available. Similar to our fixed-weight polynomial sampling algorithms, the operands for the greater-than comparison are bounded (e.g., 15 bits in the case of FrodoKEM), which allows us to

use different optimal operation sequences. The comparison results are accumulated with a masked addition. Thus, both operations are executed $|T|$ times for each sample and dominate the computation time. Note that for the case of Hawk, the CDT entries have a size of 80 bit, which renders a non-bitsliced implementation infeasible.

3.4 Masked BIKE Rotation

A naïve implementation of the polynomial rotation with word granularity in BIKE could be structured very similarly to Algorithm 4. However, this would lead to many unnecessary loads and stores. For example, when the array is rotated by one, the naïve implementation would begin by loading the first and second entries, conditionally move the second to the first and then store both entries back. Then, the same procedure is done with the second and third entries; therefore, most entries are loaded and stored two times. Following the optimized, unmasked BIKE implementation for the Cortex-M4 [CCK21], the memory accesses can be reduced to one per entry by operating on chains of values and thus keeping an entry in the registers during the cmov to the next entry and the cmov from the previous entry. The rotation obviously relies heavily on the masked cmov.

Bitslicing is not applicable here, because the conditional move already operates on the full register width. Our implementation expects a pointer to three consecutive instances of the masked polynomial. Three instances are required because the number of registers for one polynomial, which is the maximum secret rotation amount r, is not a power of two. As the side-channel secure rotation works bitwise, it covers rotation amounts up to $2^{\lceil \log_2 r \rceil} - 1 > r$, and thus, extends over the second copy. Apart from $\lceil \log_2 r \rceil$ masked shift operations that are required to select the desired bit for each round, the rotation mainly consists of $\lceil \log_2 r \rceil r$ masked cmov operations.

3.5 Masked Keccak

As already explained before, Keccak consists of three linear steps, one affine step, and a quadratic step. For the linear and affine steps, the masked xor instructions are expected to yield a moderate speed-up, while the biggest contribution to an overall acceleration will come from the masked and instructions that speed up the quadratic χ step. Note, however, that in contrast to the other use cases, Keccak does not profit from the additional cmov and cmp instructions.

4 Hardware Implementation

Our hardware implementation is based on the SCARV CPU[1], an embedded class processor implementing the RISC-V 32-bit base ISA, as well as the multiply and compressed instruction set extensions. Internally, it employs a 5-stage pipeline with forwarding paths to improve performance, but does not feature more advanced features such as branch prediction or a cache subsystem.

The ISE for masking by Gao et *al.* is tightly integrated into the core without the need to manage separate register banks, but at the cost of additional engineering effort on the hardware level. They provide Boolean (and, or, ...) and arithmetic (add, sub, ...) instructions for operands masked in the Boolean ($a = a_0 \oplus a_1$, where $\oplus$ represents the bitwise addition mod2) and arithmetic domain ($a = a_0 + a_1 \bmod 2^{32}$) as well as conversion between the two domains. The Arithmetic Logic Unit (ALU) is extended with a *masked* ALU module, which implements all necessary primitives. The shares are stored in the general-purpose registers, allowing seamless integration into existing RISC-V code. However, to enable the CPU to load both shares during the register fetch phase, they

[1] https://github.com/scarv

Algorithm 3 cmov Implementation

Data: masked 32-bit values dest, new, cond
Result: new is moved into dest if the least significant bit in cond is 1

function CMOV(dest,new,cond)
  $a \leftarrow$ dest $\oplus$ new ▷ share-wise
  $b \leftarrow a \wedge$ cond ▷ using secure and
  return $b \oplus$ dest ▷ share-wise
end function

must be stored in adjacent registers. Additional data paths and logic were added to the forwarding and hazard unit to account for the shares. Furthermore, one of the shares is stored in bit-reversed form preventing accidental combination in other pipeline stages. The original representation is only restored inside the masked ALU.

We add three new instructions for the *dedicated* acceleration of the frequent core operations that we identified in the PQC applications. This requires us to modify the masked ALU as well as the data paths inside the processor.

Randomness Generation Flaw. During the implementation and subsequent testing of our extension, we detected leakage when using both existing and our new instructions. We found that the authors used the whole state of two 32-bit Linear-Feedback Shift Registers (LFSRs) to generate 64 bits of randomness. However, it has been shown that this LFSR configuration is not ideal and can, in fact, create detectable leakage [CMM+23]. For our testing and evaluation purposes, we changed the randomness generation to use a unique LFSR for each pseudo-randomly generated bit, instead of employing the whole state.

General Architecture. RISC-V instructions support up to two source operands and one destination operand. In the SCARV CPU, the destination operand only acts as a pointer to a register which is utilized in the writeback stage. However, we require three operands to implement the cmov instruction. More specifically, we have to use the value of the destination register as a third input to be able to hide the condition. Thus, we have to change the register module to allow the transfer of three operands in their shared representation during the decode and register fetch phase. The decoder will fetch the destination register's value whenever a cmov instruction is detected. We modify the forwarding and hazard unit to account for a third operand and extend the data path to the CPU, however, the third operand is only routed into the masked ALU. Accidental share combinations are again prevented by loading and storing a bit-reversed representation.

