

A Holistic Approach Towards Side-Channel Secure Fixed-Weight Polynomial Sampling

Markus Krausz¹, Georg Land^{1,2}, Jan Richter-Brockmann¹, and
Tim Güneysu^{1,2}

¹ Ruhr University Bochum, Horst Görtz Institute for IT Security, Germany

² DFKI GmbH, Cyber-Physical Systems, Bremen, Germany
`firstname.lastname@rub.de`

Keywords: PQC, Fixed Weight Polynomial Sampling, Higher-order Masking, Cortex-M4

Abstract. The sampling of polynomials with fixed weight is a procedure required by round-4 Key Encapsulation Mechanisms (KEMs) for Post-Quantum Cryptography (PQC) standardization (BIKE, HQC, McEliece) as well as NTRU, Streamlined NTRU Prime, and NTRU LPRime. Recent attacks have shown in this context that side-channel leakage of sampling methods can be exploited for key recoveries. While countermeasures regarding such timing attacks have already been presented, still, there is no comprehensive work covering solutions that are also secure against power side channels.

To close this gap, the contribution of this work is threefold: First, we analyze requirements for the different use cases of fixed weight sampling. Second, we demonstrate how *all* known sampling methods can be implemented securely against timing and power/EM side channels and propose performance-enhancing modifications. Furthermore, we propose a new, comparison-based methodology that outperforms existing methods in the masked setting for the three round-4 KEMs BIKE, HQC, and McEliece. Third, we present bitsliced and arbitrary-order masked software implementations and benchmarked them for all relevant cryptographic schemes to be able to infer recommendations for each use case. Additionally, we provide a hardware implementation of our new method as a case study and analyze the feasibility of implementing the other approaches in hardware.

1 Introduction

With the potential advent of large-scale quantum computers, rendering “classic” asymmetric cryptosystems like Elliptic Curve Cryptography (ECC) insecure, wide deployment of Post-Quantum Cryptography (PQC) has become inevitable. After three rounds of thorough analysis and many broken cryptosystems, a first set of algorithms has been selected for standardization. To enable further diversification of security assumptions, a fourth round of standardization has been launched, consisting of the three code-based schemes BIKE, HQC, and McEliece.

One building block for all round-four candidates is fixed-weight polynomial sampling. Additionally, this is also required in the three lattice-based schemes

NTRU, which may replace Kyber if potential patent issues are not resolved, Streamlined NTRU Prime, which is currently the default algorithm in OpenSSH 9, and NTRU LPRime. The output of this sampling is a uniform random binary or ternary polynomial of a specific size with a fixed number of non-zero coefficients. Multiple algorithmic approaches have been proposed [5, 9, 10, 15, 18] for this.

Karabulut et al. presented the first power side-channel attack on fixed weight sampling [14], targeting NTRU, Streamlined NTRU Prime, and Dilithium. Recently, Guo et al. [12] introduced an attack on HQC and BIKE utilizing the fixed weight polynomial sampling with variable timing depending on the seed. Sendrier [18] seized their approach and presented suitable countermeasures for BIKE. While this attack exploits timing differences, there is no reason to believe that a power side channel cannot be exploited analogously.

On the defense end, however, there is no comprehensive analysis of effective countermeasures against this type of attack. In particular, given these recent attacks, it becomes urgent to develop also power side-channel secure methodologies for fixed-weight polynomial sampling.

Hence, we present a holistic examination of the fixed-weight polynomial sampling problem with different attacker models, parameters, sampling methods, and implementation variants. We show how power side-channel secure variants of all suitable algorithms can be realized, propose performance-enhancing modifications, and provide bitsliced masked software implementations for arbitrary masking order which we make publicly available¹. Additionally, we develop a new probabilistic sampling method accompanied by a hardware implementation and a new methodology for Boolean masked comparison which is a core component for multiple algorithms. We benchmark and evaluate our implementations for all relevant PQC schemes.

2 Preliminaries

The two most important parameters for the fixed-weight polynomial sampling problem are the length of the polynomial and the weight (number of non-zero coefficients) denoted by N and W throughout the paper.

Binomial Distribution. For the Binomial probability distribution, we denote the probability mass function as

$$\mathcal{B}(k, n, p) = \binom{n}{k} p^k (1 - p)^{n-k} \quad (1)$$

where k is the number of successes in n independent Bernoulli trials, each with probability p . We know that $\mathcal{B}(k, n, p)^{-1}$ is the expected number of repetitions of the overall experiment until *exactly* k out of n successes are reached.

¹ <https://github.com/Chair-for-Security-Engineering/maskedFWPS>

2.1 Side-Channel Analysis

In this work, we consider timing behavior and power consumption of a target implementation of a cryptographic algorithm as possible side channels that could be exploited by an attacker. For timing attacks, we consider runtime differences caused by memory or cache accesses, branching on sensitive data, or secret-dependent arithmetic operations.

For power side-channel attacks, we distinguish between single-trace and multi-trace attacks. In the single-trace scenario, the attacker has given only one single trace of the cryptographic operation, i.e., the attacker cannot invoke the system multiple times with the same secret key. However, we additionally assume that an attacker can mount template attacks. In this case, the attack profiles a target device to create a power template which is used to match a single trace to the correct key.

For multi-trace attacks, we assume that an attacker can collect as many traces as possible. These traces are used for Differential Power Analysis (DPA) including statistical analyses like Correlation Power Analysis (CPA).

2.2 Masking

Masking is a well-established countermeasure against physical Side-Channel Analysis (SCA) and is based on the strong theoretic foundation of secret sharing. A secret value x is split into $d + 1$ shares x_i with $0 \leq i \leq d$. To provide the desired security, $d - 1$ shares are chosen uniformly at random while the remaining share is determined such that $x = x_0 \circ x_1 \circ \dots \circ x_d$ holds. The group operator $\circ$ is usually addition, either in $\mathbb{F}_2$ (Boolean masking) or a larger field (additive masking). The parameter d defines the security order based on the d -probing model [13], where an attacker is assumed to obtain the exact values of up to d intermediate values of the target design. Hence, if the adversary does not learn anything about the secret values using d probes, the implementation is assumed to be secure against d -order attacks.

Functions that can be applied share-wise such that $f(x) = f(x_0) \circ f(x_1) \circ \dots \circ f(x_d)$ are easy and efficient to mask. One of these linear functions is for example a XOR in the Boolean masking domain. Non-linear functions, for example, an AND, cannot be applied share-wise and need to be expressed differently. The challenge in masking cryptographic implementations relies upon avoiding or efficiently implementing non-linear functions.

2.3 Bitslicing

An important method for efficient Boolean-masked software implementations is bitslicing. Bitslicing changes the representation of values. Instead of storing n values in n distinct n -bit registers (32-bit in our case), we aggregate the i -th bit of each value in one register. This corresponds to a matrix transposition. If the maximum bit-length of the values is below the register width, bitslicing allows a condensed representation and simultaneously fewer Boolean instructions. Bitslicing is especially useful for algorithms that operate on single bits at a time because it allows doing single-bit operations on n values simultaneously

with one instruction, comparable to Single Instruction Multiple Data (SIMD) instructions. For masked implementations, bitslicing helps to reduce the number of costly non-linear operations.

2.4 Random Integer Sampling from Range

Sampling a uniform random integer from a given range is not always as simple as it seems. Both in software and hardware we can obtain uniform random bits from e.g., a Pseudorandom Number Generator (PRNG). By concatenating l random bits, we get a random value in the range of $[0, 2^l)$.

If we need a random value r in the range of $[0, x)$ (which we denote with $\text{rand}(x)$ in the following), where x is not a power of two, we can sample r from $[0, 2^l)$, with the smallest l such that $x < 2^l$, and reject r if it is not smaller than x . The closer x is to 2^l , the fewer rejections occur, in the worst case, however, the chance for rejection is almost 50%.

Instead of rejecting values, one can alternatively use a function that maps the values from $[0, 2^l)$ to $[0, x)$. An obvious function for this is computing $r \bmod x$. Given l random bits stored in r and a bit width of t for the target range x one can alternatively compute an $(l + t)$ -bit multiplication rx and take the upper t bits of the result, which again will be a value between 0 and $x - 1$.

The drawback of both of these mapping methods is that they introduce a bias. When x is not a power of two, 2^l will not divide x , therefore some values in the output range $[0, x)$ will be more likely than others. With increasing 2^l compared to x , the bias becomes neglectable and the output becomes close to uniform random.

If we want to sample an integer from a range $[i, x)$ that is not starting at 0, we can use the previous methods and compute $i + \text{rand}(x - i)$.

2.5 Applications

Fixed weight polynomial sampling is a part of many PQC schemes, and many of them can potentially become (or already are) a standard determined by the National Institute of Standards and Technology (NIST).

BIKE. Bit Flipping Key Encapsulation (BIKE) has among three other KEMs advanced to the fourth round of NIST’s standardization process and is a code-based scheme relying on Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) codes. Polynomials live in the cyclic polynomial ring $\mathcal{R} := \mathbb{F}_2[X]/(X^r - 1)$, thus coefficients are either 0 or 1 and the number of coefficients is determined by the parameter r of the reduction polynomial. During key generation, two random fixed-weight polynomials are sampled: (h_0, h_1) with $|h_0| = |h_1| = W/2$. Moreover, during encapsulation and decapsulation, two fixed weight polynomials e_0, e_1 are sampled with $|e_0| + |e_1| = t$ where t is a publicly known and fixed parameter.

HQC. HQC also advanced to the fourth round of standardization. HQC also deploys fixed-weight sampling in key generation, encapsulation, and decapsulation. Analogously, polynomials in HQC have the polynomial ring $\mathcal{R} := \mathbb{F}_2[X]/(X^r - 1)$. Apart from parameters, the only difference then is that the polynomial e_0, e_1 are sampled separately rather than with a joint fixed weight.

