IACR Communications in Cryptology
ISSN 3006-5496, Vol. 3, No. 3, 26 pages.

https://doi.org/10.62056/avl83ziuc

Check for updates

Isochronous Fixed-Weight Sampling in Hardware

Lightweight, High-Performance and Side-Channel Secure

Adrian Marotzke[a]

NXP Semiconductors Germany GmbH, Hamburg, Germany

Abstract. We present hardware implementations of the recently proposed isochronous fixed-weight sampling algorithm by Décio Luiz Gazzoni Filho, Tomás S. R. Silva, Julio López (CiC vol. 1, 2024) and apply them to the post-quantum cryptographic algorithms NTRU-HPS, Streamlined NTRU Prime and Classic McEliece. We offer multiple implementations, optimized for different targets: A high-area high-performance implementation, a lightweight low-area implementation, as well as a side-channel secure implementation using gadget-based masking. We verify the security of our masked implementation using the PROLEAD leakage detection tool. We show that the sampling algorithm results in highly efficient and effective implementations for the NTRU-like schemes, outperforming existing hardware implementations for fixed-weight sampling.

Keywords: Post-quantum cryptography · Fixed-Weight Sampling · Hardware Implementation · NTRU · Streamlined NTRU Prime · Classic McEliece · FPGA · Masking · Gadget-based Masking

1 Introduction

Post-quantum cryptography (PQC) is now widespread with multiple standards available, notably those from NIST [NIS26]. These ciphers have numerous implementations in cryptographic libraries and protocols. NTRU-HPS, Streamlined NTRU Prime and Classic McEliece are three ciphers that, despite not being standardized by NIST, have seen deployment, e.g. in SSH [cB25, Ope22] and VPN software [Mul23] as well as standardization in other forms [FMJ26, Tea26]. All three algorithms share a common component, namely they require the sampling of a so-called fixed-weight vector or polynomial. Efficient implementation of this routine is thus a key part of accelerating these PQC algorithms. Furthermore, hardware implementations are also highly beneficial for widespread adoption, as dedicated hardware can dramatically increase performance, reduce power consumption as well as improve an implementation's resistance to side-channel attacks (SCA). Décio Luiz Gazzoni Filho, Tomás S. R. Silva and Julio López recently proposed a new algorithm for fixed-weight sampling, and their software implementation outperforms existing methods [GFSL24]. The question thus arises whether their algorithm also shows similar improvements for hardware implementations, a question we aim to answer in this work.

Our Contribution

- A high-speed hardware implementation of the fixed-weight sampling algorithm from [GFSL24], optimized for maximum performance.

E-mail: adrian.marotzke@nxp.com (Adrian Marotzke)
[a]Additional email: adrian.marotzke@tuhh.de

This work is licensed under a "CC BY 4.0" license.
Received: 2026-07-27 Accepted: 2026-09-25

- A low-area hardware implementation of [GFSL24], optimized for minimal area consumption. This variant requires no memory or multipliers.
- An arbitrary-order masked hardware implementation of [GFSL24] using HPC2 gadget-based masking. The side-channel security has been verified with the PROLEAD tool [MM22]. To our knowledge, this is the first side-channel protected hardware implementation of fixed-weight sampling for lattice schemes in the literature.
- We apply and benchmark our implementations to NTRU-HPS, Streamlined NTRU Prime and Classic McEliece on FPGAs and compare them to existing state-of-the-art implementations of fixed-weight sampling in the literature.
- We release the HDL source code of our implementation under an open-source license. It is available at `https://github.com/NXP-Research/IsochronousSampling`.

2 Background

2.1 Fixed-Weight Sampling

Fixed-weight sampling is a core component of many PQC algorithms, in particular lattice- and code-based algorithms. Fixed-weight in this context means that a vector (or a polynomial) is randomly sampled with only a specific number of non-zero entries, typically -1 or 1 (in which case it is called ternary sampling). For example, during Streamlined NTRU Prime key generation and encapsulation, a polynomial of degree n with exactly w coefficients being 1 or -1 is sampled (all other coefficients are zero), with the value of n and w depending on the exact parameter set [BCLvV17, BBC+20]. Similarly, NTRU-HPS requires the sampling of a random polynomial of degree $n-1$ with exactly $w/2$ coefficients equal to 1, $w/2$ coefficients equal to -1 and all other coefficients equal to zero [CDH+20], again with the value of n and w dependent on the parameter set. Classic McEliece samples a binary fixed-weight vector, where the vector of size n has exactly w entries with the value of 1 and the rest 0 [ABC+22]. An overview of cryptographic algorithms that incorporate fixed-weight sampling is given in Table 1. An important distinction to make with fixed-weight sampling is whether the to-be-sampled vector is *dense* or *sparse*. Dense sampling is when the weight is of similar magnitude as the vector length, such as $n/w \geq 0.1$. This is for example the case for lattice-based schemes such as *sntrup*761, where $n = 761$, $w = 286$ and $n/w = 0.376$. On the other hand, sparse sampling is when the weight is much smaller than the vector length, such as $n/w \leq 0.02$. This is the case for code-based schemes such as Classic McEliece and HQC.

For the lattice algorithms NTRU and Streamlined NTRU Prime, the respective authors recommend to use a sorting algorithm as a permutation, in order to shuffle a vector with the desired weight [CDH+20, BBC+20]. One key reason for this is that a sorting algorithm can be implemented in constant time, which is crucial for cryptographic algorithms. An example for such a sorting algorithm is djbsort [Ber26, BCLvV17], a form of a bitonic odd-even sorting network. Other ways of shuffling such a vector, such as Fisher–Yates, can be difficult to implement in a constant-time fashion which is required for cryptographic purposes [GFSL24]. Nevertheless, Fisher–Yates shuffle has been used in hardware implementations of NTRU [ABPS23] and Classic McEliece [WSN17]. Sorting-based sampling has been widely used for hardware implementations, with various different sorting algorithms being presented, such as mergesort in [DMG21] and radix sort in [PMT+23, Mar25]. Sorting-based methods and Fisher–Yates shuffling are best suited for PQC algorithms where the to-be-sampled vector is dense, such as NTRU and Streamlined NTRU Prime. This is due to the fact that for both methods, the runtime complexity is dependent on the vector length n, and is independent on the weight w. For lattice-based schemes such as

Table 1: An overview of cryptographic algorithms that make use of fixed-weight sampling.

Cipher	Parameter	Operation	n	w	n/w	Dense?	Type
BIKE [ABB+22]	L1	Enc. & Dec.	24646	134	0.005	×	binary
	L1	Keygen	12 323	71	0.006	×	binary
	L3	Enc. & Dec.	49 318	199	0.004	×	binary
	L3	Keygen	24 659	103	0.004	×	binary
	L5	Enc. & Dec.	81 194	264	0.003	×	binary
	L5	Keygen	40 973	137	0.003	×	binary
HQC [AAB+22]	128	Enc. & Dec.	17669	75	0.004	×	binary
	128	Keygen	17 669	66	0.004	×	binary
	192	Enc. & Dec.	35 851	114	0.003	×	binary
	192	Keygen	35 851	100	0.003	×	binary
	256	Enc. & Dec.	57 637	149	0.003	×	binary
	256	Keygen	57 637	131	0.003	×	binary
Classic McEliece [ABC+22]	348864	Enc.	3488	64	0.018	×	binary
	460896	Enc.	4608	96	0.021	×	binary
	6688128	Enc.	6688	128	0.019	×	binary
	6960119	Enc.	6960	119	0.017	×	binary
	8192128	Enc.	8192	128	0.016	×	binary
NTRU [CDH+20]	NTRU-HPS509	Keygen & Enc.	509	254	0.499	✓	ternary
	NTRU-HPS677	Keygen & Enc.	677	254	0.375	✓	ternary
	NTRU-HPS821	Keygen & Enc.	821	510	0.379	✓	ternary
SNTRUP [BBC+20]	*sntrup*653	Keygen & Enc.	653	288	0.441	✓	ternary
	*sntrup*761	Keygen & Enc.	761	286	0.376	✓	ternary
	*sntrup*857	Keygen & Enc.	857	322	0.376	✓	ternary
	*sntrup*953	Keygen & Enc.	953	396	0.416	✓	ternary
	*sntrup*1013	Keygen & Enc.	1013	488	0.442	✓	ternary
	*sntrup*1277	Keygen & Enc.	1277	492	0.385	✓	ternary
NTRULPR [BBC+20]	*ntrulpr*653	Keygen & Enc.	653	252	0.386	✓	ternary
	*ntrulpr*761	Keygen & Enc.	761	250	0.329	✓	ternary
	*ntrulpr*857	Keygen & Enc.	857	281	0.328	✓	ternary
	*ntrulpr*953	Keygen & Enc.	953	345	0.362	✓	ternary
	*ntrulpr*1013	Keygen & Enc.	1013	392	0.387	✓	ternary
	*ntrulpr*1277	Keygen & Enc.	1277	429	0.336	✓	ternary
SMAUG-T [CCC+26]	128	Keygen	512	140	0.273	✓	ternary
	192	Keygen	768	264	0.344	✓	ternary
	256	Keygen	1024	348	0.340	✓	ternary

NTRU with dense sampling, n is comparatively small, and w is comparatively high, with w growing roughly linearly proportional to n i.e. $w = \mathcal{O}(n)$ [GFSL24].

On the other hand, code-based PQC algorithms such as HQC and BIKE sample sparse vectors, with one parameter set of HQC for example having a length n of 17669 and a weight w of just 66 [AAB+22]. In other words, n is very large while w is small (i.e. $w = o(n)$) [GFSL24]. In these cases, other sampling methods are more suitable than shuffling or the sorting-based approaches from [BBC+20, CDH+20]. Instead, methods such as the sampling of the index of non-zero elements while rejecting index collisions are more suitable [GFSL24]. The runtime of these methods depends on the target w (which is small) and not the vector length n. This form of rejection-based sampling was investigated in e.g. [GFAB+26]. Another method for sparse sampling is for example Constant Weight Word (CWW) sampling, originally proposed in [Sen21]. This approach has also been applied to hardware, for example to HQC in [DXN+24]. However, these methods, while highly efficient for sparse sampling, are not suitable for dense fixed-weight sampling according to the investigations by [GFSL24]. For more details and background on sparse sampling, we refer to [GFAB+26].