Masked ALU. Our new instructions cmov and cmpeq are implemented as additional units, while cmpgt is an extension to an existing unit inside the masked ALU. The cmov module receives three input operands in their shared representation: the destination register's contents, a value and a 1-bit condition, which decides whether the value is moved into the destination register or not. Internally, the 1-bit condition is expanded to 32-bits. The result is computed according to Algorithm 3.

Notably, the cmov operation completes within a single clock cycle, as we reuse the Domain-Oriented Masking (DOM) and gadget provided by Gao et *al.*, which latches its input on the positive edge and adds randomness on the negative edge.

The cmpeq instruction generates a 1-bit condition based on the equality of two input operands and is implemented as its own subunit inside the masked ALU. It checks for equality by first xoring both operands and afterwards, the result is propagated through

an or-tree, which at each stage computes the inclusive or of adjacent bits within the intermediate result. Thus, each stage compresses the input by half of its bit length. Therefore, a 32-bit input will be compressed to just a single bit after five stages. If the inputs are equal, the input to the or-tree will be zero and the final output will be zero. Vice versa, if they are unequal, the input will at least contain one digit which is not zero and the final output will be one. We again reuse the DOM and gadget to build the required inclusive or gate. Because the protected and gate can compute its result in just one cycle, the overall latency of this instruction is five cycles.

Lastly, the cmpgt operation compares two input values and sets a 1-bit condition based on which of the two inputs was greater. Unlike the previous instructions, this operation is implemented as a minor extension to the existing bit arithmetic module, which provides masked addition as well as subtraction over Boolean shares. When subtracting a larger value from a smaller value, a carry bit is generated, which is currently unused and not extracted from the Kogge-Stone adder present in the design. Therefore, we modify the adder to output the carry bit in its shared representation, which acts as our 1-bit condition. The adder computes the result of a 32-bit subtraction within six cycles, thus cmpgt exhibits the same performance overhead.

5 Evaluation

We evaluate our modified SCARV Core using the ChipWhisperer CW305 FPGA board, which features an Artix `xc7a100tftg256` FPGA as well as connections to supply power from a separate power supply and to read out the power consumption. Furthermore, we use the SCARV SoC project to connect the Central Processing Unit (CPU) to an Advanced extensible Interface (AXI) providing access to memory, peripherals such as serial and General Purpose Input/Output (GPIO), thus allowing us to send inputs to the system as well as setting up precise triggers for our measurements. The CPU has access to 256 kB of shared RAM/ROM and runs at 25 MHz. In the following, we present our results regarding hardware overhead, software performance and side-channel measurements.

Table 2: Hardware performance results. Note that for the Field-Programmable Gate Array (FPGA) results cover the whole core, whereas the Application-Specific Integrated Circuit (ASIC) numbers only feature the ALU.

(a) FPGA performance results.

		Area			**Freq.**
		[LUTs]	[FFs]	[Slices]	[MHz]
Masked ISE	Bool.,arith.	7203	2670	2324	29.29
	cmov	128	64	52	
	cmpeq	127	116	81	
	cmpgt	64	8	35	
	other	655	64	137	
	combined	8177	2922	2629	26.93
	Overhead	13.5 %	9.4 %	13.1 %	-8.1 %

(b) Hardware overhead of our modified ALU using 45 nm technology.

	Bool.,arith.	all
cmpeq	—	891.09 GE
cmov	—	836.12 GE
Add/Sub	2582.24 GE	2627.02 GE
Other Masking	6265.88 GE	6265.88 GE
Other	729.17 GE	834.28 GE
Total Area	9577.31 GE	11454.40 GE

5.1 Hardware

Table 2a shows that the hardware overhead of the additional instructions is moderate. Overall, there is an increase of approximately 13 % in combinatorial logic and 10 % in registers. For the maximum frequencies, we note that we did not perform iterative

synthesizing to optimize the frequency, but rather set the target to 25 MHz. The maximum frequency from Table 2a is then computed from the reported worst negative slack.

We note that SCARV is a very simple core, without a cache subsystem or floating point unit. Gao et *al.* report a frequency drop of 5 MHz as well as a 80 % LUT and a 25 % FF overhead for their masked ISE. Thus, when adding our extension on top of their modified core, the overhead is overall moderate.

The area costs in Table 2a that are not directly associated with one of our three instructions, but rather introduced as *other*, mainly come from the extended data path for a third operand, which is necessary for our cmov instruction.

Furthermore, we determined the area overhead created by our modified masked ALU for an ASIC implementation by synthesizing it using Synopsys Design Compiler and Silvaco's Open-Cell 45nm FreePDK. The results are shown in Table 2b. In general, the extended ALU features 1877.09 additional gates, which are mainly comprised of the cmpeq and cmov modules. The cmpgt instruction only requires minimal modifications to the subtraction circuit.