McEliece. The third remaining fourth-round candidate also uses fixed-weight sampling, but only during encapsulation to sample the “message”. McEliece is deemed to be the most conservative candidate during the whole standardization, being based on the more than 40 years old original McEliece cryptosystem.

NTRU. NTRU is a lattice-based Key Encapsulation Mechanism (KEM) and comes in two “flavors”: HRSS and HPS. For both, four polynomial rings are deployed. Fixed-weight sampling is used only during key generation of the HPS parameter sets. Furthermore, NTRU-HPS imposes the special requirement of having *exactly* $W/2$ coefficients $+1$ and $W/2$ to be -1 .

Streamlined NTRU Prime and NTRU LPRime. Streamlined NTRU Prime is a lattice-based KEM and is, together with X25519, currently the default algorithm for OpenSSH 9. NTRU LPRime is a merger with Streamlined NTRU Prime during the second round of NIST standardization. Both require fixed-weight sampling in their respective key generations, similar to NTRU with a ternary target space. However, no requirement is set on the number of $+1$ and -1 .

Dilithium. Dilithium is the designated PQC digital signature standard. It is based on the Module-Learning With Errors problem and operates on polynomials in the ring $\mathbb{Z}_q[X]/(X^{256} + 1)$ with $q = 8\,380\,417$. Security is scaled through the matrix parameters. Being constructed with the help of the Fiat-Shamir with aborts technique, it simulates the verifier by querying a random oracle to sample a challenge during signature generation. This challenge has the specific form of a fixed-weight polynomial with ternary coefficients and no special restrictions on the number of coefficients with value -1 . Based on several abort checks, a signature candidate may get rejected, starting over the whole signature generation including computing a new challenge c . Thus, it is not directly clear that c from rejected iterations is public information, even though the final c is part of the signature.

Previous work on the GLP signature scheme, which is a predecessor of Dilithium, has found that if the rejected challenges are viewed as public information together with their respective commitment, either one has to live with an additional heuristic security assumption or add a statistically hiding commitment scheme, tolerating the additional communication cost [3]. This is also stated regarding Dilithium in a recent preprint [1], where they state that rejected challenges are public and the commitment as well, but based on the Learning with Rounding assumption. To avoid this additional assumption, in our opinion it would be also feasible to perform the rounding masked, hashing w_1 in masked domain, obtaining a masked bit-string $\tilde{c}$, which is already a representation of the challenge. This can then be unmasked (since we know that also rejected challenges are non-sensitive) and used to perform fixed-weight sampling.

3 Conceptual Considerations

Although the fixed weight polynomial sampling problem at its core is simple, its application comes with multiple problem dimensions depending on the algorithmic scheme, and implementation target.

Attacker Model. Sampling can be used in different parts of a KEM. If it is part of the key generation that is only executed once for one key, only single-trace side-channel attacks are applicable. The profiled Simple Power Analysis (SPA) is assumed as the strongest attacker model in our case.

Since in encapsulation, no secret key is used at all, usually no multi-trace attacks are eligible. In the current setting and applications, fixed weight sampling is used once during encapsulation to sample the message or an error. For decapsulation, multi-trace attacks are possible if the KEM key is non-ephemeral.

Target Space. Some use cases require binary polynomials while others sample ternary polynomials. For the ternary polynomials, it then can vary how the weight must be split between the ones and the minus ones.

Target Representation. The classic representation for a polynomial is an array of length N with one element for each coefficient (coefficient representation). However, polynomials can also be expressed by a list of non-zero indices (index representation). The cryptographic scheme may require different representations and the sampling methods output different representations. It is possible to convert one representation into the other.

Determinism. If the sampling is used in the encapsulation and decapsulation, it is usually required to provide the same output when given the same input seed. This can be achieved for all algorithmic approaches by using a PRNG as the source of randomness that is initialized with the seed. Determinism is usually not required in the key generation.

Secret Seed. In some use cases, the input seed for the PRNG is a secret value, thus the sampling algorithm must be constant-time not only with respect to the sampled polynomial but also with respect to the input seed. Concrete attacks have been presented recently in [12, 18].

Parameters N and W . The most important parameters that determine the performance of the sampling methods are the number of coefficients N and the number of non-zero coefficients W or the weight of the polynomial. In particular, N can vary distinctly from values between 256 to 81 194.

Target Platform. Implementing hardware or software influences the performance of an algorithm. Parallelism is important in either case, in software it can sometimes be achieved with bitslicing as introduced in Section 2.3, while in hardware, more fine-grained parallelism and trade-offs are possible.

3.1 Requirement Analysis

In Table 1 we give an overview of the most important parameters and requirements of each relevant scheme for the fixed-weight polynomial sampling.

The parameter sets of BIKE and HQC include relatively large N and small to medium W , and therefore a small W/N ratio which are all important factors

Table 1: Requirements for all potential applications

Scheme	Param.	Where?	N	W	W/N	Target Space	Det.	Sec. Seed
BIKE	L1	en/decaps	24646	134	0.005	binary	yes	yes
BIKE	L1	keygen	12323	71	0.006	binary	no	no
BIKE	L3	en/decaps	49318	199	0.004	binary	yes	yes
BIKE	L3	keygen	24659	103	0.004	binary	no	no
BIKE	L5	en/decaps	81194	264	0.003	binary	yes	yes
BIKE	L5	keygen	40973	137	0.003	binary	no	no
HQC	128	en/decaps	17669	75	0.004	binary	yes	yes
HQC	128	keygen	17669	66	0.004	binary	no	no
HQC	192	en/decaps	35851	114	0.003	binary	yes	yes
HQC	192	keygen	35851	100	0.003	binary	no	no
HQC	256	en/decaps	57637	149	0.003	binary	yes	yes
HQC	256	keygen	57637	131	0.003	binary	no	no
McEliece	348864	encaps	3488	64	0.018	binary	no	no
McEliece	460896	encaps	4608	96	0.021	binary	no	no
McEliece	6688128	encaps	6688	128	0.019	binary	no	no
McEliece	6960119	encaps	6960	119	0.017	binary	no	no
McEliece	8192128	encaps	8192	128	0.016	binary	no	no
NTRU	hps2048509	keygen	509	254	0.499	$W/2$ ternary	no	no
NTRU	hps2048677	keygen	677	254	0.375	$W/2$ ternary	no	no
NTRU	hps4096821	keygen	821	510	0.379	$W/2$ ternary	no	no
sNTRU Prime	653	keygen	653	288	0.441	uni. ternary	no	no
NTRU LPRime	653	keygen	653	252	0.386	uni. ternary	no	no
sNTRU Prime	761	keygen	761	286	0.376	uni. ternary	no	no
NTRU LPRime	761	keygen	761	250	0.329	uni. ternary	no	no
sNTRU Prime	857	keygen	857	322	0.376	uni. ternary	no	no
NTRU LPRime	857	keygen	857	329	0.384	uni. ternary	no	no
sNTRU Prime	953	keygen	953	396	0.416	uni. ternary	no	no
NTRU LPRime	953	keygen	953	345	0.362	uni. ternary	no	no
sNTRU Prime	1013	keygen	1013	448	0.442	uni. ternary	no	no
NTRU LPRime	1013	keygen	1013	392	0.387	uni. ternary	no	no
sNTRU Prime	1277	keygen	1277	492	0.385	uni. ternary	no	no
NTRU LPRime	1277	keygen	1277	429	0.336	uni. ternary	no	no

for the sampling algorithms. Both schemes are also the only ones, that require seed security and a deterministic sampling algorithm for their encapsulation and decapsulation. Their polynomials have coefficients that are either 0 or 1, this is also the case for McEliece. NTRU on the other hand has ternary coefficients that are either 0, 1 or -1 and the fixed number of nonzero coefficients W must be equally split between the 1s and -1 s. For the ternary coefficients in Streamlined NTRU Prime and NTRU LPRime, this relation is uniformly random. The schemes with ternary coefficients also have in common that the sampling is only used during the key generation, security against single-trace side-channel attacks is therefore sufficient.

4 Designing Masked Fixed Weight Sampling

In the following sections, we present multiple side-channel secure approaches for fixed-weight polynomial sampling. The different approaches can be categorized into three different groups. The rejection method in Section 4.4 and its bounded variant in Section 4.5 solve the problem by sampling W distinct values in the range $[0, N)$, which represent the indices of the non-zero coefficients. The methods based on Fisher-Yates in Section 4.2 and sorting in Section 4.3 utilize

shuffling of fixed input polynomials. ANDing in Section 4.7 and our comparison method in Section 4.6 both sample fixed-weight polynomials by randomly setting bits.

For each approach, we start by explaining the fundamental idea, then we clarify how to achieve a timing side-channel secure (constant-time) variant that is a necessity for a power side-channel secure implementation. Based on this, we explain how to realize a masked and efficient variant. In Section 5, we provide more details about our implementations. We present the algorithms only for the binary use case, in most cases they can easily be adapted for the ternary use case. If this adoption is not obvious, we explain how it can be achieved. Some of the algorithms have a small bias, so their output is only close to uniform random. Before actually deploying a scheme with one of the biased methods one needs to diligently prove that the bias does not impair the security.

Masked Sampling by Coron et al. In recent work [9], Coron et al. present an approach of side-channel-secure fixed-weight sampling for NTRU, which proposes the following strategy:

1. Initialize an empty polynomial with the first $W/2$ coefficients set to -1 , the subsequent $W/2$ coefficients to $+1$, and the remaining coefficients set to 0 .
2. Generate a fresh arithmetic masking of this polynomial.
3. Shuffle each share with the same permutation.
4. Re-share the arithmetic sharing.
5. Repeat the last two steps a total of $d + 1$ times, every time using a new permutation.