A slightly special case is Classic McEliece, where despite the vectors being sparser than the NTRU-like schemes, it has also seen the use of sorting-based sampling [ZZC+23, DMG21, WSN18] (all three use mergesort) and Fisher–Yates [WSN17]. At the same time, rejection-based approaches have also shown to be highly efficient, such as in [CCD+22].

2.2 The Rejection-Based Shuffle of Filho et *al.*

In [GFSL24], Filho et *al.* presented a new approach for dense fixed-weight sampling, namely a constant-time rejection-sampling based shuffle. It offers better asymptotic complexity over the sorting-based methods, resulting in an improved runtime performance when benchmarking their software implementation in comparison to previous methods [GFSL24]. At the same time, it also reduces the consumption of fresh randomness compared to previous sampling methods. The algorithm is also the specified sampling algorithm for the fixed-weight sampling of the SMAUG-T cryptoscheme, which won the Korean KpqC competition [Kpq26].

Filho's et *al.* algorithm consists of two parts, described in Algorithm 1 and 2. For the proof of correctness as well as a detailed explanation of them, we refer to their paper [GFSL24]. In Algorithm 2, the number of -1's is set implicitly by c_0 and c_1, with

Algorithm 1 RejSamplingMod(n): Generation of an array of unbiased uniformly random integers modulo $n-1, n-2, \ldots, 2$ [GFSL24].

Input n
Output si (output array of $n-1$ integers, such that $\mathbf{si}[i] \sim \mathcal{U}(0, n-1-i)$)

for i from 0 to $n-2$ **do**
  $\mathbf{t}[i] \leftarrow 2^L \bmod (n-1-i)$ ▷ Precomputation step, done prior
for i from 0 to $n-2$ **do**
  repeat
    $x \xleftarrow{\$} 0, 1, \ldots, 2^L - 1$ ▷ Sample a uniform random integer of L-bits
    $s \leftarrow n-1-i$
    $m \leftarrow x \cdot s$
    $l \leftarrow m \bmod 2^L$ ▷ Reduction can be done via a bitmask
  until $l \geq \mathbf{t}[i]$
  $\mathbf{si}[i] \leftarrow \lfloor m/2^L \rfloor$ ▷ Divide using a right-shift by L-bits
return si

Algorithm 2 Shuffle(n, c_0, c_1): Ternary fixed-weight sampling by shuffling [GFSL24].

Input n
Input c_0, c_1 (the number of coefficients equal to 0 and 1, respectively)
Output v (output array of $n-1$ integers)

si ←RejSamplingMod(n)
for i from 0 to $n-2$ **do**
  if $\mathbf{si}[i] < c_0$ **then** ▷ These if-clauses must be done in constant time
    $\mathbf{v}[i] \leftarrow 0$
    $c_0 \leftarrow c_0 - 1$
  else if $\mathbf{si}[i] < c_0 + c_1$ **then**
    $\mathbf{v}[i] \leftarrow 1$
    $c_1 \leftarrow c_1 - 1$
  else
    $\mathbf{v}[i] \leftarrow -1$
return v

the number of -1's being equal to $n - c_0 - c_1$. With NTRU-HPS, the number of -1's in the sampled vector is equal to the number of 1's, allowing c_0 to simply be set to the specified value depending on the parameter set. However, this is not the case for Streamlined NTRU Prime where no such restriction exists. [GFSL24] provided a suggestion for this, namely to sample an array of $n - c_0$ random bits and then setting c_1 to the Hamming weight of this

array. This corresponds to a binomial distribution and is shown in Algorithm 3.

Algorithm 3 Sample$_{c_1}(n, c_0)$: Sample c_1 according to a binomial distribution [GFSL24].

Input n
Input c_0
Output c_1

1: $c_1 \leftarrow 0$
2: **for** i from 0 to $n - c_0$ **do**
3: $\quad x \xleftarrow{\$} 0, 1$ ▷ Sample a uniform random bit
4: $\quad c_1 \leftarrow c_1 + x$
5: **return** c_1

[GFSL24] claim that their method of fixed-weight sampling is best suited for PQC algorithms where the to-be-sampled vector is dense, such as NTRU and Streamlined NTRU Prime (see also Table 1 for more algorithms). They also reason that their algorithm would not be suitable for sparse sampling. We expect both of these aspects to also apply to hardware implementations.

The applicability of the algorithm of [GFSL24] to sparse sampling was investigated in [GFAB+26], which confirmed that the algorithm is indeed not suitable for the sparse sampling of HQC. However, whether the algorithm of [GFSL24] is suitable for Classic McEliece was still left open. Applying Algorithm 1 and 2 to Classic McEliece is straightforward: Algorithm 1 can be left unmodified, and in Algorithm 2 all lines related to c_1 as well as Line 10 are removed.

2.3 Side-Channel Attacks

In reality, cryptographic algorithms must run on physical devices. This opens up the possibility of *side-channel attacks* (SCA), where an attacker can measure e.g., the timing or power consumption of a device running a cryptographic algorithm, and attempt to extract secret data from said measurements. For example, [GHJ+22] demonstrate timing attacks against the fixed-weight sampling of HQC and BIKE. The authors of [GFSL24] had already taken timing side-channels into account during their work, and their algorithm ensures that an attacker cannot extract any secret data based on timing information. However, they did not address more powerful side-channel attacks such as differential power analysis [KJJ99]. In [KAA21], the authors demonstrate successful single-trace side-channel attacks against the software implementations of fixed-weight sampling of both NTRU-HPS and Streamlined NTRU Prime. Their attack recovers the session key and shows that this step is indeed vulnerable and requires protection.

To properly model the security of side-channel protected designs, [ISW03] introduced the d-probing model. This allows an attacker to insert up to d probes in the circuit, reading the contents of the wires. However, this abstraction did not model certain physical phenomena of hardware circuits, such as glitches or transitions. It was extended to the robust d-probing model in order to incorporate these aspects [BGI+18, FGM+18].

As a countermeasure to advanced SCAs, *masking* has proved to be an efficient countermeasure. Masking, based on Shamir's secret sharing, splits sensitive data into so-called shares, and operates on these individual uniform random shares. Boolean masking, for example, splits the secret value x into $d + 1$ shares $x^{(i)}$, such that $x = \bigoplus_{i=0}^{d} x^{(i)}$. We use d as the masking order. In the robust d-probing model, the attacker can insert d probes, but since each sensitive value is split up into $d + 1$ shares, the attacker does not gain any information on the secret values. A higher masking order thus corresponds to a circuit more secure against SCAs. We will write $x^{(0:d)}$ to indicate a masked variable x at order d.

Side-channel secure fixed-weight sampling using masking was extensively studied in [KLRBG23], though as it predates [GFSL24], they could not take the new algorithm into account. While [KLRBG23] focused mainly on software implementations, they also studied masked hardware implementations of fixed-weight sampling for the code-based scheme BIKE [ABB+22]. However, as the sampled vector for BIKE is highly sparse, their method is not suitable for the dense fixed-weight sampling used in schemes like NTRU-HPS and Streamlined NTRU Prime.

2.4 Gadget-Based Masking

The approach of gadget-based masking is to replace individual hardware gates with secure masked version, called gadgets. These gadgets are atomic building blocks which can be composed together to build more complex circuits. The robust d-probing model by itself is not sufficient for this type of composability. In [BBD+15], the concept of Non-Interference (NI) was introduced, which was extended to Strong Non-Interference in [BBD+16]. Both use the concept of *simulatability*: if any set of internal probes on a gadget can be simulated by probes on the gadget's inputs and outputs, then any internal probes can be replaced with probes on the gadget's inputs or outputs. This can then be repeated recursively for composite gadgets, which means that any composition of secure gadgets is automatically secure. However, SNI compliant gadgets came with a significant overhead, namely additional register stages and the consumption of fresh randomness. This overhead was reduced with the Probe-Isolating Non-Interference (PINI) concept in [CS20], which allows trivial and efficient composability in the robust d-probing model.

An example of a PINI gadget is Hardware Private Circuit (HPC), proposed by [CGLS21]. The authors implement a secure, masked AND gadget, which can be universally composed to build arbitrary circuits at any masking order. They propose two variants HPC1 and HPC2, with HPC2 requiring less fresh randomness per cycle. We will be using HPC2 throughout this paper. One of the main benefits of masking using HPC2 is its simplicity and composability: Unlike masking methods such as Threshold Implementations [NRR06], we can arbitrarily connect HPC2 gadgets together, and the resulting circuit will be secure. Furthermore, the masking order can be easily adjusted via a single parameter, and does not require any design changes. Gadget-based designs can also be formally verified using tools such as VERICA [RBFSG22], though this may be infeasible if the design is too large.

Gadget-based masking has already been used to secure full PQC hardware implementations, such as Streamlined NTRU Prime [LMRBG24] and Kyber [BLH+25]. However, it has not yet been applied to the task of fixed-weight sampling. The authors in [LMRBG24] only implemented the decapsulation of Streamlined NTRU Prime, which does not require fixed-weight sampling.

2.5 PROLEAD

PROLEAD is an automated tool designed to analyze the side-channel security of masked hardware circuits [MM22]. Rather than formal verification, it uses logic simulations to evaluate the security, allowing PROLEAD to handle larger and more complicated circuits. PROLEAD only requires the gate-level netlist of a design, and can also properly deal with the presence of glitches and transition leakages in the robust d-probing model. While simulating the gate-level netlist, PROLEAD performs a G-test [McD14] to check for statistical independence between fixed and random inputs. The value of 10^{-5} is used as a threshold to indicate the presence of leakages, in a similar fashion to the widely used t-test [SM15]. During the simulation, the tool also outputs the statistical confidence of the results, allowing the user to gauge whether more simulations are required. The tool is available at `https://github.com/ChairImpSec/PROLEAD`. It has proven itself to be able to successfully detect vulnerabilities in masked circuits [MM22, RM25].

3 Hardware Implementation

In this section, we will present our high-performance and low-area hardware implementation of Filho's et *al.* fixed-weight sampling algorithm, and discuss implementation-specific aspects.