5.2 Software

Figures 3, 4, 5a and 5b show our software benchmark results for the four different applications. Note that for each use case, the generation of input randomness is not benchmarked, and may take another significant part of the cycle counts. This is especially the case if a masked Keccak is used (e.g., as XOF).

Fix-weight Sampling. Table 3 shows our benchmark results for all fix-weight sampling algorithms applied to a representative selection of parameter sets such that our results can be generalized to other parameter sets easily. For our selection, we choose polynomial sizes (N) from each relevant order of magnitude, and consider different W/N ratios. Moreover, we include use cases from key generation, encapsulation, and decapsulation, which all have different requirements, as pointed out in more details in [KLRBG23]. Particularly, for the BIKE decapsulation, the last parameter set, only Fisher-Yates and sorting are secure algorithms, because their runtime is independent of the randomness, which is extended from a secret value in this case. This counters a recent attack [GHJ+22], and our implementation results can be generalized to HQC decapsulation as well. In general, our results show that bitslicing improves the performance for all algorithms significantly, when no accelerators are available.

Fisher-Yates and rejection sampling generate polynomials in index representation, sorting and the comparison method generate polynomials in coefficient representation. In most applications, subsequent operations on the polynomials require the coefficient representation. For these applications, comparison sampling is superior, when implementing for the base ISA, confirming the results from Krausz et *al.* [KLRBG23]. When index representation is required, rejection sampling is clearly faster. For the fourth use case, where N is 12323, rejection sampling combined with the conversion is even faster than the comparison method.

Because sorting requires more than the available 256 kB of RAM, for the BIKE decapsulation use case only Fisher-Yates remains as a suitable algorithm. To get the required coefficient representation, the costs of the index-to-coefficient conversion lead to unpractical cycle counts of over 360 million cycles.

The picture changes distinctly when we utilize masked instructions. With the ISE, the non-bitsliced standard implementations are faster for Fisher-Yates, sorting and rejection sampling, as they get accelerated in many cases by a factor of 30.

Due to the speed-up of more than an order of magnitude, sorting becomes fastest for small N. With greater values for N, comparison sampling is again faster. For the index representation, rejection sampling remains the preferable method. The advantage of the

Table 3: Masked fixed-weight polynomial sampling performance with different parameters and ISEs in kilo cycles. For each parameter set, we underline the cycle count – for masked ISA "Bool., arith.", which represents the core by Gao et *al.*, and "all" separately – of the fastest algorithm with output in coefficient representation. For Fisher-Yates and Rejection sampling, we also take the index-to-coefficient (I2C) conversion into account. Sorting requires too much memory with the BIKE decapsulation parameters on our evaluation platform, and other options are not available because of existing timing attacks (cf. [KLRBG23] for an extensive discussion on applicable algorithms).

Parameter set	Algorithm	Poly. Format	Bitsliced		Standard		
			Base ISA	Masked ISE	Base ISA	Masked ISE	
				Bool., arith.		Bool., arith.	all
NTRU key gen. $N = 677$ $W = 254$	Fisher-Yates	index	6 876	6 455	44 729	2 059	1 674
	Sorting	coeff.	13 678	4 569	25 017	<u>944</u>	<u>850</u>
	Rejection	index	1 214	909	4 144	156	138
	Comparison	coeff.	2 243	1 010	73 289	1 137	1 105
	I2C Conv.	coeff.	5 605	4 273	236 071	9 312	5 871
sNTRU Prime key gen. $N = 953$ $W = 396$	Fisher-Yates	index	12 112	11 108	108 352	4 490	3 551
	Sorting	coeff.	19 687	6 509	38 289	<u>1 433</u>	<u>1 288</u>
	Rejection	index	2 660	1 927	6 741	247	231
	Comparison	coeff.	4 791	2 124	147 623	2 305	2 260
	I2C Conv.	coeff.	11 186	8 348	517 980	20 415	12 866
McEliece encaps. $N = 6688$ $W = 128$	Fisher-Yates	index	3 244	3 082	11 474	680	583
	Sorting	coeff.	217 048	66 040	446 669	16 191	14 479
	Rejection	index	402	311	2 073	77	70
	Comparison	coeff.	24 173	<u>9 705</u>	661 901	10 426	10 223
	I2C Conv.	coeff.	27 020	18 765	1 175 949	46 506	29 353
BIKE key gen. $N = 12323$ $W = 71$	Fisher-Yates	index	1 854	1 762	3 651	351	321
	Sorting	coeff.	460 872	136 838	941 111	33 854	30 231
	Rejection	index	187	152	1 554	58	52
	Comparison	coeff.	34 208	<u>13 269</u>	899 499	14 164	13 908
	I2C Conv.	coeff.	29 568	20 529	1 203 042	47 761	30 203
BIKE decaps. $N = 49318$ $W = 199$	Fisher-Yates	index	5 990	<u>5 518</u>	27 594	1 431	<u>1 195</u>
	Sorting	coeff.	—	—	—	—	—
	I2C Conv.	coeff.	360 683	<u>243 527</u>	13 475 741	532 081	335 510

additional masked instructions compared to the plain Boolean and arithmetic instructions is in the range of about 10 to 20 percent.