This high-level procedure is proven to be secure in the d -probing model. For their proof, however, the applied permutation is assumed to be a black box. Thus, we believe that it will be very hard, if not impossible, to instantiate securely in practice. Moreover, Karabulut et al. show a single-trace attack that targets the permutation itself [14] and there is no reason to believe that an attacker is not able to attack multiple subsequent executions of different permutations successfully. Hence, it is reasonable to assume that this countermeasure does not protect against SPA attackers comprehensively.

4.1 Core Operations

The masked algorithmic approaches for fixed-weight polynomial sampling that we present in the following sections, share a small set of operations that are repeatedly used and contribute distinctly to the overall performance. In this section, we explain how to perform a masked conditional move and different integer comparisons in the Boolean domain with little non-linear operations.

Conditional Move in Boolean Domain. A very important building block for our masked algorithms is the conditional move. The semantic of `cmov( $d, s, c$ )` is that d is overwritten by s , if the condition flag c is set and d remains unchanged if c is 0.

For non-masked, but constant-time implementations, a conditional move is most efficiently expressed in software with a dedicated instruction, but can generally be expressed with a short sequence of arithmetic or Boolean instructions to avoid branching on the secret condition c and thus leak c via timing differences. A straightforward sequence would be $d = (d \wedge \neg c) \vee (s \wedge c)$.

This solution is, however, costly to mask, because it includes three non-linear operations, two ANDs, and one OR. It is possible to reduce the number of non-linear operations to one by using XOR operations: $d = d \oplus ((d \oplus s) \wedge c)$ evaluates to $d = d \oplus d \oplus s = s$, if c is true and to $d = d \oplus 0 = d$, if c is false.

Integer Comparison in Boolean Domain. Let $a[l-1:0], b[l-1:0]$ be bit vector variables representing integers in the range $[0, 2^l)$. To check whether $a < b$, we can simply compute $a - b$ and then check whether the result is negative, in which case we know that $a < b$, and else, $a \geq b$. Thus, in Boolean domain, we can employ a Ripple-Carry subtractor which computes $r[l:0] = a[l-1:0] - b[l-1:0]$. Then, $r[l]$ is the uppermost carry-out bit, which decides whether or not the result is negative. The Ripple-Carry subtractor performs the following computations:

$$r[0] = a[0] \oplus b[0] \tag{2}$$

$$c[0] = \overline{a[0]} \wedge b[0] \tag{3}$$

$$r[i] = a[i] \oplus b[i] \oplus c[i-1] \quad \forall 1 \leq i < l \tag{4}$$

$$c[i] = (c[i-1] \wedge (\overline{a[i]} \oplus b[i])) \oplus (\overline{a[i]} \wedge b[i]) \quad \forall 1 \leq i < l \tag{5}$$

$$r[l] = c[l-1] \tag{6}$$

This is usually done in Central Processing Units (CPUs), where the subtraction instruction is also used for integer comparison, but without writing the result back to the registers. In the masked case, however, we aim to achieve a very low number of secure AND gates. Thus, as we only want to recover $r[l]$ rather than the full subtraction result, we propose an alternative approach.

$t = a \oplus b$ gives us the bits, where a and b differ. The highest set bit of t determines the bit or rather the index g in a and b that determines which of the two variables is greater. Because we know that a and b differ at this bit, it is enough to look at one of them. E.g. if b_g is set, b is greater than a . To perform this concept in constant-time we iterate over all bits, starting from the lowest bit, and update our output with b_i if t_i is set, which ultimately results in b_g in our output. With our output initialized with 0, it will result in 0 if $a \geq b$ and 1 if $a < b$. Algorithm 1 describes this idea formally.

At first sight, Algorithm 1 does not need any expensive AND gadgets, but for implementing the conditional move securely we require one AND, as explained in the previous subsection. Compared to the traditional approach via subtraction, we can half the amount of expensive non-linear gadgets. The overall asymptotic runtime is determined by the bit length of the inputs. In the algorithms presented in the following, we compare values bounded by N , so the cost for one comparison is $\lceil \log_2(N) \rceil$.

Algorithm 1 Optimized Integer Comparison in Boolean Domain

Require: $a = \sum_{i=0}^l a_i 2^i$, $b = \sum_{i=0}^l b_i 2^i$

Ensure: $res \leftarrow a < b ? 1 : 0$

```
function CMPL( $res, a, b$ )  
   $t \leftarrow a \oplus b$   
   $res \leftarrow 0$   
  for  $i \leftarrow 0$  to  $l$  do  
     $\text{cmov}(res, b_i, t_i)$   $\triangleright res := res \oplus ((res \oplus b_i) \wedge t_i)$   
  end for  
end function
```

Comparison with Fixed Public Input. We can simplify this further when we have one fixed and public input b rather than two variable ones. Then, to compare whether or not $a < b$, we first employ the same procedure as in Algorithm 1. However, for each t_i , we now know publicly that it is either

- a_i in the case that $b_i = 0$, or
- $\neg a_i$ in the case that $b_i = 1$.

Thus, we have

- for $b_i = 0$, $res := res \oplus ((res \oplus 0) \wedge a_i) = res \wedge \neg a_i$, and
- for $b_i = 1$, $res := res \oplus ((res \oplus 1) \wedge \neg a_i) = res \vee \neg a_i$.

This does not save non-linear gates, as we still need one per bit, but it saves several XOR operations, which are cheap, but not free. Moreover, we can completely omit all lower bits until the first $b_i = 1$, since we start with $res = 0$, which sets all subsequent intermediate res to zero.

Comparison on Equality. Evaluating whether two masked values are equal or not is even cheaper to realize in the Boolean domain. $c = a \oplus b$ is only zero if a is equal to b . Thus we can iterate over all bits in c and condense them to one bit with masked OR operations. After flipping the resulting bit, res will be one, if c is zero and thus a is equal to b and zero otherwise, denoted with $\text{cmpeq}(res, a, b)$ in the following. The asymptotic runtime cost is again $\mathcal{O}(\log_2(N))$.

4.2 Fisher-Yates

The Fisher-Yates shuffle is an algorithm to get a uniform random permutation of a fixed input sequence in $\mathcal{O}(N)$ time. Similar to the sorting approach explained in Section 4.3 it can be directly applied to a fixed polynomial with the correct weight to get a random polynomial with the correct weight.

Alternatively, one can apply Fisher-Yates to an array with length N with distinct integers from 0 to N and treat the first W elements of the output as the indices respectively the coefficients of the polynomials which are non-zero. In this case, the permutation of the elements beyond the first W elements is irrelevant and the algorithm can be stopped after W iterations because the first W elements are not affected by further shuffling.

In its original version, Fisher-Yates is not timing side-channel-secure, because its memory accesses reveal the permutation and (only relevant for a secret seed) it requires uniform random numbers from a varying range, which requires a rejection step.

Sendrier [18] tackled these problems with two modifications. First of all, he showed for BIKE that the security of the cryptographic scheme is not necessarily impaired when the sampling is only close to uniform random if the parameters are correctly chosen. This eliminates the need for the rejection step by allowing a slightly biased constant-time approach as explained in Section 2.4. The secret dependent memory accesses can also be circumvented, but this comes with quadratic runtime instead of the original linear runtime. The solution for the index sampling method is depicted in Algorithm 2.

Algorithm 2 Constant-Time Fisher-Yates [18]

Require: N, W
Ensure: W distinct elements of $0, \dots, N - 1$
function FISHER-YATES(N, W)
 for $i \leftarrow 0$ **to** $W - 1$ **do**
 $p[i] \leftarrow i + \text{rand}(N - i)$
 end for
 for $i \leftarrow W - 1$ **to** 0 **do**
 for $j \leftarrow i + 1$ **to** $W - 1$ **do**
 $\text{cmpeq}(cond, p[j], p[i])$
 $\text{cmov}(p[j], i, cond)$
 end for
 end for
end function

For masking the constant-time Fisher-Yates algorithm two components need to be protected. The first component is sampling a random integer in the range of $[0, N - i]$. Sendrier [18] proposed to compute a random value $r \bmod N - i$, but the implied division is a costly operation. Furthermore, in most CPUs, a division is an instruction with a variable cycle count depending on the input and thus not constant time. A modulo reduction with a constant modulo might be translated by a compiler to a constant-time Barrett reduction, but there is no guarantee for this.

We propose to use the faster multiplication approach as explained in Section 2.4 instead. Multiplication instructions are constant-time for most CPUs. In the additive masking domain, the multiplication with the public range value and the addition of the public index i can be efficiently performed sharewise.

The second component is the comparison of equality and the following conditional move, both can be done in Boolean domain, therefore a transformation from arithmetic to Boolean domain between the two components is necessary. The inner loop in Algorithm 2 containing the comparison and conditional move can be computed in parallel for multiple j , because the iterations are independent of each other.

A masked implementation of this Fisher-Yates algorithm results in an asymptotic runtime of $\mathcal{O}(W^2 \log_2(N))$, for sampling a close-to-uniform polynomial in the index representation without leaking a secret seed.

4.3 Sorting

An alternative approach to obtain a uniformly random permutation of a set is to attach distinct random values to each element and sort the pairs according to the random value. Bernstein [5] suggested applying this principle to sampling fixed-weight polynomials by starting with a polynomial with the desired weight and then getting a random permutation by sorting.

Algorithm 3 Sort based Sampling [5]

Require: $N, W, l, p[N]$
Ensure: random bitpolynomial in $p[N]$ with weight W

```

function SORTSAMPLING( $N, W$ )
  for  $i \leftarrow 0$  to  $N - 1$  do
    if  $i < W$  then
       $t \leftarrow 1$ 
    else
       $t \leftarrow 0$ 
    end if
     $r \leftarrow \text{rand}(2^l)$ 
     $p[i] \leftarrow (r \ll 1) + t$ 
  end for
  sort( $p$ )
  for  $i \leftarrow 0$  to  $N - 1$  do
     $p[i] \leftarrow p[i] \wedge 1$ 
  end for
end function

```

To get *distinct* random values one can use rejection sampling, e.g., for each new randomly sampled value one checks if it collides with one of the values sampled before. If yes, the new value gets rejected and one continues until enough distinct values are sampled. Bernstein showed that the rejection step can be skipped if the size of the random value is big enough compared to the number of elements such that the chance of a collision becomes neglectable. With a constant-time sorting algorithm, the entire procedure is constant-time with respect to the sampled polynomial and the seed for the PRNG. The runtime depends on the implementation of the sorting algorithm and the polynomial size N as a parameter. This approach can be directly applied to sampling binary and ternary polynomials.