3.1 High-Performance Implementation

Our high-speed design is optimized for highest performance. For this, we fully pipeline the second for-loop of Algorithm 1. We set the pipeline depth to 3, as this allows all register stages to be absorbed into the DSP unit of the FPGA, without consuming any additional flip-flops. We choose $L = 16$ for NTRU-based designs, the same as [GFSL24], as they showed that it strikes a good balance between the chance of rejection, randomness consumption and implementation simplicity: Larger L decreases the rejection probability in Line 9 in Algorithm 1, but increases the randomness usage per attempt. For the Classic McEliece design, we choose $L = 19$. This is due to the larger n in Classic McEliece, as $\lceil log_2(n) \rceil \leq 10$ for the NTRU and Streamlined NTRU Prime parameter sets, but for Classic McEliece we have $\lceil log_2(n) \rceil = 13$ for all parameter sets. Increasing L by 3 thus gives a similar balance between rejection chance and randomness consumption as for the NTRU. In addition, multipliers for these values of L are readily available in FPGAs in the form of DSP units [Xil18].

To prevent the pipeline from stalling, we check if a sampled value is valid or must be rejected (Line 9 in Algorithm 1). If valid, we store the sampled value in RAM at the corresponding address i (Line 10). However, if the value is rejected, we instead store the index i in an extra FIFO memory. Once the for-loop is complete and the pipeline has been flushed, we read the indices from the FIFO, and repeat the sampling for these values. In case of another rejection, the index is stored once again in the FIFO. This repeats until the FIFO is completely empty and the pipeline has been flushed, at which point **si** is valid and Algorithm 1 is complete.

One question here is how deep the FIFO should be. Should the number of rejections exceed the FIFO depth, then the entire algorithm must restart. We compute an upper bound of probability of this occurring in the following equation:

$$P(rej > t) = \sum_{k=t+1}^{n-1} \binom{n-1}{k} \cdot (1 - p_{min})^k \cdot (p_{min})^{n-1-k} \tag{1}$$

Where $\binom{n-1}{k}$ is the binomial coefficient, t is the FIFO depth and p_{min} is the minimum probability of a non-rejection sample for *any* index for a given L and vector length $n - 1$

Based on Equation 1, we write a Sagemath script that calculates the probability for different FIFO depths and vector lengths of NTRU-HPS and Streamlined NTRU Prime (see Appendix A). We choose a FIFO depth of 128. This gives a worst-case probability of the FIFO filling up of $2^{-173.27}$, which is the case for the Streamlined NTRU Prime parameter set *sntrup*1277. All other NTRU and NTRU Prime parameter sets have a lower probability. In practice of course, the actual probability of the FIFO being full is even lower, as Equation 1 always uses the worst-case rejection probability p_{min} for simplicity, rather than the true rejection probability for a given index. What's more, in the unlikely event that the number of rejected samples exceeds 128 and the FIFO is full, then the entire algorithm simply aborts and starts from the beginning.

Our variant of RejSamplingMod, called PipeRejSamplingMod is depicted in Algorithm 4. A diagram of the PipeRejSamplingMod module is shown in Figure 1. Once **si** has been generated, the values can be read from memory sequentially and compared with the values c_0 and $c_0 + c_1$, as per Algorithm 2. This is straightforward in hardware, and can be done with adders and multiplexers in a single clock cycle to ensure constant-time execution.

Furthermore, as we only perform a single pass over **si**, we can use a RAM with separated read and write ports to already start the next iteration of Algorithm 4. This allows us to interleave the operation of Algorithm 4 and 2, saving further time. In the case of Streamlined NTRU Prime, we first also perform Algorithm 3 to get the value of c_1, before starting Algorithm 2.

An option we did not pursue is to use multiple parallel multipliers. This would mean effectively vectorizing the inner for loops in Algorithm 4. One downside, aside from the extra hardware for additional multipliers, is that this would make dealing with rejections more complex: Rather than store the individual rejected indexes in the FIFO, we would store the whole vectorized batch and resample them later, in a similar fashion to how [GFSL24] vectorized their software implementation using SIMD (Algorithm 7 in their paper). However, if higher throughput is desired, it is far simpler to instantiate the entire fixed-weight sampling module multiple times, and have them running in parallel.

Algorithm 4 PipeRejSamplingMod(n): Generation of an array of unbiased uniformly random integers modulo $n-1, n-2, \ldots, 2$, in a manner suitable for pipelining in hardware.

Input n
Output **si** (output array of $n-1$ integers, such that $\mathbf{si}[i] \sim \mathcal{U}(0, n-1-i)$)

1: **for** i from 0 to $n-2$ **do**
2: $\quad \mathbf{t}[i] \leftarrow 2^L \bmod (n-1-i)$ ▷ Precomputation step, done prior
3: **for** i from 0 to $n-2$ **do** ▷ All operations in this loop can be fully pipelined
4: $\quad x \xleftarrow{\$} 0, 1, \ldots, 2^L - 1$
5: $\quad s \leftarrow n-1-i$
6: $\quad m \leftarrow x \cdot s$
7: $\quad l \leftarrow m \bmod 2^L$
8: $\quad$ **if** $l \geq \mathbf{t}[i]$ **then**
9: $\quad\quad \mathbf{si}[i] \leftarrow \lfloor m/2^L \rfloor$
10: $\quad$ **else**
11: $\quad\quad$ **f** enqueue i ▷ Store index i in the FIFO **f**
12: **wait until** pipeline is flushed
13: **while** **f** is not empty **do** ▷ If **f** is empty, this loop is skipped
14: $\quad x \xleftarrow{\$} 0, 1, \ldots, 2^L - 1$ ▷ All operations in this loop can be fully pipelined
15: $\quad i \leftarrow$ **f** dequeue ▷ read index i from the FIFO **f**
16: $\quad s \leftarrow n-1-i$
17: $\quad m \leftarrow x \cdot s$
18: $\quad l \leftarrow m \bmod 2^L$
19: $\quad$ **if** $l \geq \mathbf{t}[i]$ **then**
20: $\quad\quad \mathbf{si}[i] \leftarrow \lfloor m/2^L \rfloor$
21: $\quad$ **else**
22: $\quad\quad$ **f** enqueue i
23: $\quad$ **wait until** pipeline is flushed
24: **return** **si**

3.2 Low-Area Implementation

For our low-area implementation, minimizing the total area is our priority. In particular, the memory to store **si** and the multiplication of $x \cdot s$ in Algorithm 1 are our main focus. There are two important details here: First is that in the $x \cdot s$ multiplication (Line 7, Algorithm 1), only the x is secret, with s being a public value. And secondly, the fact that Algorithm 2 reads **si** sequentially in a single pass.

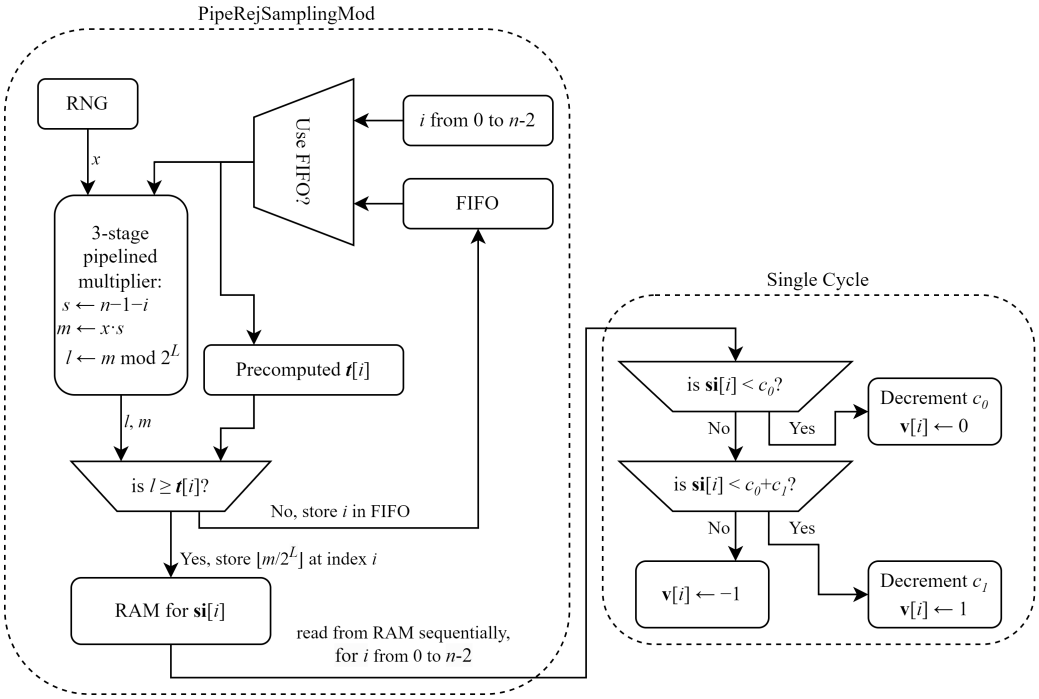

Figure 1: An architecture diagram of PipeRejSamplingMod module as well as module for Algorithm 2 for the high-speed design. The area marked "single cycle" is implemented using multiple multiplexers controlled by a finite state machine and takes a single cycle.

The fact that s is public allows us to replace the $x \cdot s$ multiplication with an iterative shift-and-add multiplication, using a simple adder. This does not expose a timing side-channel, as the timing variability depends solely on the number of set bits of s, which is public. To be precise, the shift-and-add multiplication takes a number of cycles equal to the hamming weight of s. In case of a rejection, we simply repeat the process for the same index immediately, with a new random x. We do not have any pipelining here, so this is straightforward.

Our next optimization is the memory to store **si**. There is no requirement in Algorithm 2 for **si** to be computed in its entirety before beginning Algorithm 2. As **si** is read sequentially in a single pass, we remove the memory to store **si** entirely, and instead compute each value one-at-a-time when needed. We can then immediately compute the corresponding output **v**. The single value of **si** can then be discarded, as it is no longer needed. This variant, which we call IterativeShuffle, is depicted in Algorithm 5. An architecture diagram is shown in Figure 2. As we output each value of **v** instead of buffering them, no memory cells (e.g. BRAM, SRAM, LUTRAM) are needed in the entire implementation.