Although Fisher-Yates is more than five times faster with the ISE compared to the bitsliced variant for the base ISA, the combined costs (including I2C conversion) remain very high at more than 240 million cycles, for the BIKE decapsulation application. With more RAM available to be able to run the sorting method, it is expected to be faster than Fisher-Yates plus the I2C conversion. Still, the estimated costs of more than 100 million cycles, would not be practical.

CDT Sampling. Our results for the CDT sampler of FrodoKEM, Haetae, and Hawk can be found in Table 4. Most importantly, the base ISA numbers for the bitsliced representation indicate that side-channel resistant sampling from a CDT is generally feasible but costly for all use cases. The bitsliced variants, moreover, are constantly faster than the implementations on standard representation for FrodoKEM, albeit the

Table 4: Masked CDT sampler performance in kilo cycles. N is contained in the parameter set name. For Haetae, all parameter sets have $N = 256$.

Algorithm	Bitsliced		Standard		
	Base ISA	Masked ISE	Base ISA	Masked ISE	
		Bool., arith.		Bool., arith.	all
FrodoKEM-640	1 197	244	26 787	400	319
FrodoKEM-976	1 578	323	34 858	530	433
FrodoKEM-1344	1 376	272	31 499	511	426
Haetae	2 294	369	49 909	545	289
Hawk-256	2 895	549	—	—	—
Hawk-512	7 535	1 425	—	—	—
Hawk-1024	15 078	2 851	—	—	—

Table 5: Masked BIKE and Keccak results.

(a) Masked BIKE rotation performance in kilo cycles.

Parameter set	N	Base ISA	Masked ISE	
			Bool., arith.	all
BIKE-1	12323	283	100	72
BIKE-3	24659	620	217	157
BIKE-5	40973	1 146	340	244

(b) Masked SHA/Keccak performance in kilo cycles. Note that one Keccak permutation consists of 24 rounds.

# Keccak Permutations	Base ISA	Masked ISE
		Bool., arith.
1	488	176
2	975	352
3	1 463	529
4	1 951	705
5	2 439	881
6	2 926	1 057

featured numbers do not include transformation into and from bitsliced representation. Consequently, this application does not profit from a dedicated acceleration of the core operations, but achieves a significant speed-up with the masked ISE of approximately factor five to six, lowering the costs from 1197 to 244 kilocycles for the smallest parameter set.

The picture is different for Haetae, where the standard representation not only yields the fastest result, but also our acceleration achieves an overall speed-up of more than two orders of magnitude. For Hawk, finally, only a bitsliced implementation is reasonable, because of the large CDT values, and we achieve a speed-up of factor five to six, similar to FrodoKEM.

Polynomial Rotation. Table 5a shows the speed-ups that can be achieved for rotation of BIKE polynomials. Again, the base ISA numbers indicate that this operation is generally feasible when implemented with side-channel countermeasures, with cycle counts ranging from 284 to 1146 thousand cycles. Still, the ISE brings a significant speed-up of about factor three with only the Boolean and arithmetic, masked instructions. With the dedicated, masked cmov instruction the rotation is even four times faster.

Keccak. As shown in Table 5b, the masked instructions achieve a speed-up of factor 2.7 in addition to eliminating microarchitectural leakage. This translates to a significant overall speed-up for various PQC schemes. As mentioned before, it is reported for Haetae that

more than 60 % of the signing cycles account for randomness generation of the hyperball sample. Then, a fully masked deterministic Haetae implementation requires at least 102 masked Keccak permutations for generating a hyperball sample, which translates to an approximated cycle count of nearly 50 million cycles without ISE, but only 18 million with masked ISE. Another example, Kyber-768 decapsulation, requires eight masked Keccak permutations where the masked ISE would yield an acceleration from approximately 3.9 million cycles down to 1.4 million cycles in addition to eliminating microarchitectural leakage for the Keccak part alone.

Table 6: Algorithm and data representation recommendations for masked fixed-weight polynomial sampling in different use cases.

Parameter set	Masked ISE	
	Bool., arith.	all
NTRU key gen., $N = 677, W = 254$	sorting standard rep.	sorting standard rep.
sNTRU Prime key gen., $N = 953, W = 396$	sorting standard rep.	sorting standard rep.
McEliece encaps., $N = 6688, W = 128$	comparison bitsliced rep.	comparison[1] bitsliced rep.
BIKE key gen., $N = 12323, W = 71$	comparison bitsliced rep.	comparison[1] bitsliced rep.
BIKE decaps., $N = 49318, W = 199$	Fisher-Yates, bs. I2C conversion, bs.	Fisher-Yates, std. I2C conversion, bs.

[1] no acceleration by additional masked instructions

5.3 Implementation Recommendations

Table 6 summarizes our recommendations for implementing masked fixed-weight sampling with masked instructions. Non-bitsliced sorting is the preferable algorithm for the exemplary NTRU and Streamlined NTRU Prime parameters. For the McEliece encapsulation and BIKE key generation use cases, the bitsliced comparison methods is the fastest option, the cmov and cmp instructions can not be applied here and therefore do not bring an advantage.