Sorting algorithms can have at lowest linear asymptotic runtime, but then usually no efficient constant-time implementation exists. A group of sorting algorithms that can be very efficiently implemented in constant-time is sorting networks, they consist only of a fixed number of comparisons and swaps.

Comparison-based sorting algorithms have at best an asymptotic runtime of $\mathcal{O}(N \log(N))$. A naive masked implementation of a sorting network mainly consists of a comparison and a conditional move depending on the comparison, both can be masked efficiently in software and in hardware.

The sorting approach is deployed in NTRU and Streamlined NTRU Prime [6] with an implementation based on Batcher’s Odd-Even mergesort [4]. For our implementation, we opted for Batcher’s Bitonic mergesort [4], because it is easier to parallelize in the bitsliced domain, which is critical for our efficient masked software implementation. Both sorting algorithms have an asymptotic runtime of $\mathcal{O}(N \log^2(N))$. Although we use our improved comparison approach explained in Section 4.1 instead of a costly subtraction, the masked comparison and the conditional move are still the overwhelming driver in cycle costs.

A major drawback of this sampling method besides its high runtime costs for large polynomial size N is the high amount of randomness required upfront resulting also in high memory usage, compared to other methods. This can be circumvented by using radixsort. Radixsort utilizes an arbitrary, stable sorting algorithm to sort numbers e.g., bit by bit, starting from the lowest bit. The stableness of the sorting algorithm ensures that the order according to the lower bits is maintained when sorting according to the higher bits¹. As radixsort only works on one bit per sorting iteration, only one random bit per element needs to be sampled and stored at a time because we are not interested in the sorted random values, but only in the permutation the sorting provides. Stable sorting networks exist, but they have a quadratic asymptotic runtime, which makes this approach more costly. Radixsort combined with an unstable sorting network does not result in correct sorting universally and coherently also not in uniform random permutations, which we confirmed for small parameters by exhaustive testing.

4.4 Rejection Sampling

Probably the most obvious solution for fixed-weight polynomial sampling is the rejection method. One samples a uniform random value r below N by rejecting values from the range $[0, 2^l)$, with the smallest l such that $x < 2^l$. Then one iterates over the already sampled indices and checks for a collision, if a collision is found, r gets rejected. The rejection sampling continues until W distinct indices are sampled as presented in Algorithm 4.

The runtime of this probabilistic algorithm varies and depends on the randomness, therefore it is not suitable for cryptographic schemes, where the seed for the PRNG is secret. This restriction in application allows early termination of loops, as soon as the rejection becomes evident. Although the result of the comparisons for equality for the collision check is public, we cannot XOR both arguments and then simply unmask the result and check if it is zero or not. In this case, we would leak the bits in which r differs from $p[c]$. So the comparisons themselves must be side-channel secure to protect the non-rejected values. This can be done with the core operations presented in Section 4.1.

¹ Stable sorting in ascending manner according to the MSB of (10, 11, 01) results in (01, 10, 11) and not (01, 11, 10).

Algorithm 4 Rejection Sampling - Index

Require: $N, W, 2^l > N, p[W]$

Ensure: W distinct elements of $0, \dots, N - 1$

```
function REJECTION-INDEX( $N, W$ )  
   $i \leftarrow 0$   
  while  $i < W$  do  
     $r \leftarrow \text{rand}(2^l)$   
     $\text{cml}(t, r, N)$   
    if  $\neg t$  then  
      continue  
    end if  
     $\text{collision} \leftarrow 0$   
    for  $c \leftarrow 0$  to  $i - 1$  do  
       $\text{cmpeq}(t, p[c], r)$   
      if  $t$  then  
         $\text{collision} \leftarrow 1$  break  
      end if  
    end for  
    if  $\text{collision} = 1$  then  
      continue  
    end if  
     $p[i] \leftarrow r$   
     $i \leftarrow i + 1$   
  end while  
end function
```

For this algorithm, N determines the probability for the first rejection step, with an N only slightly greater than the closest power of two this probability can be close to 50%. W/N determines the probability of the second rejection when checking for collisions. With a W/N close to 0.5, the chance for a collision for a single value gets close to 50% for the last iterations when i reaches W , so on average the probability for rejection due to a collision for a single value can be up to 25%. Drucker et al. [10] already pointed out that the fixed weight polynomial sampling problem is symmetric such that for $W/N > 0.5$, one can solve it for $(N - W)/N$ and invert the result.

4.5 Bounded Rejection Sampling

The idea of a bounded rejection sampling algorithm as presented by Drucker et al. [15] for the BIKE use case, is to transform the rejection sampling method as presented in Section 4.4 such that it is constant-time also with respect to the PRNG seed. This idea has also been implemented similarly by Guo et al. [12] for HQC.

For this, the rejections must not influence the path taken in the algorithm and therefore branches in a software implementation and the memory access pattern must be independent of the randomness. This is done by keeping track of the number of valid samples with a secret counter that indicates where to input the next valid index into the array and does not get incremented if a

Algorithm 5 Bounded Rejection Sampling - Index [15]

Require: $N, W, B, 2^l > N, p[W]$

Ensure: W distinct elements of $0, \dots, N - 1$

```
function BOUND-REJECTION-INDEX( $N, W, B$ )  
   $cntr \leftarrow 0$   
  for  $i \leftarrow 0$  to  $B - 1$  do  
     $r \leftarrow \text{rand}(2^l)$   
     $dup \leftarrow 0$   
    for  $c \leftarrow 0$  to  $W - 1$  do  
       $\text{cmpeq}(t, p[c], r)$   
       $dup \leftarrow dup \vee t$   
       $\text{cmpeq}(f, c, cntr)$   
       $\text{cmov}(p[c], r, f)$   
    end for  
     $\text{cmpl}(t, r, N)$   
     $t \leftarrow !dup \wedge t$   
     $cntr \leftarrow cntr + t$   
  end for  
end function
```

sample gets rejected so that the next sample can overwrite the rejected one. Early termination of loops is not possible anymore, so with every sampled value one has to iterate over the entire array of indices and securely check for a collision. These comparisons can however be performed in parallel. Also, the comparison of the current index with the counter and the conditional move can be parallelized, as the comparison only outputs 1 for a single index for one complete iteration over the array. Thus the counter can be conditionally incremented only once after iterating over the array and remains constant during the loop.

The second challenge for any seed-independent timing is the number of random values that need to be sampled which can not be determined exactly upfront, but they can be estimated. Depending on the parameters N and W one can compute a loose upper bound B of iterations or rather samples, within with overwhelming probability at least W valid indices are found.

The majority of the algorithm can be masked with Boolean components that we already discussed in previous algorithms. The incrementation of the secret counter is most efficient in the additive masking domain, however, the counter is also required in the Boolean domain for the comparison. To avoid the costly transformations between the domains, we propose performing the addition with a single bit in the Boolean domain with half-adders implying $\lceil \log_2(N) \rceil$ masked ANDs.

Algorithm 5 demonstrates this approach, since B is a multiple of W the runtime is $\mathcal{O}(W^2 \log_2(N))$. The asymptotic view indicates a similar performance to the Fisher-Yates algorithm, but a closer inspection reveals that first, bounded rejection takes more than W^2 iterations compared to $\frac{1}{2}W^2$ for Fisher-Yates and the rejection method requires two masked comparisons for each iteration compared to one comparison for Fisher-Yates. When the sampling $\text{rand}(N - i)$

Algorithm 6 Bounded Rejection Sampling - Coefficient

Require: $N, W, B, 2^l > N$, $p[N]$ initialized with zeros

Ensure: W random coefficients in p are set to 1

function BOUND-REJECTION-COEFF(N, W, B)

```
   $cntr \leftarrow 0$ 
  for  $i \leftarrow 0$  to  $B - 1$  do
     $t \leftarrow 0$ 
     $r \leftarrow \text{rand}(2^l)$ 
     $\text{cmpeq}(f0, cntr, W)$ 
    for  $c \leftarrow 0$  to  $N - 1$  do
       $\text{cmpeq}(f1, c, r)$ 
       $\text{cmpeq}(f2, p[c], 0)$ 
       $f \leftarrow \neg f0 \wedge f1 \wedge f2$ 
       $\text{cmov}(p[c], 1, f)$ 
       $t \leftarrow t \vee f$ 
    end for
     $cntr \leftarrow cntr + t$ 
  end for
end function
```

of W values in Fisher-Yates does not contribute significant costs, the bounded rejection is probably less performant when masked.

Alternatively, the sampling of values less than N can be realized with the biased multiplication method as we use it for Fisher-Yates. For some parameter sets of HQC, this might be faster, because N is close to the next lower power of two, thus the chance of rejection when $r \geq N$ is high and the upper bound B is higher. In this case, however, the runtime comparison to Fisher-Yates is even more clear and indicates that Fisher-Yates is the faster solution.

In Algorithm 6 we show how the bounded rejection method can be adapted to output polynomials in the coefficient representation instead of the index representation with asymptotic runtime $\mathcal{O}(WN \log_2 N)$.

The bounded rejection sampling is only relevant for cryptographic schemes, where the PRNG input needs to be protected as the protection comes with a performance overhead compared to the simple rejection method.