4 Masked Hardware Implementation

In this section, we will present our masked implementation for fixed-weight sampling. Its design is based on the low-area design (Algorithm 5 and Figure 2) from Section 3.2, which we secure using gadget-based masking. The key feature is that the low-area design does not require a full multiplier, but can instead be implemented with an adder. Gadget-based masked adders have previously been presented in e.g. [BG22, LMRBG24, BLH+25]. Specifically, we re-use the 13-bit Sklansky adder from [LMRBG24] (which in turn is based

Algorithm 5 IterativeShuffle(n, c_0, c_1): Ternary fixed-weight sampling by shuffling.

Input n
Input c_0, c_1 (the number of coefficients equal to 0 and 1, respectively)
Output $\mathbf{v}$ (output array of $n-1$ integers)

1: **for** i from 0 to $n-2$ **do**
2: $\quad \mathbf{t}[i] \leftarrow 2^L \bmod (n-1-i)$ ▷ Precomputation step, done prior
3: **for** i from 0 to $n-2$ **do**
4: $\quad s \leftarrow n-1-i$
5: $\quad$ **repeat**
6: $\quad\quad x \xleftarrow{\$} 0, 1, \ldots, 2^L - 1$
7: $\quad\quad m \leftarrow x \cdot s$ ▷ Multiplication using an iterative shift-and-add
8: $\quad\quad l \leftarrow m \bmod 2^L$
9: $\quad$ **until** $l \geq \mathbf{t}[i]$
10: $\quad \text{si} \leftarrow \lfloor m/2^L \rfloor$ ▷ Compute single value si
11: $\quad$ **if** $\text{si} < c_0$ **then** ▷ The if-clauses are constant time, see Figure 2
12: $\quad\quad \mathbf{v}[i] \leftarrow 0$
13: $\quad\quad c_0 \leftarrow c_0 - 1$
14: $\quad$ **else if** $\text{si} < c_0 + c_1$ **then**
15: $\quad\quad \mathbf{v}[i] \leftarrow 1$
16: $\quad\quad c_1 \leftarrow c_1 - 1$
17: $\quad$ **else**
18: $\quad\quad \mathbf{v}[i] \leftarrow -1$
19: **return** $\mathbf{v}$

on the Sklansky adder from [BG22]), which we modify to a width of 16-bit. [BG22] showed that the parallel-prefix adder design of Sklansky [Skl60] offers a good balance of area, latency and randomness consumption for masked designs. This masked variant, which we call MaskedIterativeShuffle, is shown in Algorithm 6.

A notable difference from the low-area design is that we choose $L = 15$ for the masked design. This means our masked iterative adder also only has an input width of 15-bit. Doing so allows us to use the 16th bit of the adder as the carry out, which in turn means we do not require an additional stage for the adder. This would increase the latency of the adder by two clock cycles (the HPC2 gadget has a latency of two clock cycles for a masked AND gate).

We implement the if-else-clause in Line 17 of Algorithm 6 using the same masked adder as above, together with a masked multiplexer. In addition, we store $c_0^{(0:d)}$ and $c_1^{(0:d)}$ as the two's complement of c_0 and c_1 respectively. This allows us to forgo masked subtraction and instead implement all arithmetic operations with the masked adder.

To update $c_0^{(0:d)}$ and $c_1^{(0:d)}$, we securely add 1 to both, and then feed the original values and the incremented ones into a masked multiplexer. This multiplexer uses the most significant bit of $\text{si}^{(0:d)} + c_0^{(0:d)}$ and $\text{si}^{(0:d)} + c_0^{(0:d)} + c_1^{(0:d)}$ respectively as the select signal. The result is then written back into $c_0^{(0:d)}$ and $c_1^{(0:d)}$. We use the same multiplexing to write $-1, 0$ or 1 into $\mathbf{v}^{(0:d)}[i]$. For the masked multiplexer itself, we reuse the design of [LMRBG24], which is open source and also uses the HPC2 gadget.

During the rejection sampling routine, we need to check whether $l^{(0:d)} \geq t[i]$ (Line 11, Algorithm 6). While the values of $t[i]$ are precomputed and public, $l^{(0:d)}$ is a secret and masked value. We implement this check as follows: using the masked adder, we subtract the value of $t[i]$ from $l^{(0:d)}$ by adding the two's complement of $t[i]$. We then *unmask* the most significant bit, which is the 16th bit and is the carry-out of the adder. This bit is then used to decide whether to reject the sampled integer or to accept it, as it is zero if $l^{(0:d)} \geq t[i]$ and one otherwise. Despite the unmasking, this does not reveal any information

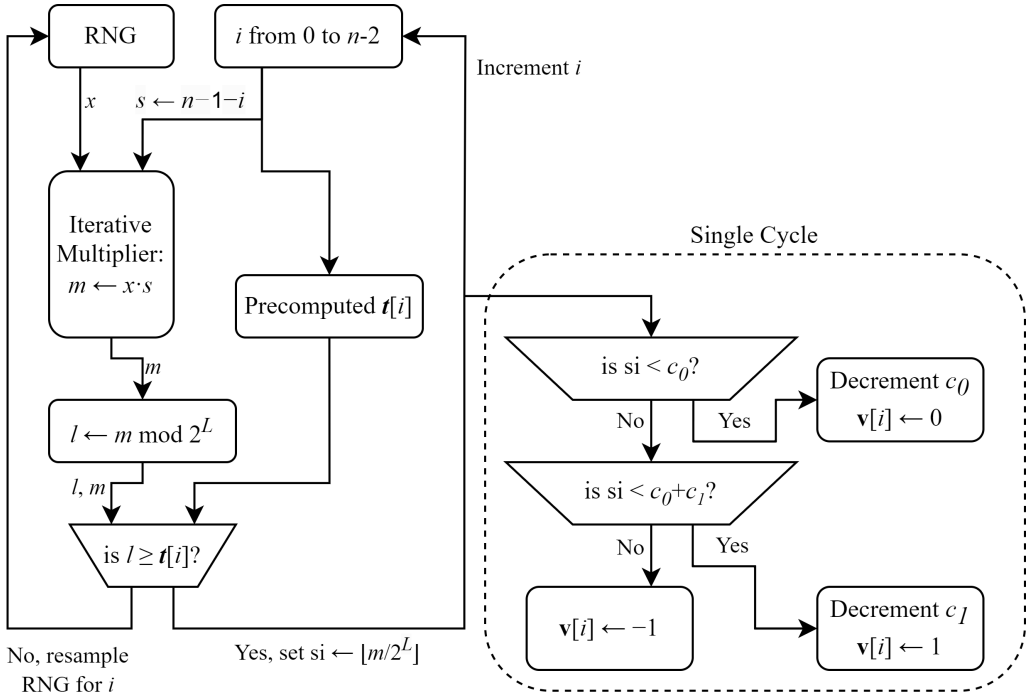

Figure 2: An architecture diagram of the IterativeShuffle module for the low-area design. The area marked "single cycle" is implemented using multiple multiplexers controlled by a finite state machine and takes a single cycle. This architecture is also used for the MaskedIterativeShuffle, with the difference that all sensitive values are masked, and all sensitive operations are done using masked gadgets.

on the value of $l^{(0:d)}$ (and thus $m^{(0:d)}$), other than whether it is smaller or larger than $t[i]$. However, this can be easily deduced by observing the timing behavior of Algorithm 1 and is not a vulnerable side-channel, as was already analyzed in [GFSL24]. As rejected values are completely discarded, an attacker does not learn any information on valid samples.

Algorithm 6 describes our masked implementation for NTRU-HPS, where the number of -1 is equal to the number of 1 in the sampled vector. For Streamlined NTRU Prime, where this is not the case, we take a slightly different approach. There, we only keep track of $c_0^{(0:d)}$. For each output value, we sample a new masked random bit $y^{(0:d)}$, which determines whether the output is -1 or 1. We then securely multiplex between $(00)^{(0:d)}$ and $(y1)^{(0:d)}$ to get the output $v[i]^{(0:d)}$, using the most significant bit of $\mathrm{si}^{(0:d)} - c_0^{(0:d)}$ as the selector. This method requires us to sample a total of $n-1$ random masked bits for $y^{(0:d)}$. It has the benefit that we do not need to implement a masked version of Algorithm 3, as well as any processing related to $c_1^{(0:d)}$. We also do not need to implement the multiplexing for the second if clause (Line 21), and the computation of $\mathrm{si}^{(0:d)} - c_0^{(0:d)} - c_1^{(0:d)}$. In all, this both simplifies and accelerates the algorithm, as well as requiring less randomness for the masked gadgets. Note that this method could also be used for the unmasked designs of Streamlined NTRU Prime in Section 3. However, this would end up requiring more randomness from the RNG than simply using Algorithm 3 and only save a negligible amount of hardware, hence there is little reason to do so.

For Classic McEliece, Algorithm 6 can be simplified even more, as we only need to perform binary sampling. This allows us to remove all parts related to keeping track of $c_1^{(0:d)}$, as well as the second if-else clause in Line 21. Instead, we simply always multiplex

Algorithm 6 MaskedIterativeShuffle(n, c_0, c_1): Ternary fixed-weight sampling by shuffling, using gadget-based masking.