For FrodoKEM, generally, the **bitsliced** sampler is the fastest, and the additional ISE does not yield an advantage. In contrast, if for Haetae only the original masked ISE is available, the **bitsliced** sampler is superior, but if the additional ISE is available, the **standard** representation is faster.

5.4 Side-Channel Security

We verify first-order power side channel resistance by creating microbenchmarks for each of our new instructions and performing a *fixed vs. random* t-test with 100 000 measurements. All measurements were performed using a sample rate of 2.5 GS/s. The results are shown in Figure 1. We performed two t-tests per instruction to ensure that the supplied or generated condition does not leak any information. The results seen in Subfigures (a)-(f) show that our implementation is first-order secure. Additionally, we verified our results by disabling the internal randomness generator for the masked ALU. As seen in Figure 2, without any randomness, the cmov instruction already shows strong signs of leakage after 40 000 measurements.

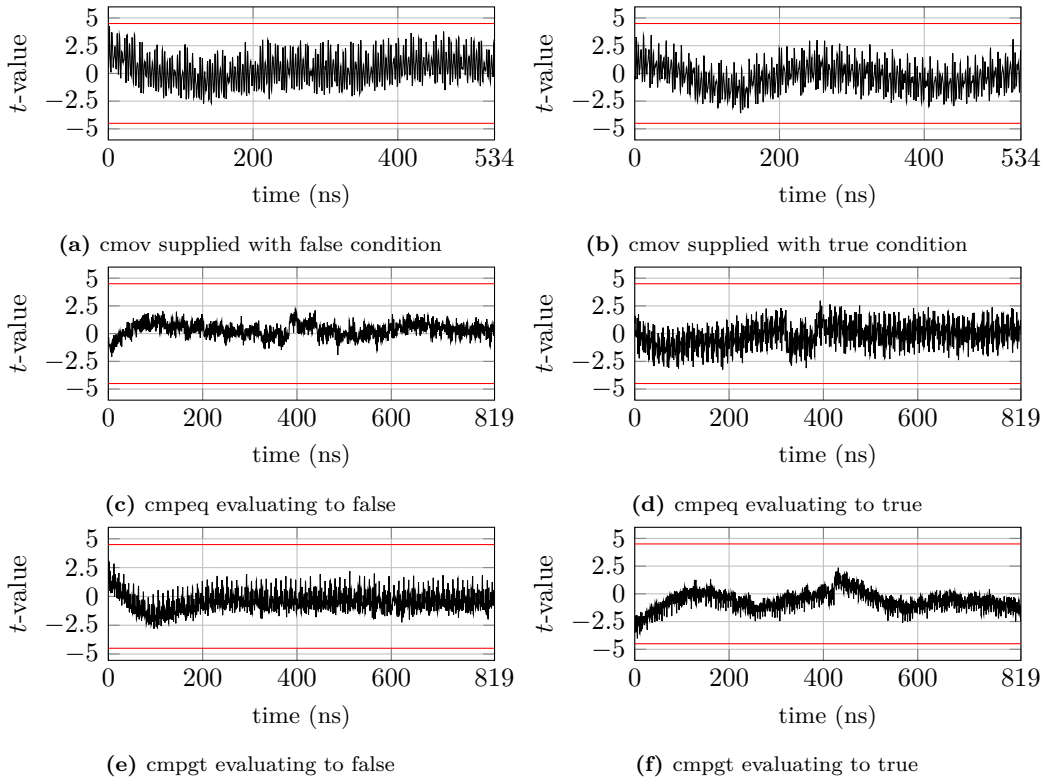


Figure 1: Side-channel evaluation of our instruction set extension using a fixed vs. random t-test with 100,000 measurements.

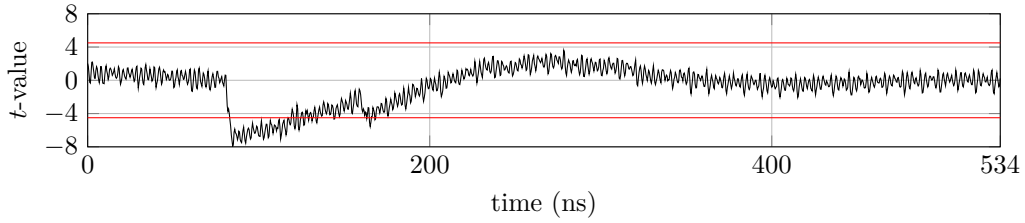


Figure 2: Side-channel measurement setup verification by performing a fixed vs. random t-test with 40,000 measurements of our cmov instruction with the internal randomness engine disabled, causing significant leakage.

6 Conclusion

In our work, we demonstrate how an ISE featuring masked instructions for Boolean and arithmetic operations, developed in previous work for symmetric cryptography can be applied to widespread PQC components. We identify common critical operations, for which we develop efficient instruction sequences based on the masked Boolean and arithmetic instructions. Furthermore, we evaluate the introduction of dedicated masked instructions for these critical operations.