4.6 Comparison Sampling

The idea of this novel approach is to sample each coefficient of the polynomial individually with an approximation of the probability W/N . This can be implemented efficiently by comparing a uniform random bit string of length ℓ with a fixed threshold t such that $t/2^\ell \approx W/N$. If t is smaller than the random ℓ -bit value, the coefficient is set to 1.

After performing this for each coefficient, a masked weight check of the polynomial is carried out and the polynomial is accepted only if the correct weight W is hit. Else, the whole procedure is repeated, which renders this approach infeasible for use cases that require runtime independent of the input seed. This method can be considered somewhat of a generalization of the RepeatedAND method by Drucker and Gueron that we cover in Section 4.7.

Table 2: BIKE Comparison sampling, for number of expected repetitions and expected random bits, see Eqs. 7 and 8

ℓ	BIKE-L1			BIKE-L3			BIKE-L5		
	t	rep.	rnd.	t	rep.	rnd.	t	rep.	rnd.
8	1	2439.74	240 518 881	1	31.88	6 289 754	1	169.06	55 415 054
9	3	21.30	2 362 385	2	31.88	7 075 973	2	169.06	62 341 935
10	6	21.30	2 624 872	4	31.88	7 862 192	3	92.48	37 892 643
11	12	21.30	2 887 359	9	29.10	7 892 628	7	30.30	13 658 519
12	24	21.30	3 149 847	17	25.46	7 533 925	14	30.30	14 900 203
13	47	21.10	3 379 948	34	25.46	8 161 752	27	29.73	15 833 552
14	94	21.10	3 639 944	68	25.46	8 789 579	55	29.34	16 829 906

For efficiency, the choice of ℓ, t is decisive and the expected number of repetitions of the overall procedure is determined by

$$\mathcal{B}(W, N, t/2^\ell)^{-1} \quad (7)$$

Therefore, these parameters must be chosen carefully for each potential use case.

Let $p = W/N$ be the target probability. Then, for ℓ random bits, the best comparison threshold t is $\lfloor p2^\ell \rfloor$. Intuitively, the larger we choose ℓ , the better we approximate p at cost of more randomness and more secure operations. Interestingly, for all applications, there exists a threshold for ℓ , from which increasing does not improve the success probability significantly.

Apart from minimizing the number of non-linear operations, we also want to minimize the number of fresh random bits that are required. For a given (N, W, ℓ, t) , we know that

$$\mathcal{B}(W, N, t/2^\ell)^{-1} \cdot N\ell \quad (8)$$

is the expected number of fresh random bits for this method, which will help us choose ℓ, t for each use case. On the lower layer, we can employ our efficient comparison from Algorithm 1 and the optimizations for comparison with one fixed operand, resulting in $\ell - 1$ non-linear operations per coefficient and $\mathcal{B}(W, N, t/2^\ell)^{-1} \cdot N(\ell - 1)$ expected non-linear operations overall for a given (N, W, ℓ, t) .

Note that these numbers refer to the *unprotected* instantiation. When masking this approach, we require $d + 1$ times as much randomness and in addition, fresh randomness for each non-linear operation.

In the following, we give details on each potential application.

BIKE and HQC Key Generation. For BIKE and HQC, we cannot deploy this method for encapsulation and decapsulation, due to the attack by [12,18]. Still, it is eligible for key generation in both cases. Table 2 and Table 3 give details on the choice of ℓ, t for both algorithms. As can be seen there, for BIKE-L1 $\ell = 9, t = 3$ is the obvious choice, as well as $\ell = 8, t = 1$ for BIKE-L3 and $\ell = 11, t = 7$ for BIKE-L5.

For HQC, $\ell = 8, t = 1$ is the best choice for HQC-128, $\ell = 10, t = 3$ for HQC-196, and $\ell = 12, t = 9$ for HQC-256. Moreover, the randomness numbers indicate that BIKE performs better with this approach.

Table 3: HQC Comparison Sampling, for number of expected repetitions and expected random bits, see Eqs. 7 and 8

ℓ	HQC-128			HQC-196			HQC-256		
	t	rep.	rnd.	t	rep.	rnd.	t	rep.	rnd.
8	1	21.77	3 076 984	1	1.5e4	4 270 634 825	1	3.7e11	1.7e17
9	2	21.77	3 461 607	1	7272.20	2 346 440 182	1	120.43	62 473 484
10	4	21.77	3 846 230	3	28.33	10 155 736	2	120.43	69 414 983
11	8	21.77	4 230 853	6	28.33	11 171 310	5	40.46	25 649 983
12	15	20.62	4 371 146	11	26.90	11 572 734	9	30.89	21 361 556
13	31	20.47	4 700 992	23	25.11	11 700 990	19	29.46	22 076 057
14	61	20.36	5 036 069	46	25.11	12 601 066	37	28.75	23 201 162

Table 4: McEliece Comparison Sampling, for number of expected repetitions and expected random bits, see Eqs. 7 and 8

ℓ	348864			460896			Parameter Set 6688128			6960119			8192128		
	t	rep.	rnd.	t	rep.	rnd.	t	rep.	rnd.	t	rep.	rnd.	t	rep.	rnd.
6	1	44.13	923655	1	965.40	26691249	1	344.42	13820908	1	43.68	1823905	1	28.16	1383882
7	2	44.13	1077597	3	49.42	1593954	2	344.42	16124392	2	43.68	2127889	2	28.16	1614530
8	5	22.66	632174	5	29.70	1094851	5	28.88	1544999	4	43.68	2431873	4	28.16	1845177
9	9	21.11	662545	11	25.49	1057213	10	28.88	1738124	9	28.43	1780973	8	28.16	2075824
10	19	19.98	696744	21	24.62	1134475	20	28.88	1931249	18	28.43	1978859	16	28.16	2306471

Table 5: NTRU HPS Comparison Sampling, for number of expected repetitions and expected random bits, see Eqs. 7 and 8. Note that these are the numbers for generating a masked *binary* polynomial. In Section 4.6 we explain the transformation into a ternary.

ℓ	2048509			Parameter Set 2048677			4096821		
	t	rep.	rnd.	t	rep.	rnd.	t	rep.	rnd.
1	1	28.32	14 414	1	5.73e+10	3.878e+13	1	1.32e+12	1.086e+15
2	2	28.32	28 827	2	5.73e+10	7.757e+13	2	1.32e+12	2.172e+15
3	4	28.32	43 241	3	31.59	64 164	5	35.75	88 043
4	8	28.32	57 655	6	31.59	85 551	10	35.75	117 391
5	16	28.32	72 069	12	31.59	106 939	20	35.75	146 739
6	32	28.32	86 482	24	31.59	128 327	40	35.75	176 087
7	64	28.32	100 896	48	31.59	149 715	80	35.75	205 435
8	128	28.32	115 310	96	31.59	171 103	159	34.85	228 911

Table 6: Streamlined NTRU Prime Comparison Sampling, for number of expected repetitions and expected random bits, see Eqs. 7 and 8

ℓ	653		761		Parameter Set				1013		1277	
	t	rnd.	t	rnd.	857	953	rnd.	rnd.	t	rnd.	t	rnd.
1	1	1966846	1	5.1e+14	1	1.3e+16	1	3.1e+10	1	3.5e+07	1	3.0e+19
2	2	3933692	2	1.0e+15	2	2.5e+16	2	6.3e+10	2	7.0e+07	2	6.0e+19
3	4	5900538	3	76571	3	91488	3	2945795	4	1.1e+08	3	222512
4	7	84497	6	102095	6	121985	7	371651	7	168223	6	296683
5	14	105621	12	127618	12	152481	13	215382	14	210278	12	370853
6	28	126745	24	153142	24	182977	27	235987	28	252334	25	360750
7	56	147869	48	178666	48	213473	53	255543	57	286495	49	396206

BIKE-L1 Optimization. We have $\ell = 9, t = 3$ and thus want to have $a = 2^9 - 4$ in Algorithm 1 to obtain a 1 output bit in $3/2^9$ cases for a random input b . Then,

Table 7: NTRU LPRime Comparison Sampling, for number of expected repetitions and expected random bits, see Eqs. 7 and 8

ℓ	Parameter Set													
	653		761		857		953		1013		1277			
	t	rnd.	t	rnd.	t	rnd.	t	rnd.	t	rnd.	t	rnd.	t	rnd.
1	1	5.7e+11	1	1.7e+24	1	4.1e+14	1	3.3e+20	1	8.6e+15	1	1.4e+35		
2	2	1.1e+12	1	6.4e+09	2	8.2e+14	1	4.1e+17	2	1.7e+16	1	1.8e+15		
3	3	72090	3	2643045	3	106026	3	150085	3	160778	3	11021700		
4	6	96120	5	155129	6	141368	6	200113	6	214370	5	1083789		
5	12	120150	11	183384	12	176711	12	250142	12	267963	11	321266		
6	25	126010	21	148387	25	199178	23	215790	25	243060	22	385519		
7	49	144495	42	173118	49	214613	46	251755	50	283570	43	378276		
8	99	163109	84	197849	98	245272	93	284544	99	315026	86	432315		

we apply the above-described optimizations for a fixed input comparison:

$$\begin{aligned}
r &= ((0 \vee b_0) \vee b_1) \wedge b_2 \wedge b_3 \wedge b_4 \wedge b_5 \wedge b_6 \wedge b_7 \wedge b_8 \\
&= (b_0 \vee b_1) \wedge \bigwedge_{i=2}^8 b_i = \neg(\neg b_0 \wedge \neg b_1) \wedge \bigwedge_{i=2}^8 b_i
\end{aligned}$$

Note that we convert the logical OR into a logical AND by De Morgan's law since this is how it is implemented with masked gadgets. Inversion is $\mathcal{O}(1)$, while **SecAnd** is $\mathcal{O}(d^2)$, so this does not increase asymptotic complexity. Still, we can save two inversions, since b_0, b_1 are random input bits, which we can assume to be inverted already. It follows that for **BIKE-L1**, the following Boolean formula can be used to obtain a random bit with approximately the correct probability of being one, using random input bits $b_0, \dots, b_8$.

$$r = \neg(b_0 \wedge b_1) \wedge \bigwedge_{i=2}^8 b_i \quad (9)$$

BIKE-L3 and HQC-128 Optimization. With $\ell = 8, t = 1$, we fall back to the repeated AND method and can just compute $r = \bigwedge_{i=0}^7 b_i$ for uniform random bits $b_0, \dots, b_7$. Notably, we can use this approach both for **BIKE-L3** and **HQC-128**.