Input n
Input d (the masking order)
Input c_0, c_1 (the number of coefficients equal to 0 and 1, respectively)
Output $\mathbf{v}^{(0:d)}$ (masked output array of $n-1$ integers)

for i from 0 to $n-2$ **do**
  $\mathbf{t}[i] \leftarrow 2^L \bmod (n-1-i)$ ▷ Precomputation step, done prior
$c_0^{(0:d)} \leftarrow -c_0$ ▷ Stored as negative number with shares $c_0^{(1:d)}$ being zero
$c_1^{(0:d)} \leftarrow -c_1$
for i from 0 to $n-2$ **do**
  $s \leftarrow n-1-i$
  repeat
    $x^{(0:d)} \xleftarrow{\$} 0, 1, \ldots, 2^L - 1^{(0:d)}$ ▷ Sample a random masked integer
    $m^{(0:d)} \leftarrow x^{(0:d)} \cdot s$ ▷ Multiplication using a masked shift-and-add
    $l^{(0:d)} \leftarrow m^{(0:d)} \bmod 2^L$
  until $l^{(0:d)} \geq \mathbf{t}[i]$ ▷ See text on details for this step
  $\mathrm{si}^{(0:d)} \leftarrow \lfloor m^{(0:d)} / 2^L \rfloor$
  $c_0'^{(0:d)} \leftarrow c_0^{(0:d)} + 1$
  $c_1'^{(0:d)} \leftarrow c_1^{(0:d)} + 1$
  $\mathrm{sic0}^{(0:d)} \leftarrow \mathrm{si}^{(0:d)} + c_0^{(0:d)}$
  $\mathrm{sic0c1}^{(0:d)} \leftarrow \mathrm{sic0}^{(0:d)} + c_1^{(0:d)}$
  if $\mathrm{sic0}^{(0:d)} < 0$ **then** ▷ Implemented using masked multiplexer
    $\mathbf{v}^{(0:d)}[i] \leftarrow 0$
    $c_0^{(0:d)} \leftarrow c_0'^{(0:d)}$
    $c_1^{(0:d)} \leftarrow c_1^{(0:d)}$
  else if $\mathrm{sic0c1}^{(0:d)} < 0$ **then** ▷ Implemented using masked multiplexer
    $\mathbf{v}^{(0:d)}[i] \leftarrow 1$
    $c_0^{(0:d)} \leftarrow c_0^{(0:d)}$
    $c_1^{(0:d)} \leftarrow c_1'^{(0:d)}$
  else
    $\mathbf{v}^{(0:d)}[i] \leftarrow -1$
    $c_0^{(0:d)} \leftarrow c_0^{(0:d)}$
    $c_1^{(0:d)} \leftarrow c_1^{(0:d)}$
return $\mathbf{v}^{(0:d)}$

between a $(0)^{(0:d)}$ and $(1)^{(0:d)}$, and output this value.

5 Evaluation

In this section, we implement and test our designs on Xilinx FPGAs, as well as compare them to the existing hardware implementations for fixed-weight sampling in the literature. We also analyze and verify the side-channel security of our masked design using the PROLEAD tool.

5.1 FPGA Evaluation

We implement our high-speed and low-area implementation for the NTRU-HPS parameter sets NTRU-HPS509, NTRU-HPS677 and NTRU-HPS821, for the Streamlined NTRU Prime

parameter sets *sntrup*761 and *sntrup*1277, as well as the Classic McEliece parameter set mceliece8192128. *sntrup*761 is the recommended parameter set to use by the Streamlined NTRU Prime authors [BBC+20] and is also the set used by e.g. OpenSSH, while *sntrup*1277 is the highest security parameter set. Our cycle benchmark numbers are the average of 1000 runs. We report area numbers for the Xilinx Zynq™ Ultrascale+™ FPGA, and we use Vivado™ v2023.1 for synthesis and implementation. Our area numbers do not include the cost of an RNG. The area and benchmark numbers are in Table 2, together with implementations from the literature. [DMG21] and [ABPS23] do not report area numbers for their NTRU-HPS version of fixed-weight sampling, so we only list the cycle count and frequency.

5.1.1 Evaluation for NTRU and Streamlined NTRU Prime

While our low-area designs consistently use more LUT than our high-speed design for Streamlined NTRU Prime, it uses equal or slightly less LUT for NTRU-HPS. The low-area designs always consume fewer Flip-Flops (FF) and no BRAM or DSP for all parameter sets. The required clock cycles are significantly higher, up to 7.7×, though the low-area designs do achieve a slightly higher maximum clock frequency. We believe this is due to the place and route algorithm having more freedom in the placement, as no DSP at a specific position is needed. For all our low-area designs, none of the LUT are used for distributed memory.

In comparison to the fixed-weight sampling designs in the literature for NTRU-HPS, our low-area design has a higher cycle count than the design from [DMG21], being about 1.9 times slower. Our design has a higher clock frequency, leading to the total execution time being lower. Compared to [ABPS23], we have both a lower cycle count and a higher clock frequency. However, this is not an apples-to-apples comparison. The sampling modules from [DMG21] and [ABPS23] are part of a unified NTRU-HPS hardware implementation. If they were standalone, it is likely that their designs could also achieve higher clock frequencies. Our high-speed designs however, are by far the fastest across all parameter sets, having a cycle count that is between 3.9 to 8.2 times lower.

Compared to the Streamlined NTRU Prime fixed-weight sampling modules in the literature, our high-speed design outperforms all other designs by over one order of magnitude, while consuming less BRAM and, in all cases but one, less LUT and FF. Our high-speed design does however require one DSP, which no other design does. Our low-area design for Streamlined NTRU Prime is on par speed-wise with existing designs, while requiring no BRAM and fewer LUT and FF. The only exception is [Mar20] which uses a sorting network based on djbsort. It consumes even less LUT and FF, but requires one BRAM and is over 12 times slower than our low-area design.

In addition, the two designs for *sntrup*761 that are slightly faster than our low-area design, [Mar25] and [DMG21], require far more resources: [Mar25] requires nearly twice the LUT and FF, while [DMG21] consumes more than three times the LUT and six times the FF. Both also require BRAM, while our low-area design does not. In other words, duplicating our low-area design and allowing them both to operate in parallel would also result in a higher throughput while still consuming less FPGA resources than [Mar25] and [DMG21].

Our results show that the rejection-based shuffle from [GFSL24] outperforms all previously presented sorting methods in hardware for dense fixed-weight sampling, irrespective of the specific sorting algorithm. It also outperforms Fisher–Yates shuffling. Our low-area designs have a latency that is on-par or better than previous approaches, while using far less FPGA resources. At the same time, our high-speed designs are far faster than any previously published implementation. One general downside of our approach is that, like any rejection-sampling approach, there is some inherent timing variability. Although this is not a timing side-channel vulnerability, applications where strict real-time guarantees

Table 2: Our unmasked designs implemented on a Xilinx Zynq™ Ultrascale+™ FPGA, as well as a comparison with designs in the literature. The design from [CCD+22] is implemented on a Xilinx Artix™-7 FPGA. The design from [WSN17] is implemented on a Altera Stratix™ V FPGA. All others are on a Xilinx Zynq™ Ultrascale+™. [CCD+22] has a 37% success probability and requires multiple passes in the case of failure. Our high-speed implementation is marked with TW-HS, our low-area is marked with TW-LA. n is the vector length and w is the target weight. Fisher–Yates shuffle is abbreviated as FY shuffle.

Cipher	Param	Design	Method	LUT	FF	BR AM	DSP	Freq. [MHz]	Cycles	Time [μs]
NTRU	n=509, w=254	TW-HS	Filho et *al.*	248	128	0.5	1	425	518	1.219
		TW-LA	Filho et *al.*	248	122	0	0	465	3826	8.228
		[ABPS23]	FY Shuffle	n/a	n/a	n/a	n/a	400	∼4250	∼10.6
	n=677, w=254	TW-HS	Filho et *al.*	298	225	0.5	1	430	686	1.595
		TW-LA	Filho et *al.*	296	128	0	0	470	5098	10.84
		[DMG21]	mergesort	n/a	n/a	n/a	n/a	250	2678	10.71
	n=821, w=510	TW-HS	Filho et *al.*	324	237	0.5	1	435	831	1.910
		TW-LA	Filho et *al.*	314	129	0	0	475	6327	13.32
		[DMG21]	mergesort	n/a	n/a	n/a	n/a	250	3343	13.37
SNTRUP	n=761, w=286	TW-HS	Filho et *al.*	341	191	0.5	1	415	790	1.904
		TW-LA	Filho et *al.*	438	138	0	0	440	5860	13.32
		[PMT+23]	radixsort	823	220	2	0	285	4837	16.97
		[DMG21]	mergesort	1441	940	3.5	0	250	2762	11.05
		[Mar20]	djbsort	231	87	1	0	279	49 400	177.1
		[Mar25]	radixsort	844	285	2	0	400	4842	12.11
	n=1277, w=492	TW-HS	Filho et *al.*	511	299	1	1	420	1323	3.15
		TW-LA	Filho et *al.*	638	149	0	0	440	10 302	23.41
Classic McEliece	n=8192, w=128	TW-HS	Filho et *al.*	1933	380	7	1	380	8234	21.67
		TW-LA	Filho et *al.*	1865	240	0	0	333	78 147	234.7
		[DMG21]	mergesort	2533	1589	33	0	250	26 646	106.4
		[ZZC+23]	mergesort	2510	3887	20	0	300	24 598	81.99
		[WSN18]	mergesort	583	411	20	0	250	147 505	590.0
		[CCD+22]	Rejection	304	148	2	0	262	1900	7.1
		[WSN17]	FY Shuffle	149	111	7	0	335	23 635	70.55

are needed may prefer a sorting-based sampling, as these are fully constant time. Note that Fisher–Yates shuffle also has a variable runtime, as the index sampling within the shuffle also uses rejection sampling to be bias-free. We conclude that the rejection-based shuffle from [GFSL24] is highly suitable for hardware implementations of cryptoschemes that employ dense fixed-weight sampling.

5.1.2 Evaluation for Classic McEliece

Unlike the designs for Streamlined NTRU Prime and NTRU, the low-area design for Classic McEliece has a lower clock frequency than our high-speed design. The lower frequency is due to a longer critical path in the iterative multiplier, which has larger width than the one for the NTRU-based designs. The low-area design however does use no BRAM or DSP, as well as less LUT and FF.

Our high-speed design for Classic McEliece has a lower cycle count than all other sorting-based designs in the literature [DMG21, ZZC+23, WSN18], as well as a higher clock frequency. Our design is also faster than the Fisher–Yates based shuffle in [WSN17], though it does use far more LUT and FF. However, it is slower than the rejection-based approach from [CCD+22]. The comparison with [CCD+22] is made somewhat more complicated in two aspects: First, [CCD+22] only has a success probability of 37%. In other words, it is highly likely that multiple passes are required, which increases the average cycle count to obtain an output. The second aspect is that implementations for the Artix™-7 used by [CCD+22] generally have a lower maximum clock frequency than for our Zynq™ Ultrascale+™. The design of [CCD+22] would have a higher clock frequency when implemented on a similar FPGA.

With regards to memory, our high-speed design for Classic McEliece requires less BRAM than any of the sorting-based designs, though it does require one DSP. In addition, our low-area design is the only implementation that does not require any BRAM at all. However, this comes at the cost of being 2.2, 2.9 and 33.7 times slower respectively than the designs from [ZZC+23], [DMG21] and [CCD+22], though still 2.5 times faster than the design from [WSN18].