For fixed-weight sampling, this leads to speed-ups of more than an order of magnitude. However, further research is required for sampling in the BIKE and HQC decapsulation, which is inefficient even with the acceleration. While we reach considerable accelerations for all applications, we note that the execution time of the sampling and rotation algorithms

is limited by their extensive memory accesses, which is further amplified by masking.

In addition to more efficient fixed-weight sampling methods that are suitable for BIKE and HQC decapsulation, fully masked implementations of both algorithms would be an important next step for research, especially in the light of a potential standardization of one of the schemes. A fully masked implementation of FrodoKEM, which is recommended by the French ANSSI and the German BSI, is now feasible with the CDT sampler presented in this work.

Acknowledgements

The work described in this paper has been supported by the German Federal Ministry of Education and Research BMBF through the projects 6GEM (16KISK038) and FlexKI (01IS22086I), by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and by the European Commission under the grant agreement number 101070374.

A BIKE Syndrome Rotation

In this section, we show that the conditional move operations in BIKE's portable C implementation [DGK] allow to perform a Simple Power Analysis (SPA) leading to a partial secret key recovery even for ephemeral keys. In the ephemeral key setting, a side-channel adversary can only acquire the power consumption of one secret key used in the decoding process. The same attack has been presented by Cheriere et *al.* in [CARG23].

Algorithm In the decapsulation of BIKE, the error polynomials, which are intentionally added to the encrypted message in the encapsulation are iteratively decoded by the BGF decoder [DGK20]. In this decoding step, the syndrome s is rotated by the secret indices of the private key (similar to the multiplication presented in [Cho16]). More precisely, these indices are represented by $\lceil \log r \rceil$ bits where r denotes the size of the polynomials used in BIKE. On a 32-bit architecture, the vector storing the syndrome s is divided into $\lceil r/32 \rceil$ chunks. Hence, applying a rotation by k bits (i.e., a secret index of the private key), can be accomplished by performing a *word-unit rotation* by the upper $\lceil \log r \rceil - 5$ bits of k (denoted by δ in the following) and a subsequent *bit-unit rotation* by the lower five bits of k. However, in the presented attack, we only measure the power consumption of the word-unit rotation (i.e., rotation by δ) as differences in the power consumption depending on the applied private key can already be distinguished by using just one single trace.

An implementation of the word-unit rotation, that is not timing side-channel secure, could simply change the pointer to the syndrome, but then subsequent operations on the syndrome would leak the rotation value via their memory access pattern. To this end, we show the constant-time implementation of the word-unit rotation in Algorithm 4. For each bit δ_i in δ, the algorithm computes a mask m which is set to `0xFFFFFFFF` in case the bit is one or to `0x00000000` in case the bit is zero. Afterwards, the implementation loops over the whole syndrome, loads the shifted and the unshifted value, and selects one of them based on m. This procedure ensures that both – the rotated and unrotated values are loaded from the memory – and no timing differences with respect to the secret index (i.e., δ) can be observed.

While the constant-time implementation protects the rotation against timing attacks, it unfortunately introduces two attack surfaces for side-channel adversaries exploiting the power consumption of the device. First, the computation of the mask m in Lines 6 in Algorithm 4 causes variations in the power consumption due to the huge difference of the Hamming weight which is directly connected to the i-th bit of the secret index δ. Second,

Algorithm 4 Word-unit rotation used in BIKE's decapsulation.

1: **Data:** Vector $v = \{s, s, s\}$ holding three consecutive copies of s, secret index δ
2: **Result:** Rotated syndrome s

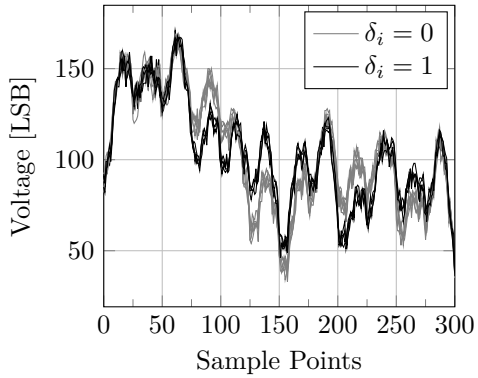
3: **function** WORD_UNIT_ROTATION(v, δ)
4: $i \leftarrow \log(r/32/2)$
5: **while** $i \geq 1$ **do**
6: $m \leftarrow (\delta > i \ ?$ `0xFFFFFFFF` : `0x00000000`) ▷ constant-time
7: $\delta \leftarrow \delta - (i \ \& \ m)$
8: **for** $j = 0$ **to** $r/32 + i$ **do**
9: $v[j] \leftarrow (v[j] \ \& \ \sim m) \mid (v[j + i] \ \& \ m)$
10: **end for**
11: $i \leftarrow i >> 1$
12: **end while**
13: **return** Lower r bits of v
14: **end function**

m is used again in Line 9 to decide whether the rotated or unrotated value of the syndrome is used to update it. Again, due to the huge difference of the Hamming weight of m, it is expected to observe corresponding variations in the power consumption of the target device.