BIKE-L5 Optimization. With $\ell = 11, t = 7$, we have $a = 2^{11} - 8$ in Algorithm 1 with random bits $b_0, \dots, b_{10}$. Then, applying the analog optimizations as above, including not inverting random input bits:

$$\begin{aligned}
r &= \left(\bigvee_{i=0}^2 b_i \right) \wedge \bigwedge_{i=3}^{10} b_i = \neg \left(\bigwedge_{i=0}^2 \neg b_i \right) \wedge \bigwedge_{i=3}^{10} b_i \\
&\sim \neg \left(\bigwedge_{i=0}^2 b_i \right) \wedge \bigwedge_{i=3}^{10} b_i
\end{aligned} \quad (10)$$

HQC-196 Optimization. Using $\ell = 10, t = 3$, we set $a = 2^{10} - 4$ in Algorithm 1 with random bits $b_0, \dots, b_9$. Applying the aforementioned optimizations, we ob-

tain

$$\begin{aligned}
r &= (b_0 \vee b_1) \wedge \bigwedge_{i=2}^9 b_i = \neg(\neg b_0 \wedge \neg b_1) \wedge \bigwedge_{i=2}^9 b_i \\
&\sim \neg(b_0 \wedge b_1) \wedge \bigwedge_{i=2}^9 b_i
\end{aligned} \tag{11}$$

HQC-256 Optimization. $\ell = 12, t = 9$ implies setting $a = 2^{12} - 10$ in Algorithm 1 with random bits $b_0, \dots, b_{11}$.

$$\begin{aligned}
r &= \left(\left(\bigwedge_{i=0}^2 b_i \right) \vee b_3 \right) \wedge \bigwedge_{i=4}^{11} b_i = \neg \left(\neg \left(\bigwedge_{i=0}^2 b_i \right) \wedge \neg b_3 \right) \wedge \bigwedge_{i=4}^{11} b_i \\
&\sim \neg \left(\neg \left(\bigwedge_{i=0}^2 b_i \right) \wedge b_3 \right) \wedge \bigwedge_{i=4}^{11} b_i
\end{aligned} \tag{12}$$

McEliece Encapsulation. For this application, we have no special restrictions, which renders the Comparison approach possible. As can be seen from Table 4, there are feasible choices of ℓ, t for each parameter set. Notably, the highest parameter set has both N and W set to a power of two, which implies that the Comparison method falls back effectively to the RepeatedAND method.

NTRU, Streamlined NTRU Prime, and NTRU LPRime Key Generation.

For NTRU, Streamlined NTRU Prime and NTRU LPRime, we have an interesting different case, since the target space is not binary, but ternary. Additionally, NTRU imposes the condition that *exactly* $W/2$ coefficients need to be $+1$ and the remaining -1 . To convert a binary polynomial to a ternary one, we employ the following strategy, assuming that we already have sampled a Boolean masked, weight- W polynomial:

1. Sample a uniform random, masked bit r_i for each coefficient a_i with $0 \leq i < N$.
2. Compute securely the masked sign $s_i := r_i \wedge a_i$ for each masked coefficient.
3. If there is a weight restriction on the number of -1 and $+1$, accumulate all s_i securely, unmask the result and check whether the correct number of -1 is hit. If not so, start over from Step 1.

Note that for NTRU, the initially sampled binary weight- W polynomial is not rejected, but only the vector of signs. This adds $\mathcal{B}(127, 254, 0.5)^{-1} \approx 20$ expected repetitions of the above procedure for NTRU-HPS2048{509,677}, and for NTRU-HPS4096821 $\mathcal{B}(255, 510, 0.5)^{-1} \approx 28.3$.

The numbers for sampling binary polynomials with correct weight are presented in Table 5, Table 6, and Table 7. It stands out that compared to the code-based schemes, a notably lower amount of randomness is required. This

is due to the smaller polynomial degrees and the more favorable ratio between W and N . However, the very low numbers for NTRU are misleading, since they do not include the additional randomness required for sampling the correct sign weight.

4.7 RepeatedAND

In [10], Drucker and Gueron propose ANDing random bit strings repeatedly with subsequent dedicated correction of the weight as a method for sampling fixed weight vectors. Starting with a zero bit string A of length N , they compute a random bit string $\bar{A}$ of the same length by repeatedly ANDing random strings so that the expected weight of the string is halved with each AND, until the weight is below or equal to the target weight W . Then, A is set to $A \vee \bar{A}$, so that the new weight of A is less or equal the sum off the individual weights of A and $\bar{A}$. As long as the weight of A is not W , a new $\bar{A}$ is computed with a target weight of the difference between the weight of A and W and ORed with A to increase its weight towards W .

Just like the simple rejection and our comparison sampling, this method is not secure for the decapsulation in HQC and BIKE.

At first sight, this method can be masked in a straight-forward manner, by checking the weight of secret intermediate vectors being the only non-trivial component. However, it makes heavy use of computing the (secret) weight of intermediate vectors, which is cheap in unmasked domain, but a big cost factor for masking. Experimentally, we found that for BIKE, HQC and McEliece, the average number of required weight checks significantly exceeds the average for our Comparison method presented in Section 4.6, with the smallest difference being McEliece-348864 (31.02 vs. 22.66), and the biggest difference being BIKE-L5 (60.38 vs. 30.30). In software, this masked weight check would predominantly determine the performance, rendering RepeatedAND obsolete for BIKE, HQC and McEliece. In hardware, however, the weight check could be performed in parallel with the ANDing. For NTRU, Streamlined NTRU Prime and NTRU LPRime the average number of comparisons are very similar, the RepeatedAND method, however, requires less randomness.

4.8 Conversions between Polynomial Representations

Some implementations of the cryptographic schemes use the index representation for fast multiplications that follow the sampling process, but one can also transform this representation to the coefficient representation in a constant-time and masked way. For each of the W non-zero indices one iterates over all coefficients N that are initialized with 0, and if the current indices are hit one replaces the 0 with a 1. We therefore need NW iterations with a `cmpeq` and a `cmov`. A conversion in the other direction from coefficient to index representation can be done similarly with comparable costs.

5 Masked Implementation

5.1 Software

We implemented all methods presented in Section 4 in software generalized for arbitrary masking order and thus secure against multi-trace power side-channel attacks. Except for the comparison method, our implementations are parametrized for N and W . We based our software implementations on masked gadgets presented in [7]. These gadgets can be proven to be secure under the assumptions provided by the d -probing model. Additionally, they fulfill certain composability notions ensuring that a design constructed by these gadgets is still secure in the d -probing model.

For Boolean masked software implementations, bitslicing is often a very efficient methodology to improve performance. All of our implementations are bitsliced as far as the algorithms allow, we also bitsliced all core operations presented in Section 4.1.

Fisher-Yates. The first component of the Fisher-Yates algorithm is to sample a random value with a varying range $[0, N - i)$. We implemented this in the additive masking domain with the biased multiplication method. To be compatible with unmasked implementations we take 32-bit Boolean masked randomness (for example from a masked Keccak) as input and transform it bitsliced to the 48-bit additive domain (modulo 2^{48}). We unbitslice the randomness and perform the multiplication with the public value $N - i$ by a simple sharewise unmasked multiplication. The result is at most 48-bit wide, therefore the additive domain modulo 2^{48} and not e.g. 2^{64} which saves us some costly non-linear operations in the Boolean to arithmetic and arithmetic to Boolean conversions.

Taking the upper 16 bit from the results can be done in the Boolean domain, which we need anyway for the second component of the algorithm. But before transforming to the Boolean domain, we add i to the upper 16 bit, which is cheaper in the additive domain. The additive to Boolean transformation is again implemented bitsliced and we keep the data in the bitsliced domain for the comparisons and conditional moves of the second component. With W padded to the next multiple of 32, we can perform the inner loop with the comparison and condition move on 32 values at a time.

Sorting. To evaluate the sorting approach presented in Section 4.3 we implemented a masked bitonic sort in software. Bitonic sort for n elements performs $n/2$ comparisons operating on all n elements in each iteration and each pair of elements that are compared has the same distance during one iteration. For the cases where the distance is greater than our register width of 32, we thus can directly compare and conditionally swap a group of 32 consecutive values with their respective pairs in the bitsliced domain where 32 values share a register for each bit.

Comparisons of elements with a distance of less than 32 are also possible in the bitsliced domain but require a transformation. When the distance is halved from one iteration to the next as is the case for most iterations, we need to swap

half of the bits of one group of 32 elements in one register with the respective other half of paired group. In the non-bitsliced domain, this would correspond to simple register swaps, in the bitsliced domain we need to swap bits by using rotations and Boolean operators. By implementing this transformation in the bitsliced domain, we are able to perform the entire sorting algorithm in the bitsliced domain and save transformations between the domains.

For distances below 32, our method works on 64 consecutive elements at a time, we therefore pad the polynomial width to the next multiple of 64 for a clear and efficient implementation. The additional coefficients appended by the padding get initialized with zero and not paired with random values, but with the highest value possible so that the nonzero lower coefficients will not be sorted to the additional indices and they can simply be cut off after sorting. Bitonic sort originally only works on power-of-two input sizes, but can be adapted to arbitrary sizes, as we did for our implementation.