Based on these results, we conclude that the rejection-based shuffle from [GFSL24] is not suitable for hardware implementations for Classic McEliece and other code-based schemes such as HQC that use sparse fixed-weight sampling. While it is preferable over sorting-based methods, methods such as [CCD+22] are preferable, as they are both faster and use less hardware resources. This matches our expected behavior, and also corroborates results from [GFSL24, GFAB+26].

5.2 FPGA Evaluation of Masked Design

We implement our masked implementation for the NTRU parameter sets NTRU-HPS509, NTRU-HPS677 and NTRU-HPS821, for the Streamlined NTRU Prime parameter sets *sntrup*761 and *sntrup*1277 as well as the Classic McEliece parameter set mceliece8192128. We give numbers for masking order $d = \{1, 2, 3\}$. Our cycle benchmark numbers are the average of 300 runs. We report FPGA area numbers on the Zynq™ Ultrascale+™ FPGA, and we use Vivado™ v2023.1 for synthesis and implementation. The area and benchmark numbers are in Table 3. We separately list the randomness required to refresh the HPC2 gadgets each cycle. This randomness is needed to ensure the security of the gadget, and is used internally to refresh masked variables [CGLS21]. We list it separately as, though it must be refreshed every clock cycle, the quality of this randomness is not as strict as that for a cryptographically secure RNG [CMM+24]. To our knowledge, this is the first masked fixed-weight sampling hardware implementation for these schemes in the literature, so we cannot compare it with other implementations, as [KLRBG23] only presented hardware implementations for BIKE. Instead, we compare it to our unmasked low-area design (where masking order $d = 0$).

As expected, we can see in Table 3 that adding masking increases both the area (LUT and FF) consumption and cycle count compared to the unmasked design, for all three PQC algorithms. The area increases further when the masking order increases, though the cycle count stays the same. This behavior is due to the HPC2 gadgets and was also observed in the gadget-based masked implementation of Streamlined NTRU Prime [LMRBG24]: The HPC2 secure AND gadget always has a latency of 2 clock cycles, independent of the masking order. Instead, the only factor influencing the cycle count is the parameter set, specifically the polynomial (or vector) degree.

Like the low-area designs, our masked implementations do not require any BRAM or DSP. For $d = 1$, the maximum clock frequency is higher than our low-area design for all parameter sets. However, as the masking order increases, the maximum clock frequency drops for NTRU-HPS and Streamlined NTRU Prime. This is due to the routing complexity of the masked Sklansky adder. Interestingly, this is not the case for Classic McEliece where the max clock frequency increases for higher masking order. The precise reason for this behavior is unfortunately not known to us, and we expect the clock frequency to decrease again for even higher masking orders.

Finally, the randomness that is consumed per cycle for refreshing the HPC2 gadgets is almost entirely independent of the parameter set. It is slightly less for the NTRU-HPS509 parameter set, and only slightly more for the *sntrup*1277 parameter set. This is due to their polynomial degree being 9-bit and 11-bit respectively, rather than 10-bit. Classic McEliece also requires additional randomness, as its vector size is 14-bit.

Table 3: Our masked design for fixed-weight sampling, implemented on a Xilinx Zynq™ Ultrascale+™ FPGA for different masking orders d. For $d = 0$, we use the numbers of our low-area design. The column Rand. bits/cycle lists the randomness per cycle needed to refresh the masking gadgets. Non of the designs require any BRAM or DSP.

Cipher	Parameter	Order	LUT	FF	Freq. [MHz]	Cycles	Time [μs]	Rand. bits/cycle
NTRU-HPS	$n = 509$	$d = 0$	248	122	465	3826	8.228	0
		$d = 1$	1596	1556	540	35 297	65.36	52
		$d = 2$	2847	3001	480	35 297	73.54	156
		$d = 3$	4796	5096	440	35 297	80.22	312
	$n = 677$	$d = 0$	296	128	470	5098	10.84	0
		$d = 1$	1727	1593	500	47 063	94.13	53
		$d = 2$	3023	3074	495	47 063	95.08	159
		$d = 3$	4993	5043	480	47 063	98.05	318
	$n = 821$	$d = 0$	314	129	475	6327	13.32	0
		$d = 1$	1748	1594	495	58 853	118.9	53
		$d = 2$	3049	3080	495	58 853	118.9	159
		$d = 3$	5013	5044	480	58 853	122.6	318
SNTRUP	$n = 761$	$d = 0$	438	138	440	5860	13.32	0
		$d = 1$	1609	1513	495	54 188	109.5	53
		$d = 2$	2841	2955	480	54 188	112.9	159
		$d = 3$	4752	4873	485	54 188	111.7	318
	$n = 1277$	$d = 0$	638	149	440	10 302	23.41	0
		$d = 1$	1738	1565	505	96 760	191.6	54
		$d = 2$	3017	3041	490	96 760	197.5	162
		$d = 3$	4971	5204	440	96 760	219.9	324
Classic McEliece	$n = 8192$	$d = 0$	1865	240	333	78 147	234.7	0
		$d = 1$	2576	1734	365	812 423	2226	57
		$d = 2$	3942	3306	385	812 423	2110	171
		$d = 3$	6019	5410	405	812 423	2006	342

5.3 Side-Channel Evaluation

Evaluating the side-channel security of a masking design is not trivial. However, using HPC2 gadgets has a benefit here, as they are trivially composable according to the PINI concept explained in Section 2.4. As a result, any combination of them is still secure. However, there are some things that could still damage security, for example by not properly connecting the randomness needed to refresh a gadget. For this reason, we use the side-channel analysis tool PROLEAD [MM22] to verify the first-order probing security of our masked iterative multiplier. This component is used for the fixed-weight sampling for all masked designs, and is one of two entities that operates on masked values. The other entity is the masked multiplexer used to implement the if-else clauses in Algorithm 6. We do not additionally analyze the masked multiplexer, as its side-channel security was formally verified in [LMRBG24].

As a sanity check that our setup is working correctly, we analyzed the masked Sklansky adder while disabling the randomness for refreshing the gadgets. The results of the PROLEAD analysis are shown in Figure 3. We can see that the $-\log_{10}(p)$-value grows very quickly and reaches infinity after around 3800 simulations (to be precise, the p value underflows to zero).

Our initial approach is to analyze the entire masked iterative multiplier as a single entity. However, this turns out to be infeasible for PROLEAD when using the default configuration: Even on a machine with 128GB of RAM (the largest available to us), the tool runs out of memory after processing roughly 180 000 simulations. The required number of simulations for statistical confidence is also not reached. However, note this this is not a tool limitation, a machine with more RAM (e.g. 1TB) would likely be able to complete the run. This incomplete run is depicted in Figure 4.

We repeat the simulation with PROLEAD running in *compact* mode. In compact mode, PROLEAD uses the Hamming weight of signals to check for leakages, rather than the full signal value. This reduces the accuracy of the simulation, but also greatly reduces the

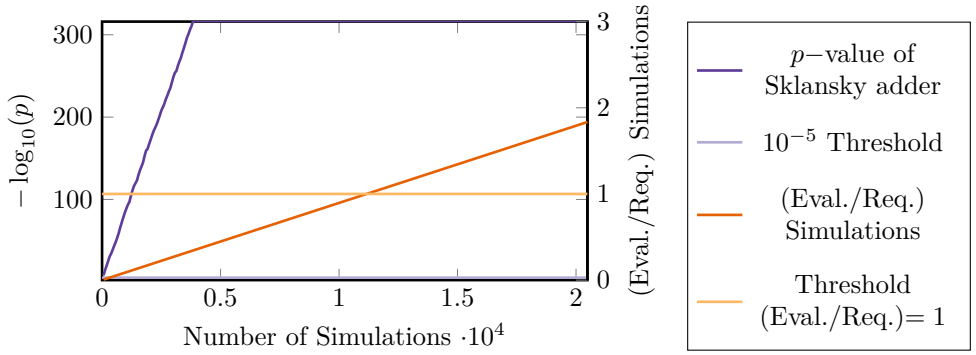

Figure 3: First-order PROLEAD simulation of the first-order 16-bit masked Sklansky adder, with the randomness for refreshing the masking gadgets disabled. As expected, the $-\log_{10}(p)$-value rises very quickly, with the threshold crossed immediately. The $-\log_{10}(p)$ value reaches infinity after around 3800 simulations. To be exact, p (stored in a 64-bit float) underflows to zero. As a result $-\log_{10}(p)$ cuts off at around 320, as $\lim_{x\to 0^+} -\log_{10}(x) = \infty$, and the smallest positive number a 64-bit float can contain is $\sim 10^{-320}$.

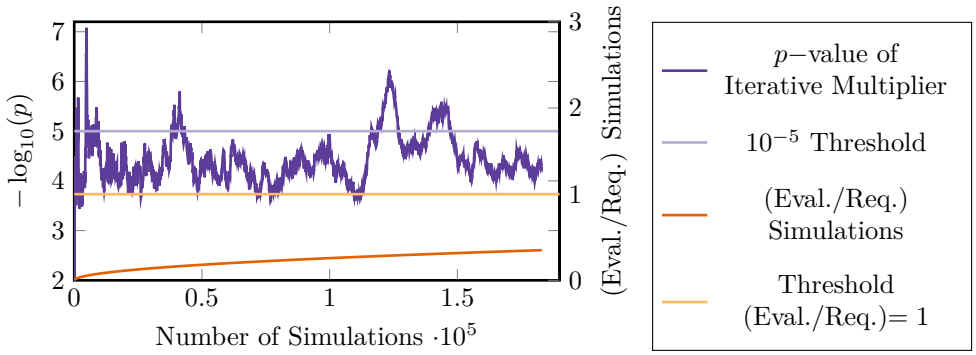

Figure 4: Incomplete first-order PROLEAD simulation of the first-order 15-bit masked multiplier, with the adder included. Due to running out of memory, only 180 000 simulations are performed. The p-value is below the threshold of 10^{-5} in the end, though the threshold of (Eval./Req.)= 1 for statistical confidence is not reached.

memory consumption [Mü25]. We ran the simulation for a total of 100 million traces, the results are in Figure 5. Although the 10^{-5} threshold for the p-value is exceeded at certain intervals, it stays below the threshold for the majority of the time. The PROLEAD documentation states that when there are a large number of probing sets (the number of positions where probes can be placed, which increases as the circuit grows more complex), the chance of false positives increases [Mü25]. The documentation also lists 100 million as the minimum number of simulations when running in compact mode. As the p-value does not continuously increase as in Figure 3, we concluded that the temporary exceeding of the threshold is a false-positive.