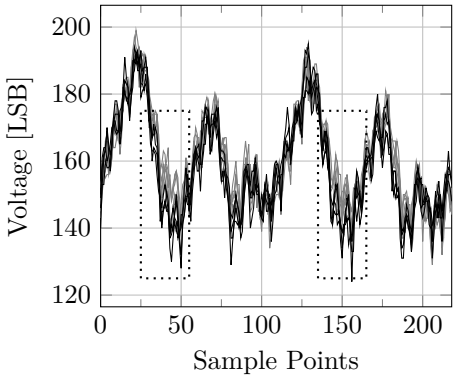
Side-Channel Attack In our side-channel attack, we first record a power trace of the decoding step of BIKE's decapsulation. Since the BGF decoder always performs seven iterations, each secret index of the private key is used exactly seven times to execute the rotation described in Algorithm 4. In order to achieve cleaner traces, we compute the mean of these seven sub-traces. Now, we select Points of Interest (PoI) by visually inspecting our preprocessed power traces, i.e., the points in the power traces where the mask m is computed or used. The selection of the PoIs is done for each bit position i separately since minor differences can occur in the power traces (e.g., in the horizontal appearance). To recover all W secret indices of one private key at the same time, we apply a *k-means* clustering of all power traces processing the i-th bit of the indices.

Practical Results We perform the described attack on BIKE's security level $\lambda = 1$ parameter set utilizing only $W = 142$ rotations. As target device, we use an ARM Cortex-M4 processor supplied with a 120 MHz clock. The measurement results for the most significant bit for ten different secret indices δ are shown in Figure 3. The differences between a one and a zero can clearly be distinguished without any further postprocessing of the traces. Repeating the same experiment for the remaining bits of δ would result in recovering the upper bits of a secret index k of BIKE's private key. The remaining lower five bits could be recovered by solving linear equation systems.

To demonstrate that our attack works reliably, we measure the power consumption of the word-unit rotation of 1 000 different secret keys. We are able to recover 98.40 % of the partial secret keys without erroneous bits. In general, 99.99 % of all upper secret bits ($8 \cdot 142 \cdot 10^3$) could be recovered correctly. Optimized versions [CCK21, CGKT22] of BIKE significantly reduce the Hamming weight, but the attack is still possible in a similar fashion as recently demonstrated in [CARG23].



(a) Calculation of the mask m (cf. Algorithm 4, Lines 7-11). Frequent visual differences over the period of the calculation.



(b) Use of the mask m (cd. Algorithm 4, Line 14) in two subsequent iterations. Visual differences are highlighted.

Figure 3: Ten raw power traces (500MS/s) of the word-unit rotation from different secret key indices δ at bit position $i = 13$. The power traces are measured on an ARM Cortex-M4 processor of the FRDM-K22F development board (rev.D) and clocked with 120 MHz. The C code is compiled with `arm-none-eabi-gcc` compiler (v9.2.1) and optimization level `-O3`.

References

[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.

[BCD+16] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1006–1018. ACM Press, October 2016.

[BCG+14] Johannes Buchmann, Daniel Cabarcas, Florian Göpfert, Andreas Hülsing, and Patrick Weiden. Discrete ziggurat: A time-memory trade-off for sampling from a Gaussian distribution over the integers. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 402–417. Springer, Heidelberg, August 2014.

[BCLv17] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime: Reducing attack surface at low cost. In Carlisle Adams and Jan Camenisch, editors, *SAC 2017*, volume 10719 of *LNCS*, pages 235–260. Springer, Heidelberg, August 2017.

[BCMP24] Alessandro Budroni, Isaac A. Canales-Martínez, and Lucas Pandolfo Perin. Sok: Methods for sampling random permutations in post-quantum cryptography. Cryptology ePrint Archive, Paper 2024/008, 2024. https://eprint.iacr.org/2024/008.

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 23–39. Springer, Heidelberg, August 2016.

[Ber22] Daniel J Bernstein. Divergence bounds for random fixed-weight vectors obtained by sorting, 2022.

[Bih97] Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 260–272. Springer, Heidelberg, January 1997.

[CARG23] Agathe Cheriere, Nicolas Aragon, Tania Richmond, and Benoît Gérard. BIKE key-recovery: Combining power consumption analysis and information-set decoding. In Mehdi Tibouchi and Xiaofeng Wang, editors, *Applied Cryptography and Network Security - 21st International Conference, ACNS 2023, Kyoto, Japan, June 19-22, 2023, Proceedings, Part I*, volume 13905 of *Lecture Notes in Computer Science*, pages 725–748. Springer, 2023.

[CCD+23] Jung Hee Cheon, Hyeongmin Choe, Julien Devevey, Tim Güneysu, Dongyeon Hong, Markus Krausz, Georg Land, Marc Möller, Damien Stehlé, and Min-June Yi. Haetae: Shorter lattice-based fiat-shamir signatures. Cryptology ePrint Archive, Paper 2023/624, 2023. https://eprint.iacr.org/2023/624.