Rejection. To be able to parallelize the comparison of r versus N we perform the outer loop on batches of 32 values. We then iterate over the batch, if the result of the comparison indicates that a value r is not less than N we directly continue with the next value. If not, we compare the value to the already sampled ones, again performing 32 bitsliced comparisons at a time. By performing 32 comparisons at a time we often perform comparisons with elements that are not yet set by the algorithm, but are initialized with a value e.g. zero. The result of these comparisons must not influence the rejection behavior, otherwise the initialization value can never be included in the output which would violate the uniform randomness requirement. We solve this by simply masking out the bits of these comparisons.

If no collision is found r is stored in the array, in contrast to the bounded rejection method, this condition is not a secret value, thus we do not need the masked `cmov` operation. But we implemented this move in the bitsliced domain, so that the array of indices can be kept in the bitsliced domain throughout the entire algorithm and only converted to the non-bitsliced domain at the end.

Bounded Rejection. Similar to the simple rejection method, the implementation of the bounded rejection method for the index representation has to deal with false collisions with the initialization values. Tracking which values are set and thus which comparisons are valid is cumbersome in this case because the amount of already correctly sampled values is secret. Instead, we implemented this by initializing the array with a value that is out of bound, e.g. N . This induces only a small overhead for the collision comparison, which now has to operate on $\lceil \log_2(N) \rceil + 1$ bits instead of $\lceil \log_2(N) \rceil$.

Again we parallelized the inner loop with bitslicing to significantly improve the performance.

We determined the bound according to the formulas provided by Drucker et al. [15]. Drucker et al. suggest bounds for BIKE Level 1 ($B = 327$) and Level 3 ($B = 488$) which give a probability to fail of less than 2^{128} . If a sampling failure does not affect the security of the scheme, a lower bound can be chosen for better

performance. We selected a bound of 704 for BIKE Level 5 and 364, 460 and 267 for the three relevant parameter sets of HQC to reach the same probability.

Comparison. Generally, this approach can be parallelized very efficiently as each coefficient is sampled individually. For software, this means that bitslicing is eligible, and for hardware, an individual trade-off between area and latency can be found.

In software, the weight check is the bottleneck of this method. Since it is hard to accumulate single masked bits in Boolean sharing on software platforms, we first deploy a bitsliced Boolean-to-additive masking conversion, which converts 32 masked bits to 32 arithmetically masked values modulo 2^z for a sufficiently large chosen z . Then, we unbitslice these values and accumulate the additively masked values share-wise. Finally, when we iterated over the whole polynomial with this procedure, we can unmask the shared accumulation value to obtain the weight of the masked polynomial.

Optimized Masked Weight Check. To check whether the masked polynomial candidate has the correct weight or not, it is required to compute the weight of the polynomial. For this operation, the intermediate weight is a sensitive information as it could reveal the position of single coefficients. The masked weight computation itself is a secure accumulation of all masked coefficients to a value of size $\lceil \log_2 N \rceil$ bits, e.g., by means of a secure $\lceil \log_2 N \rceil$ -plus-one-bit adder. It is worth noting that for all three code-based applications, though, W is *much smaller* than N . It follows that most of the upper bits of such a secure adder are not required with overwhelming probability.

Since we know the expected weight of our polynomial candidate (under the assumption that no biased randomness is used as input), we can decrease the secure accumulator size and accept the possibility of an overflow happening. An overflow of the accumulator is not critical as long as it does not lead to a false-positive result, i.e., approving a polynomial that has not the correct weight.

Let z be the bit length of the secure accumulator output. Then, for a given (N, W, ℓ, t) as explained in Section 4.6, we have probability p_{fp} of a false positive:

$$p_{\text{fp}} = \sum_{\substack{i=-\lfloor \frac{W}{2^z} \rfloor \\ i \neq 0}}^{\lfloor \frac{N-W}{2^z} \rfloor} \mathcal{B}\left(W + i \cdot 2^z, N, \frac{t}{2^\ell}\right) \quad (13)$$

Obviously, it is desirable to have a negligible p_{fp} , but also a low z , since this affects the efficiency of the weight check. We find that for all use cases and parameter sets, choosing $z = 8$ (i.e., an 8-bit secure accumulator), yields $p_{\text{fp}} < 2^{-200}$.

RepeatedAND. Using the same weight check module as above, we also implemented the RepeatedAND method presented in [10]. This time, however, we cannot use the optimization shown above, since we do not check for equality, but rather whether a weight is bigger or smaller. Thus, we use 10-bit secure

accumulation, which is enough for nearly all parameter sets of NTRU, Streamlined NTRU Prime, and NTRU LPRime. For Streamlined NTRU Prime- and NTRU LPRime-1277, the probability that an intermediate weight is greater or equal than 2^{10} is negligible. This approach is not efficient for McEliece, BIKE and HQC, because compared to the Comparison approach, significantly more and bigger weight checks are required.

5.2 Hardware

As a case study, we implement the comparison sampling approach for BIKE in hardware. Additionally, we give some remarks on how hardware implementations of the other algorithms could be realized.

For hardware implementations, we generally have similar restrictions compared to embedded software platforms. Most importantly, only very limited memory is available, rendering sorting-based methods for high polynomial degrees infeasible. As an example, the smallest BIKE parameter set already would require $32 \cdot 12323 \cdot 2 = 788672$ bit storage for first-order masking assuming that 31 bit randomness per coefficient would be sufficient. On the other hand, comparison-based sorting networks can be implemented very efficiently for smaller N as in NTRU and its variants and parallelized in a more fine-grained manner than for software platforms. This allows for precise trade-offs between latency and area demand.

To reduce the latency of comparisons, a parallel-prefix subtractor could be deployed by optimizing it to only obtain the uppermost carry-out bit. In return, this would require more secure non-linear gadgets compared to our comparison method presented in Section 4.1.

For Fisher-Yates, the boolean to arithmetic and vice versa transformations could be implemented with secure Boolean adders, which is possible efficiently and pipelined [17]. Then, the relatively big integer multiplications are a major cost factor in hardware, as they involve many bit operations.

For the RepeatedAND method, we certainly expect a higher control overhead due to the more complex algorithm compared to the comparison approach. Also, an intermediate masked vector must be stored in addition to the output vector, which results in a higher memory requirement. On the other hand, in contrast to software implementations, where the weight check is the bottleneck, we can execute the weight check in parallel to the secure AND operations. This could make this method efficient for the BIKE and HQC key generations and McEliece encapsulation.

Comparison Method. Since we aim for a masked implementation, we store each share of the target sampled polynomial in a separate memory (for BIKE level 1, we instantiate one 18 KB memory for each share). Each of these memory modules can be accessed via a 32-bit interface. As explained in Section 4.6, the approach requires ℓ bits of randomness to sample one bit. Due to the 32-bit interface of the memory modules, our hardware design samples 32 bit in parallel which leads to $\ell \cdot d \cdot 32$ bits of randomness required as input to the fixed input comparison. Since our target is to implement a side-channel resistant design,

we replace all non-linear gates by secure gadgets (in our case study, we used Domain-Oriented Masking (DOM) gadgets [11]). As shown in Section 4.6, the comparison for BIKE level 1 consists of eight secure multiplication gadgets where each gadget requires $\frac{d \cdot (d+1)}{2}$ bit of fresh randomness.

To track the Hamming weight of the sampled masked polynomial, we instantiate a masked Hamming weight computation unit. The design follows the implementation concept of the unmasked Hamming weight unit from [16]. However, we realize each adder stage by masked Ripple-Carry Adder (RCA) generated from HPC2 gadgets [8]. Eventually, we obtain a masked six-bit result for each 32-bit block which is fed into an accumulation stage. The accumulator is implemented by a fully pipelined masked 8-bit Sklansky adder as proposed in [2]. Since the adder consists of eight register stages, we obtain eight masked intermediate results that need to be accumulated to a final result. For this, we utilize the same adder and cleverly feed in the intermediate results from the adder to its input to add up all intermediate results. The final result is not secret and can be unmasked in order to compare it to the desired weight W . The procedure needs to be repeated in case the weight is not met.

6 Evaluation

6.1 Software

The target of our software implementations is the 32-bit Cortex-M4 microcontroller on the STM32F4 discovery board. To measure the cycle counts we set the frequency to 24 MHz to make the cycle counts independent of the memory speed. We used the `arm-none-eabi-gcc-10.3.1` compiler with optimization-level 03 and report average cycle counts of 10 runs for algorithms without data-dependent branching and average counts of 1000 runs otherwise. For comparison sampling, we measure the non-branching execution of one iteration and report this value multiplied by the expected number of repetitions.

We excluded the generation of randomness required by calls to `rand` in our measurements so that only the performance of the fixed weight polynomial sampling algorithm is measured and not the performance of the PRNG. The generation of randomness required by masked operations is however included.

Tables 8 show our measurements in kilo cycle counts for first-order masking on the Cortex-M4.

From our measurements, we can first of all conclude that masked fixed-weight polynomial sampling is expensive in software.

For masked sampling of fixed-weight polynomials in the index representation, the simple rejection method is the fastest and can always be applied when there is no need for seed security. If seed security is required, one could alternatively use the bounded method, which is always slower compared to the simple rejection. We thus only benchmarked the bounded rejection for the use cases in BIKE and HQC. Fisher-Yates also provides seed security, and outputs in the index representation and is faster than the bounded rejection for all parameter sets that we measured.

Table 8: Performance on the Cortex-M4 in kilo cycles for first order masking. Entries marked with – are irrelevant combinations that we did not implement/measure.