For further analysis, we split the circuit, analyzing the Sklansky adder separately from the masked iterative multiplier. The PROLEAD evaluation of the Sklansky adder is shown in Figure 6, where we performed a total of 5 million simulations. The p-value stays below the threshold except for a brief moment after around 3.5 million traces, and does not rise continuously. For this reason, we conclude that the adder does not leak. The results for the masked iterative multiplier with the adder excluded are shown in Figure 7, also with a

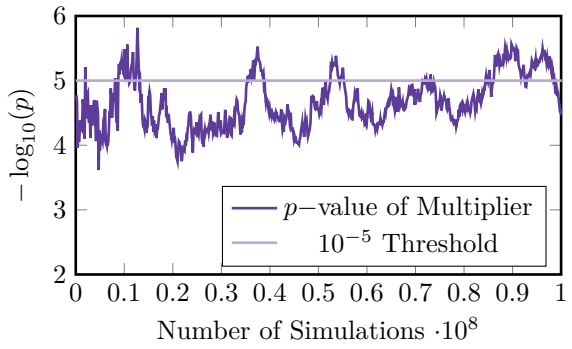

Figure 5: First-order PROLEAD simulation of the first-order 15-bit masked multiplier, with the adder included, in compact mode. In compact mode, PROLEAD uses the Hamming weight of signals to check for leakages, rather than the actual signal value. PROLEAD does not output the (Eval./Req.) for statistical confidence when running in compact mode. Instead we run more simulations, for a total of 100 million.

total of 5 million simulations. At no point does the p-value exceed the threshold. We thus conclude that our masked design does not exhibit leakages.

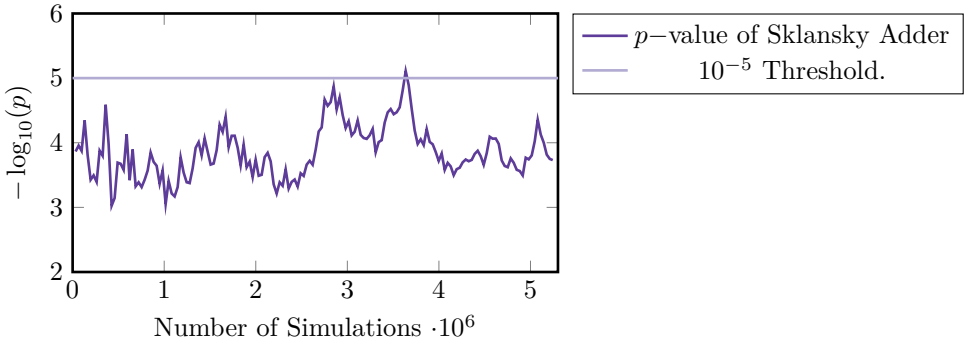

Figure 6: First-order PROLEAD simulation of the first-order 16-bit masked Sklansky adder with 5 million traces. The (Eval./Req.) simulations plot for statistical significance is not depicted, as the requirement is reached after just 30 000 simulations.

An aspect that we exclude from our side-channel analysis is the RNG used to generate the randomness for the sampling, as well as the RNG to create the randomness to refresh the gadgets. Especially the RNG for the sampled values (Line 8 in Algorithm 6) would also require side-channel protection. However, these parts aren't unique to our design, and are a requirement for all masked sampling implementations. For this reason, we consider the RNG to be out of scope of our analysis.

6 Conclusion

In this work, we presented multiple hardware implementations for fixed-weight sampling, a key part of PQC algorithms such as NTRU-HPS and Classic McEliece. We optimized each implementation with different targets in mind: Maximum performance, minimal area, or side-channel security. Our designs outperform existing hardware implementations of fixed-weight sampling for NTRU-HPS and Streamlined NTRU Prime in the literature, having either better performance, lower area, or both. Our low-area implementations are also

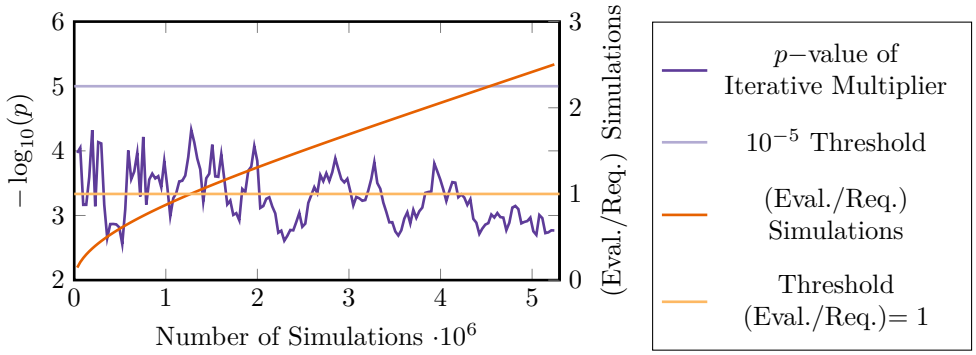

Figure 7: First-order PROLEAD Simulation of the first-order 15-bit masked multiplier, with the 16-bit masked Sklansky adder excluded. The required number of simulations for statistical confidence is reached after around 1 200 000 simulations.

the only fixed-weight sampling methods that do not require any memory blocks. To our knowledge, our masked design is the first side-channel protected hardware implementation for fixed-weight sampling for NTRU-like schemes in the literature. For Classic McEliece, our designs outperform previous implementations that utilize sorting, but cannot keep up with rejection-based approaches that are designed for sparse fixed-weight sampling.

Our work shows that the performance and efficiency improvements of the new fixed-weight sampling algorithm from [GFSL24] also extend to hardware implementations. We thus support the conclusion from [GFSL24] that this new approach should be considered for future standardization of PQC algorithms that utilize fixed-weight sampling, especially when the to-be-sampled vectors are dense. This includes all cryptoschemes from Table 1 where $n/w \geq 0.1$. Our work also confirms statements from [GFAB+26] that this type of fixed-weight sampling algorithm is not efficient when applied to sparse vectors such as those found in code-based algorithms (in Table 1, those with $n/w \leq 0.03$.).

With regards to future work, one aspect of interest which we did not address are ASIC implementations. An ASIC synthesis of our low-area design would be particularly interesting, as the design requires no memory or multiplier blocks, both of which can consume significant silicon area. As our low-area design is the only implementation of fixed-weight sampling in the literature that does not require any memory, this could result in a very small ASIC footprint compared to other designs. Our masked fixed-weight sampling design can also be used to enable side-channel secure hardware implementations of NTRU-HPS and Streamlined NTRU Prime encapsulation and key generation, as these operations require fixed-weight sampling. A further interesting future work would be the application of other masking schemes, such as domain-oriented masking [GMK16], to our hardware designs. In addition, methods such as those from [SVGV26] could be applied to reduce the randomness consumption of the first-order masked designs. Finally, our design could also be applied to hardware implementations of SMAUG-T.

7 Acknowledgments

We would like to thank the anonymous reviewers for their constructive criticisms and feedback, as well as Mustafa Ibrahim and Melissa Azouaoui. This work was supported by the Federal Ministry of Research, Technology and Space (BMFTR) of the Federal Republic of Germany (grant 16KIS1572K, SASVI), as well as the European Commission under the grant agreement number 101070374 (CONVOLVE). This work was supported by the European Cybersecurity Competence Centre under the European Union's Horizon Europe

research and innovation program under grant agreement number 101225859 (FOCAL).

References

[AAB+22] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. HQC. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[ABB+22] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar-Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zémor, Valentin Vasseur, Santosh Ghosh, and Jan Richter-Brokmann. BIKE. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[ABC+22] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions.

[ABPS23] Francesco Antognazza, Alessandro Barenghi, Gerardo Pelosi, and Ruggero Susella. A Flexible ASIC-Oriented Design for a Full NTRU Accelerator. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, ASPDAC '23, page 591–597, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3566097.3567916.

[BBC+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, Berlin, Heidelberg, April 2015. doi:10.1007/978-3-662-46800-5_18.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 116–129. ACM Press, October 2016. doi:10.1145/2976749.2978427.

[BCLvV17] 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: 24th Annual International Workshop on Selected Areas in Cryptography*, volume 10719 of *Lecture Notes in Computer Science*, pages 235–260. Springer, Cham, August 2017. doi:10.1007/978-3-319-72565-9_12.

[Ber26] Daniel J. Bernstein. djbsort, 2026. [Online; accessed 21-05-2026]. URL: https://sorting.cr.yp.to/index.html.

[BG22] Florian Bache and Tim Güneysu. Boolean masking for arithmetic additions at arbitrary order in hardware. *Applied Sciences*, 12(5), 2022. URL: https://www.mdpi.com/2076-3417/12/5/2274, doi:10.3390/app12052274.

[BGI+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, Cham, April / May 2018. doi:10.1007/978-3-319-78375-8_11.

[BLH+25] Fabian Buschkowski, Georg Land, Niklas Höher, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. HADES: Automated Hardware Design Exploration for Cryptographic Primitives. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(4):1–45, Sep. 2025. URL: https://tches.iacr.org/index.php/TCHES/article/view/12402, doi:10.46586/tches.v2025.i4.1-45.

[cB25] brian m. carlson and Taylor Blau. Post-quantum security for SSH access on GitHub. https://github.blog/engineering/platform-security/post-quantum-security-for-ssh-access-on-github/, 2025. [Online; accessed 11-11-2025].

[CCC+26] Jung Hee Cheon, Hyeongmin Choe, Joongeun Choi, Dongyeon Hong, Jeongdae Hong, Chi-Gon Jung, Honggoo Kang, Janghyun Lee, Seonghyuck Lim, Aesun Park, et al. SMAUG-T: the Key Exchange Algorithm based on Module-LWE and Module-LWR, 2026. URL: https://kpqc.cryptolab.co.kr/smaug-t.