[CCHY23] Jung Hee Cheon, Hyeongmin Choe, Dongyeon Hong, and MinJune Yi. Smaug: Pushing lattice-based key encapsulation mechanisms to the limits. Cryptology ePrint Archive, Paper 2023/739, 2023. https://eprint.iacr.org/2023/739.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the intel haswell and ARM cortex-M4. *IACR TCHES*, 2021(3):97–124, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8969.

[CD23] Wouter Castryck and Thomas Decru. An efficient key recovery attack on SIDH. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 423–447. Springer, Heidelberg, April 2023.

[CGKT22] Ming-Shing Chen, Tim Güneysu, Markus Krausz, and Jan Philipp Thoma. Carry-less to BIKE faster. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 22*, volume 13269 of *LNCS*, pages 833–852. Springer, Heidelberg, June 2022.

[CGTZ23] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. High-order masking of NTRU. *IACR TCHES*, 2023(2):180–211, 2023.

[Cho16] Tung Chou. QcBits: Constant-time small-key code-based cryptography. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 280–300. Springer, Heidelberg, August 2016.

[CMM+23] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. Randomness generation for secure hardware masking - unrolled trivium to the rescue. *IACR Cryptol. ePrint Arch.*, page 1134, 2023.

[CP23] Hao Cheng and Daniel Page. eliminate: a leakage-focused ise for masked implementation. Cryptology ePrint Archive, Paper 2023/966, 2023. https://eprint.iacr.org/2023/966.

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal Gaussians. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 40–56. Springer, Heidelberg, August 2013.

[DGK] Drucker, Gueron, and Kostic. Additional Implementation of BIKE (Bit Flipping Key Encapsulation). https://github.com/awslabs/bike-kem. Accessed: 2023-05-20.

[DGK20] Nir Drucker, Shay Gueron, and Dusan Kostic. QC-MDPC decoders with several shades of gray. In Jintai Ding and Jean-Pierre Tillich, editors, *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*, pages 35–50. Springer, Heidelberg, 2020.

[DN12] Léo Ducas and Phong Q. Nguyen. Faster Gaussian lattice sampling using lazy floating-point arithmetic. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 415–432. Springer, Heidelberg, December 2012.

[DPPvW22] Léo Ducas, Eamonn W. Postlethwaite, Ludo N. Pulles, and Wessel P. J. van Woerden. Hawk: Module LIP makes lattice signatures fast, compact and simple. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part IV*, volume 13794 of *LNCS*, pages 65–94. Springer, Heidelberg, December 2022.

[FBR+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. Masked accelerators and instruction set extensions for post-quantum cryptography. *IACR TCHES*, 2022(1):414–460, 2022.

[GGM+21] Si Gao, Johann Großschädl, Ben Marshall, Dan Page, Thinh Pham, and Francesco Regazzoni. An instruction set extension to support software-based masking. *IACR TCHES*, 2021(4):283–325, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9067.

[GHJ+22] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander Schröder. Don't reject this: Key-recovery timing attacks due to rejection-sampling in HQC and BIKE. *IACR TCHES*, 2022(3):223–263, 2022.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 197–206. ACM Press, May 2008.

[Inc] NewAE Technology Inc. ChipWhisperer). https://www.newae.com/chipwhisperer. Accessed: 2023-06-25.

[KAA21] Emre Karabulut, Erdem Alkim, and Aydin Aysu. Single-trace side-channel attacks on ω-small polynomial sampling: With applications to ntru, NTRU prime, and CRYSTALS-DILITHIUM. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2021, Tysons Corner, VA, USA, December 12-15, 2021*, pages 35–45. 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, Heidelberg, May 2023.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking NIST PQC on ARM cortex-M4. Cryptology ePrint Archive, Report 2019/844, 2019. https://eprint.iacr.org/2019/844.

[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, *Constructive Side-Channel Analysis and Secure Design - 14th International Workshop, COSADE 2023, Munich, Germany, April 3-4, 2023, Proceedings*, volume 13979 of *Lecture Notes in Computer Science*, pages 149–169. Springer, 2023.

[MP21] Ben Marshall and Dan Page. SME: Scalable masking extensions. Cryptology ePrint Archive, Report 2021/1416, 2021. https://eprint.iacr.org/2021/1416.

[MPW22] Ben Marshall, Dan Page, and James Webb. MIRACLE: MIcRo-ArChitectural leakage evaluation A study of micro-architectural power leakage across many devices. *IACR TCHES*, 2022(1):175–220, 2022.

[ND22] Dusan Kostic Nir Drucker, Shay Gueron. Isochronous implementation of the errors-vector generation of BIKE. https://github.com/awslabs/bike-kem, 2022. Accessed: 2022-10-25.

[Pei10] Chris Peikert. An efficient and parallel Gaussian sampler for lattices. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 80–97. Springer, Heidelberg, August 2010.

[Por] Thomas Pornin. Why Constant-Time Crypto? https://www.bearssl.org/constanttime.html. Accessed: 2023-06-30.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, Heidelberg, May 2013.

[Sen21] Nicolas Sendrier. Secure sampling of constant-weight words – application to BIKE. Cryptology ePrint Archive, Report 2021/1631, 2021. https://eprint.iacr.org/2021/1631.