Scheme	N	W	Sort	Fisher-Y.	Reject	B. Reject	RepAND	Comp.	I2C	Trans.
BIKE	24646	134	–	7128	–	34077	–	–	–	770708
BIKE	12323	71	–	2854	647	–	101629*	45838	–	195945
BIKE	49318	199	–	13206	–	69140	–	–	–	2394245
BIKE	24659	103	–	4901	1255	–	156631*	129050	–	592411
BIKE	81194	264	–	21680	–	131135	–	–	–	5497931
BIKE	40973	137	–	7514	2176	–	320522*	234007	–	1372560
HQC	17669	75	–	3063	–	25803	–	–	–	309894
HQC	17669	66	–	2852	620	–	185348*	63242	–	272707
HQC	35851	114	–	5377	–	41500	–	–	–	999526
HQC	35851	100	–	5034	1282	–	391503*	183833	–	876778
HQC	57637	149	–	7808	–	28930	–	–	–	2094589
HQC	57637	131	–	7367	2132	–	837777*	348099	–	1841552
McEliece	3488	64	108596	1847	462	–	19519 [†]	12948	–	32246
McEliece	4608	96	160777	3326	972	–	31778 [†]	20392	–	68236
McEliece	6688	128	240949	5044	1555	–	63766 [†]	31652	–	131539
McEliece	6960	119	249618	4848	1386	–	59875 [†]	34571	–	127568
McEliece	8192	128	300713	4591	1527	–	62867 [†]	34609	–	161312
NTRU	509	254	9699	11532	4709	–	2141	1666	–	15342
NTRU	677	254	14958	12445	4833	–	3559	2935	–	22674
NTRU	821	510	18338	17737	7022	–	4140	3921	–	32655
sNTRU Prime	653	288	14958	15086	6345	–	3023	3033	–	24650
NTRU LPRime	653	252	14958	12390	4806	–	3177	3299	–	21515
sNTRU Prime	761	286	16464	15063	6005	–	3699	3336	–	27828
NTRU LPRime	761	250	16464	12350	4570	–	3457	3773	–	24264
sNTRU Prime	857	322	19848	20249	7461	–	4125	4012	–	34948
NTRU LPRime	857	329	19848	20569	7805	–	4482	4650	–	35825
sNTRU Prime	953	396	21564	27494	11253	–	4403	6266	–	47664
NTRU LPRime	953	345	21564	20867	8404	–	4496	6617	–	41385
sNTRU Prime	1013	448	23405	31680	14421	–	4836	5763	–	57395
NTRU LPRime	1013	392	23405	27380	10843	–	5428	7015	–	50228
sNTRU Prime	1277	492	32361	42388	18445	–	6861	9612	–	85013
NTRU LPRime	1277	429	32361	33673	13822	–	7285	9245	–	74318

* average over 3 executions

† average over 10 executions

The sorting method required too much stack to fit into the 192-KB SRAM of our board for the large N of BIKE and HQC, but the results of McEliece with medium sized N already indicate high costs for large N . As the runtime for sorting grows sub-quadratic in N , but Fisher-Yates performance is mainly determined by its $\mathcal{O}(W^2)$ loop iterations, the sorting method is faster for the higher parameter sets of Streamlined NTRU Prime and NTRU LPRime which have medium sized N and relatively high W .

However, sorting is always outperformed by the two other coefficient representation methods, the RepeatedAND and our comparison method. For BIKE, HQC and McEliece the comparison method is superior to the RepeatedAND in runtime costs. For NTRU the performance of both methods is very similar, for Streamlined NTRU Prime and NTRU LPRime, RepeatedAND is mostly faster.

In the last column of Table 8 we present the cycle counts for a masked transformation from index to coefficient representation. In general, it depends on the implementation of the scheme which representation is required for further operations, the index representation of sparse polynomials can for example be used for efficient multiplications. The high costs for a masked transformation indicate, that if only a single representation is required, a method that directly outputs the correct representation is usually preferable.

To summarize the recommendations for masked software implementations derived from our performance measurements: if no seed security is required, the simple rejection method is the fastest index sampling method and either RepeatedAND or our comparison method is preferable for an output in coefficient representation. For the scenarios in BIKE and HQC, where the seed must be kept secret, Fisher-Yates yields the best performance with an output in index representation, and one could use a masked transformation to get the coefficient representation, or implement a less memory-consuming variant of the sorting method, by utilizing radixsort, as explained in Section 4.3.

Two schemes have parameter sets that lead to a special case for some algorithms. When N is a power of two which is the case for one parameter set of McEliece, then Fisher-Yates and rejection sampling become easier because checking if $r < N$ is not necessary. We adopted our code accordingly when benchmarking this parameter set and the effect shows clearly in the cycle numbers of Fisher-Yates that are lower compared to the next smaller parameter set of McEliece. The second special case is the highest parameter set for NTRU where $W > N/2$. In this case, the symmetry of fixed weight sampling allows to sample with $W' = N - W$ instead.

6.2 Hardware

Table 9 shows the hardware implementation results for the comparison approach presented in Section 4.6 for BIKE level 1. Therefore, we implement our design for a Xilinx Artix-7 xc7a200 Field-Programmable Gate Array (FPGA) and report the required resources and performance numbers. As a baseline, we first implement an unprotected design that consumes just 100 slices and finishes on average one sampling process in 33.4 μ s. Note, the number of required Block-RAMs (BRAMs) is reported in 36 KB memory modules. Therefore, the unprotected design requires only one 18 KB memory to store the final polynomial.

Table 9: Implementation results for the comparison sampling approach for BIKE level 1 on an Artix-7 FPGA.

d	Resources				Performance		
	Logic	Memory		Area	Cycles	Frequency	Latency
	LUT	FF	BRAM	Slices	Cycles	MHz	μs
0	194	115	0.5	100	8 350	250	33.400
1	1 957	2 721	1	627	9 756	250	39.024
2	5 075	5 815	1.5	1 548	9 756	250	39.024
3	9 038	10 085	2	2 584	9 756	250	39.024

The next three lines in Table 9 show the implementation results for a first, second, and third-order protected design. The first-order protected implementation consumes 627 slices compared to 100 slices of the unprotected design. However, all protected implementations of the sampler can be executed with the same frequency, but have a slightly higher latency due to additional register stages introduced by the masking approach.

7 Conclusion

In this work, we demonstrated how all fixed-weight polynomial sampling methods in the literature can be masked at arbitrary order. Our implementations indicate that despite bitslicing and optimized subcomponents, the existing algorithms are costly for masked software. Drucker and Gueron [10] benchmarked a subset of our algorithms and schemes without power side-channel countermeasures, their numbers indicate that the relative performance of the sampling algorithms for a given scheme is equal for masked and nonmasked software implementations.

The flexibility of hardware implementations allows faster solutions, further implementations would be an interesting target for future work. Additionally, we identified that shuffling should be investigated for the sampling algorithms as an efficient countermeasure against single-trace attacks.

Acknowledgments The work described in this paper has been supported by the German Federal Ministry of Education and Research BMBF through the project QuantumRISC (16KIS1038) and PQC4Med (16KIS1044), the German Research Foundation DFG under Germany’s Excellence Strategy - EXC 2092 CASA - 390781972 and the European Commission under the grant agreement number 101070374. We thank Eike Kiltz and Gregor Leander for their valuable comments.

References

1. Melissa Azouaoui, Olivier Bronchain, Gaëtan Cassiers, Clément Hoffmann, Yulia Kuzovkova, Joost Renes, Markus Schönauer, Tobias Schneider, François-Xavier Standaert, and Christine van Vredendaal. Leveling dilithium against leakage:

- Revisited sensitivity analysis and improved implementations. Cryptology ePrint Archive, Paper 2022/1406, 2022. <https://eprint.iacr.org/2022/1406>.
2. Florian Bache and Tim Güneysu. Boolean Masking for Arithmetic Additions at Arbitrary Order in Hardware. *Applied Sciences*, 12(5):2274, 2022.
 3. Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP Lattice-Based Signature Scheme at Any Order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT*, volume 10821, pages 354–384. Springer, 2018.
 4. Kenneth E. Batcher. Sorting Networks and Their Applications. In *AFIPS Conference*, volume 32, pages 307–314. Thomson Book Company, Washington D.C., 1968.
 5. Daniel J Bernstein. Divergence bounds for random fixed-weight vectors obtained by sorting, 2020.
 6. Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime: Reducing attack surface at low cost. In *SAC*, volume 10719 of *Lecture Notes in Computer Science*, pages 235–260. Springer, 2017.
 7. Olivier Bronchain and Gaëtan Cassiers. Bitslicing Arithmetic/Boolean Masking Conversions for Fun and Profit with Application to Lattice-Based KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):553–588, 2022.
 8. Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware Private Circuits: From Trivial Composition to Full Verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.
 9. Jean-Sebastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. High-order masking of NTRU. Cryptology ePrint Archive, Report 2022/1188, 2022. <https://eprint.iacr.org/2022/1188>.
 10. Nir Drucker and Shay Gueron. Generating a Random String with a Fixed Weight. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 141–155. Springer, 2019.
 11. Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. In *TIS@CCS*, page 3. ACM, 2016.
 12. 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 Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(3):223–263, 2022.
 13. 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, Heidelberg, August 2003.
 14. 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 HOST*, pages 35–45. IEEE, 2021.
 15. Dusan Kostic Nir Drucker, Shay Gueron. Isochronous implementation of the error-vector generation of BIKE. <https://github.com/aws-labs/bike-kem>, 2022. Accessed: 2022-10-25.
 16. Jan Richter-Brockmann, Johannes Mono, and Tim Güneysu. Folding BIKE: Scalable Hardware Implementation for Reconfigurable Devices. *IEEE Trans. Computers*, 71(5):1204–1215, 2022.
 17. Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic addition over boolean masking - towards first- and second-order resistance in hardware. In *ACNS*, volume 9092 of *Lecture Notes in Computer Science*, pages 559–578. Springer, 2015.
 18. Nicolas Sendrier. Secure sampling of constant-weight words – application to BIKE. Cryptology ePrint Archive, Report 2021/1631, 2021. <https://eprint.iacr.org/2021/1631>.