[CCD+22] Po-Jen Chen, Tung Chou, Sanjay Deshpande, Norman Lahr, Ruben Niederhagen, Jakub Szefer, and Wen Wang. Complete and improved FPGA implementation of classic McEliece. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):71–113, 2022. doi:10.46586/tches.v2022.i3.71-113.

[CDH+20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hulsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRU. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware Private Circuits: From Trivial Composition to Full Verification. *IEEE Transactions on Computers*, 70(10):1677–1690, 2021. doi:10.1109/TC.2020.3022979.

[CMM+24] 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 Communications in Cryptology (CiC)*, 1(2):4, 2024. doi:10.62056/akdkp2fgx.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference. *IEEE Transactions on Information Forensics and Security*, 15:2542–2555, 2020. doi:10.1109/TIFS.2020.2971153.

[DMG21] V Dang, Kamyar Mohajerani, and Kris Gaj. High-speed hardware architectures and fair FPGA benchmarking of CRYSTALS-Kyber, NTRU, and Saber. In *The NIST Third Standardization Conference*, pages 1–48, 2021.

[DXN+24] Sanjay Deshpande, Chuanqi Xu, Mamuri Nawan, Kashif Nawaz, and Jakub Szefer. Fast and efficient hardware implementation of HQC. In Claude Carlet, Kalikinkar Mandal, and Vincent Rijmen, editors, *SAC 2023: 30th Annual International Workshop on Selected Areas in Cryptography*, volume 14201 of *Lecture Notes in Computer Science*, pages 297–321. Springer, Cham, August 2024. doi:10.1007/978-3-031-53368-6_15.

[FGM+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):89–120, 2018. URL: https://tches.iacr.org/index.php/TCHES/article/view/7270, doi:10.13154/tches.v2018.i3.89-120.

[FMJ26] Markus Friedl, Jan Mojzis, and Simon Josefsson. Secure Shell (SSH) Key Exchange Method Using Hybrid Streamlined NTRU Prime sntrup761 and X25519 with SHA-512: sntrup761x25519-sha512. RFC 9941, April 2026. URL: https://www.rfc-editor.org/info/rfc9941, doi:10.17487/RFC9941.

[GFAB+26] Décio Luiz Gazzoni Filho, Gora Adj, Slim Bettaieb, Alessandro Budroni, Jorge Chávez-Saab, and Francisco Rodríguez-Henríquez. Sota voce: Low-noise sampling of sparse fixed-weight vectors. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(1):353–375, Jan. 2026. URL: https://tches.iacr.org/index.php/TCHES/article/view/12685, doi:10.46586/tches.v2026.i1.353-375.

[GFSL24] Décio Luiz Gazzoni Filho, Tomás S. R. Silva, and Julio López. Efficient isochronous fixed-weight sampling with applications to NTRU. *IACR Communications in Cryptology*, 1(2), 2024. doi:10.62056/a6n59qgxq.

[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 Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):223–263, 2022. doi:10.46586/tches.v2022.i3.223-263.

[GMK16] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security*, TIS '16, page 3, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2996366.2996426.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, Berlin, Heidelberg, August 2003. doi:10.1007/978-3-540-45146-4_27.

[KAA21] Emre Karabulut, Erdem Alkim, and Aydin Aysu. Single-Trace Side-Channel Attacks on w-Small Polynomial Sampling: With Applications to NTRU, NTRU Prime, and CRYSTALS-DILITHIUM. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 35–45, 2021. doi:10.1109/HOST49136.2021.9702284.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, Berlin, Heidelberg, August 1999. doi:10.1007/3-540-48405-1_25.

[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: 26th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 13941 of *Lecture Notes in Computer Science*, pages 94–124. Springer, Cham, May 2023. doi:10.1007/978-3-031-31371-4_4.

[Kpq26] KpqC. Selected Algorithms from the KpqC Competition Round 2, 2026. [Online; accessed 21-05-2026]. URL: https://kpqc.or.kr/competition_02.html.

[LMRBG24] Georg Land, Adrian Marotzke, Jan Richter-Brockmann, and Tim Güneysu. Gadget-based masking of streamlined NTRU prime decapsulation in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(1):1–26, 2024. doi:10.46586/tches.v2024.i1.1-26.

[Mar20] Adrian Marotzke. A Constant-Time Full Hardware Implementation of Streamlined NTRU Prime. In Pierre-Yvan Liardet and Nele Mentens, editors, *Smart Card Research and Advanced Applications - 19th International Conference, CARDIS 2020, Virtual Event, November 18-19, 2020, Revised Selected Papers*, volume 12609 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2020. doi:10.1007/978-3-030-68487-7_1.

[Mar25] Adrian Marotzke. *Secure and efficient hardware implementations of NTRU Prime*. PhD thesis, Hamburg University of Technology, 2025. https://doi.org/10.15480/882.15172.

[McD14] John H McDonald. *Handbook of biological statistics*. Sparky House Publishing, Baltimore, Maryland. USA, 2014.

[MM22] Nicolai Müller and Amir Moradi. PROLEAD A probing-based hardware leakage detection tool. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):311–348, 2022. doi:10.46586/tches.v2022.i4.311-348.

[Mul23] Mullvad. Stable quantum-resistant tunnels in the app, 2023. [Online; accessed 11-11-2025]. URL: https://mullvad.net/en/blog/2023/4/6/stable-quantum-resistant-tunnels-in-the-app.

[Mü25] Nicolai Müller. PROLEAD Wiki. https://github.com/ChairImpSec/PROLEAD/wiki/Results#interpretation, 2025. [Online; accessed 11-11-2025].

[NIS26] NIST. Post-Quantum Cryptography, 2026. [Online; accessed 15-07-2026]. URL: https://csrc.nist.gov/projects/post-quantum-cryptography.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *ICICS 06: 8th International Conference on Information and Communication Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, Berlin, Heidelberg, December 2006. doi:10.1007/11935308_38.

[Ope22] OpenSSH Team. OpenSSH Release 9.0. https://www.openssh.org/txt/release-9.0, 2022. [Online; accessed 11-11-2025].

[PMT+23] Bo-Yuan Peng, Adrian Marotzke, Ming-Han Tsai, Bo-Yin Yang, and Ho-Lin Chen. Streamlined NTRU prime on FPGA. *Journal of Cryptographic Engineering*, 13(2):167–186, June 2023. doi:10.1007/s13389-022-00303-z.

[RBFSG22] Jan Richter-Brockmann, Jakob Feldtkeller, Pascal Sasdrich, and Tim Güneysu. VERICA - verification of combined attacks automated formal verification of security against simultaneous information leakage and tampering. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):255–284, 2022. doi:10.46586/tches.v2022.i4.255-284.

[RM25] Hemin Rahimi and Amir Moradi. One More Motivation to Use Evaluation Tools, This Time for Hardware Multiplicative Masking of AES. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7, 2025. doi:10.23919/DATE64628.2025.10993221.

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

[Skl60] J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(2):226–231, 1960. doi:10.1109/TEC.1960.5219822.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, Berlin, Heidelberg, September 2015. doi:10.1007/978-3-662-48324-4_25.

[SVGV26] Dilip Kumar S. V, Benedikt Gierlichs, and Ingrid Verbauwhede. A graph-theoretic framework for randomness optimization in first-order masked circuits. In *2026 Design, Automation and Test in Europe Conference (DATE)*, pages 1–7, 2026. doi:10.23919/DATE69613.2026.11539466.

[Tea26] Classic McEliece Team. Classic McEliece: ISO, 2026. [Online; accessed 15-07-2026]. URL: https://classic.mceliece.org/iso.html.

[WSN17] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based key generator for the Niederreiter cryptosystem using binary Goppa codes. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 253–274. Springer, Cham, September 2017. doi:10.1007/978-3-319-66787-4_13.

[WSN18] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-Based Niederreiter Cryptosystem Using Binary Goppa Codes. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 77–98, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-79063-3_4.

[Xil18] Xilinx, Inc. *UG479: 7 Series DSP48E1 Slice*, 1.10 edition, March 2018. https://docs.amd.com/v/u/en-US/ug479_7Series_DSP48E1.

[ZZC+23] Yihong Zhu, Wenping Zhu, Chen Chen, Min Zhu, Zhengdong Li, Shaojun Wei, and Leibo Liu. Mckeycutter: A High-throughput Key Generator of Classic McEliece on Hardware. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023. doi:10.1109/DAC56929.2023.10247918.

A Sage Script for calculating FIFO depth

```
1  L = 16
2  #L = 19
3
4  # NTRU for degree n, samples a vector of length n-1.
5  # NTRU Prime and  Classic McEliece, for degree n,
6  # sample a vector of length n.
7  # As the code below assumes vector length of n-1, the
8  # NTRU Prime and  Classic McEliece degrees
9  # are increased by 1 in the n_array
10 # to compensate for this.
11 n_array = [509,677,762,821,1278, 8193]
12
13 # Different FIFO depths.
14 fifo_depth_array = [16,32,64,128,256]
15
16 for n in n_array:
17     print("Array length:", n-1)
18
19     min_prop=min([(1-(2^L%(n-1-i))/2^L) for i in range(n-1)])
20     print("Worst case rejection probability:", RR(1-min_prop))
21
22     avg_prop=mean([(1-(2^L%(n-1-i))/2^L) for i in range(n-1)])
23     print("Mean rejection probability:", RR(1-avg_prop))
24
25     for fifo_depth in fifo_depth_array:
26         f = sum(binomial(n-1,k)*(1-min_prop)**k*min_prop**(n-1-k
               ) for k in range(fifo_depth+1,n-1))
27         print("Upper bound of probability of exceeding FIFO
               depth of", fifo_depth, "is", RR(f), ", 2^", RR(
               log(f,2)),sep='')
28
29         f_avg = sum(binomial(n-1,k)*(1-avg_prop)**k*avg_prop**(n
               -1-k) for k in range(fifo_depth+1,n-1))
30         print("Estimated probability (based on mean rejection
               probability) of exceeding FIFO depth of",
               fifo_depth, "is", RR(f_avg), ", 2^", RR(log(f_avg
               ,2)),sep='')
31     print("")
```

Listing 1: A Sage script to compute the probability for a given sampled vector length that the number of rejected samples during the first pass exceeds a certain FIFO depth. We ran this with SageMath 9.2. In addition to using the worst-case probability for a rejection, the script also computes an estimated probability using the average rejection probability.